Informática

Lógica de programación

Omar Iván **Trejos** Buriticá

ediciones de la U

Informática

Lógica de Programación

Omar Iván **Trejos** Buriticá

ediciones de la U

Trejos Buriticá, Omar Iván
Lógica de programación -- 1a. edición. Bogotá : Ediciones de la U, 2017.
432 p. ; 24 cm.
ISBN 978-958-762-720-6 e-ISBN 978-958-762-721-3
1. Programación 2. Lógica 3. Variables, constantes y operadores
4. Algoritmos 5. Ciclos 6. Matrices I. Tít.
519.7cd 21 ed.

Área: Informática
Primera edición: Bogotá, Colombia, noviembre de 2017
ISBN. 978-958-762-720-6

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: Digiprint Editores SAS
Calle 63 #70D-34, Pbx. (57+1) 7217756

Impreso y hecho en Colombia
Printed and made in Colombia

ediciones de la U

Apreciad@ cliente:

Es gratificante poner en sus manos estas obras, por esta razón le invitamos a que se registre en nuestra web: **www.edicionesdelau.com** y obtenga beneficios adicionales como:

- **Complementos digitales de las obras**
- **Actualizaciones de las publicaciones**
- **Interactuar con los autores a través del blog**
- **Información de nuevas publicaciones de su interés**
- **Noticias y eventos**

Adquiere nuestras publicaciones en formato e-book

Visítanos en:

www.edicionesdelau.com

Sus pedidos a:
Carrera 27 # 27-43 • Barrio Teusaquillo
PBX. (57-1) 3203510 • (57-1) 3203499 • Móvil: 310 - 6256033
comercial@edicionesdelau.com - gerencia@edicionesdelau.com
Bogotá - Colombia

Av. Coyoacán 1812 A. Acacias Benito Juárez C.P. 03240
PBX. (52) 55-63051703 • Cel. 044 5544439418
janethcr@gruporamadelau.com
México D. F. - México

Contenido

A Natalia y a Juan José,
¡¡¡mi maravilloso todo!!!

Introducción

Durante muchos años he dedicado gran parte de mi tiempo no solo a la enseñanza de la Lógica de Programación, sino al análisis de la enseñanza de dicha Lógica debido, precisamente, a que me he encontrado con que muchas personas confunden la Programación con la Lógica de Programación. La primera involucra el conocimiento de técnicas e instrucciones de un determinado lenguaje a través de los cuales se hace sencillo lograr que el computador obtenga unos resultados mucho más rápido que nosotros. La segunda involucra, de una manera técnica y organizada, los conceptos que nos permiten diseñar, en términos generales, la solución a problemas que pueden llegar a ser implementados a través de un computador.

El estudio de la Lógica de Programación no exige ningún conocimiento previo de computadores ni de tecnología en general; tampoco exige la presencia de algún lenguaje de programación específico, aunque no puedo negarle que este podría permitirle, solo después de que usted maneje bien los conceptos de Lógica de Programación, la implementación de las soluciones lógicas a sus objetivos.

Fueron muchos los alumnos que con el tiempo me fueron solicitando que les enseñara cuáles eran los conceptos realmente básicos para aprender a programar, o sea, aquellos conceptos con los cuales es suficiente para enfrentarse a cualquier lenguaje de programación o, mejor aún, enfrentarse a lograr cualquier objetivo a través de un computador. Poco a poco, fui buscando soluciones a las preguntas que mis alumnos me planteaban y veía que, en sus dudas, siempre estaba presente la búsqueda de conceptos esenciales que los liberara de las ataduras que tienen los lenguajes de programación cuando estos son lo primero que se conoce en computadores.

Luego de muchos años de estudio de estos factores, pude condensar en este libro los que, considero, son los conceptos fundamentales para aprender realmente a programar, o sea, lo que he llamado *la esencia de la Lógica de Programación*, pues busco que usted conozca estos elementos conceptuales y, luego

de dominarlos, se enfrente sin ningún problema no solo a cualquier objetivo que pueda ser alcanzable a través de computadores, sino además a cualquier lenguaje de programación.

Puedo garantizarle que, si usted lee este libro hoja por hoja y desarrolla los ejercicios aquí planteados, al llegar al final del mismo, podrá entender que programar no es más que buscar soluciones muy lógicas utilizando unos conceptos muy sencillos. Espero, pues, que este libro cumpla el objetivo planteado, pues pensando en usted fue como se concibió. No se vaya a afanar por leerlo de una sola vez; tómese su tiempo para razonar y asimilar los conceptos que aquí se plantean. No olvide que, para resolver problemas computables, usted no aplica su propia lógica, sino que toma prestada una lógica que no es la natural.

Este libro en ninguna de sus partes le mostrará conceptos complejos, debido precisamente a que la lógica de programación es la unión de muchos (pero muchos) conceptos sencillos para el diseño de soluciones muy (pero muy) lógicas.

Omar Iván Trejos Buriticá, PhD

¿Cómo usar este libro?

En primera instancia, le recomiendo que lo lea pausadamente. No se afane en avanzar; la apropiación y asimilación de los conceptos pertinentes a la lógica de programación implica tiempo, pues no son exactamente los mismos conceptos que subyacen a la lógica natural y deliberativa que tenemos los seres humanos.

Lea y vuelva a leer, piense en lo que ha leído y, ante todo, realice (o, por lo menos, intente realizar) los ejercicios propuestos. Pregunte cuando tenga dudas y siempre propóngase terminar los ejercicios. No los deje a medio camino, pues cada vez que usted lleve un ejercicio hasta el final verá cómo su lógica humana se amplía para acudir a la lógica computacional en los casos donde corresponda.

Para el profesor

Utilice este libro al ritmo que sus estudiantes le permitan. Usted y yo sabemos que lo más importante no es avanzar en un contenido, sino que los estudiantes realmente aprendan lo que se puede asimilar de ese contenido. Recuerde que asimilar una nueva lógica toma tiempo y lo que para algunos puede ser muy obvio para otros puede ser complejo. Esta consideración será de gran utilidad para intentar comprender los diferentes niveles de aprendizaje y apropiación de la lógica por parte de cada uno de los estudiantes que son y serán nuestra razón de ser.

Para el estudiante

Siga el ritmo que le indique el profesor. Revise los ejercicios resueltos, resuelva los ejercicios propuestos y, siempre, sin excepción, pregunte. No se quede con dudas. Por simples que le parezcan, recuerde que, cuando se trata de Lógica, las dudas son también lógicas, pero si no se resuelven, simplemente van quedando lagunas cuya resolución posteriormente puede llegar a ser más

compleja. Comparta soluciones con sus compañeros. Es muy enriquecedor alimentarse de la lógica de los demás y aportarles lo que, desde nuestra lógica, se pueda aportar. La forma excelsa para aprender a nadar es nadando; asimismo, el camino óptimo para asimilar la lógica de programación es practicando, practicando y practicando.

¿Qué es programar?

Programar es encontrar soluciones, basadas en lógica de programación, que permiten que el computador alcance por nosotros un determinado objetivo. El artífice de que el computador logre dichos objetivos es la persona que lo programó.

Capítulo 1

La Lógica

1.1. Hablando de Lógica

Recuerdo que, en mi niñez, alguna vez me abroché mal la camisa, en un momento en que toda mi familia estaba afanada por salir. Una familiar me vio la camisa mal abrochada y me dijo fuertemente que me había abrochado mal la camisa, que si era que yo no tenía lógica. Luego de acomodármela adecuadamente, o sea, de manera que cada botón coincidiera con su respectivo ojal, comencé a pensar que era posible que no tuviera lógica, pues me parecía sorprendente que no me hubiera dado cuenta de que para que la camisa estuviera colocada correctamente solo hay una forma y es que coincidan par botón-ojal. Llegué a otra conclusión y es el hecho de que es más fácil ponerse bien una camisa que ponérsela mal o, dicho en otras palabras, es muchísimo más fácil colocársela correctamente en lo que a botones y ojales corresponde.

Fui creciendo y poco a poco me di cuenta de que son muchas las cosas que, pareciendo obvias, por un extraño error no hacemos bien y vuelve a mi memoria la voz de mi familiar diciendo "¿¿¿Usted no tiene lógica o qué???". Estudié mi carrera universitaria Ingeniería de Sistemas porque allí encontré por qué era tan importante aquello de la lógica. Luego de buscar muchos significados de la palabra lógica, llegué a una que al fin me convenció. Le pregunté a una amiga "¿Qué es la lógica...?". Ella respondió en un lenguaje muy común: "*Pues lógica es... es.... es.... es como algo muy lógico*". Su respuesta no me satisfizo, pues había incluido en la definición el término que quería definir, lo cual significa que no me había dicho nada. Cuando pregunté por qué era difícil definirlo, ella respondió: "*No es fácil definir algo tan lógico*". Ella tenía clara la idea del significado, simplemente no era capaz de definirlo.

Luego le pregunté a don Luis, un viejo tendero que por diez años lo había visto llegar todas las mañanas a abrir su tienda desde donde atendía a su clientela.

Él me respondió: *"Lo único que puedo decir es que lo que es lógico es todo aquello que nunca es ilógico"*. Esa definición me pareció bastante racional, pero seguía siendo lejana de lo que yo había esperado. Yo veía que el proceso de abrir su tienda tenía unos pasos definidos y siempre los hacía de forma bastante lógica.

Después le pregunté a un profesor de la materia Español y él me entregó una excelente definición de esas tomadas de un diccionario: *"Lógica es la rama del conocimiento que nos permite definir que algo está respaldado por la razón como bien deducido o bien pensado"*. Para mí era una buena definición y me bastaba con que apareciera en el diccionario *Pequeño Larousse* para no discutirla. Sin embargo me exigía más reflexiones de las necesarias para poder entenderla, pues me parecía increíble que la definición de la palabra lógica fuera tan compleja, o sea, que su definición no fuera lógica. Eso mismo me había motivado a buscar una definición sencilla y lógica que no me exigiera muchas reflexiones adicionales.

Buscando una definición que me dejara satisfecho, fui a preguntarle a un matemático. Yo suponía que él podría definir qué era la lógica. Cuando le pregunté, me respondió: *"Lógica es la ciencia que estudia la estructura, fundamentos y uso de las expresiones del conocimiento humano"*. Esa era una definición muy exacta pero, al igual que la definición del *Pequeño Larousse*, me exigía demasiados razonamientos como para poder digerirla.

Me animé a preguntarle a alguien, un desconocido, qué era la lógica y su respuesta desprevenida me gustó porque la entendí fácilmente: *"La lógica es como una serie coherente de ideas y razonamientos"*. Compartí su definición y me pareció apropiada. Además que pude descubrir que todas las personas a quienes les preguntaba tenían muy claro el concepto de lógica, así algunas de esas personas no la pudieron definir de manera clara.

Finalmente, busqué a un campesino al que los avances tecnológicos no lo hubieran tocado aún. Alguien para quien el mundo moderno era un conjunto de problemas en vez de soluciones. Le pregunté: "¿Qué es la lógica?" y mirándome con extrañeza me dijo: *"Mire patrón, pues eso es la forma más OBVIA y FÁCIL de hacer cualquier cosa"*. Entonces vi que, de todas las definiciones que había recogido, esta era la que me parecía más lógica. Concluí que eso es la LÓGICA.

Los libros de tecnología citan que, para resolver un problema a través del computador, se requiere tener muy buena lógica. Creo que la base para ello es ser muy lógicos, o sea, poder vislumbrar el camino más obvio y más fácil para lograr un objetivo. Este libro busca orientar su lógica natural de manera que para usted sea muy sencillo hablar de la lógica computacional.

1.2. Fundamentos conceptuales

Vamos a comenzar por plantear una opinión sobre María... ¿y quién es María? Es la figura que nos va a acompañar en esta explicación. Se lo voy a decir muy claro: "María es alta". Inmediatamente usted, querido lector, se imaginará una mujer con más de 1,70 m de estatura, que puede ser más alta que usted o que, por lo menos, es de su misma estatura (si es que usted se considera alto). Apenas yo digo "María es alta", usted deberá hacer unos razonamientos lógicos y concluyentes para captar lo que yo le quise decir. ¿Qué fue lo que yo le describí de María? Muy fácil, describí un **atributo** de ella. ¿Qué es un atributo? Es una característica que identifica a un **ente informático**. ¿Y qué es un ente informático? Todo aquello que se puede describir basándose en sus características.

¿Qué características tiene un atributo? La primera consiste en que obedece a una serie de razonamientos humanos, lo cual quiere significar que debe existir todo un juego de razonamientos previos. La segunda característica es que es muy relativo. Si María mide 1,65 y vive en Occidente, puede no ser tan alta; si ella vive en Europa, sería una persona baja y si de pronto ella vive en Oriente, sería una persona notoriamente alta. Los atributos siempre van a estar sujetos a la mirada relativa con la que observa quien emane el concepto. Similar es el caso que se presenta cuando un hombre dice que una mujer es muy hermosa, pues lo que para él es una mujer hermosa, es posible que para otros no lo sea tanto.

Como se puede ver, estas dos características, en unión con un conjunto de conceptos y vivencias provenientes de la cultura de la región en donde hemos crecido, logran que se afiance más lo relativo de un atributo. Debido a esta relatividad conceptual sobre los atributos, estos son inmanejables porque van a depender del observador que los esté usando.

Se ha hecho necesario a través de la Historia que los atributos sean medidos a través de escalas, ya que esto los hace manejables y dejan de ser relativos (sin decir con esto que se vuelvan absolutos). Es por ello que surge la gran vedete de la Informática que es el **dato**. Nuestra frase inicial "María es alta" se puede cambiar a decir "María mide 1,73 m". A pesar de que los razonamientos y las conclusiones podrían ser los mismos, se pueden dejar al libre concepto de quien los observe. ¿Qué es un dato? Es un atributo "codificado" en unos términos que sean entendibles a un sistema de información, que sean manejables y comparables y que son, en gran medida, absolutos.

Un atributo es "codificado" si ha sido convertido a una escala determinada para poder ser más manejable, lo cual quiere decir que se puede operar con otros

atributos de la misma escala, es decir, se pueden realizar comparaciones y obtener resultados de ellas. Un dato en solitario no significa nada, excepto que se tenga claro cuál es el atributo que se está describiendo. Si yo le dijera: "Amigo lector, le cuento que el dato es 4", ¿qué pensaría usted que significa este dato? La cantidad de carros del autor o la cantidad de libros del autor o la cantidad de amigos del autor o... realmente usted no tendría certeza de su significado. Viene un concepto que aclara las cosas.

Nuestra frase inicial "María es alta", que luego se convirtió en "María mide 1,73 m", podríamos ahora plantearla como "La estatura de María es 1,73". En este momento, ya tenemos identificado de manera clara y con un nombre el atributo que se intenta describir. Este es el concepto de **campo**, que es el nombre que se le coloca a un dato para identificar el atributo que se propone describir. Así, nuestra frase "La estatura de María es 1,73" presenta tres campos bien identificados. El 1º de ellos es la estatura, campo con el que se ha estado realizando toda la explicación; el 2º corresponde al nombre de la persona de quien se está hablando, que es María en este ejemplo, y el 3º es el sexo, pues se puede suponer que María es de sexo femenino. La tabla 1 presenta la información de manera organizada.

Tabla 1. Información organizada

Nombre de la persona	Estatura de la persona	Sexo de la persona
María	1,73 m	Femenino

Se tiene aquí un conjunto de campos de forma que en cada campo está consignado un dato, en donde todos los datos pertenecen a un mismo ente informático. Con esto le acabo de entregar la definición de lo que es un **registro.** En esas condiciones se le puede colocar un nombre identificador al registro del ejemplo y lo vamos a llamar *Persona*. También se le pueden adicionar otros campos y llenarlos con datos del mismo ente informático, tal como se muestra en la tabla 2.

Tabla 2. Información organizada con más campos

Registro *Persona*					
Nombre	Estatura	Sexo	Fecha de nacimiento	No. cédula	Salario
María	1,73 m	Femenino	21-ago-78	42.522.301	560.000,00

Se puede pensar en organizar de una forma más apropiada la información que corresponde a María buscando que sea más presentable y más manejable. Una versión del registro de estudio mejor organizado se presenta en la tabla 3.

Tabla 3. Registro organizado

Registro *Persona*					
No. cédula	Nombre	Sexo	Fecha de nacimiento	Estatura	Salario
42.522.301	María	Femenino	21-ago-78	1,73 m	560.000,00

¿Cuántos campos pueden pertenecer a un registro?.Todos los que usted necesite, es decir, todos los campos en donde los datos sean útiles para usted. Una característica adicional que debe cumplir un registro es que debe ser manejado como una sola unidad, es decir, que todos los campos que lo conforman se encuentren en el mismo lugar físico o lógico de manera que pueda ser manipulado como un todo. Ahora, ¿qué sucedería si además de los datos de María tenemos también los datos de Luis, Pedro, Aníbal, Marta, Elena y Julián obteniendo de cada uno los mismos campos que obtuvimos de María? Pues simplemente que se ha conformado un **archivo**, que es un conjunto de registros que tienen la misma estructura y que se puede manejar como una sola unidad. Hablar de registros con la misma estructura quiere decir que tienen los mismos campos pero no los mismos datos. La tabla 4 presenta la información de varias personas.

Tabla 4. Archivo de varias personas

Registro *Persona*					
No. cédula	Nombre	Sexo	Fecha de nacimiento	Estatura	Salario
42.522.301	María	Femenino	21-ago-78	1,73 m	960.000,00
10.544.676	Luis	Masculino	20-ene-75	1,60 m	800.000,00
16.432.435	Pedro	Masculino	25-feb-70	1,55 m	980.000,00
10.398.789	Aníbal	Masculino	18-abr-55	1,98 m	890.000,00
41.884.556	Marta	Femenino	20-jul-79	1,77 m	900.000,00
38.756.986	Elena	Femenino	16-sep-65	1,58 m	999.000,00

Si lo que se necesita es almacenar tanta información que debemos guardarla en varios archivos pero que estén interrelacionados, entonces se está hablando de una **base de datos**, que es un conjunto de archivos o tablas organizados bajo unas técnicas específicas de normalización.

Estas definiciones nos han llevado desde un concepto completamente humano, como es el atributo, hasta un concepto absolutamente técnico, como es la base de datos. Si miramos el trasfondo de toda esta secuencia, podemos descubrir cuál es su objetivo. El objetivo es hablar de **información.** ¿Cómo se puede definir la información? Información es un conjunto de datos suficientemente organizados y entendibles (algunas veces se organizan a través de la tecnología).

- Escriba su estatura.
- Escriba su edad y su fecha de nacimiento.
- Escriba su número de identificación.
- Organice toda esa información en forma de registro.
- Escriba los mismos datos de cuatro amigos suyos.
- Organice la información en forma de tabla.

Capítulo 2

Metodología para solucionar un problema computable

Siempre que vamos a resolver un problema, nos enfrentamos con la dificultad de tener que encontrar precisamente eso: una solución. Pocas veces nos detenemos a pensar que existe un camino estructural que nos permite resolver cualquier problema (en términos generales) teniendo, como es obvio, que entrar en la minucia del detalle dependiendo del problema.

2.1. El objetivo

¿Cuál es el primer paso que debemos dar cuando nos enfrentamos a un problema? Lo primero que debemos tener muy claro es: ¿cuál es el problema a resolver? Es evidente que no podemos avanzar hacia la casa de un amigo si no sabemos en donde vive porque las posibilidades de que lleguemos son casi nulas. De manera que lo primero a conocer muy bien es el problema cuya solución la vamos a denotar con el nombre **OBJETIVO**.

Tener claro el objetivo nos va a permitir obtener dos beneficios que, a la postre, serán más grandes de lo que podemos pensar:

a. Nos permite saber hacia dónde vamos.
b. Nos permite saber dónde debemos parar.

Estas dos definiciones parecieran ser lo mismo pero no lo son. Usted puede tener muy claro hacia dónde va pero podría no saber hasta dónde debe llegar o, dicho en otras palabras, podría no saber en dónde debe parar o podría saber en dónde debe parar pero no tener ni idea por cuál ruta llegar. El OBJETIVO se ha de convertir en la razón de ser en la solución de un problema.

Alguna vez, antes de irme a estudiar a la ciudad de Bogotá, mi padre, en una de esas tardes en las cuales se sentaba a aconsejarme, me dijo: *"Te vas a ir a Bogotá con el objetivo de estudiar. Vas a tener toda la libertad del mundo para hacer todo lo que quieras pero, eso sí, independiente de todo lo que hagas en la capital, métete en tu cabeza que la clave del éxito para cualquier cosa en la vida es* ***no perder de vista el objetivo*** *cualquiera que este sea".* Desde allí entendí que realmente tener un objetivo claro, verdaderamente claro, exageradamente claro, es más importante que cualquier otra cosa, porque gracias a ello puede uno ir detrás de dicho objetivo hasta lograrlo. Cada paso que se dé, debe ir en pos del OBJETIVO.

En nuestro caso, podemos decir que, para llegar a la solución de un problema, la clave está en **tener muy claro cuál es el objetivo** y **no perderlo nunca de vista.** Tal vez usted tendrá alguna inquietud en cuanto a la insistencia de este tópico, pero la realidad es que muchas veces creemos tener claro el objetivo y, solo cuando nos empeñamos en lograrlo, vemos que no era así.

2.2. El algoritmo

Tener claro el objetivo nos permite algo adicional. Aquí voy a utilizar una frase que, aunque un poco romántica, nos va a ilustrar claramente: el OBJETIVO es el faro que, solo cuando está bien claro, nos ilumina el camino para lograrlo. Cuando el objetivo está suficientemente claro, podemos vislumbrar un camino lógico para llegar hasta él. Ese camino lógico va a tener un nombre, dada la orientación de este libro, y ese nombre es ALGORITMO.

¿Qué es un ALGORITMO? Es un conjunto de pasos secuenciales y ordenados que permiten lograr un objetivo. Que sean pasos secuenciales significa que deben ser ejecutados uno después de otro y que sean pasos ordenados quiere decir que deben llevar un orden que, en algunos casos, podría ser obligatorio. Como puede notar, el ALGORITMO permite lograr un OBJETIVO, o sea, que este es el camino que necesitamos para lograrlo.

De nuevo, ¿cuándo podemos vislumbrar claramente el algoritmo? Solo cuando el OBJETIVO está realmente claro. Siempre que usted, en el desarrollo de la solución de un problema, vea que en algún momento no sabe para dónde coger, no sabe qué hacer o se siente perdido, no busque más, simplemente quiere decir que realmente usted no tenía tan claro el objetivo como había pensado.

¿Cómo se estructura un objetivo? Muy sencillo; esto le va a parecer muy obvio, pero aun así se lo voy a decir. Un algoritmo se estructura comenzando en un

inicio y terminando en un *fin*. Mucho de lo que encuentre en este libro notará que es exageradamente lógico, pero no olvide que ese es el tema que evoca el título de este libro.

Veamos, entonces, un ejemplo: desarrollar un algoritmo que nos permita adquirir el libro ***El coronel no tiene quien le escriba*** de Gabriel García Márquez.

Objetivo: adquirir el libro *El coronel no tiene quien le escriba* de Gabriel García Márquez. Mucha atención al objetivo. Solamente es adquirirlo, en ningún momento el objetivo es leerlo o resumirlo ni nada, solamente adquirirlo.

Algoritmo: salimos del lugar en donde estemos y nos dirigimos hacia una librería. En caso de que ya estemos en una librería, solicitamos si tienen el libro. Si lo tienen, lo adquirimos y si no lo tienen, vamos a otra librería en donde repetimos el proceso.

Explicado así, el algoritmo no va a pasar de ser un breve texto explicativo que nos va a permitir lograr algo y que, en este caso, es la adquisición de un libro determinado. Pero podríamos organizar este algoritmo de manera que fuera un poco más estético y, por qué no decirlo, un poco más entendible, comenzando por el hecho de que esta vez le vamos a colocar un nombre al algoritmo y que lo vamos a generalizar para conseguir cualquier libro siempre y cuando esté completamente definido.

Algoritmo Adquisicion_Libro
Inicio
1. Saber cuál es el libro que se quiere adquirir
2. Desplazarnos hacia una librería
3. Preguntar si tienen el libro que necesitamos
4. Si lo tienen
adquirirlo y parar allí (dentro de este algoritmo)
Si no lo tienen
ir al paso 2
Fin

Note algunas puntualizaciones al respecto de este algoritmo:

a. Casi todas las líneas van numeradas, pero no todas.

b. En la línea 1, se debe cumplir esa orden para poder continuar con el resto del algoritmo, porque se asume en el algoritmo que no solo se pasa por encima de las líneas, sino que se realizan las tareas allí indicadas.

c. Si realizamos todos los pasos que indica este algoritmo, podremos obtener el libro que sea porque la connotación de este es absolutamente genérico sin restricciones, ya que en ningún momento se está diciendo que nos desplacemos hacia una librería que quede en la ciudad.

d. Si luego de recorrer todas las librerías de todos los países de todo el mundo vimos que no pudimos conseguir el libro, entonces podemos obtener dos conclusiones: una es que el libro que buscábamos no lo tiene ninguna librería porque está agotado y la otra es que el libro es posible que nunca haya existido.

e. Si probamos este ejemplo con el libro en mención (o sea *El coronel no tiene quien le escriba*) tendremos un alto porcentaje de seguridad de que lo conseguiremos a menos que esté agotado.

Este tipo de algoritmos son conocidos como informales, es decir, aquellos algoritmos (según los libros) que no pueden ser implementados a través de un computador. Yo sería un poco menos drástico. Yo diría que son algoritmos informales aquellos que no son fácilmente implementables en un computador. Segundo, y precisamente debido a que son algoritmos informales, deben hacerse una cantidad de reflexiones antes y después de ellos. Reflexiones que tienen una connotación puramente humana.

2.3. La Prueba de escritorio

En la reflexión e. se habló de una prueba. Textualmente dice: "*Si probamos este ejemplo...*", lo cual significa que todo algoritmo debe ser probado antes de ser ejecutado con el propósito de que tengamos una alta certeza en cuanto al logro del objetivo. Precisamente este es el tercer concepto que abordaremos y que se conoce como prueba de escritorio.

¿Qué es la prueba de escritorio? Es la simulación de la puesta en marcha de un algoritmo. Con la prueba de escritorio podemos determinar si el algoritmo que hemos diseñado logra el objetivo propuesto. De no ser así, podremos concluir que se debe corregir el algoritmo hasta lograr que satisfaga el objetivo propuesto.

Por lo que usted ha podido observar en el algoritmo de ejemplo, cada línea numerada del algoritmo puede considerarse a su vez como otro algoritmo, ya que

el solo hecho de saber cuál es el libro que se quiere adquirir nos obliga a realizar una serie de pasos ordenados y secuenciales para poderlo lograr. Entonces, surge una inquietud: ¿qué tan detallado debe ser un algoritmo? Su respuesta, como todo lo que va a encontrar en este libro, es muy lógica y muy sencilla.

Un algoritmo debe tener el nivel de detalle suficiente como para que no exista ninguna duda en su puesta en marcha, es decir, como para que cada línea pueda ser realizada sin el más mínimo asomo de inquietud. Esto quiere decir que algunos algoritmos pueden ser más entendibles para unas personas que para otras, dada su misma definición racional.

Como todo dentro del conocimiento humano requiere una clasificación y los conceptos de los cuales estamos hablando no son la excepción, los algoritmos se podrían clasificar en las categorías que se explican a continuación.

2.4. Algoritmos informales

Definidos como todos aquellos algoritmos que no son realizables a través de un computador (al menos, no fácilmente). Son aquellos algoritmos en donde el ejecutor real es el ser humano, como el algoritmo para dar un beso, el algoritmo para fritar unos huevos o el algoritmo para conseguir un libro. Escribo que *"...al menos no fácilmente"* porque la tecnología ha avanzado tanto que muchos algoritmos que en el pasado no eran implementables a través de un computador en la actualidad lo son y de manera mucho más sencilla, como es el caso del algoritmo para conseguir un libro que anteriormente se pensaba en librerías y ahora se piensa en un concepto más globalizado: Internet con sus buscadores, con más posibilidad de conseguirlo y con menos trabajo. De manera que vamos a considerar aquellos algoritmos informales como los que son preferiblemente realizables por el ser humano.

2.5. Algoritmos computacionales

Se consideran como tales todos aquellos algoritmos que deben ser preferiblemente implementados en un computador para aprovechar su velocidad de procesamiento. Un ejemplo de estos puede ser el algoritmo que genere los primeros 100 números primos, recordando que un número primo es aquel que solo puede ser dividido exactamente entre la unidad y entre sí mismo que, si bien podrían ser calculados utilizando un papel y un lápiz, la utilización de un computador en unión con el algoritmo adecuado nos va a dar un resultado mucho más rápido y absolutamente confiable (lo cual depende de que el programa se base en un algoritmo confiable). Son precisamente estos algoritmos los que vamos a tratar de definir y poner en práctica en el desarrollo de este libro.

En el desarrollo de los algoritmos computacionales, los cuales nos van a ocupar en lo sucesivo, la metodología para llegar a la solución final que permita lograr un objetivo (igualmente computacional) continúa con los siguientes pasos:

2.5.1. Transcripción

Este es el proceso a través del cual "convertimos" un algoritmo, escrito en términos muy coloquiales e informales, en un listado de instrucciones entendibles a un computador y que se ajustan a las reglas sintácticas de determinado lenguaje de programación. Podríamos decir que es la "traducción" de un algoritmo con la "ortografía" de un lenguaje de programación.

¿Qué son las reglas sintácticas de un lenguaje de programación? Son todas las restricciones técnicas (y algunas veces caprichosas) sobre las cuales está construido el lenguaje. Por ejemplo, y solo con el ánimo de ilustrar lo que acabo de decir, si estamos utilizando el lenguaje de programación C, la orden para leer un dato se da con la instrucción *cin*, así como está escrito y sin ningún tipo de modificación por más mínima que esta pudiera ser. Será un error, entonces, escribir esta instrucción como *Cin* o como *cim*.

El lenguaje C solo entiende la instrucción *cin* tal como sus creadores la diseñaron. De tal forma que, para escribir un algoritmo computacional en términos entendibles a un computador, lo único que necesitamos saber son las reglas sintácticas de un lenguaje de programación cualquiera. El algoritmo escrito con dichas reglas se llamará **programa**. ¿Qué es, pues, un programa? Es un algoritmo escrito con las instrucciones, las restricciones y las reglas de un lenguaje de programación.

2.5.2. Digitación

Es el proceso a través del cual le escribimos al computador el programa que hemos acabado de escribir en papel. Para ello nos valemos de un programa llamado Editor de texto o Entorno Integrado de Desarrollo (IDE, por sus siglas en inglés, *Integrated Development Environment*) que nos permite escribir un programa y grabarlo. Visto neutralmente, un programa no es más que un texto escrito bajo la óptica de algunas reglas preestablecidas por los creadores de un lenguaje de programación.

2.5.3. Compilación

Es muy normal que al reescribir un algoritmo con las reglas sintácticas de un lenguaje de programación, es decir, al escribir un programa, omitamos algunas

reglas y se nos vayan, sin querer, algunos errores. Por ejemplo, que en alguna parte del programa abrimos un paréntesis que luego se nos olvidó cerrar. Para ello, el computador nos facilita una herramienta que revisa la sintaxis del programa, nos dice si tiene errores y, en los casos más depurados, nos dice en qué líneas del programa están los errores y hasta nos sugiere la corrección.

Entonces, ¿qué es la compilación? Es el proceso a través del cual el computador revisa que el programa que hemos digitado se ajuste a las reglas sintácticas de un determinado lenguaje de programación. ¿Quién realiza realmente el proceso llamado compilación? Lo realiza un programa llamado compilador, que es el encargado de evaluar dos tipos de errores:

a) Errores de sintaxis.- Podríamos asociar los errores de sintaxis en un lenguaje de programación con los errores de ortografía en nuestro idioma. Son aquellos errores representados en la omisión de alguna o algunas reglas sintácticas (hablando de un lenguaje de programación). Por ejemplo, es normal que algunas veces, en medio de una expresión matemática, abramos un paréntesis que luego se nos olvida cerrar. En ese caso, al momento de compilar, el compilador nos indicará precisamente ese error.

b) Errores de precaución.- Algunos compiladores nos hacen, por decirlo así, cierto tipo de recomendaciones para efectos de mejoramiento o aseguramiento de nuestros programas. Este tópico lo veremos de manera más detallada en la medida que se vayan desarrollando los temas de este libro.

¿Por qué se habla de algunos compiladores? Pues porque, dado que existen varios lenguajes de programación, cada lenguaje de programación tiene su propio compilador, o sea, su propio revisor sintáctico. Podríamos decir de nuevo (y aunque sea mal dicho sirve en este caso) que la sintaxis es a un lenguaje de programación como la ortografía es a un idioma.

¿Por qué existen varios lenguajes de programación? Esto sí obedece a dos factores: el primero es la especificidad de los lenguajes, ya que son desarrollados para que cumplan de la mejor manera ciertos objetivos (refiriéndonos al mundo de la informática) y el segundo es un factor netamente comercial, pues los lenguajes de programación son producidos por empresas fabricantes de *software*.

En un programa, los errores son de tres tipos: errores de sintaxis y errores de precaución que, como ya se dijo, son revisados por el compilador. Son los errores fáciles porque los compiladores actuales no solo le dicen a uno cuál es el error, sino que además le indican, más o menos, en donde está e incluso algunas veces le sugieren la corrección. Los errores difíciles realmente de encontrar en un programa son el tercer tipo de error y son los errores lógicos, ya que

el compilador no le va a discutir acerca de lo que usted quiere hacer y cómo quiere hacerlo.

¿...Y en dónde se detectan los errores lógicos? Pues en la prueba de escritorio. Allí y solo allí usted podrá determinar si su algoritmo está realmente bien o no, es decir, si logra o no el objetivo propuesto. Ha de tenerse especial cuidado cuando un algoritmo sea transcrito, ya que el cambio de cada línea de un algoritmo por su correspondiente instrucción en un programa a veces cambia un poco la lógica inicial si no se conocen bien las reglas sintácticas del lenguaje de programación.

2.5.4. Ejecución o puesta en marcha

Luego de que hemos realizado las correcciones pertinentes para que nuestro compilador nos reporte cero errores de sintaxis y cero errores de precaución, ya estamos en condiciones de poner a "correr" nuestro programa, o sea, en condiciones de ser ejecutado por el computador. Si lo que queríamos inicialmente (o sea, nuestro objetivo) era generar los 100 primeros números pares, entonces al momento de la ejecución deberán aparecer en pantalla los 100 primeros números pares.

2.6. Verificación de resultados

Este último paso es útil ya que, con lo que nos entregue la ejecución del programa, podremos saber si se cumplió el objetivo inicial o no. En caso de que no se haya cumplido el objetivo inicial (al llegar a este punto), podría ser por algunas de las siguientes razones:

a. No teníamos claro el objetivo y fallamos en todo el proceso.

b. No realizamos bien la Prueba de Escritorio y nos la saltamos creyendo que el algoritmo estaba bien.

c. No conocíamos bien las reglas sintácticas del lenguaje con el que pensábamos trabajar y el programa transcrito final terminó siendo una representación técnica diferente del algoritmo inicial.

Lo que sí podemos asegurar es que, si mantenemos esta metodología paso a paso y cada uno lo realizamos concienzudamente, siempre al realizar la verificación de resultados se va a satisfacer con estos el objetivo inicial.

2.7. Ejercicios propuestos sobre algoritmos informales

La única forma como uno puede realmente aprender a nadar o a tirarse desde un paracaídas es haciéndolo, por eso lo invito a que se siente pacientemente a desarrollar estos algoritmos pensados para que usted encuentre una gran coincidencia entre unos y otros a pesar de tener objetivos diferentes. Sé que surgirán muchas dudas en cuanto a algunos de ellos, pero también estoy seguro de que, si usted lee este libro detenidamente, va a despejar todas las dudas que tenga. Por eso, tome al azar cualquiera de los siguientes enunciados y siéntese a practicar y a poner a funcionar, un poquito, esa lógica humana que tan pocas veces ejercitamos.

- Desarrollar un algoritmo que permita adquirir una revista.
- Desarrollar un algoritmo que permita entrar a una casa que está con llave.
- Desarrollar un algoritmo que permita dar un beso.
- Desarrollar un algoritmo que permita empacar un regalo.
- Desarrollar un algoritmo que permita encender un vehículo.
- Desarrollar un algoritmo que permita fritar un huevo.
- Desarrollar un algoritmo que permita mirar por un telescopio.
- Desarrollar un algoritmo que permita botar la basura.
- Desarrollar un algoritmo que permita tomar un baño.
- Desarrollar un algoritmo que permita estudiar para un examen.
- Desarrollar un algoritmo que permita tocar determinada canción con un instrumento musical.
- Desarrollar un algoritmo que permita viajar en avión.
- Desarrollar un algoritmo que permita encender un bombillo.
- Desarrollar un algoritmo que permita encender una vela.
- Desarrollar un algoritmo que permita apagar una vela.
- Desarrollar un algoritmo que permita apagar un bombillo.
- Desarrollar un algoritmo que permita parquear un vehículo.
- Desarrollar un algoritmo que permita almorzar.
- Desarrollar un algoritmo que permita ir de la casa al trabajo.
- Desarrollar un algoritmo que permita colocarse una camisa.
- Desarrollar un algoritmo que permita quitarse la camisa.
- Desarrollar un algoritmo que permita escuchar un determinado disco.
- Desarrollar un algoritmo que permita abrir una ventana.
- Desarrollar un algoritmo que permita ir a la tienda a comprar algo.

- Desarrollar un algoritmo que permita tomar una fotografía.
- Desarrollar un algoritmo que permita hacer deporte.
- Desarrollar un algoritmo que permita cortarse el cabello.
- Desarrollar un algoritmo que permita hacer un avión con una hoja de papel.
- Desarrollar un algoritmo que permita manejar una bicicleta.
- Desarrollar un algoritmo que permita manejar una motocicleta.
- Desarrollar un algoritmo que permita manejar un monociclo.
- Desarrollar un algoritmo que permita maquillarse.
- Desarrollar un algoritmo que permita hacer un pastel.
- Desarrollar un algoritmo que permita hacer un almuerzo.
- Desarrollar un algoritmo que permita adquirir un pantalón.
- Desarrollar un algoritmo que permita hacer un mercado pequeño.
- Desarrollar un algoritmo que permita leer el periódico.
- Desarrollar un algoritmo que permita saludar a un amigo.
- Desarrollar un algoritmo que permita arrullar a un bebé hasta que se duerma.
- Desarrollar un algoritmo que permita hacer un gol en fútbol.
- Desarrollar un algoritmo que permita jugar ping-pong.
- Desarrollar un algoritmo que permita nadar.
- Desarrollar un algoritmo que permita tirarse desde un avión con un paracaídas.
- Desarrollar un algoritmo que permita tirarse desde un avión sin un paracaídas.
- Desarrollar un algoritmo que permita descifrar un jeroglífico.
- Desarrollar un algoritmo que permita amarrarse un zapato.
- Desarrollar un algoritmo que permita quitarse los zapatos.
- Desarrollar un algoritmo que permita silbar.
- Desarrollar un algoritmo que permita elevar una cometa.
- Desarrollar un algoritmo que permita desarrollar algoritmos.

Capítulo 3

Variables, constantes y operadores

3.1. Variable

Informalmente, algo variable es algo que puede cambiar de un momento a otro. Técnicamente, una variable es un campo de memoria al que se le puede cambiar su contenido cuantas veces sea necesario. Primera aclaración: un campo de memoria es un pedacito de la memoria principal del computador en donde podemos guardar un dato. Segunda aclaración: a pesar de que en la memoria es donde se guarda la información, exactamente esta se almacena en variables. Esto le ha de representar a usted que es, a través de variables, como se puede utilizar la memoria del computador.

¿Ha notado usted que la maleta de una guitarra es diferente a la maleta de un violín o de una trompeta? Sabe entonces ¿qué es lo que diferencia la maleta de un instrumento musical de la maleta de otro instrumento musical? Pues precisamente la única diferencia es su contenido, es decir, el instrumento en sí. Y esto, ¿qué tiene que ver con el tema que estamos tratando? Pues muy sencillo: la diferencia entre una variable y otra radica precisamente en su contenido o, más bien, en el tipo de su contenido.

Para poder utilizar variables en el desarrollo de un programa de computador se debe primero decir qué tipo de dato se va a almacenar, pues las variables son como unas cajitas de diferentes tamaños y, por tal motivo, se deben declarar previamente para que el computador las dimensione de acuerdo a las necesidades. ¿Cuáles son los tipos de datos que pueden ser almacenados en una variable? A pesar del avance de la tecnología, los tipos de datos de las variables se explican a continuación.

3.1.1. Tipo entero

Un dato de tipo entero es un número que no tiene punto decimal, por lo tanto, en sus operaciones jamás va a generar decimales. Por ejemplo 25, -96 y 0. El hecho de que los datos de tipo entero no generen decimales significa que operan con un juego de reglas llamado aritmética entera. Una variable que se declare de tipo entero podrá almacenar solamente datos de tipo entero.

3.1.2. Tipo real

Un dato de tipo real es un número que tiene punto decimal, por lo tanto, en sus operaciones puede generar decimales. Por ejemplo 12.3, -78.56 o 45.0. El hecho de que los datos de tipo real generen decimales significa que operan con un juego de reglas llamado aritmética real. Una variable que se declare de tipo real podrá almacenar solamente datos de tipo real.

Por lo dicho en las anteriores dos definiciones, ¿qué tipo de dato sería *5.*? (así, con el punto y todo). Pensaríamos que es un entero, pero en realidad no. La definición de dato entero es que no tiene punto decimal y la de dato real es que tiene punto decimal, por lo tanto, 5. es un dato real.

3.1.3. Tipo carácter

Un dato tipo carácter es un equivalente del código ASCII (*American Standard Code for Interchange Information*). ¿Qué es el código ASCII? Es el código internacional de equivalencias internas en el sistema binario. A nivel mundial, los computadores están construidos en un sistema numérico llamado sistema binario, sistema que se basa solamente en la utilización de unos (1) y ceros (0). Este sistema tiene una relación directa con el sistema decimal y, por lo tanto, fue adoptado, ya que permite aprovechar características físicas de los componentes electrónicos. Dada la gran importancia que poco a poco fueron adquiriendo los computadores, se adoptó un solo código interno para la interpretación de todas y cada una de las teclas de su teclado.

De esta forma, cuando usted presiona en su teclado la letra A, realmente se genera por dentro de su computador el número 65 pero expresado en código binario, es decir, 0100 0001, y cuando usted presiona la tecla 1, se genera internamente el número 49, pero expresado igualmente en código binario, es decir, 0011 0001.

Cada una de las teclas que usted presione tendrá un equivalente interno y por supuesto expresado (internamente) en sistema binario. Cada cero o cada uno

utilizado en este sistema se conoce como bit (abreviatura de *binary digit*) y un conjunto de 8 bits (medida en la cual se expresa el código ASCII) se conoce como un byte (pronúnciese *bait*).

Como el código ASCII está expresado en bytes y cada byte tiene 8 bits y cada bit puede tener un 0 o un 1 (o sea, dos estados), entonces se puede concluir que el código completo consta de 2^8 combinaciones (o sea, 256 equivalencias). A continuación relaciono la tabla completa de equivalencias ASCII.

Tabla 5. Fragmento del código ASCII

CÓDIGO ASCII					
Carácter	**Equivalencia sistema decimal**	**Equivalencia sistema binario**	**Carácter**	**Equivalencia sistema decimal**	**Equivalencia sistema binario**
0	48	0011 0000	G	71	0100 0111
1	49	0011 0001	H	72	0100 1000
2	50	0011 0010	I	73	0100 1001
3	51	0011 0011	J	74	0100 1010
4	52	0011 0100	a	97	0110 0001
5	53	0011 0101	b	98	0110 0010
6	54	0011 0110	c	99	0110 0011
7	55	0011 0111	d	100	0110 0100
8	56	0011 1000	e	101	0110 0101
9	57	0011 1001	f	102	0110 0110
A	65	0100 0001	g	103	0110 0111
B	66	0100 0010	h	104	0110 1000
C	67	0100 0011	i	105	0110 1001
D	68	0100 0100	j	106	0110 1010

Como puede usted notar, estas son apenas algunas de las 256 equivalencias que tiene la tabla ASCII. Es obvio pensar que también tienen equivalencia los caracteres especiales como la coma, el punto o el paréntesis.

Cuando se tiene un conjunto de caracteres, se dice técnicamente que se tiene una cadena, por lo tanto, el nombre del autor "OMAR" es una cadena. El contenido de una cadena no es evaluado por el computador y se acostumbra acotarlo o encerrarlo entre comillas dobles; así, la cadena "5 – 7 es igual a 8", a pesar de no ser lógica ni correcta matemáticamente, es válida para el computador ya que él, en ningún momento, evalúa las cadenas.

3.2. Asignaciones

¿Cómo se llevan los datos a las variables?, o sea, ¿cómo se "cargan" las variables? Pues a través de un signo muy conocido por usted y es el signo =.

Este signo tiene, en el caso de los algoritmos computacionales y programas, una connotación un poco diferente a la que se le da en matemáticas. El signo igual (=) significa que el computador va a realizar lo que está a la derecha del igual y lo va a almacenar en la variable que se encuentre a la izquierda del igual.

De manera que ya usted puede ver claramente en esta definición que a la izquierda del igual solo puede haber una variable y al lado derecho del igual puede haber una constante, una variable o una expresión. De manera que cualquiera de los siguientes esquemas es válido:

a = 8 Le indica al computador que guarde la constante 8 en la variable a.

b = a Le indica al computador que guarde en la variable b el contenido de la variable a que en la instrucción había sido "cargada" con 8, por lo tanto, en la variable b queda el valor de 8 al igual que en la variable a.

c = a + bLe indica al computador que guarde en la variable c el resultado de sumar el contenido de la variable a con el contenido de la variable b. Como la variable a tenía el contenido 8 y la variable b también tenía el contenido 8, entonces el computador sumará 8+8 y ese 16 de resultado lo almacenará en la variable c.

Puede notarse en este ejemplo que en la variable a se ha almacenado una constante, en la variable b se ha almacenado el contenido de otra variable y en la variable c se ha almacenado el resultado de una expresión. Y qué pasará si luego de tener estas tres instrucciones adicionamos la siguiente:

b = 9 Pues, muy sencillo; el anterior contenido de la variable b que era 8 va a ser reemplazado por el nuevo contenido de la variable b que es 9. Esto significa que, cada vez que se asigna un nuevo valor (proveniente de una constante, una variable o como resultado de una expresión), el valor anterior de la misma variable se pierde.

De esta forma, si se quisieran escribir los contenidos de las variables a, b y c, el computador nos reportaría para a el contenido 8, para b el contenido 9 y para c el contenido 16.

Todo lo que debe tener en cuenta con la asignación o carga de las variables es lo siguiente:

a. Al lado izquierdo del igual solo puede haber una variable.

b. Al lado derecho del igual puede haber una constante, una variable o una expresión.

c. El computador siempre resuelve lo de la derecha del igual y su resultado lo almacena en la variable que esté a la izquierda del mismo.

d. Cada vez que se le entra un nuevo valor a una variable, el valor anterior se pierde.

De acuerdo a lo dicho, vamos a resolver el siguiente conjunto de instrucciones:

Entero: A, B, C	Declara de tipo entero las variables A, B y C, de manera que solo podrán almacenar datos enteros.
A = 10	Almacena la constante 10 en la variable A.
B = 15	Almacena la constante 15 en la variable B.
C = 20	Almacena la constante 20 en la variable C.
A = A + B	Almacena en la variable A el resultado de sumar el contenido de A más el contenido de B, o sea, 10+15, que es igual a 25.
B = B + 8	Almacena en la variable B el resultado de sumar el contenido de B con la constante 8, o sea, 15+8, que es igual a 23.
C = C + A	Almacena en la variable C el resultado de sumar el contenido de la variable C más el contenido de la variable A, o sea, 20+25, que es igual a 45. Recuerde que en esta línea se utiliza el último valor almacenado en la variable A.
A = A + 5	Almacena en la variable C el resultado de sumar el contenido de la variable A más la constante 5, es decir, 25+5, que es igual a 30.
B = B + 3	Almacena en la variable B el resultado de sumar el contenido de la variable B más la constante 3, o sea, 23+3, que es igual a 26.
C = C + 2	Almacena en la variable C el resultado de sumar el contenido de la variable C más la constante 2, o sea, 45+2, que es igual a 47.
A = A – B	Almacena en la variable A el resultado de restarle al contenido de la variable A el contenido de la variable B, o sea, 30-26, que es igual a 4.
B = A – B	Almacena en la variable B el resultado de restarle al contenido de la variable A el contenido de la variable B, o sea, 4-26, que es igual a -22.

C = A – B	Almacena en la variable C el resultado de restarle al contenido de la variable A el contenido de la variable B, o sea, 4- (-22), que por propiedades algebraicas es igual a 4+22, o sea, 26.

Los resultados finales en las tres variables son:

Variable A	4
Variable B	-22
Variable C	26

No olvide que para el manejo de variables cada nuevo valor que se le asigne a una variable borra el valor anterior. Nótese que en este conjunto de instrucciones las tres últimas son iguales en su forma pero no en sus resultados. Para hacerlo más breve, el seguimiento de este conjunto de instrucciones podríamos detallarlo de la siguiente forma:

Variables

Entero: A, B, C	**A**	**B**	**C**
A = 10	10		
B = 15	10	15	
C = 20	~~10~~	15	20
A = A + B	25	~~15~~	20
B = B + 8	25	23	~~20~~
C = C + A	~~25~~	23	45
A = A + 5	30	~~23~~	45
B = B + 3	30	26	~~45~~
C = C + 2	~~30~~	26	47
A = A – B	4	~~26~~	47
B = A – B	4	-22	~~47~~
C = A – B	4	-22	26

Era evidente que teníamos que llegar al mismo resultado. Esto que acabamos de hacer es precisamente la PRUEBA DE ESCRITORIO de este conjunto de instrucciones. También puede notarse que cada nuevo valor asignado a cada variable reemplaza el valor anterior de la misma variable, por esa razón, por cada nuevo resultado (en una determinada variable), se va tachando el resultado anterior para indicar que ese ya no es válido.

3.3. Ejercicios

1.

a = 10
b = 20
c = 5
a = a + 3
b = b + 4 – a
c = a + b + c
a = a + c
b = 4
c = c + 3 - b + 2

¿Qué valores quedan almacenados en las variables a, b y c?

2.

a = 5
b = 18
c = 15
d = 25
a = a + 10
b = b + 5 – c
c = c + 4 + b
d = d + b + a
a = a + 1
b = b + c
c = b + c
d = b + b

¿Qué valores quedan almacenados en las variables a, b, c y d?

3.

a = 9
b = 6
a = a + 4
b = b + 2
a = a + 10

b = b – 25
a = a – 20
b = b + 5
a = a + 4
b = b + 2
a = a + 10
b = b – 10

¿Qué valores quedan almacenados en las variables a y b?

4.

a = 18
b = 18
c = 18
d = 18
a = a + b
b = a - b
c = a + b
d = a - b
a = a - b
b = a + b
c = a - b
d = a + b

¿Qué valores quedan almacenados en las variables a, b, c y d?

5.

a = 10
b = 5
a = a - 5
b = b + 6
a = a + 18
b = b – 23
a = a – 21
b = b - 5
a = a - 4

b = b - 2
a = a + 10
b = b + 10

¿Qué valores quedan almacenados en las variables a y b?

6.

a = 8
b = 7
c = 5
d = 8
a = a + b - c + d
b = a + b - c + d
c = a + b - c + d
d = a + b - c + d
a = a + b - c + d
b = a + b - c + d
c = a + b - c + d
d = a + b - c + d

¿Qué valores quedan almacenados en las variables a, b c y d?

3.3. Operadores

Los operadores son signos que nos permiten expresar relaciones entre variables y/o constantes, relaciones de las cuales normalmente se desprende un resultado. Ya hemos manejado dos operadores, que son el de la suma (+) y el de la resta (-), pero no son los únicos. En un algoritmo computacional, también se pueden utilizar los siguientes operadores:

^ Para expresar la potenciación
* Para expresar la multiplicación
/ Para expresar la división

Debo anotar que la notación para potenciación que vamos a utilizar en este libro no es estándar para todos los lenguajes de programación y, en algunos casos, el mismo signo tiene otro significado. Por tal motivo, sugiero que, cuando vaya a utilizar este operador en un programa determinado donde necesite

realizar operaciones de potenciación, consulte primero el manual del lenguaje en el cual esté trabajando. Por lo menos lo vamos a utilizar en el desarrollo de este libro.

Algo que debemos tener en cuenta cuando vamos a escribir una expresión es que el computador solo entiende las expresiones en formato linealizado, esto quiere decir que son expresiones escritas en una sola línea. De tal manera que si queremos escribir la ecuación:

$$\text{var} = \frac{a+b}{c+d}$$

no se la podemos entregar al computador tal y cual como está aquí escrita, sino que debemos "transformarla" de manera que quede escrita en una sola línea. Supondríamos en primera instancia que su equivalente (en una sola línea) sería:

$$\text{var} = \text{a} + \text{b} / \text{c} + \text{d}$$

Sin embargo, aunque a primera vista pareciera ser la misma ecuación, esta expresión podría tener varias interpretaciones. Le pregunto a usted amigo lector, la ecuación computacional:

$$\text{var} = \text{a} + \text{b} / \text{c} + \text{d}$$

¿a cuál de las siguientes ecuaciones reales correspondería?

$$\text{var} = a + \frac{b}{c} + d \qquad \text{var} = \frac{a+b}{c+d}$$

$$\text{var} = a + \frac{b}{c+d} \qquad \text{var} = \frac{a+b}{c} + d$$

Gran pregunta... pues es muy obvio que cada una de estas ecuaciones va a dar un resultado diferente. Para solucionar esta gran inquietud, todos los computadores tienen implementada una jerarquía de operadores que no es más que un conjunto de reglas que le permiten a un computador evaluar de una forma (y solo una) una expresión aritmética para que no haya espacio para ambigüedades.

Lo primero que el computador evalúa y realiza son las potencias, revisándolas de derecha a izquierda. Lo segundo que el computador evalúa y realiza son las

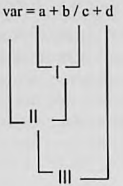
multiplicaciones y divisiones y lo último que revisa son las sumas y restas. Tanto para el nivel de multiplicaciones y divisiones como para el nivel de sumas y restas, la evaluación es totalmente indistinta; esto quiere decir que, en la medida que va encontrando sumas y restas (si están en este nivel), las va ejecutando.

Veamos entonces el ejemplo inicial:

$$var = a + b / c + d$$

Será interpretado por el computador de la siguiente manera. Primero evalúa en esta expresión si existen potencias. Como no las hay, pasa al siguiente nivel y vuelve a recorrer la expresión evaluando si existen (indistintamente y no necesariamente en ese orden) multiplicaciones y divisiones y encuentra que existe una división, de manera que lo primero que realiza es la división de b/c. Luego vuelva a recorrer la expresión buscando (en el tercer nivel) sumas y restas (indistintamente y no necesariamente en ese orden) y encuentra la suma de a más lo que ya había calculado y luego realiza la suma de este último resultado más d.

¿Qué es lo que se persigue con esta jerarquía de operadores? Pues sencillamente que cuando el computador vaya a resolver una expresión, en donde por supuesto participen operadores aritméticos, siempre tenga listos los valores que va a operar. De esta forma, la evaluación de la expresión en mención se hace en los siguientes pasos:



Por lo tanto, escribir la expresión así

$$var = a + b / c + d$$

SOLAMENTE equivale a la expresión

$$var = a + \frac{b}{c} + d$$

Y si queremos alterar esa jerarquía porque la expresión que queríamos escribir no era esta, ¿qué hacemos? Para eso se hicieron los paréntesis... precisamente para alterar esta jerarquía.

El hecho de que los computadores se basen en esta jerarquía de operadores para realizar sus operaciones es lo único que garantiza que, para una determinada expresión, el resultado en cualquier computador sea el mismo. Cuando se utilizan paréntesis, el computador detecta el primer paréntesis más interno y dentro de él aplica la tabla de jerarquía de operadores. ¿Cómo sabe el computador que se encuentra dentro de un "paréntesis más interno"? El computador considera un par de paréntesis como "más interno" cuando dentro de ellos no existe ningún otro par de paréntesis.

Haciendo uso de la facilidad de los paréntesis, podemos entonces expresar computacionalmente las siguientes fórmulas así:

$$\text{var} = a + \frac{b}{c} + d$$

VAR = (a + b) / (c + d)

$$\text{var} = \frac{a+b}{c+d}$$

VAR = a + b / c + d

$$\text{var} = \frac{a+b}{c} + d$$

VAR = a + b / (c + d)

$$\text{var} = a + \frac{b}{c+d}$$

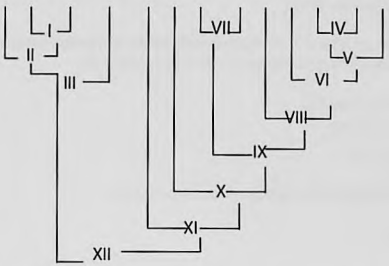
VAR = (a + b) / c + d

Puede usted, querido lector, suponer el papel tan importante que hacen aquí los paréntesis, precisamente porque, cuando a través de ellos se altera la jerarquía de operadores, es cuando se llega a las fórmulas que queremos que el computador realice. Igualmente, ha de saber que un paréntesis mal colocado finalmente hace que los resultados que calcule el computador sean diferentes a los esperados. Veamos el siguiente ejemplo:

VAR = (a + b / c – d) / (a + b / (c ^ d+ d / (a – b / c * d)))

Recuerde, el computador busca los paréntesis más internos, sobre ellos aplica la tabla de jerarquía de operadores (primero potencias, segundo multiplicaciones y divisiones y tercero sumas y restas). Luego el orden de resolución de esta ecuación, suponiendo que son variables que tienen ya unos valores asignados, es el siguiente:

VAR = (a + b / c – d) / (a + b / (c ^ d+ d / (a – b / c * d)))



El objetivo fundamental de mantener esta jerarquía es que, cuando el computador vaya a realizar una operación entre dos operandos, siempre va a tener definidos los operandos. Veámoslo paso a paso y vamos reemplazando por cada uno de los resultados que va encontrando el computador señalando cada resultado por el número ordinal del paso.

Primero se ubica en el primer paréntesis más interno y dentro de él aplica la jerarquía de operaciones:

VAR = (a + b / c – d) / (a + b / (c ^ d+ d / (a – b / c * d)))
VAR = (a + I – d) / (a + b / (c ^ d+ d / (a – b / c * d)))
VAR = (II – d) / (a + b / (c ^ d+ d / (a – b / c * d)))
VAR = (III) / (a + b / (c ^ d+ d / (a – b / c * d)))

Luego se ubica en el siguiente paréntesis más interno. Recuerde que un paréntesis es "más interno" si no tiene más paréntesis adentro.

VAR = (III) / (a + b / (c ^ d+ d / (a – IV * d)))
VAR = (III) / (a + b / (c ^ d+ d / (a – V)))
VAR = (III) / (a + b / (c ^ d+ d / (VI)))

Sigue buscando y resolviendo los paréntesis que vayan quedando, aplicando en cada uno la tabla de jerarquía de operadores:

VAR = (III) / (a + b / (c ^ d+ d / VI))
VAR = (III) / (a + b / (VII + d / VI))

VAR = (III) / (a + b / (VII + VIII))
VAR = (III) / (a + b / (IX))

En la medida en que se van resolviendo completamente los paréntesis, estos van desapareciendo y la expresión se va simplificando:

VAR = (III) / (a + b / IX)
VAR = (III) / (a + X)
VAR = (III) / (XI)

Finalmente, la expresión queda reducida a resolver:

VAR = III / XI
VAR = XII

Bueno, y si quisiéramos saber esta fórmula linealizada a qué fórmula algebraica correspondería no es sino seguir los mismos pasos que siguió el computador para resolverla y llegaremos a la siguiente fórmula:

$$VAR = (a + b / c - d) / (a + b / (c \wedge d + d / (a - b / c * d)))$$

Equivale algebraicamente a:

$$\text{var} = \cfrac{a + \cfrac{b}{c} - d}{a + \cfrac{b}{c^{d} + \cfrac{d}{a - \cfrac{b}{c} * d}}}$$

3.4. Ejercicios

Todos los siguientes ejercicios deberán desarrollarse utilizando las reglas de la aritmética entera.

1. a = 10
   b = 20
   c = 10
   a = a + 15

b = b + 12
c = a * c

¿Qué valores quedan en las variables a, b y c?

2. a = 3
b = 8
c = 1
a = 5
b = 9
c = 7
a = a + 1
b = b + 2
c = c + 3

¿Qué valores quedan en las variables a, b y c?

3. a = 10
b = 5
c = 10
a = a + b - 5
b = a + b - 5
c = a + b - 5
a = a + 5 * b / 2
b = a + 5 * b / 2
c = a + 5 * b / 2

¿Qué valores quedan en las variables a, b y c?

4. a = 5
b = 5
c = 5
a = a + a
b = b + b
c = c + c
a = a + b + c
b = a + b + c

c = a + b + c

¿Qué valores quedan en las variables a, b y c?

5. a = 10
   b = 10
   c = 10
   a = a + 5
   b = a + 3
   c = a + 2
   a = b + 4
   b = b + 5
   c = c + 8

¿Qué valores quedan en las variables a, b y c?

6. a = 10
   b = 1
   c = 4
   a = a + c
   b = a + c
   c = a + c
   a = c + 5
   b = c + b
   c = a + b + c

¿Qué valores quedan en las variables a, b y c?

7. a = 1
   b = 1
   c = 1
   a = a + a
   b = b + a
   c = c + a
   a = a + a
   b = b + a
   c = c + a

c = a + b + c

¿Qué valores quedan en las variables a, b y c?

5. a = 10
   b = 10
   c = 10
   a = a + 5
   b = a + 3
   c = a + 2
   a = b + 4
   b = b + 5
   c = c + 8

¿Qué valores quedan en las variables a, b y c?

6. a = 10
   b = 1
   c = 4
   a = a + c
   b = a + c
   c = a + c
   a = c + 5
   b = c + b
   c = a + b + c

¿Qué valores quedan en las variables a, b y c?

7. a = 1
   b = 1
   c = 1
   a = a + a
   b = b + a
   c = c + a
   a = a + a
   b = b + a
   c = c + a

¿Qué valores quedan en las variables a, b y c?

8. a = 10
   b = 50
   c = 30
   a = a – b
   b = b – c
   c = c – a
   a = a – 1
   b = b – a
   c = c + a – b

   ¿Qué valores quedan en las variables a, b y c?

9. a = 1
   b = 2
   c = 3
   a = a + b
   b = a – b
   c = a * b
   a = a – b
   b = a + b
   c = a * b

   ¿Qué valores quedan en las variables a, b y c?

10. a = 1
    b = 2
    c = 3
    a = a + 2
    b = a + 2 + b
    c = a + 2 + c
    a = a / 2
    b = b / 2
    c = c / 2

    ¿Qué valores quedan en las variables a, b y c?

"Linealizar" las siguientes expresiones (no se olvide que linealizar significa escribir una expresión algebraica en una sola línea). En cada uno de los siguientes ejercicios, escribir el orden en que el computador realizaría las operaciones.

11.

$$x = \frac{a + \frac{b}{c}}{\frac{a}{b} + c}$$

12.

$$x = \frac{a + b + \frac{a}{b}}{c}$$

13.

$$x = \frac{\frac{a}{a+b}}{\frac{a}{a-b}}$$

14.

$$x = \frac{a + \frac{b}{a + b + \frac{b}{c}}}{a + \frac{b}{c + a}}$$

15.

$$x = \frac{a+b+c}{a+\frac{b}{c}}$$

16.

$$x = \frac{a+b+\frac{c}{d*a}}{a+b*\frac{c}{d}}$$

17.

$$x = \frac{a+\frac{b}{c}+d}{a}$$

18.

$$x = \frac{\frac{a}{b}+\frac{b}{c}}{\frac{a}{b}-\frac{b}{c}}$$

Capítulo 4

Estructuras básicas y técnicas para representar algoritmos

4.1 El concepto de estructura

Una estructura se define como un esquema que nos permite representar de manera simplificada alguna idea y que, bajo condiciones normales, es constante. Ello significa que, de alguna manera, el pensamiento del ser humano, en lo que se refiere a los algoritmos, está enmarcado en algún tipo de estructuras que no solo le permiten tener un medio más simplificado y a la mano para expresar las ideas, sino que además permite "restringir" un poco el horizonte de la lógica algorítmica.

Es pertinente, pues, hacer un breve paréntesis para explicar por qué es importante "restringir" un poco el horizonte de la lógica algorítmica. Comencemos con dos breves preguntas:

1. ¿Con cuántos algoritmos las señoras de la casa pueden preparar los fríjoles?

2. ¿Cuántas personas ve usted cerca que lleven puesta una camisa y un pantalón exactamente igual al suyo?

La primera pregunta se resuelve preguntándoles a tres o cuatro señoras acerca de su forma de preparar los fríjoles. Tenga la seguridad de que todas van a tener una manera diferente (o sea, un algoritmo diferente) para prepararlos si los vemos detalladamente, pero lo que va a ser coincidente en todas también es que logran el mismo objetivo, que es dejar listos los fríjoles para ser degustados. Esto nos va a demostrar, en primera instancia, que cada persona concibe algorítmicamente el mismo proceso de manera diferente, pero que

pueden llegar al mismo objetivo sin importar el camino que hayan escogido para lograrlo.

La segunda es todavía más reveladora y la voy a hacer en otro sentido. ¿Sabe usted por qué ninguna, o casi ninguna, persona lleva puesta una camisa y un pantalón exactamente igual al suyo? Pues sencillamente porque todas las personas están cumpliendo, en condiciones normales, con el objetivo de estar vestidos más no exactamente de la misma forma.

Esa variabilidad en cuanto a la concepción de un determinado algoritmo es lo que llevó a pensar en que la parte técnica también podría llegar a ser igualmente variable o, más bien, exageradamente variable. ¿Qué pasaba si una persona concebía un algoritmo computacional en unas condiciones lógicas que prácticamente solo ella la entendiera? Pues precisamente que el día que esa persona fuera despedida de la empresa o se fuera o falleciera, la empresa se vería en un verdadero y grande problema.

A nivel informal, la variabilidad de ópticas en cuanto a la concepción del mundo es lo que le ha permitido a este avanzar y es de allí que se han podido extractar tecnologías, modas, teorías y muchos avances del mundo moderno; pero a nivel técnico, sí resulta muy importante que la lógica para desarrollar un algoritmo computacional sea tan clara y tan "estándar" (si se puede decir así) que se pueda lograr que un programa desarrollado por una persona sea fácilmente entendible por cualquier otra, dado que podríamos llegar a encontrarnos con programas tan confusos que solo llegarían a ser entendibles por su creador.

Esa es la razón fundamental por la cual se buscó "uniformar" la lógica para desarrollar algoritmos computacionales y poder llegar a unas estructuras básicas sobre las cuales se pueda decir que está fundamentada dicha lógica.

4.2. Consideraciones algorítmicas sobre el pensamiento humano

Luego de analizar desde muchos ángulos el pensamiento humano, y teniendo en cuenta los cónceptos de algoritmo informal y algoritmo computacional, se llegó a la conclusión de que dicho pensamiento se mueve entre tres estructuras básicas:

4.2.1. Secuencia

Cuando usted está planeando ir este fin de semana a pasear con la familia, lo que en su mente se va dibujando poco a poco es una secuencia de acciones a

realizar y que le permitan pasar un fin de semana bien bueno. Cuando usted tiene que pensar que debe ir hasta el paradero de buses a tomar el transporte, lo que va organizando en su mente es una secuencia de acciones que le permitan acercarse al paradero, esperar el bus correcto y tomarlo para irse para su casa. Pues bien, esa es la primera estructura sobre la cual se mueve el pensamiento humano y es la estructura llamada **SECUENCIA**.

Permanentemente, usted está inmerso en esta estructura y, generalmente, usted primero planea cada secuencia de acciones (consciente o inconscientemente) antes de ejecutarlas. Cada una de las cosas que hacemos diariamente no son más que secuencias de acciones que hemos planeado para poder cumplir con nuestros objetivos en la sociedad.

4.2.2. Decisión

Usted ha llegado al paradero de buses, ve cómo pasan y pasan buses pero ninguno tiene la ruta que necesita porque usted vive en ese barrio para el cual hay un servicio de transporte muy deficiente. Por cada bus que pasa, usted le mira la ruta y, al ver que no es, espera el siguiente bus y así sucesivamente hasta que ve llegar al bus que usted necesita.

Usted está planeando el fin de semana, pero no sabe si pasar el domingo en el balneario que queda a una hora de la ciudad en donde vive o aprovechar e ir hasta la finca de aquel tío que hace mucho tiempo que no visita y que queda también a una hora. En cada alternativa, encuentra ventajas y desventajas y usted sabe que visitar al tío es bueno porque hace mucho tiempo que no lo ve, pero también sabe que la finca del tío no tiene piscina y el balneario sí y que le gustaría ver a su familia divertirse en ella.

Usted va a ver las noticias en algún noticiero de las 9:30 de la noche, pero aún no sabe en qué noticiero verlas, pues a esa hora presentan un noticiero diferente en cada uno de los canales de televisión.

Precisamente usted está *ad portas* de conocer la segunda estructura sobre la cual se basa el pensamiento (o razonamiento) humano. Esta es la estructura de **DECISIÓN**.

Gracias a la cual usted puede escoger lo que para usted sea la mejor alternativa de entre varias opciones, y hago hincapié en esto, porque cuando usted tiene (como erradamente dicen los periodistas) una sola alternativa, pues sencillamente no tiene alternativa y no hay caminos para escoger. La decisión se da siempre que usted tenga que escoger de entre, por lo menos, dos caminos lógicos.

4.2.3. Ciclos

Usted acostumbra todos los días a ver el noticiero de las 9:30 de la noche, acostumbra a ir al trabajo a la misma hora y a esperar el bus en el mismo paradero, acostumbra saludar de la misma forma a su esposa y acostumbra dormir en el mismo lado y en la misma posición. Usted sencillamente vive practicando la tercera estructura y son los **CICLOS**.

Que no es más que la estructura que nos permite repetir una o varias acciones una cantidad definida de veces dependiendo de una condición. Todos los días usted almuerza en su casa, según lo cual estará en el ciclo de ir a almorzar siempre, pero en pleno almuerzo, el hecho de que usted lleve muchas veces la cuchara del plato a su boca representa que usted estará haciendo lo mismo mientras en el plato exista todavía algo más para comer. Puede notar que permanentemente (e infortunadamente) estamos también realizando tareas cíclicas. Cuando nos aferramos mucho a estos ciclos de vida, es cuando la vida se nos vuelve tremendamente monótona.

Por ahora, lo que importa es que usted tenga claro que todo lo que usted hace, sin importar qué sea, cualquier acción o conjunto de acciones que usted haga siempre estarán enmarcadas en estas tres estructuras: secuencias de acciones, decisión de acción y ciclos de acciones.

También conviene que sepa que tomar una decisión depende de una determinada condición y que repetir un conjunto de acciones depende de que se cumpla o se deje de cumplir igualmente una condición.

4.3. Estructuras básicas expresadas técnicamente

Precisamente, y con el ánimo de facilitar unos patrones técnicos que permitan describir las ideas lógicas de una manera uniforme, se han desarrollado unos esquemas que nos van a permitir escribir las estructuras mencionadas anteriormente.

4.3.1. Las secuencias de órdenes

Para escribir una secuencia de órdenes o acciones, todo lo que tiene que hacer es colocar una nueva orden o una nueva acción después de la última que haya colocado. De esta manera, se entiende la secuencialidad y la ordinalidad en la ejecución de esas acciones.

Vamos a desarrollar un algoritmo que nos permita asomarnos a la ventana, pero vamos a asumir que la ventana a donde nos queremos asomar ya está abierta y que no estamos muy distantes de la ventana. Entonces podríamos decir:

Algoritmo para asomarnos a la ventana
Inicio
Ubicar la ventana por la que nos queremos asomar
Levantarnos del lugar en donde estemos sentados
Avanzar hacia la ventana
Llegar hasta tener la ventana muy muy cerquita
Asomarnos por la ventana
Fin

Tal vez usted puede notar que el enunciado de este ejercicio tiene unas condiciones que parecen inoficiosas. La razón de la presencia de estas condiciones es que, solo por el ejemplo, no quería que intervinieran otro tipo de estructuras.

En el ejemplo dado, usted puede ver que cada acción está antes de una y después de otra (excepto por supuesto la primera y la última). También puede notar que, para que este algoritmo nos permita asomarnos a la ventana, todo lo que tenemos que hacer es realizar cada acción en el orden en que están planteadas y sencillamente realizar una a la vez. Eso nos va a permitir lograr el objetivo propuesto.

Si queremos realizar el algoritmo para colocarnos una camisa (asumimos que la camisa está en nuestro ropero doblada y abrochada), entonces:

Algoritmo para colocarnos una camisa
Inicio
Dirigirnos a nuestro ropero
Abrir el ropero
Tomar una camisa
Desabrocharla
Abrir la camisa
Meter un brazo por una de sus mangas
Meter el otro brazo por la otra de sus mangas
Ajustar la camisa al tronco
Abotonarla (botón a botón)
Fin

Al igual que en el ejemplo anterior, todo lo que tenemos que hacer es ejecutar cada acción en el orden indicado y hacerlo paso a paso y entonces podremos lograr el objetivo de colocarnos la camisa.

Puede usted notar que, para utilizar la estructura de secuencia (que a veces parece ser tan obvia), todo lo que tenemos que hacer es ir colocando una acción tras otra y, por supuesto, ser muy racionales en el orden de dichas acciones, porque estoy seguro que, hasta el momento, usted ha podido notar que, en cuestión de algoritmos, el orden de los factores sí altera el resultado.

4.3.2. Las decisiones

Siempre que tenemos que tomar una decisión o, más bien, siempre que tengamos que utilizar la estructura de decisiones, vamos a depender de una condición. La condición es la que nos permite que podamos decidir cuál es el camino lógico correcto a tomar.

Vamos a desarrollar el mismo algoritmo de asomarnos a la ventana, pero esta vez no le vamos a colocar las condiciones de que estamos cerca de la ventana y de que esta está abierta. Para ello, vamos a incorporar unas líneas de decisión que nos permitan tener un algoritmo más genérico y que nos permitan lograr mejor el objetivo, así:

Algoritmo para asomarnos a la ventana
Inicio
    Ubicar la ventana por la que nos queremos asomar
    Si estamos sentados
        Levantarnos del lugar en donde estemos sentados
        Orientarnos hacia la ventana
    Sino
        Orientarnos hacia la ventana
    Avanzar hacia la ventana
    Llegar hasta tener la ventana muy muy cerquita
    Si está cerrada
        Abrirla
    Asomarnos por la ventana
Fin

Ya puede usted notar que nuestro algoritmo ha cambiado un poco y, por lo tanto, ahora tiene unas condiciones que le permiten ser una secuencia de acciones más racional. En estas condiciones, el algoritmo se convierte en algo más depurado y mucho más aproximado a la realidad. Note usted varias cosas en este algoritmo:

1. Las palabras *Si* que aparecen son exclusivamente condicionales y no afirmativas como pudiera pensarse en algunos casos.

2. Después de cada *Si* condicional va una condición, que es la que permite que se haga una cosa u otra. La condición regula las acciones que vienen después y que dependen del *Si* condicional inicial. En la decisión

Si estamos sentados

Levantarnos del lugar en donde estemos sentados

Orientarnos hacia la ventana

Sino

Orientarnos hacia la ventana

Notamos que estar sentados es la condición de la cual depende si hacemos las dos acciones (en caso de que la condición sea VERDADERA)

Levantarnos del lugar en donde estemos sentados

Orientarnos hacia la ventana

O si solo hacemos la acción (en caso de que la condición sea FALSA)

Orientarnos hacia la ventana

3. Puede usted notar que una decisión completa involucra:

Una pregunta que evalúa una condición.

Un conjunto de acciones a realizar en caso de que la condición sea evaluada como VERDADERA.

Un conjunto de acciones a realizar en caso de que la condición sea evaluada como FALSA.

Esta última parte, dentro del numeral 3, es la razón de ser de la existencia de la acción *Sino*.

4. No siempre que exista un condicional *Si* debe existir un *Sino* asociado a él. Siempre que exista un *Sino* es porque está asociado a un *Si* condicional determinado. Tal es el caso de la decisión

Si está cerrada

Abrirla

En donde, *Si la ventana está abierta* no hay que hacer más que asomarse por ella, pero si está cerrada debemos primero abrirla para poder asomarnos por ella.

Retomando el segundo ejemplo, y sabiendo que contamos con una estructura para mejorar los algoritmos, podremos adecuarlo de manera que el algoritmo para colocarnos una camisa quede de la siguiente forma:

Algoritmo para colocarnos una camisa
Inicio

Dirigirnos a nuestro ropero
Si está cerrado
Abrirlo
Tomar una camisa
Si está abrochada
Desabrocharla
Abrir la camisa
Si está doblada
Desdoblarla
Meter un brazo por una de sus mangas
Meter el otro brazo por la otra de sus mangas
Ajustar la camisa al tronco
Si es una camisa de botones
Abotonarla (botón a botón)
Ajustarla al cuerpo
Sino
Ajustarla de manera que quede bien puesta
Fin

Claramente aquí se puede notar una utilización alta de condicionales *Si* que no tienen mucha necesidad de tener un *Sino* por las razones lógicas del mismo algoritmo. Es obvio que usted podrá tener muchos "reparos" a este algoritmo porque alguno o algunos de los pasos aquí consignados no coinciden con su lógica, pero tenga en cuenta que todos los algoritmos planteados en este libro son solo una idea del autor y que si su idea propia (amigo lector) es acertada, es decir, logra los mismos objetivos, así el algoritmo sea diferente, estará completamente correcto.

Sé que han de existir muchas diferencias de concepción, sobre todo en cuanto a este tipo de algoritmos informales, pero lo importante es que usted se vaya

acostumbrando a una filosofía propia de los algoritmos para expresar cualquier idea.

4.3.3. Los ciclos

Vamos a suponer, para ampliar nuestros ejemplos, que usted es un supervisor de una fábrica y que cada media hora, a lo largo de todo el día, debe estar vigilando determinada acción a través de una ventana. El algoritmo para cumplir su objetivo, que es el de *Vigilar* (como supervisor de la fábrica), parte de una unidad muy sencilla y es *Asomarse por una ventana*. En palabras sencillas, usted tendrá que asomarse por una ventana mientras no termine el día cada media hora y durante el tiempo que usted no esté asomado, lo que tiene que hacer es seguir en su puesto de trabajo. De esta forma, y partiendo de lo que ya tenemos, usted podrá estructurar un algoritmo de la siguiente manera:

```
Algoritmo para Vigilar desde una ventana
Inicio

        Llegar puntual a la hora de inicio de la jornada laboral
        Ubicarnos en nuestro escritorio
        Mientras no sea el fin del día
                Ubicar la ventana por la que nos queremos asomar
                Si estamos sentados
                        Levantarnos del lugar en donde estemos sentados
                        Orientarnos hacia la ventana
                Sino
                        Orientarnos hacia la ventana
                Avanzar hacia la ventana
                Llegar hasta tener la ventana muy muy cerquita
                Si está cerrada
                        Abrirla
                Asomarnos por la ventana
                Regresar a nuestro escritorio
                Mientras no haya pasado Media Hora
                        Permanecer en nuestro escritorio
                Fin_Mientras
        Fin_Mientras
Fin
```

Varios factores nuevos entran en este algoritmo:

1. La palabra *Mientras* establece (en relación con una condición) el inicio de un conjunto de acciones que se repiten precisamente *Mientras* esa condición lo permita.

2. Todo *Mientras* (por efectos de clarificación del algoritmo) debe tener un finalizador que indique hasta dónde llega el bloque de acciones que debemos repetir.

3. La indentación o lo que corrientemente se conoce como el "sangrado" del texto, es decir, el hecho de que algunas acciones estén más adentro de la hoja que otras, representa que existen bloques de acciones que tienen una característica.

Las acciones contenidas entre el *Inicio* y el *Fin* indican que son las acciones que conforman el algoritmo en mención.

Las acciones comprendidas entre *Mientras no sea Fin del día* y su correspondiente *Fin_Mientras* son el conjunto o bloque que se debe repetir (o iterar) precisamente mientras la condición sea *Verdadera* o sea *Mientras no sea fin del día*.

La acción comprendida entre *Mientras no haya pasado Media Hora* y su correspondiente *Fin_Mientras* es la acción que se deberá realizar hasta cuando se complete media hora.

4. Cada ciclo de acciones que se inicie con *Mientras* deberá tener un *Fin_Mientras* asociado y a su vez cada *Fin_Mientras* deberá finalizar con uno y solo un ciclo iniciado con *Mientras*.

Supongamos que usted es el inspector de calidad de un almacén de ropa y su trabajo consiste en medirse algunas de las camisas que están en los roperos del almacén para verificar su ajuste en cuanto a la talla. Entonces, mientras no termine su jornada de trabajo, usted lo que hará será ir de ropero en ropero tomando una camisa y midiéndosela. De esta forma, si partimos del algoritmo de colocarnos una camisa que ya tenemos, entonces este nuevo objetivo puede cumplirse de la siguiente forma:

Algoritmo Inspeccionar las camisas en un almacén de ropa
Inicio
    Llegar puntuales al inicio de la jornada laboral
    Mientras no sea fin de la jornada laboral
        Dirigirnos a un ropero
        Si está cerrado
            Abrirlo
        Tomar una camisa
        Si está abrochada
            Desabrocharla
            Abrir la camisa
        Si está doblada
            Desdoblarla
        Meter un brazo por una de sus mangas
        Meter el otro brazo por la otra de sus mangas
        Ajustar la camisa al tronco
        Si es una camisa de botones
            Abotonarla (botón a botón)
            Ajustarla al cuerpo
        Sino
            Ajustarla de manera que quede bien puesta
        Emitir el concepto de calidad sobre la camisa
    Fin_Mientras
Fin

Las apreciaciones acerca de este algoritmo coinciden en su mayoría con las apreciaciones acerca del algoritmo anterior (dentro de este mismo tema). Es de anotar que así como, por claridad, se utiliza un *Fin_Mientras* para indicar en donde termina el bloque de instrucciones que se deben operar, es conveniente utilizar un *Fin_Si* para indicar hasta dónde llega completamente una decisión y, en unión con la indentación de acciones, tener claridad en cuanto a los "bloques" de acciones que se formen dentro del algoritmo.

Estos algoritmos informales están expresados tal como desprevenidamente cualquier persona los expresaría y puede entonces suponer usted que la variabilidad de algoritmos que cumplan los mismos objetivos sería inmensa si no existieran unas técnicas uniformes para facilitar la expresión de estas ideas, particularmente en algoritmos computacionales.

4.4. Técnicas para representar algoritmos

4.4.1. Diagramas de flujo

Un diagrama de flujo parte de unos símbolos que nos permiten decir lo mismo que dijimos hace un momento en los algoritmos pero de una manera gráfica y, por supuesto, un poco más entendible. Los siguientes son algunos de los símbolos (y el significado de ellos) que se han acordado utilizar dentro de los diagramas de flujo o flujogramas.

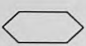
Un rectángulo representa un proceso que es una acción o una orden a ejecutarse de manera clara y concreta. Un ejemplo típico de proceso es la asignación de un valor a una variable.

Este símbolo nos permite representar una decisión. En su interior, podemos escribir la condición de la cual depende la decisión y por sus extremos derecho (o izquierdo) e inferior se pueden colocar las salidas para los casos en que la condición sea falsa o sea verdadera.

Este símbolo nos permite expresar un proceso de entrada o salida, teniendo en cuenta que una entrada en un algoritmo se concibe como el proceso a través del cual se recibe información y una salida es el proceso a través del cual se entrega información.

Este símbolo permite representar la escritura de un resultado o lo que técnicamente se conoce como una salida.

Este símbolo representa el inicio o el fin de un algoritmo. Todo lo que se tiene que hacer es escribir la palabra *Inicio* o *Fin* y ubicarlo apropiadamente dentro del diagrama de flujo.



Este símbolo permite que coloquemos en él los parámetros de inicio de un ciclo cuando se ajustan a una de las

formas establecidas por las normas de programación. En el capítulo de ciclos desglosaremos un poco más esta definición.

Este símbolo representa una entrada de datos utilizando el teclado del computador. Todo lo que tenemos que escribir en su interior es el nombre de la variable (o las variables) en donde queremos que se almacene el dato que entra por el teclado.

Estos símbolos se conocen como conectores lógicos. Nos permiten representar la continuación de un diagrama de flujo cuando este es tan largo que no cabe en una sola hoja.

Este símbolo permite representar una lectura de datos. Representa una tarjeta perforada, pues esta técnica fue establecida cuando aún se leían los datos a través de tarjetas perforadas. Actualmente, este símbolo representa sencillamente una lectura.

Este símbolo genera una salida de datos. Representa una cinta perforada porque, al igual que el símbolo anterior, esta técnica fue establecida cuando aún se generaba la salida de datos a través de una tarjeta perforada. En la actualidad, este símbolo representa sencillamente una salida o una escritura de datos.

Este símbolo representa una salida de datos pero escrita en la pantalla del computador. Es un símbolo un poco más moderno para efectos de los diagramas de flujo.

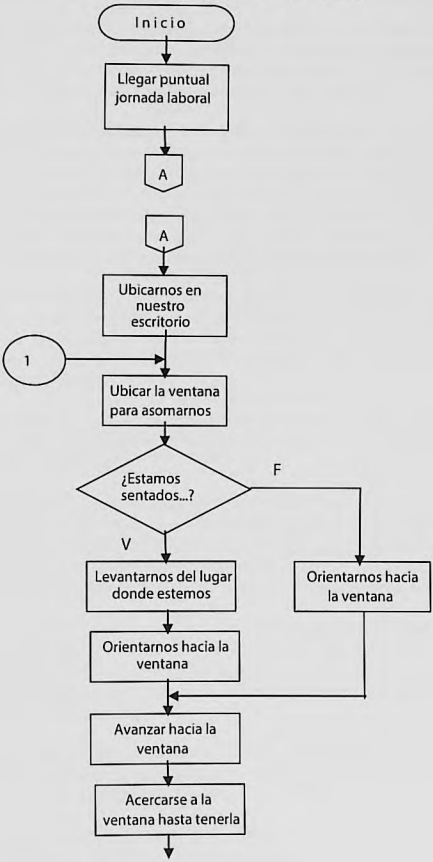
Las flechas son los símbolos que nos van a permitir representar la forma de conexión entre los demás símbolos, determinando igualmente el flujo de ejecución o realización de acciones.

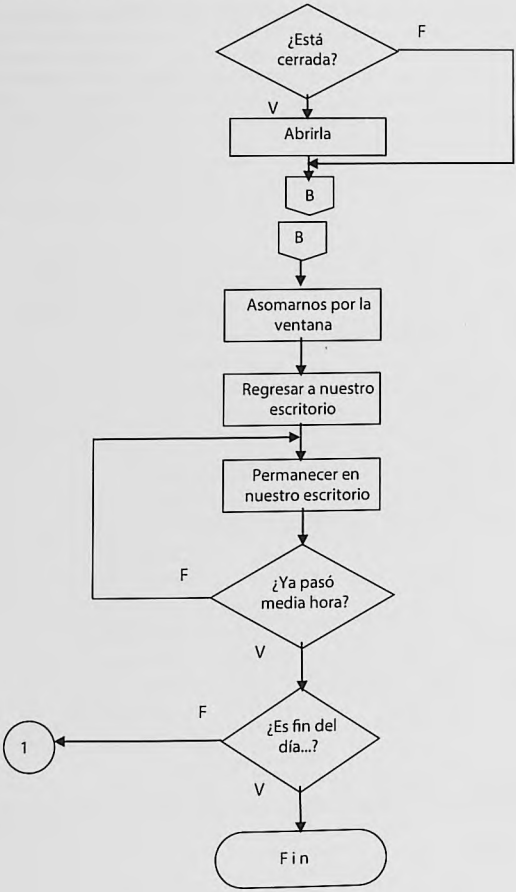
Estos símbolos (en unión con otros símbolos que para efectos de nuestro libro tal vez no haya necesidad de citar) fueron utilizados por mucho tiempo para representar gráficamente una idea o un algoritmo. ¿Cómo se utiliza entonces esta simbología? Tomemos el caso de los dos algoritmos que construimos mientras conocíamos las estructuras básicas. El enunciado final buscaba desarrollar un algoritmo que nos permitiera *Vigilar una empresa desde una ventana asomándonos cada media hora por ella*. El algoritmo lo habíamos planteado como sigue a continuación:

```
Algoritmo para Vigilar desde una ventana
Inicio
        Llegar puntual a la hora de inicio de la jornada laboral
        Ubicarnos en nuestro escritorio
        Mientras no sea el fin del día
                Ubicar la ventana por la que nos queremos asomar
                Si estamos sentados
                        Levantarnos del lugar en donde estemos sentados
                        Orientarnos hacia la ventana
                Sino
                        Orientarnos hacia la ventana
                Avanzar hacia la ventana
                Llegar hasta tener la ventana muy muy cerquita
                Si está cerrada
                        Abrirla
                Asomarnos por la ventana
                Regresar a nuestro escritorio
                Mientras no haya pasado Media Hora
                        Permanecer en nuestro escritorio
                Fin_Mientras
        Fin_Mientras
Fin
```

Si queremos llevarlo a la simbología de diagramas de flujo, su equivalente sería el siguiente:

Diagrama de flujo para *Vigilar desde una ventana*




¿Está cerrada?
F
V
Abrirla
B
B
Asomarnos por la ventana
Regresar a nuestro escritorio
Permanecer en nuestro escritorio
F
¿Ya pasó media hora?
V
F
1
¿Es fin del día...?
V
Fin

Cabe destacar algunos detalles significativos en este diagrama de flujo:

1. Toda decisión, como es obvio, tiene dos caminos: un camino nos lleva a la acción o a las acciones a realizar en el caso de que la respuesta a la pregunta sea verdadera y el otro camino es el que nos dice qué debemos hacer en caso de que la respuesta a la pregunta sea falsa.

2. Lo que en el algoritmo eran unos ciclos, en el diagrama se cambiaron por unas decisiones en donde uno de los caminos se devuelve (instrucciones atrás). Al realizar un seguimiento de este diagrama de flujo, usted notará que se podrá devolver tantas veces como lo permita la condición de la decisión que queda al final y que solo se va a salir de ese ciclo cuando la condición sea verdadera, o sea, que el ciclo se mantiene mientras la condición sea falsa, lo cual concuerda con la teoría de los ciclos.

3. En la última decisión, el camino *Falso* nos lleva a una burbuja que tiene un número 1 adentro. Número que también está al principio del diagrama, pero con la flecha en el otro sentido (es decir, no saliendo del diagrama, sino entrando a él). Se utiliza esta notación solo para simplificar un poco el diagrama de flujo.

4. Con el diagrama de flujo, usted puede ver un gráfico de la solución y con ello hacerse una idea clara de la secuencia de pasos que necesitaría para alcanzar el objetivo.

5. Siempre que vaya a desarrollar un diagrama de flujo, trate de ser muy organizado y muy estético, pues no se olvide que si vamos a representar un algoritmo computacional (en donde se busca que el computador logre un objetivo por nosotros), al momento de la transcripción será muy importante el orden que usted haya tenido en la utilización de esta técnica.

6. Cuando diseñe un ciclo, no se olvide verificar que, lógicamente, la decisión por la cual reemplace el ciclo al momento de diseñar su diagrama de flujo tenga el mismo comportamiento, es decir, permita que bajo las mismas condiciones una acción o un conjunto de acciones se repitan una cantidad finita de veces.

7. Si el algoritmo que usted tiene para lograr este mismo objetivo es diferente, tenga presente que el diagrama de flujo también va a ser diferente, ya que este es un reflejo gráfico de aquel.

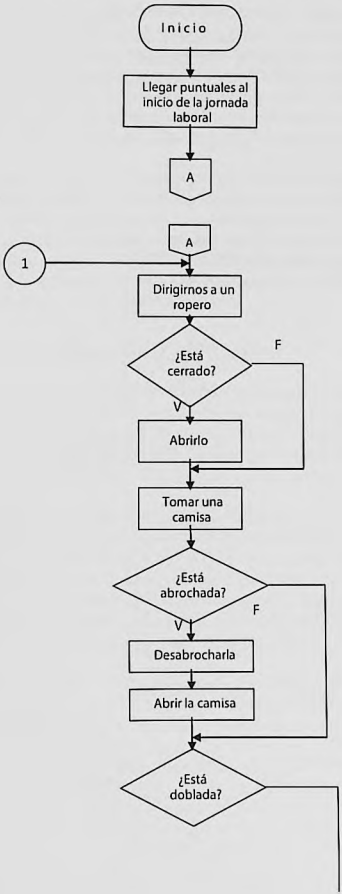
8. Es muy importante que sepa que el simple hecho de cambiar la cabeza de una determinada flecha cambia completamente el algoritmo. Puede

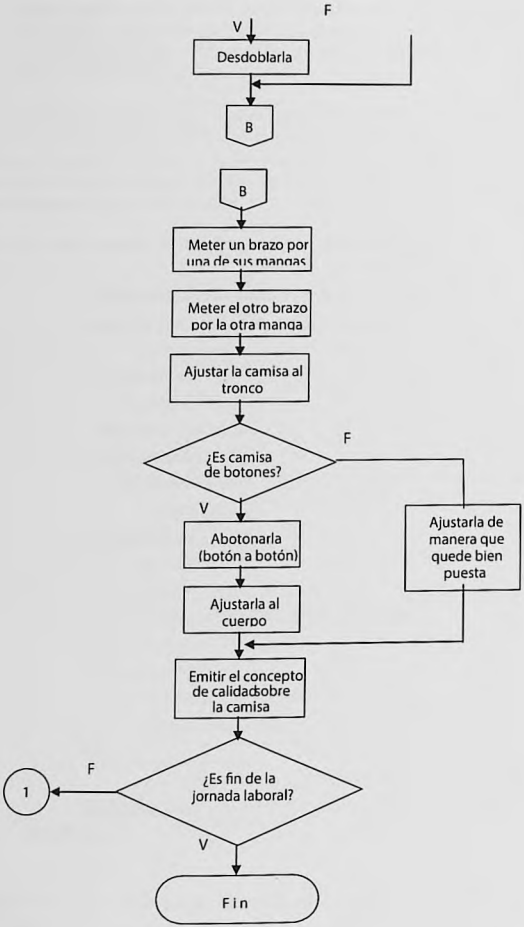
usted notar que la utilización de los símbolos resulta ser una tarea muy simplificada, pero lo que sí es delicado es la colocación de las flechas, ya que ellas son las que representan el sentido con que se va a "mover" el flujo de nuestra lógica.

Para clarificar aún más lo que hasta ahora hemos dicho, vamos a diseñar el diagrama de flujo del algoritmo para inspeccionar las camisas en un almacén de ropa. Para ello, y tal como lo hicimos en el algoritmo anterior, partamos de la solución final que dimos a este algoritmo en donde involucrábamos secuencias de acciones, decisiones y ciclos.

Algoritmo para Inspeccionar las camisas en un almacén de ropa
Inicio
Llegar puntuales al inicio de la jornada laboral
Mientras no sea fin de la jornada laboral
Dirigirnos a un ropero
Si está cerrado
Abrirlo
Tomar una camisa
Si está abrochada
Desabrocharla
Abrir la camisa
Si está doblada
Desdoblarla
Meter un brazo por una de sus mangas
Meter el otro brazo por la otra de sus mangas
Ajustar la camisa al tronco
Si es una camisa de botones
Abotonarla (botón a botón)
Ajustarla al cuerpo
Sino
Ajustarla de manera que quede bien puesta
Emitir el concepto de calidad sobre la camisa
Fin_Mientras
Fin

Llevado a la simbología de un diagrama de flujo, su equivalente sería el siguiente:


Inicio
Llegar puntuales al inicio de la jornada laboral
A
A
1
Dirigirnos a un ropero
¿Está cerrado?
F
V
Abrirlo
Tomar una camisa
¿Está abrochada?
F
V
Desabrocharla
Abrir la camisa
¿Está doblada?


F
V
Desdoblarla
B
B
Meter un brazo por una de sus mangas
Meter el otro brazo por la otra manga
Ajustar la camisa al tronco
¿Es camisa de botones?
F
V
Abotonarla (botón a botón)
Ajustarla al cuerpo
Ajustarla de manera que quede bien puesta
Emitir el concepto de calidad sobre la camisa
F
1
¿Es fin de la jornada laboral?
V
Fin

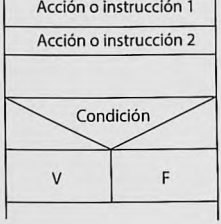
Estoy seguro de que usted no estará de acuerdo conmigo en algunos de los pasos o acciones de este algoritmo (o del anterior), pero no se preocupe. Cada uno de nosotros actúa bajo una lógica propia pero enmarcada dentro de unas normas que son generales a todas. Si usted no está de acuerdo con los diagramas expuestos hasta aquí será muy normal, pues eso le demuestra que su algoritmo puede ser diferente al mío y diferente al de cualquier otra persona. Sin embargo, estoy seguro que tanto su algoritmo como el mío lograrán el objetivo en ambos casos.

Diagrame su propia idea de estos dos algoritmos y verá, con seguridad, que su diagrama es completamente diferente a los expuestos en este libro. No olvide que en este libro aparecen solo las soluciones de una persona. El hecho de que sus soluciones no coincidan con las mías no necesariamente quiere decir que su solución esté mal. Recurra siempre a la Prueba de Escritorio antes de dudar de sus soluciones y, además, verifique las soluciones de este libro valiéndose de la misma herramienta.

4.4.2. Diagramas rectangulares estructurados

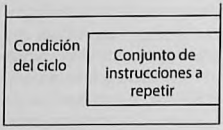
Una de las dificultades de los diagramas de flujo radica en que así como brinda la posibilidad de representar gráficamente el flujo de una idea o el "recorrido" de la solución a un problema también abre el espacio para que un programador desordenado ponga flechas de flujo a diestra y siniestra y, finalmente, obtenga una representación más compleja que la idea misma.

Precisamente, la técnica de Diagramas Rectangulares Estructurados (DRE) nos permite tener unas herramientas gráficas para representar la solución a un problema con la ventaja de que nos brinda la posibilidad de que seamos desordenados en nuestra concepción. Gráficamente, se basa en representar todo el algoritmo dentro del marco de un rectángulo y, a diferencia de la técnica anterior, la DRE se basa en la utilización de tres símbolos que corresponden a cada una de las estructuras básicas de la lógica de programación. Estas representaciones son las siguientes:



Para representar secuencias de instrucciones, todo lo que tenemos que hacer es colocar cada instrucción en una línea "enmarcada".

Para representar una decisión, se utiliza este símbolo en donde por el lado izquierdo podemos colocar las acciones o instrucciones que corresponderían ejecutar en el caso de que la condición fuera verdadera y por el lado derecho colocaríamos las acciones o instrucciones a ejecutar cuando la condición fuera falsa.

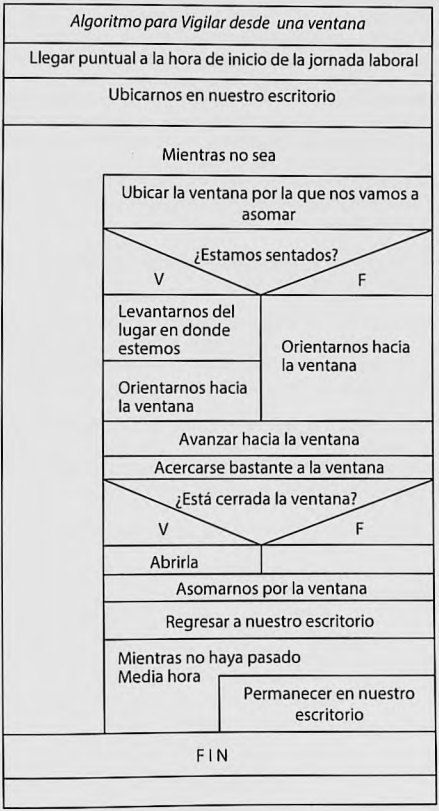


Para representar un ciclo, sencillamente en la esquina superior izquierda del bloque correspondiente colocamos la condición y dentro del bloque colocamos las instrucciones o acciones que se deben repetir y que, a su vez, por supuesto, dependen de la condición.

Pero, definitivamente, la utilización efectiva de esta técnica de representación se ve con un ejemplo. Vamos a estructurar en diagramación rectangular estructurada los dos algoritmos de ejemplo que ya representamos en la técnica de diagrama de flujo. Para ello, volvamos a recordar el primer enunciado y su correspondiente solución a nivel algorítmico. Se trataba de desarrollar un algoritmo que nos permitiera cada media hora durante la jornada de trabajo laboral vigilar desde una ventana. El algoritmo que fue planteado como solución final fue el siguiente:

```
Algoritmo para Vigilar desde una ventana
Inicio
        Llegar puntual a la hora de inicio de la jornada laboral
        Ubicarnos en nuestro escritorio
        Mientras no sea el fin del día
                Ubicar la ventana por la que nos queremos asomar
                Si estamos sentados
                        Levantarnos del lugar en donde estemos sentados
                        Orientarnos hacia la ventana
                Sino
                        Orientarnos hacia la ventana
                Avanzar hacia la ventana
                Llegar hasta tener la ventana muy muy cerquita
                Si está cerrada
                        Abrirla
                Asomarnos por la ventana
                Regresar a nuestro escritorio
                Mientras no haya pasado Media Hora
                        Permanecer en nuestro escritorio
                Fin_Mientras
        Fin_Mientras
Fin
```

Ahora bien, llevado este algoritmo a nivel de diagramación rectangular estructurada, el resultado sería el siguiente:



Es imperante hacer algunas precisiones acerca de este diagrama:

a. Puede usted notar que la correspondencia entre nuestra idea y su representación (bajo esta técnica) es mucho más exacta que en el caso del diagrama de flujo en donde tuvimos que hacer algunos pequeños cambios lógicos para que el diagrama correspondiera a la solución planteada.

b. La técnica de diagramación rectangular estructurada obliga a ser mucho más ordenado y no da ningún espacio para que nuestro algoritmo sea inentendible, dado que las estructuras son relativamente rígidas.

c. Para la utilización de esta técnica, solo tenemos que conocer tres símbolos y, con ellos, representamos todo lo que queramos, dado que nuestra lógica se basa en esas tres estructuras.

d. Enmarcar nuestra idea en un rectángulo nos brinda una concepción más concreta de la solución planteada.

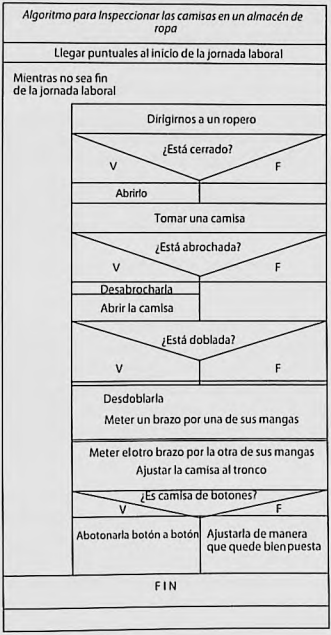
e. Realizar una prueba de escritorio con un diagrama basado en esta técnica se reduce a seguir la secuencia de instrucciones y (al igual que con los diagramas de flujo) a realizar una a una y tal como están allí las instrucciones o acciones, las decisiones y la revisión de las condiciones de los ciclos.

Tomemos el segundo algoritmo y realicemos su correspondiente diagrama rectangular estructurado. El enunciado buscaba diseñar un algoritmo para inspeccionar la calidad de las camisas en un almacén de ropa. La solución algorítmica que se planteó fue la siguiente:

Algoritmo para Inspeccionar las camisas en un almacén de ropa
Inicio
    Llegar puntuales al inicio de la jornada laboral
    Mientras no sea fin de la jornada laboral
        Dirigirnos a un ropero
        Si está cerrado
            Abrirlo
        Tomar una camisa
        Si está abrochada
            Desabrocharla
            Abrir la camisa
        Si está doblada
            Desdoblarla
        Meter un brazo por una de sus mangas
        Meter el otro brazo por la otra de sus mangas

Ajustar la camisa al tronco
Si es una camisa de botones
Abotonarla (botón a botón)
Ajustarla al cuerpo
Sino
Ajustarla de manera que quede bien puesta
Emitir el concepto de calidad sobre la camisa
Fin_Mientras
Fin

Llevándolo a su equivalente en diagramación rectangular estructurada, obtendríamos el siguiente diagrama:



Es importante anotar que cuando exista una decisión podremos tener o no acciones o instrucciones por alguno de sus ramales. Con ello quiero decir que es absolutamente correcto tener una decisión que tenga acciones en el caso de que la condición sea evaluada como verdadera y no tenga nada en el caso de que sea evaluada como falsa dicha condición. Cabe anotar que también es correcto tener una decisión dentro de otra decisión o tener un ciclo dentro de otro ciclo o tener una decisión de un ciclo o un ciclo dentro de una decisión, puesto que ninguna de las estructuras es excluyente.

4.4.3. Seudocódigos

La tercera técnica para representar algoritmos es la más obvia y seguramente usted no le va a encontrar nada nuevo: es la técnica de los seudocódigos. ¿Qué es un seudocódigo...? Pues sencillamente es la representación textual de un algoritmo de manera que dicho texto se encuentre enmarcado en algunas normas técnicas que faciliten su posterior transcripción a un lenguaje de programación.

Por lo dicho anteriormente, es evidente que la técnica de seudocódigo está muy orientada hacia los algoritmos computacionales. Cuando se habla de algunas normas, estamos diciendo que existen unos requisitos que, si bien pueden ser violados, facilitan la posterior transcripción del algoritmo a un lenguaje de programación para ser cumplidos a cabalidad. No debemos perder el faro que todas estas técnicas nos deben facilitar para la posterior transcripción de los algoritmos.

Para escribir pues un algoritmo bajo la forma de seudocódigo, algunas de las normas son las siguientes:

Primera norma. Siempre se le ha de colocar un nombre al algoritmo de manera que sea lo primero que se lea. Es conveniente acostumbrarse a que dicho nombre sea altamente mnemónico, o sea, que su nombre haga una referencia aproximada a lo que hace. Si a un seudocódigo lo llamamos Seudocódigo X, es posible que más adelante no nos sea muy claro su objetivo; pero si lo llamamos Seudocódigo Liquidar, es muy factible que cada vez que lo veamos nos acordemos de que su objetivo era la liquidación de un determinado valor.

Pero si lo llamamos Seudocódigo LiqSalNe, es muy posible que cada vez que veamos este nombre nos acordemos de que ese seudocódigo es el que nos permite Liquidar el Salario Neto. Resulta muy conveniente, por todas las razones que usted se imagine, que el nombre de los algoritmos expresados en seudocódigos sea lo más mnemónico posible, pues no sabemos cuándo tendremos que retomarlos y es allí en donde vamos a ver la gran importancia del buen nombre de un algoritmo.

Segunda norma. Luego de colocado el nombre del seudocódigo, debemos a continuación declarar las variables con las cuales vamos a trabajar durante el programa. Todas las variables que vayan a ser utilizadas deben ser declaradas.

Declararlas significa escribir el tipo de dato que van a almacenar y el nombre que dichas variables van a llevar. No se olvide que vamos a trabajar con tres tipos estándar de datos, como son los datos de tipo entero, datos de tipo real y datos de tipo carácter, que tienen cada uno unas características y unas restricciones.

Tercera norma. Todo el cuerpo del algoritmo deberá ir "encerrado" entre las palabras *Inicio* y *Fin* indicando en donde comienza y en donde termina el seudocódigo.

Cuarta norma.

a. Cuando quiera que salga un título en la pantalla, todo lo que tiene que hacer es utilizar la orden *Escriba* y, a continuación, colocar entre comillas dobles lo que quiera que salga en pantalla. Por ejemplo:

   Escriba "Esta es una demostración"
   Generará en pantalla el título *Esta es una demostración*

b. Si usted quiere que lo que salga en pantalla sea el contenido de una variable, todo lo que tiene que hacer es utilizar la orden *Escriba* y, a continuación, y sin comillas dobles, el nombre de la variable que quiere escribir. Por ejemplo

   N = 5
   Escriba N
   Mostrará en pantalla el valor *5*

c. Si quiere que en pantalla salga un título y a continuación salga el contenido de la variable, todo lo que tiene que hacer es colocar el título entre comillas dobles y, luego de haberlas cerrado, colocar el nombre de la variable que usted quiere escribir al lado del título. Por ejemplo:

   N = 8
   Escriba "El valor es" N
   Generará en pantalla *El valor es 8*

d. Si quiere mostrar en pantalla el contenido de varias variables, entonces simplemente a continuación de la orden *Escriba* y separadas por comas puede escribir los nombres de las variables que usted quiere escribir. Por ejemplo:

N = 8

M = 4

Escriba "Los valores son" N, M

Escribirá en pantalla *Los valores son 8 4*

Quinta norma.

a. Cuando usted vaya a leer un dato para que sea almacenado en una variable determinada, utilice la orden *Lea*. Para no tener que escribir (por ejemplo):

***Lea** un dato entero y guárdelo en la variable **N** que también es entera*

Solo tiene que escribir

Lea N

y el computador lo entenderá correctamente.

b. Cuando necesite leer más de un dato para ser almacenado en diferentes variables, todo lo que tiene que hacer es utilizar la orden *Lea* y escribir las variables separadas por comas. Por ejemplo:

Lea a, b

Suponiendo que tanto *a* como *b* son variables de tipo entero, esta orden le indicará al computador que lea un dato entero y lo almacene en la variable *a* y luego que lea otro dato entero y lo almacene en la variable *b*.

c. No necesariamente, cuando se lean dos o más variables utilizando una sola orden *Lea*, estas deben ser del mismo tipo. Por ejemplo:

Lea var_e, var_r, var_c

Asumiendo que *var_e* es una variable de tipo entero, *var_r* es una variable de tipo real y *var_c* es una variable de tipo carácter, esta orden le indicará al computador que lea un valor entero y lo almacene en la variable *var_e*, luego que lea un valor real y lo almacene en la variable *var_r* y luego que lea un carácter y lo almacene en la variable *var_c*.

Sexta norma. Cuando necesite tomar una decisión, deberá utilizar la orden ***Si***, a continuación escribir la *condición* correspondiente y luego las instrucciones que se han de realizar en caso de que la condición sea *Verdadera*. En caso de que la condición sea *Falsa* y tenga instrucciones a realizarse cuando así sea,

entonces deberá existir una alternativa ***Sino***. Al finalizar toda la decisión, deberá existir un indicador ***Fin_Si***. La estructura entonces será la siguiente:

```
Si Condición
        .
        .
        Instrucciones a ejecutar si la condición es Verdadera
        .
        .
Sino
        .
        .
        Instrucciones a ejecutar si la condición es Falsa
        .
        .
Fin_Si
```

Las condiciones pueden ser expresadas utilizando los siguientes ***operadores relacionales***, que son los símbolos que nos van a permitir obtener una respuesta *Verdadera* o *Falsa*:

>	Mayor que
<	Menor que
>=	Mayor o igual a
<=	Menor o igual a
=	Igual a (también llamado igual de comparación)
< >	Diferente de

Es importante anotar que el signo igual (=), utilizado como operador relacional, tiene una función diferente que el signo igual (=) utilizado para asignarle un valor a una variable. En el primer caso, sería utilizado para preguntar, por ejemplo, si el contenido de la variable ***a*** es igual al valor 5 y en el segundo caso estaríamos asignando a la variable ***a*** el valor 5. Son dos usos diferentes.

Algunos lenguajes diferencian sintácticamente el igual de comparación del igual de asignación precisamente para que el compilador no tenga la opción de realizar interpretaciones ambiguas. En este libro, será claro que la utilización del igual de asignación será en instrucciones de asignación y el igual de comparación será utilizado en decisiones.

También podemos conectar expresiones relacionales (tales como a<b) a través de los ***operadores booleanos***, que son unos signos que nos van a permitir expresar dichas relaciones.

Cuando éramos niños y nos decían, al mandarnos a la tienda, *tráigame una gaseosa y un pan de $1000*, teníamos varias opciones:

a. Si no traíamos ninguna de las dos cosas, entonces no habíamos cumplido la orden.

b. Si no traíamos la gaseosa pero sí traíamos el pan de $1000, tampoco habíamos cumplido la orden.

c. Si traíamos la gaseosa pero no traíamos el pan de $1000, tampoco habíamos cumplido la orden.

d. Si traíamos la gaseosa y también traíamos el pan de $1000, entonces allí sí habíamos cumplido la orden completamente.

Para este ejemplo, y tal como lo hacía nuestra progenitora, asumimos que cumplir la orden es hacer el "mandado" completo. Igualmente, vamos a asumir la frase *Hemos cumplido la orden* como un *Verdadero* (V) y la frase *No hemos cumplido la orden* como un *Falso* (F). De esta forma, podríamos organizar la situación según se muestra en la tabla 6.

Tabla 6. Tabla de verdad operador Y (AND)

Condición 1 Tráigame una gaseosa	Condición 2 Tráigame un pan de $1000	Cond1 y Cond2	Explicación textual
F	F	F	No cumplimos completamente la orden
F	V	F	No cumplimos completamente la orden
V	F	F	No cumplimos completamente la orden
V	V	V	Sí cumplimos completamente la orden

Puede notar usted que solamente habremos cumplido la orden completamente si hemos cumplido cada una de las condiciones. Igualmente, observe que en el título de la tercera columna hay una **Y** un poco más grande que el resto del texto. Precisamente, esa es el primer operador booleano que vamos a conocer. Es el operador **Y**, que en la mayoría de los libros se conoce como el operador **AND** y que precisamente aquí lo vamos a llamar **AND**.

Su tabla de verdad es la que está expuesta en ese ejemplo, por lo cual podemos concluir que, cuando utilizamos un operador **AND**, solamente genera *Verdadero* si ambas condiciones se cumplen (no olvide que se habla de ambas condiciones porque el operador AND puede "conectar" solamente dos condiciones) y en cualquier otro caso genera *Falso*.

Cuando nos decían *Tráigame una gaseosa litro o una bolsa de leche* nos estaban dando las siguientes opciones:

a. Si no traemos ninguna de las dos cosas, entonces no hemos cumplido la orden.

b. Si no traemos la gaseosa litro y traemos la bolsa de leche, entonces hemos cumplido la orden.

c. Si traemos la gaseosa litro y no traemos la bolsa de leche, entonces hemos cumplido la orden.

d. Si traemos ambas cosas, hemos cumplido sobradamente la orden.

Podemos, pues, con este razonamiento organizar la siguiente tabla de verdad (asumiendo que cumplir la orden se representará con *Verdadero* y no cumplirla se representará con *Falso*). La tabla 7 presenta las situaciones posibles.

Tabla 7. Tabla de verdad operador O (OR)

Condición 1 Tráigame una gaseosa litro	Condición 2 Tráigame una bolsa de leche	Cond1 O Cond2	Explicación textual
F	F	F	No cumplimos la orden
F	V	V	Cumplimos la orden
V	F	V	Cumplimos la orden
V	V	V	Requete cumplimos la orden

Puede notar que en la tercera columna de esta tabla hay una **O** que une a la Condición1 con la Condición2 y que es precisamente el nombre del segundo operador booleano. Técnicamente, lo vamos a llamar **OR**. Cuando el operador OR une dos condiciones, toda la expresión es verdadera si, al menos, una de las dos es verdadera. Es obvio pensar que, en el caso en que las dos condiciones sean verdaderas, entonces toda la expresión será más que verdadera.

El tercer operador booleano es el operador **NOT.** Este operador actúa sobre una sola expresión y lo que hace es que invierte el sentido de la condición, es decir, cuando el operador **NOT** va antes de una condición, entonces toda la expresión será verdadera si deja de cumplirse la condición. Veámoslo con un ejemplo:

```
A = 10
Si NOT( A = 12 )
```

En la primera línea, estamos asignando el valor 10 a la variable A y en la segunda línea estamos preguntando que *si A no es igual a 12*, condición que es verdadera debido a que la variable A es igual a 10. Luego, cuando escribimos la siguiente condición

```
Si NOT ( A > B )
```

Es como si hubiéramos escrito

```
Si ( A < = B )
```

O sea, dicho textualmente, ¿cuándo A no es mayor que B...? Pues cuando es menor o igual a B.

Con estos tres operadores booleanos podemos construir una gran cantidad de decisiones y permitir que el computador las ejecute correctamente. Cabe anotar que evaluar una decisión y determinar si es verdadera o falsa es lo que más tiempo le toma a un computador, cuando dicha decisión está implementada en un lenguaje de programación.

Séptima norma. Se utilizará como estructura de ciclo básica la siguiente:

```
Mientras Condición Haga
        .....
        .....
        .....
        Cuerpo del ciclo
        .....
        .....
Fin_Mientras
```

Algunos libros escriben *Mientras que* en vez de *Mientras* solo, pero esas minucias no son de importancia pues obedecen más a capricho de autores que a alguna razón de fondo frente a la lógica. En el cuerpo del ciclo se colocan las órdenes que se van a repetir (o iterar) mientras la condición sea *Verdadera*. El *Fin_Mientras* le indicará, posteriormente, hasta dónde llega el bloque de instrucciones u órdenes y así determinar a partir de dónde se devuelve el control del algoritmo para evaluar la condición. La forma de ejecución de los ciclos se explicará apropiadamente en el capítulo de ciclos, así como las otras formas referenciales que existen para expresar ciclos.

Octava norma. Cada vez que usted vaya a utilizar un conjunto de instrucciones, deberá indicar claramente en donde comienza y en donde termina utilizando apropiadamente las palabras *Inicio* y *Fin*.

Tal vez usted estará esperando que escribamos los dos ejemplos iniciales en la forma de seudocódigo. Efectivamente lo voy a hacer a continuación, pero espero que usted vea que en algoritmos informales la utilización del seudocódigo no es muy práctica ya que no existe mucha diferencia entre el algoritmo como tal y su respectivo equivalente en seudocódigo (excepto algunos detalles que son mínimos comparados con las otras técnicas).

Verá una gran utilidad cuando estemos escribiendo algoritmos computacionales utilizando esta técnica. Por ahora, la versión en seudocódigo que le podría brindar acerca de los dos algoritmos es la siguiente: recuerde que el objetivo era realizar un algoritmo que nos permitiera vigilar por una ventana asomándonos por ella cada media hora, por lo tanto, la versión de este algoritmo en seudocódigo es la siguiente:

```
Algoritmo para Vigilar desde una ventana
Inicio
    Llegar puntual a la hora de inicio de la jornada laboral
        Ubicarnos en nuestro escritorio
        Mientras no sea fin del día
            Ubicar la ventana por la que nos queremos asomar
            Si estamos sentados
                Levantarnos del lugar en donde estemos sentados
                Orientarnos hacia la ventana
            Sino
                Orientarnos hacia la ventana
            Avanzar hacia la ventana
            Llegar hasta tener la ventana muy muy cerquita
            Si está cerrada
                Abrirla
            Asomarnos por la ventana
            Regresar a nuestro escritorio
            Mientras no haya pasado Media Hora
                Permanecer en nuestro escritorio
            Fin_Mientras
        Fin_Mientras
Fin
```

Y para el segundo algoritmo, en donde el objetivo era inspeccionar las camisas en un almacén de ropa emitiendo nuestro concepto acerca de su calidad, la solución es la siguiente:

```
Algoritmo para Inspeccionar las camisas en un almacén de ropa
Inicio
        Llegar puntuales al inicio de la jornada laboral
        Mientras no sea fin de la jornada laboral
                Dirigirnos a un ropero
                Si está cerrado
                        Abrirlo
                Tomar una camisa
                Si está abrochada
                        Desabrocharla
                        Abrir la camisa
                Si está doblada
                        Desdoblarla
                Meter un brazo por una de sus mangas
                Meter el otro brazo por la otra de sus mangas
                Ajustar la camisa al tronco
                Si es una camisa de botones
                        Abotonarla (botón a botón)
                        Ajustarla al cuerpo
                Sino
                        Ajustarla de manera que quede bien puesta
                Emitir el concepto de calidad sobre la camisa
        Fin_Mientras
Fin
```

¿Qué hay de diferente? Pues nada, porque en este tipo de algoritmos los seudocódigos (como le dije en un párrafo anterior) son de muy poca utilidad. No se olvide que la técnica de los seudocódigos está diseñada fundamentalmente para ser utilizada en algoritmos computacionales.

Es por eso que, en este momento, tal vez quede en su mente una nebulosa de dudas acerca de la utilidad de la técnica en sí, pero no hay nada de qué preocuparse, pues precisamente la utilización eficiente de esta técnica será unos de los objetivos fundamentales de este libro para que usted, a través de ella, pueda expresar cualquier algoritmo computacional y obtenga una solución que luego sea fácilmente codificable en cualquier lenguaje de programación.

4.3.4. Cuadro comparativo

Nombre de la técnica	Ventajas	Desventajas
Diagrama de flujo	a. Permite visualizar gráficamente el camino que sigue la solución a un problema. b. Por ser tan simplificado, es muy entendible. c. No se necesitan muchos conocimientos técnicos para utilizar esta técnica.	a. Dado que los flujos (representados con flechas) pueden ir de cualquier lugar a cualquier lugar, da espacio para que el diagrama llegue a ser casi inentendible. b. Deben conocerse bien los símbolos que se van a utilizar. c. No todos los símbolos están estandarizados. d. Los ciclos deben ser reinterpretados para poder ser diagramados en esta técnica. e. No siempre es muy entendible. f. Algunas veces, la analogía entre el diagrama y la codificación en el lenguaje de programación resulta ser compleja.
Diagramación rectangular estructurada	a. Permite tener un marco referencial concreto y definido para la representación de los algoritmos. b. Solo tiene tres esquemas que le permiten a su vez representar las tres estructuras básicas. c. Exige orden en la representación de un algoritmo. d. Es muy entendible. e. La analogía entre la codificación y el diagrama normalmente es directa y, por lo tanto, muy sencilla.	a. Exige una fundamentación técnica que permita representar la solución a cualquier problema a través de las tres estructuras básicas. b. No una técnica muy popularizada.
Seudocódigo	a. Permite expresar la solución algorítmica a un problema en nuestro propio lenguaje y casi con nuestras propias reglas. b. La codificación se facilita demasiado, dado que la transcripción es directa. c. Si el programador es ordenado, esta puede llegar a ser la técnica más entendible.	a. Exige mucho orden para ser utilizada eficientemente. b. Exige el mantenimiento claro de los conceptos de algoritmos como tales. c. Las decisiones deben estar encasilladas dentro de los alcances de los operadores lógicos y operadores booleanos.

Aún a pesar de que algunos libros y algunos profesionales de la programación aceptan única y exclusivamente la técnica de los seudocódigos, mi concepto personal es que cada una de estas técnicas tiene unas ventajas y unas

desventajas que las hacen comparables con las demás. Considero de suprema importancia darle prioridad de uso a la técnica que facilite la codificación, ya que el computador no ejecutará los algoritmos escritos en estas técnicas, sino escritos en términos de un lenguaje de programación y puedo garantizar que la técnica que más facilita la transcripción es el seudocódigo, sin desconocer las ventajas de cada una de las otras técnicas.

4.4. Ejercicios

Utilizando las técnicas explicadas, REPRESENTAR los siguientes algoritmos:

1. Desarrollar un algoritmo que permita adquirir una revista.
2. Desarrollar un algoritmo que permita entrar a una casa que está con llave.
3. Desarrollar un algoritmo que permita dar un beso.
4. Desarrollar un algoritmo que permita empacar un regalo.
5. Desarrollar un algoritmo que permita encender un vehículo.
6. Desarrollar un algoritmo que permita fritar un huevo.
7. Desarrollar un algoritmo que permita mirar por un telescopio.
8. Desarrollar un algoritmo que permita botar la basura.
9. Desarrollar un algoritmo que permita tomar un baño.
10. Desarrollar un algoritmo que permita estudiar para un examen.
11. Desarrollar un algoritmo que permita tocar determinada canción con un instrumento musical.
12. Desarrollar un algoritmo que permita viajar en avión.
13. Desarrollar un algoritmo que permita encender un bombillo.
14. Desarrollar un algoritmo que permita encender una vela.
15. Desarrollar un algoritmo que permita apagar una vela.
16. Desarrollar un algoritmo que permita apagar un bombillo.
17. Desarrollar un algoritmo que permita parquear un vehículo.
18. Desarrollar un algoritmo que permita almorzar.
19. Desarrollar un algoritmo que permita ir de la casa al trabajo.
20. Desarrollar un algoritmo que permita colocarse una camisa.
21. Desarrollar un algoritmo que permita quitarse una camisa.
22. Desarrollar un algoritmo que permita escuchar un determinado disco.
23. Desarrollar un algoritmo que permita abrir una ventana.
24. Desarrollar un algoritmo que permita ir a la tienda a comprar algo.

25. Desarrollar un algoritmo que permita tomar una fotografía.
26. Desarrollar un algoritmo que permita hacer deporte.
27. Desarrollar un algoritmo que permita cortarse el cabello.
28. Desarrollar un algoritmo que permita hacer un avión con una hoja de papel.
29. Desarrollar un algoritmo que permita manejar una bicicleta.
30. Desarrollar un algoritmo que permita manejar una motocicleta.
31. Desarrollar un algoritmo que permita manejar un monociclo.
32. Desarrollar un algoritmo que permita maquillarse.
33. Desarrollar un algoritmo que permita hacer un pastel.
34. Desarrollar un algoritmo que permita hacer un almuerzo.
35. Desarrollar un algoritmo que permita adquirir un pantalón.
36. Desarrollar un algoritmo que permita hacer un mercado pequeño.
37. Desarrollar un algoritmo que permita leer el periódico.
38. Desarrollar un algoritmo que permita saludar a un amigo.
39. Desarrollar un algoritmo que permita arrullar a un bebé hasta que se duerma.
40. Desarrollar un algoritmo que permita hacer un gol en fútbol.
41. Desarrollar un algoritmo que permita jugar ping-pong.
42. Desarrollar un algoritmo que permita nadar.
43. Desarrollar un algoritmo que permita tirarse desde un avión con un paracaídas.
44. Desarrollar un algoritmo que permita tirarse desde un avión sin un paracaídas.
45. Desarrollar un algoritmo que permita descifrar un jeroglífico.
46. Desarrollar un algoritmo que permita amarrarse un zapato.
47. Desarrollar un algoritmo que permita quitarse los zapatos.
48. Desarrollar un algoritmo que permita silbar.
49. Desarrollar un algoritmo que permita elevar una cometa.
50. Desarrollar un algoritmo que permita desarrollar algoritmos.

Capítulo 5

La tecnología

Un mundo como el moderno, en donde casi todo se concibe a la luz de la tecnología, en donde el hombre pasa a ser una presa fácil de las grandes fábricas desde donde lo parasitan argumentándole comodidad pero estableciéndole reglas en su vida social, reglas basadas precisamente en la misma tecnología, es un mundo en el cual se hace imposible no hablar de ella en cualquier libro técnico.

Sería pues imposible ignorar la tecnología cuando lo que se busca al hablar de algoritmos computacionales es aprovecharla para poder lograr, de la manera más eficiente, los objetivos que se hayan propuesto. El computador como herramienta tecnológica nos brinda su velocidad para que, en unión con nuestros algoritmos, se puedan obtener velozmente resultados que, de otra forma, tomarían *muchísimo* tiempo, sí, así como está escrito, *muchísimo* tiempo. Encontrar un dispositivo o un aparato que puede trabajar en términos de millonésimas de segundo es muy útil y, mucho más, si podemos aprovechar dicha velocidad para nuestra conveniencia.

Ya en su momento, explicábamos que, luego de que se ha concebido apropiadamente un algoritmo, es decir, luego de que hemos comprobado a través de una "prueba de escritorio" que el algoritmo realmente sí nos permite alcanzar el objetivo propuesto, entonces pasamos a una segunda etapa que corresponde a la intervención de la máquina en la solución del problema inicial. Esta etapa se inicia con la transcripción, que no es otra cosa que reescribir nuestro algoritmo pero en términos de un determinado lenguaje de programación. Pues bien, el lenguaje de programación lo podemos definir como ese puente perfecto que permite que el computador ejecute lo que nosotros habíamos concebido como un algoritmo (y además que lo haga a altas velocidades).

Un lenguaje de programación es, técnicamente hablando, un conjunto de instrucciones que son entendibles y ejecutables por un computador. No

podemos esperar, por lo menos no por ahora, que el computador ejecute lo que nosotros concebimos como algoritmo (aunque sería lo óptimo) y por eso debemos incorporar al desarrollo técnico un paso o un conjunto de pasos más y son lo que involucran los lenguajes de programación.

Esto significa que si nosotros hemos desarrollado un algoritmo computacional que es muy útil, de nada nos va a servir si no conocemos las reglas sintácticas de un lenguaje de programación, si no escribimos el algoritmo con esas reglas, si no contamos con el compilador del lenguaje en el cual vamos a trabajar y si no sabemos interpretar los errores. Por lo tanto, el contacto técnico que vamos a tener con el computador va mucho más allá de desarrollar solo el algoritmo, ya que lo ideal es hacer realidad el algoritmo a través del lenguaje de programación.

5.1. Lenguajes de bajo nivel

Son aquellos lenguajes en los cuales sus instrucciones son para nosotros complejas de entender pero que son extremadamente sencillas para ser entendidas por el computador. Tal es el caso del Lenguaje Assembler, según el cual las instrucciones están basadas en una serie de mnemónicos que no siempre facilitan la transcripción. Este lenguaje es el lenguaje verdadero de los computadores. Esto quiere decir que, internamente, lo único que entienden los computadores realmente son instrucciones del Lenguaje Assembler.

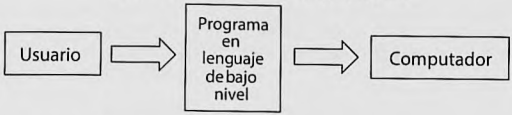
La programación a bajo nivel fue la que se utilizó en los primeros años de la Historia de la Programación. Cada vez más, los programadores tenían que ser mucho más especializados dada la complejidad de entendimiento de las instrucciones utilizadas en este lenguaje. No se podía desconocer el inmenso poderío de los lenguajes de bajo nivel, pero tampoco se podía desconocer la complejidad de su estructura.

Si usted encuentra una instrucción como la siguiente

mov bp, sp

No es muy fácil deducir qué hace o para qué le serviría en el desarrollo de un programa. Sin embargo, puede tener la seguridad de que esta instrucción es perfectamente clara para el computador, ya que su interpretación es inmediata. El esquema 1 muestra la situación explicada.

Esquema 1. Relación usuario-lenguaje de bajo nivel



5.2. Lenguajes de alto nivel

Precisamente, pensando en estas desventajas de la programación a bajo nivel y sabiendo que quien realmente desarrollaba los programas era el ser humano y no la máquina y que la máquina gracias a su velocidad era solo la encargada de ejecutar la orden que se le diera, se pensó en crear unos lenguajes de programación que fueran más entendibles al ser humano, o sea, unos lenguajes de programación en donde las órdenes fueran tan sencillas de comprender que programar se convirtiera en algo realmente fácil.

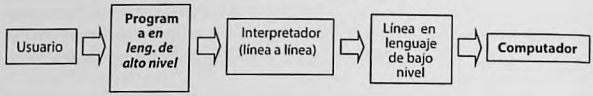
Estos lenguajes fueron llamados lenguajes de alto nivel, sabiendo que de todas maneras el computador necesitaría otro programa que tomara lo que se escribiera y lo convirtiera a lenguaje de bajo nivel (ya que no podemos olvidar que este es el verdadero lenguaje del computador). De allí surgió la idea de los *interpretadores* y los *compiladores*.

5.2.1. Lenguajes interpretados

Son aquellos lenguajes de programación en donde existe un programa interpretador, que es un programa que "toma" nuestro programa y lo convierte, línea a línea, a lenguaje de bajo nivel y, así mismo, lo va ejecutando (o sea, línea a línea). Estos lenguajes tenían como inconveniente que, si el programa tenía un error en una de las últimas líneas, solo se detectaba dicho error cuando el interpretador llegaba allí, luego de haberse ya ejecutado (posiblemente) un gran bloque de instrucciones.

Sin embargo, y para los inicios de la programación a través de lenguajes de alto nivel, esta era la solución precisa para poder programar con lenguajes más entendibles al ser humano que al computador. De esta manera, la utilización de lenguajes interpretados tomó mucha popularidad por las grandes facilidades que le brindaban al programador, que ya no tenía que invertir la mayor parte de su tiempo en especializarse en entender instrucciones de alguna manera complejas. El esquema 2 presenta la situación.

Esquema 2. Relación usuario–lenguaje interpretado



Se incorpora pues un elemento adicional como es el interpretador (línea a línea) que le facilita el trabajo al programador, pues este ya puede utilizar instrucciones más entendibles como ***print*** que le representará *Imprimir* y que con solo leerla da una idea de qué hace la instrucción. Esa es precisamente la diferencia exacta entre las instrucciones de un lenguaje de bajo nivel y un lenguaje de alto nivel y radica, dicho de una manera sencilla, en el hecho de que las instrucciones son más entendibles, los programas y el proceso de transcripción son más sencillos y menos exigentes técnicamente.

Sin embargo, la dificultad presentada por los lenguajes interpretados, en cuanto al hecho de que convertían línea a línea el programa y así lo ejecutaban, estableció el riesgo de que en programas comerciales, en donde es fácil encontrar 15000 o 20000 líneas en un programa, era demasiado costoso a nivel empresarial y demasiado riesgoso a nivel técnico saber que, en cualquier momento, por un error determinado, el programa podía ser interrumpido (o "abortado", como técnicamente se le dice) debido a que el interpretador iba revisando línea a línea. Esto llevó a pensar en un esquema mucho más práctico y menos arriesgado tanto para programadores como para empresarios.

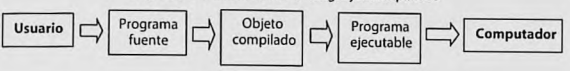
5.2.2. Lenguajes compilados

Son aquellos lenguajes en donde un programa llamado compilador toma TODO el programa que hemos escrito (que normalmente se denomina *programa fuente*), lo revisa y solo hasta donde esté completamente bien, solo hasta allí, lo convierte a su equivalente en lenguaje de bajo nivel para ser ejecutado por el computador. De esta manera, se reducía el riesgo de evaluación que representaban los lenguajes compilados y se podía saber si TODAS las instrucciones del programa eran correctas. De hecho, era de suponer que, bajo condiciones normales, el programa entonces no se abortaría cuando estuviera en su momento de ejecución (cosa que no es 100% cierta). Lo que sí se puede decir es que la cantidad de interrupciones de los programas elaborados con lenguajes compilados se redujo notoriamente frente a las interrupciones que sufrían los programas elaborados con lenguajes interpretados.

De allí que, cuando se compila el programa fuente, se genera un segundo programa (que solo es entendible por el computador) conocido como *objeto*

compilado y cuando se va a ejecutar, entonces este programa es "organizado" de manera que cada instrucción sea perfectamente entendible y ejecutable por el computador. A este nuevo programa se le conoce como *programa ejecutable*. De esta manera, podemos resumir el proceso de utilización de los lenguajes compilados según el esquema 3.

Esquema 3. Relación usuario–lenguajes compilados



De aquí surgen entonces los pasos que inicialmente se habían explicado en cuanto a la metodología para la solución de un problema cuando este se trata de un algoritmo computacional. Ya podemos también concluir que un lenguaje de programación está diseñado para facilitarnos el trabajo de hacer que el computador haga algunas cosas por nosotros. Entonces se pregunta uno: ¿por qué existen tantos lenguajes de programación? La respuesta es muy sencilla y se puede plantear bajo dos criterios:

La primera razón es estrictamente comercial, ya que detrás de cada lenguaje de programación existe una gran empresa vendiéndole al mercado informático su producto (que es el compilador y sus utilidades). Muchas empresas se han dedicado al desarrollo de lenguajes de programación (entre otros productos) y son en la actualidad una de las empresas más grandes y más rentables del mundo. El ánimo competitivo a nivel empresarial en el mundo informático, y más exactamente en lo que se refiere al desarrollo de lenguajes de programación, es lo que ha hecho que exista una gran cantidad de lenguajes de programación.

La segunda razón es más técnica y menos comercial, ya que no se puede desconocer que los lenguajes de programación también se han desarrollado para lograr de una manera más fácil y eficiente unos objetivos específicos. De esta manera, algunas empresas se han especializado en colocar a disposición de los programadores un conjunto de instrucciones que les permitan lograr ciertos objetivos concretos de una forma muy sencilla y sin tanto rigor técnico.

5.3. Errores en un programa

Cuando se dice que un compilador revisa un programa, se está dando una definición muy amplia de lo que realmente es esa revisión. Primero que nada, vamos a revisar los dos tipos de errores que puede tener un programa. Ellos son: los errores humanos y los errores detectables por un compilador.

5.3.1. Errores humanos

Son todos aquellos errores que dependen exclusivamente de la participación del ser humano en el proceso de escritura de un programa (a partir de un algoritmo), obviamente. Es evidente que son errores que no son detectados, en estos tiempos, por un compilador, ya que dependen exclusivamente de la visión, práctica, experiencia y conocimientos que tenga un programador. Los errores humanos son de tres tipos:

5.3.2. Errores de concepción

Este es el tipo de error que se presenta cuando el programador cree que tiene el objetivo claramente identificado y entendido y resulta que no es así (pero él no lo sabe), con lo cual es evidente que, finalizado el programa, seguramente habrá logrado algo totalmente distinto a lo que inicialmente necesitaba lograr.

Por ejemplo, un programador quiere implementar en un computador un programa que realice cierto tipo de liquidación de un estado de cuentas de un cliente. Averigua y se entera de que, en la empresa para la cual trabaja, la liquidación de los clientes se hace de una manera diferente a la que hacen las demás empresas, ya que en esta se tiene en cuenta el tiempo de vinculación con la empresa para reducir su deuda final. El programador investiga y creyó entender cómo realmente lo hacían cuando en realidad no era así, pues no captó la utilización de un factor que de manera autocrática es establecido por el criterio del gerente. El resultado final es una liquidación que, a pesar de ser muy detallada y muy completa, no coincide con lo que realmente se necesitaba en la empresa.

Pareciera ser muy poco frecuente este error y resulta ser lo contrario, pues yo le pregunto lo siguiente: *¿quién le garantiza a usted que lo que usted cree que entendió es porque realmente lo entendió...?* Pues nadie, porque no es fácil saber si alguien captó realmente lo que le quisimos decir. En este sentido, es importante que se tenga en cuenta que debemos buscar muchos métodos para demostrar, sobre todo a nosotros mismos, que sí entendimos lo que queríamos entender.

Para el caso de ese programador del ejemplo, ¿qué debió hacer él antes de empezar a programar? Pues debió haber solicitado los datos completos de varios clientes y haber realizado MANUALMENTE la liquidación. Solo de esta manera hubiera sabido que no le era claro aquel factor que colocaba el gerente o, por lo menos, hubiera encontrado un camino para aproximarse a dicho factor. De manera que este error podemos decir que es el más difícil de detectar si no establecemos los mecanismos para ello, sobre todo porque el computador en ninguno de sus procedimientos va a detectarlos.

5.3.3. Errores lógicos

Son los errores que se presentan cuando no se ha comprobado apropiadamente a través de la "prueba de escritorio" la efectividad de un algoritmo. Estos errores solo son detectables a través de dicha prueba; el computador nunca nos va a hacer recomendaciones dentro del campo de la lógica. Por eso es tan importante que sepamos que, luego de reconocido el objetivo y de desarrollado un algoritmo, es imprescindible realizarle una prueba de escritorio. Cuando esta no se realiza, es muy posible (extremadamente posible) que nuestro algoritmo no satisfaga el objetivo propuesto.

Por ejemplo, en una empresa en donde primero se realiza un determinado descuento sobre el valor bruto de una compra y luego se liquida la factura para el cliente, será importante que solo en ese orden el programador de turno lo realice porque si primero liquida la factura del cliente y luego le aplica el descuento, los resultados podrán ser (y lo serán) completamente diferentes a los esperados y, en ese momento, el computador no le va a decir qué fue lo que hizo malo o lo que dejó de hacer.

5.3.4. Errores de procedimiento

Son los errores que se presentan cuando se tiene claro el objetivo y se ha desarrollado un algoritmo aproximadamente bien pero no se realiza bien la "prueba de escritorio", es decir, se hace la prueba de escritorio pero se hace mal y no nos damos cuenta y, además de ello, quedamos convencidos que el algoritmo quedó bien. Por eso no solo es importante que se realice la prueba de escritorio para evitar los errores lógicos, sino que además se realice bien, apegada a las instrucciones. Recuerde que lo único que necesita para realizar una buena prueba de escritorio es que usted, mientras la esté realizando, no "razone", o sea, que actúe como si usted fuera un computador (que no piensa) y de manera autómata realice y obedezca una a una las órdenes establecidas en el algoritmo. Esa será la única forma para usted de estar seguro de que realizó bien la prueba de escritorio porque este es otro de los factores que no es fácil de detectar, ya que muchas veces podemos hacerlas y no tener la certeza de si lo hicimos bien o no.

Sin temor a equivocarme, puedo decirle que son los errores del ser humano los más difíciles de detectar en el desarrollo de un algoritmo, dado que dependen solo de la metodología y del orden que tenga, para solucionar el problema, el mismo ser humano. También podríamos decir que dichos errores son los más complejos porque su solución se remite a cambiar una concepción acerca de un problema o incluso a realizar algunos ajustes a la lógica de programación

para acercarnos más a la lógica computacional y abandonar un poco la lógica humana. Los errores humanos son los más difíciles porque ningún aparato nos dice en dónde están. Tenga pues en cuenta que solo de usted va a depender que no tenga que cometer estos errores, ya que al final resultan ser los más costosos si se analizan desde una óptica estrictamente empresarial.

5.3.5. Errores detectados por un compilador

Son los errores más sencillos, ya que los compiladores modernos no solo nos dicen cuál es el error, sino que además nos orientan en dónde puede estar dicho error. Es muy importante que sepa que, en cuanto a los errores detectados por un compilador, usted deberá saber dos cosas:

¿Qué significan? Debido a que normalmente los compiladores son desarrollados fuera del país y los errores salen como un aviso en inglés, saber qué significan simplemente es conocer su traducción literal y no más.

¿Qué representan? Esto es lo más importante en un aviso de error generado por un computador, ya que la representación de un error es lo que realmente nos va a permitir corregirlo apropiadamente. Tomemos un breve ejemplo: usted ha realizado un programa en el que ha utilizado variables de las cuales se le olvidó declarar una variable que dentro del programa usted ha llamado *conta1*. En algún momento del programa, usted asigna el valor 35 a dicha variable. Cuando lo compila le sale el siguiente error:

Impossible assign value in variable conta1

¿Qué significa? Imposible asignar valor en la variable *conta1* (no olvide que el significado es la traducción literal).

¿Qué representa? Que el computador no puede asignar el valor porque la variable en mención (o sea, *conta1*) no está declarada y, por lo tanto, para él no existe.

La familiarización con el significado y la representación de los errores de un determinado lenguaje de programación se da solamente en la medida en que usted codifique muchos programas en dicho lenguaje. Este puede ser un buen termómetro para determinar si una persona realmente tiene experiencia en un determinado lenguaje de programación o no.

Pasemos pues a describir los dos tipos de errores que genera un compilador cuando está "revisando sintácticamente" un programa. Estos errores son:

5.3.6. Errores de sintaxis

Son las omisiones que cometemos cuando transcribimos el programa. Es normal que se nos olvide cerrar un paréntesis que hemos abierto o declarar una variable. El compilador nos dice, a través de un aviso, cuál es el error y nos ubica aproximadamente en donde está. Cabe anotar que estos errores no dejan que el programa sea ejecutado.

5.3.7. Errores de precaución

Son recomendaciones técnicas que el compilador nos hace para que el computador nos garantice el correcto funcionamiento del programa. Podría decirse que no son errores como tales, pues el programa puede ejecutarse aun a pesar de tener errores de precaución, con la pequeña diferencia de que es muy posible que los resultados finales no sean los que esperamos.

5.4. Desarrollo histórico de la programación

Como todas las ramas del conocimiento humano, la programación también ha ido avanzando, haciendo que esta sea cada vez más simplificada para el programador y brindando día a día más herramientas técnicas que permitan la utilización de los computadores de una manera sencilla y simplificada. También se han experimentado algunos cambios en cuanto a la concepción del mundo de la programación y sus efectos y utilización en el mundo de la informática.

Cuando comienzan los computadores o lo que en esos tiempos era un computador, no existían, como es obvio, las facilidades tecnológicas que hoy existen, razón por la cual el concepto de programación se realizaba a través de circuitos eléctricos y electrónicos directamente, de tal manera que los "programadores" de aquellos tiempos eran unos tremendos en electrónica, ya que las soluciones las construían con partes como tales. Esa programación se conoció como ***programación directa o real***, dado que el contacto entre el programador y la máquina era directo y requería un altísimo conocimiento técnico no solo de partes electrónicas, sino también de lo que en ese entonces era la programación a bajo nivel.

Poco a poco, la tecnología fue avanzando, permitiendo que, en este campo, el ser humano tuviera cada vez más y mejores herramientas de trabajo. Fue entonces cuando se pensó en la programación tal y como se concibe en el día de hoy, es decir, permitir que, a través de órdenes dadas al computador, este pudiera realizar unas tareas a altas velocidades. Este concepto comenzó

a ser trabajado y poco a poco empezaron a surgir en el mercado lenguajes de programación como tales. Estos lenguajes permitían realizar un gran número de tareas con la simple utilización correcta de unas instrucciones. La metodología de la utilización de estas instrucciones fue en esos tiempos algo muy libre, razón por la cual a esta etapa de la programación se le conoció como ***programación libre***.

Bajo esta técnica de programación, la persona que estuviera al frente del computador podía realizar todas las tareas que pudiera o, más bien, que el lenguaje le permitiera, basándose solamente en su lógica propia aplicada a la libre utilización de dichas instrucciones. Aun a pesar de que en principio esta forma de utilizar los lenguajes de programación fue la solución para muchos problemas y de que el mundo había comenzado a ser más eficiente en el tratamiento de la información gracias precisamente a la utilización de órdenes para programar los computadores, los problemas no esperaron para dejarse venir.

Cuando un programador se sentaba con su lógica propia a resolver un problema utilizando las instrucciones que un lenguaje de programación le permitía, muchas veces (y casi siempre) llegaba a soluciones que solamente él entendía y cuando este programador era sacado de la empresa o se retiraba o se moría, entonces la empresa se veía en la penosa obligación de conseguir otro programador que, en la mayoría de los casos, lo que hacía era volver a hacer todo lo que el primero había hecho pero con su propia lógica, quedando la empresa en manos de este nuevo programador y teniendo previsto el gran problema que se originaría cuando este se retirara o se muriera o hubiera que echarlo.

Fue allí en donde comenzó a pensarse en la lógica estructurada de programación o más bien se comenzó a pensar que los programas, por diferentes que fueran, obedecían a una serie de normas que eran comunes en cualquier algoritmo, término que se comienza a acuñar en esa época. A través de muchos estudios, se llegó a la conclusión de que la lógica de programación se basaba solo en tres estructuras, como son las secuencias, las decisiones y los ciclos. Se puso a prueba esta teoría y se descubrió que era cierta, pues ningún algoritmo se salía de estas tres estructuras.

Pensar en unas estructuras básicas del pensamiento al momento de la programación facilitó enormemente el hecho de que el programa desarrollado por un programador fuera entendido sin mayores complicaciones por otro. También esta forma de programación restringió el desorden de algunos programadores porque le colocó unos límites a la lógica computacional que era la que había que utilizar cuando se necesitara escribir un programa. A esta forma de trabajo se le llamó ***programación estructurada***, que no es más que la técnica a través de la cual se utilizan los lenguajes de programación utilizando

las estructuras básicas y permitiendo que los programas sean mucho más entendibles, ya que no son concebidos al libre albedrío del programador, sino basados en unas normas técnicas.

Esta técnica de programación comenzó a tomar mucha fuerza en el desarrollo de la programación debido, precisamente, a que ya un programador podía tomar los programas de otro y entenderlos con muchísima facilidad. Se desarrollaron lenguajes que permitieran precisamente la sana utilización de estas estructuras y a estos se les llamó lenguajes estructurados. Además, dichos lenguajes se podría decir que casi obligaban al programador a no salirse del marco conceptual de las estructuras básicas.

El mundo y el ser humano, ávido de soluciones para sus necesidades, utilizaron esta técnica de programación estructurada por mucho tiempo sin cuestionarla hasta que las mismas necesidades de programación comenzaron a cuestionar lo que hasta ese momento había funcionado tan perfectamente. Se partió de la teoría de que la programación no es más que una interpretación del mundo real y su simulación a través de un computador; por tal motivo, se pensó en aproximar mucho más los conceptos de programación al mundo real y fue allí en donde se encontró que todo lo que nos rodea tiene unas características y sirve para algo. Por ejemplo, un lápiz tiene peso, color, olor, sabor (si se quiere), longitud, espesor, torque, textura y muchas otras características. Al mismo tiempo, un lápiz sirve para escribir, para separar una hoja de un libro, para rascarse la espalda, para defenderse de un atraco, para señalar un punto, para dibujar, para manchar a alguien y para miles de cosas más.

Esta concepción llevó a una gran revolución en la historia de la programación, pues se crearon dos vertientes dentro de la lógica de programación: la programación estructurada que ya definimos y la ***programación orientada a objetos***, por medio de la cual se podía modelar el mundo en el computador tal y como es. Su aporte principal era el concepto de objeto. ¿Qué es, pues, un objeto? En términos generales, un objeto no es más que un ente informático que tiene características (técnicamente llamadas *atributos*) y que sirve para algo (técnicamente se dice que tiene unos *métodos* asociados).

Así se creó pues este concepto y se comenzarían a utilizar los objetos (en programación), que como ya dijimos no son más que tipos de datos con atributos y métodos propios. Toda una teoría se comenzó a derivar de esta nueva concepción del mundo y se fue aplicando poco a poco en la programación, ya que se empezaron a descubrir unas relaciones entre objetos, unas operaciones entre objetos y, en general, un montón de conceptos nuevos en cuanto a lo que inicialmente no habían sido más que los objetos.

Mientras se desarrollaba esta teoría y se ponía en práctica en muchos de los lenguajes de programación comerciales, también se seguía utilizando la técnica de programación estructurada, pues estas dos técnicas no eran excluyentes. Dependía pues del programador que tuviera una verdadera concepción acerca del problema que quería solucionar y la decisión de saber por cuál técnica de programación (programación estructurada o programación orientada a objetos) era más apropiado resolverlo.

Ello exigía simultáneamente que el programador no solo conociera muy bien los conceptos de programación, sino que también conociera muy bien el problema que iba a solucionar y las características de cada una de las técnicas de programación. Tenía que ser un profesional integral de la programación, pues ahora no solo se necesitaba que supiera de computadores o de electrónica o que se supiera las instrucciones de un simple lenguaje. Ahora tenía que conocer teorías y combinarlas de manera que pudiera llegar a la mejor solución aprovechando la tecnología existente. Debo decir que, cuando comenzó a tomar fuerza la teoría de la programación orientada a objetos, no todos los lenguajes de programación (o mejor, no todos sus compiladores) estaban acondicionados para que aceptaran la nueva forma de programar.

De esta manera, también era necesario que el programador supiera si el problema que iba a solucionar a través de un programa era implementable fácilmente con el lenguaje de programación que tuviera a mano, pues debe usted saber que no es fácil inducir la compra de un lenguaje de programación (o sea, de su compilador) en una empresa cuando todo el sistema de información está basado en otro lenguaje de programación. Esta filosofía de programación fue tomando mucha fuerza y con ella se fueron fortaleciendo los lenguajes que habían iniciado la aceptación de esas nuevas características. Empresas fabricantes que hasta ese momento habían sido competitivas se convirtieron en verdaderos imperios de la informática. La programación, definitivamente, había dado un salto impresionante hacia la solución de muchos problemas que eran, en algunos casos, más complejos de resolver con programación estructurada que con programación orientada a objetos.

Poco a poco, algunos fabricantes de lenguajes de programación se fueron introduciendo en el mercado y aprovechando las características de la nueva técnica de programación y fueron dejando de lado, de alguna manera, la programación estructurada que algunos libros han llamado programación tradicional. En ese avance tecnológico y con el ánimo de entregar al mercado de la programación mejores herramientas de trabajo, se empezó a manejar un concepto muy importante en programación como es el concepto de **interfaz**. Una interfaz no es más que la forma como usted puede mostrar la información por medio de algún dispositivo de salida. Ya se sabía que cuanto más clara y entendible fuera la información podría decirse que los programas serían mejores, ya

que lo que finalmente el usuario de un programa necesitaba era que la información que le arrojaba un computador fuera claramente entendible.

Se fue notando, pues, por parte de las empresas fabricantes de lenguajes de computadores, que el tiempo de un programador se iba en su mayor parte en el diseño de las interfaces, o sea, en el diseño de la presentación de los datos. Por tal motivo, se pensó que, en unión con la teoría de programación orientada a objetos y con las herramientas que ella facilitaba, se hacía necesario diseñar lenguajes de programación que facilitaran el diseño de interfaces para que el programador invirtiera su tiempo mejor en el diseño de procesos o de manipulación y tratamiento de datos.

Fue entonces cuando entraron al mercado los lenguajes visuales y se incorporó al desarrollo de la programación la ***programación visual***, que no es más que una forma de programar en donde se cuenta con una gran cantidad de herramientas prediseñadas para facilitar, precisamente, el diseño de interfaces. Este tipo de programación ha llevado a que en el mundo de la informática y exactamente en la programación se llegue a unos resultados mucho más convenientes y mejores a nivel técnico, pues en la actualidad se pueden obtener aplicaciones de computador mucho más entendibles y manejables por el usuario gracias a la filosofía incorporada por la programación visual.

A este nivel, la programación requería menos conceptos técnicos y más lógica de programación, que era lo que realmente se necesitaba para desarrollar un programa. Es normal ver cómo una persona con unos modestos conocimientos de computación puede, a través de lenguajes visuales, desarrollar aplicaciones verdaderamente útiles y además muy bien presentadas. Lo que poco a poco se fue afianzando fue la necesidad de tener unos conceptos de lógica de programación bien fundamentados para poder aprovechar de una manera eficiente los recursos que la informática le entregaba al computador.

Como se busca modelar con el computador al mundo que nos rodea y en ese avance la tecnología cada vez se ha ido mejorando más y más, se espera que dentro de muy poco se podrá hablar de una ***programación virtual*** en donde el programador pueda ver en tres dimensiones (3D) todo el escenario que necesita para crear sus aplicaciones. Es muy posible que, cuando este libro esté en sus manos, algunos de estos lenguajes de programación ya estén en el mercado.

Capítulo 6

Metodología, técnica y tecnología para solucionar un problema computable

Hasta este momento tenemos una metodología para solucionar un problema, conocemos unas técnicas para representar la solución y hemos hablado de la tecnología a nivel de lenguajes de programación para que el computador cumpla por nosotros el objetivo propuesto. Todo esto se une en una teoría que nos permite acercarnos a la lógica de programación y, por supuesto, gracias al primer contacto que ya tenemos con los lenguajes, a la programación como tal. Es importante saber que cuando se habla de lógica de programación se está hablando de ese conjunto de normas técnicas que nos permiten que, de una manera sencilla, nosotros desarrollemos un algoritmo entendible para la solución de un problema. Cuando se habla de programación como tal, se habla de la utilización de lenguajes que permiten que nuestra solución sea entendida y ejecutada por un computador.

Precisamente, y con el ánimo de ilustrar toda la teoría que hasta el momento hemos visto, vamos a plantear tres enunciados y vamos a resolverlos aplicando la metodología para solucionar un problema, utilizando las técnicas de representación y codificándolos en unos lenguajes de programación.

6.1. Concepción del problema

Es muy importante que cuando tengamos un enunciado podamos tener una concepción acertada de él, de manera que podamos alcanzar su objetivo (solución) y que ese objetivo sea el que realmente necesita ser solucionado. La concepción del problema es el camino para tener la certeza de que lo hemos entendido correctamente y que lo que buscamos solucionar coincide con lo que se busca solucionar en el problema.

6.1.1. Clarificación del objetivo

Por lo dicho en capítulos anteriores, es muy importante que a través de un razonamiento teórico y textual nos sentemos a reflexionar en cuanto a los alcances de nuestro objetivo (enunciado como un problema), ya que con eso tendremos muy claro no solo hacia dónde debemos ir, sino hasta dónde debemos llegar.

6.1.2. Algoritmo

Es el conjunto de pasos que nos permiten llegar (ojalá de la mejor de las formas) a alcanzar el objetivo propuesto. Debe ser organizado y, ante todo, ordenado para que sea absolutamente entendible.

6.1.3. Prueba de escritorio

Es la prueba reina de un algoritmo. Nos permite saber si realmente está bien o no. ¿Cuándo un algoritmo está bien? Solamente cuando realmente alcanza el objetivo propuesto. Si un algoritmo no alcanza el objetivo que inicialmente se propuso, estará mal así haga maravillas en su desarrollo.

6.2. Técnicas de representación

Es importante que usted conozca y domine las técnicas de representación porque con ello usted podrá evaluar ventajas y desventajas reales (y para usted) y podrá determinar cuál es la técnica más apropiada para la representación de sus algoritmos. No está de más decir que cuando se quiere representar un algoritmo solamente se utiliza una de las técnicas, pero para los objetivos explicativos de este libro representaremos los algoritmos de este capítulo y de otros subsiguientes con las tres técnicas, solo para que usted encuentre diferencias entre ellos y esto le permita realizar una correcta evaluación y establecer unos criterios firmes acerca de su utilización.

6.2.1. Diagramas de flujo

Representados por signos en donde el hilo conductor de la lógica se basa en la utilización de flechas que conectan pequeños gráficos (con significado) que van a indicar la dirección del flujo de la idea.

6.2.2. Diagramación rectangular estructurada

Esquema en donde se utiliza un rectángulo como base y utilizando solo tres tipos de notaciones se puede representar todo lo que para nosotros sea parte de un algoritmo.

6.2.3. Seudocódigo

Texto basado en unas normas técnicas que lo hace muy entendible y, sobre todo, muy fácil de codificar y que representa, obviamente, la solución que hayamos planteado a través de un algoritmo.

6.3. Transcripción o codificación

Es la representación de un algoritmo a través de un lenguaje de programación. En este capítulo, utilizaremos los lenguajes *Basic*, *Pascal*, *C* y *Cobol* como ejemplos y explicaremos brevemente y de manera muy somera, ya que no es el objetivo del libro, algunos tópicos acerca de cada uno de los lenguajes. También es importante que usted sepa que, cuando vaya a desarrollar realmente programas aplicativos, solo va a tener que codificar en un solo lenguaje de programación. En este libro lo haremos en cuatro lenguajes solo por el ánimo explicativo del libro y para establecer algunas diferencias entre uno y otro lenguaje.

6.4. Primer enunciado

Desarrollar un programa que permita leer un número entero positivo y determinar si es par.

Concepción del problema

Clarificación del objetivo

Se trata de recibir un número entero (para lo cual utilizaremos una variable de tipo entero), verificar que es un número positivo y determinar si es un número par. Recordemos, pues, que son números pares aquellos que son divisibles exactamente entre dos, o sea, aquellos que al dividirlos entre 2 su residuo es cero. Algunos números pares son 18, 6, 4, 56 y 88. Algunos números que no son pares son 45, 7, 19, 23 y 99, ya que no cumplen con las condiciones de los números pares.

En caso de que el número leído sea par, avisaremos a través de un título que el *número sí es par* y, en caso de que no sea así, entonces haremos lo mismo avisando que el *número no es par*. Apenas hayamos avisado a través de un título que el número es par o que no lo es, entonces allí deberá terminar nuestro algoritmo.

Algoritmo

Algoritmo para determinar si un número es par
Inicio
    Leer un número y guardarlo en una variable entera
    Si ese número es negativo
        Escribir que ese número no sirve para nuestro propósito
    Sino
        Preguntar si el número es par
        Si lo es entonces escribir que el número leído es par
        Si no lo es escribir que el número leído no es par
Fin

Ya de por sí debemos tener en cuenta que el algoritmo es en sí la esencia de nuestra idea tal y como está representado aquí. Ahora lo que tenemos que hacer es ir mutando nuestra idea para que se convierta en una solución más aproximada a lo técnico que a lo informal. Comencemos pues con un análisis detallado de cada una de las órdenes que aparecen en este algoritmo:

Si vamos a convertir este algoritmo en un programa, entonces el nombre debe ser un poco más técnico. De esta manera, no lo vamos a llamar *Algoritmo para determinar si un número es par*, sino que lo vamos a llamar *Algoritmo Número_Par* y será nuestra obligación recordar que el *Algoritmo Número_Par* es el algoritmo que nos permite leer un número y determinar si es par.

Como vamos a tener la necesidad de utilizar una variable para que almacene el número que se va a leer, entonces es necesario que al inicio del algoritmo declaremos una variable de tipo entero a la cual vamos a llamar (por conveniencia técnica) *num*; de esta forma, la cabecera de nuestro algoritmo que estaba así:

Algoritmo para determinar si un número es par
Inicio

Se va a transformar en:

Algoritmo Número_Par
Variables
    Entero: *num*
Inicio

Esto significa que durante el algoritmo vamos a utilizar una variable que la vamos a llamar *num*, que solo podrá almacenar datos de tipo entero y que, cuando se utilice en operaciones, sus resultados se van a regir por las reglas de la aritmética entera (es decir, sin decimales).

Como ya tenemos una variable en donde vamos a almacenar el número que se lea, entonces la orden *Leer un número y guardarlo en una variable entera* se convertirá conceptualmente en *Leer un número y guardarlo en la variable entera num* que, por lo dicho en capítulos anteriores, es lo mismo que decir *Lea num* (orden dada al computador). Entonces, el algoritmo que, hasta el momento, era:

Algoritmo para determinar si un número es par
Inicio
    Leer un número y guardarlo en una variable entera

Se convierte ahora en:

Algoritmo Número_Par
Variables
    Entero : num
Inicio
    Lea num

Como ya tenemos el valor guardado en una variable, entonces preguntar por el número leído es lo mismo que preguntar por el contenido de la variable y hemos de recordar que, cuando se utiliza el nombre de una variable en un algoritmo, eso representará que nos estamos refiriendo al contenido de dicha variable. Igualmente, preguntar si un número es negativo se reduce a preguntar si dicho número es menor que 0 valiéndonos de un operador relacional (<). En esas condiciones la pregunta:

    Si ese número es negativo
        Escribir que ese número no sirve para nuestro propósito

Se convierte en:

Si num < 0
Escriba "El número debe ser positivo"

Y, por lo tanto, nuestro algoritmo que originalmente era:

Algoritmo para determinar si un número es par
Inicio
Leer un número y guardarlo en una variable entera
Si ese número es negativo
Escribir que ese número no sirve para nuestro propósito

Se ha transformado, hasta el momento, en:

Algoritmo Número_Par
Variables
Entero : num
Inicio
Lea num
Si num < 0
Escriba "El número debe ser positivo"

Si esta última pregunta es falsa, querrá decir que el número es mayor que o igual a 0 y, por lo tanto, pasaremos a realizar la siguiente pregunta:

Sino
Preguntar si el número es par
Si lo es entonces escribir que el número leído es par
Si no lo es escribir que el número leído no es par
Fin

Que consistirá en determinar si el número es par para avisar a través de un título que sí lo es o que no lo es. Pero, ¿cómo convertimos técnicamente la pregunta *Si el número es par* para que el computador la pueda ejecutar y obtener la respuesta apropiada? Una de las formas es aprovechando las características de la aritmética entera. Recuerde que en esta aritmética no se generan decimales, por lo tanto, si nosotros tomamos un número y lo dividimos entre dos, eso nos dará un resultado. Si ese resultado lo multiplicamos por dos, ¿nos volverá a dar el mismo número inicial? Sí, pero solamente cuando este haya sido par, ya que si hubiera sido impar al hacer la división entre dos se pierden sus decimales.

Vamos a hacerlo con un ejemplo: si dividimos 7 entre 2, ¿cuánto nos da? (Recuerda que son datos enteros y que estamos trabajando con aritmética entera por ser estos dos datos enteros). Pues el resultado es 3, ya que en aritmética entera no se generan decimales. Y si ahora tomamos este resultado y lo multiplicamos por 2, nos va a dar 6 que no es igual al 7 inicial, por lo tanto, podemos decir que como 6 no es igual a 7 entonces el 7 no es par.

Tal vez usted pensará que todos sabemos cuándo un número es par o no. Pero no se olvide que el que va a ejecutar este algoritmo (convertido obviamente en programa) es el computador y ese sí que no fue a la escuela como nosotros. Igualmente, y para continuar con nuestro ejemplo, si el número 8 es dividido entre 2, obtenemos el resultado 4 y si ese resultado es multiplicado por 2, obtenemos el 8 inicial. Por lo tanto, podemos decir que 8 es un número par.

Luego para determinar si un número cualquiera es par, todo lo que tenemos que hacer es dividirlo entre 2 y multiplicarlo por 2. Si al final se obtiene el resultado inicial, es porque el número es par. Si no se obtiene el resultado inicial, es porque el número no es par. De tal forma que nuestra pregunta:

```
Sino
        Preguntar si el número es par
        Si lo es entonces escribir que el número leído es par
        Si no lo es escribir que el número leído no es par
Fin
```

Se convierte en:

```
Sino
        Si num / 2 * 2 = num
                Escriba "El número leído es par"
        Sino
                Escriba "El número leído no es par"
Fin
```

Cuando se vaya a resolver la pregunta *Si num / 2 * 2 = num* no se olvide de la jerarquía de operadores para que el resultado sea el correcto y, por ende, la respuesta a dicha pregunta.

Entonces, nuestro algoritmo, que inicialmente era:

```
Algoritmo para determinar si un número es par
Inicio
        Leer un número y guardarlo en una variable entera
        Si ese número es negativo
                Escribir que ese número no sirve para nuestro propósito
        Sino
                Preguntar si el número es par
                Si lo es entonces escribir que el número leído es par
                Si no lo es escribir que el número leído no es par
Fin
```

Se ha convertido ahora en un algoritmo técnico así:

```
Algoritmo Número_Par
Variables
        Entero : num
Inicio
        Lea num
        Si num < 0
                Escriba "El número debe ser positivo"
Sino
                Si num / 2 * 2 = num
                        Escriba "El número leído es par"
                Sino
                        Escriba "El número leído no es par"
Fin
```

¿Cuál es la verdadera diferencia entre uno y otro? Pues la diferencia es que la segunda versión de este algoritmo es fácilmente codificable en un lenguaje de programación y la primera versión no es tan fácilmente codificable dada la gran cantidad de razonamientos que hay que hacer. Debo aclararle que la primera versión es el algoritmo puro como tal y sin ningún tipo de retoque. La segunda versión es el mismo algoritmo pero expresado bajo la técnica del seudocódigo que no es más que una representación técnica textual de un algoritmo.

Prueba de escritorio

¿Cómo se hace realmente una prueba de escritorio? Muy sencillo. Usted va a tener dos elementos que manejar en una prueba de escritorio: el primero es la memoria en donde se van a manejar las variables que intervengan en el programa y el segundo es la pantalla (o unidad de salida, cualquiera que esta sea) por

donde usted va a obtener los resultados de su algoritmo. Entonces, desarrolle paso a paso lo que diga el algoritmo utilizando las variables que el mismo le indique y colocando en pantalla los títulos que el mismo algoritmo le indique.

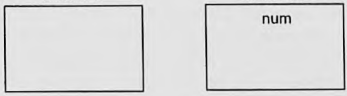
Sencillamente, suponga que usted es el computador. Cuando llegue al *Fin* del algoritmo, todo lo que tiene que hacer es mirar en la pantalla (o unidad de salida que usted haya representado) y ver si lo que dice allí coincide con el objetivo que inicialmente se había propuesto. De ser así, su algoritmo estará bien. Si no es así, el algoritmo estará mal y usted tendrá que corregirlo para volver a realizarle una prueba de escritorio.

De esta manera, si tomamos el algoritmo técnico final y le realizamos una prueba de escritorio, obtenemos lo siguiente:

```
Algoritmo Número_Par
Variables
        Entero : num
Inicio
        Lea num
        Si num < 0
                Escriba "El número debe ser positivo"
Sino
                Si num / 2 * 2 = num
                        Escriba "El número leído es par"
                Sino
                        Escriba "El número leído no es par"
Fin
```

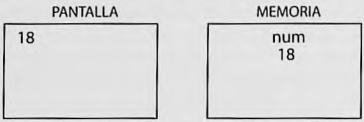
Tal como lo indica el algoritmo en su parte inicial en memoria, tenemos una variable que se llama *num* y en pantalla inicialmente estamos en blanco.

```
Algoritmo Número_Par
Variables
        Entero : num
```



A partir de este momento somos computadores. Solo vamos a hacer lo que el algoritmo nos diga... El algoritmo nos dice *Lea num*, entonces vamos a asumir

que el número 18 se recibe a través del teclado y se refleja automáticamente en la pantalla. Dicho número es almacenado en la variable *num* porque así se lo hemos dicho a través del mismo algoritmo (no se olvide que *Lea num* significa *Lea un número entero y guárdelo en la variable num*).

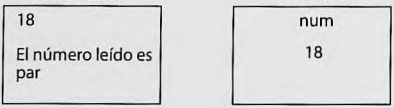


A continuación, sigue la pregunta *Si num < 0*. Como sabemos que cuando se refiera a la variable *num* realmente se está refiriendo al contenido de la variable *num*, entonces internamente la pregunta sería *Si 18 < 0*, a lo cual podemos responder (y así lo haría el computador) que es Falso. Por lo cual optamos por desarrollar las órdenes que se encuentran después del *Sino* (que son las órdenes que se deben ejecutar cuando la condición sea falsa, como en este caso). Nuestra situación tanto de pantalla como de memoria siguen iguales, no ha habido ningún cambio en ninguna de las dos. A continuación, y siguiendo con el *Sino*, nos encontramos con la pregunta *Si num / 2 * 2 = num*. Para ello, lo primero que debemos hacer es resolver la primera parte de la pregunta, o sea, realizar la operación que se derive de *num / 2 * 2* y obtener su resultado. Así, reemplazando *num* por su contenido y aplicando la jerarquía de operadores, obtenemos que la expresión y su resultado serían:

$$num / 2 * 2 = num$$
$$18 / 2 * 2 = 18$$
$$9 * 2 = 18$$
$$18 = 18$$

Con lo cual la pregunta inicial que era *Si num / 2 * 2 = num* se convierte en *Si 18 = 18*, a lo cual nuestra respuesta (como computadores que somos en este momento) es Verdadero; entonces procedemos a realizar la acción de escribir en pantalla *"El número leído es par"* tal como lo indica nuestro algoritmo.

Luego en nuestra pantalla aparece:



Verificamos finalmente lo que hay en pantalla y vemos que está el número *18* y después la frase *El número leído es par* que es Verdad referente al número 18. Como nuestro objetivo era desarrollar un programa que nos permitiera leer un número y determinar si era un número par, entonces podemos decir que este algoritmo SÍ cumple con el objetivo, o sea, que está bien.

Ahora usted deberá realizarle la prueba de escritorio a este algoritmo suponiendo que el número inicial leído es el 25. No se olvide de la jerarquía de los operadores y de que mientras usted esté haciendo una prueba solo debe acatar y ejecutar las órdenes tal y como se las indique el algoritmo, pues es así como el computador va a hacer realidad nuestros programas.

Técnicas de representación

Una vez desarrollado el algoritmo y habiéndosele realizado una correcta prueba de escritorio, se procede a representarlo usando alguna de las técnicas vistas. Hago hincapié en que son algunas porque para efectos de aprendizaje en este libro representaremos este algoritmo usando las tres técnicas, pero en la realidad se utiliza solamente una de ellas. Es muy importante que sepa que la representación se debe hacer cuando el algoritmo está escrito técnicamente. Para ello, recordemos que hemos, hasta el momento, desarrollado el mismo algoritmo de dos formas: la forma informal y la forma técnica. ¿Se acuerda cuál es la forma informal? Pues aquí se la presento de nuevo:

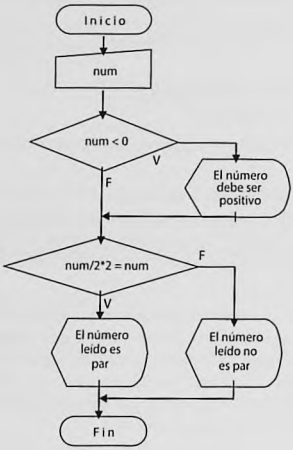
Algoritmo para determinar si un número es par
Inicio
    Leer un número y guardarlo en una variable entera
    Si ese número es negativo
        Escribir que ese número no sirve para nuestro propósito
    Sino
        Preguntar si el número es par
        Si lo es entonces escribir que el número leído es par
        Si no lo es escribir que el número leído no es par
Fin

Como puede ver, la característica principal es que la forma informal nos da una idea muy coloquial de nuestra solución, pero se sale mucho de los esquemas técnicos que necesitamos para poder llevar este algoritmo a un computador. Por eso, la forma técnica es la que nos permite realmente llevar al computador lo que hemos considerado como solución a nuestro objetivo. ¿Cuál es la forma técnica? Pues aquí se la presento también:

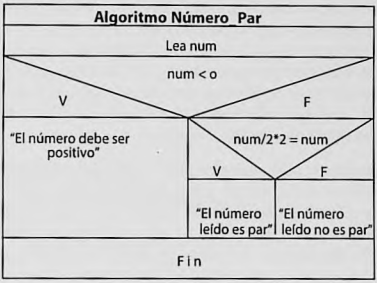
```
Algoritmo Número_Par
Variables
        Entero : num
Inicio
        Lea num
        Si num < 0
                Escriba "El número debe ser positivo"
Sino
                Si num / 2 * 2 = num
                        Escriba "El número leído es par"
                Sino
                        Escriba "El número leído no es par"
Fin
```

En la forma técnica, muchas de las órdenes son equivalentes a instrucciones de un lenguaje de programación y ello es lo que nos permite que fácilmente podamos convertir el algoritmo en un programa. Esto significa que es esta última forma la que vamos a representar utilizando las técnicas estudiadas.

Diagrama de flujo



Diagramación rectangular estructurada



No se olvide que lo que va entre comillas dobles en la diagramación rectangular estructurada representa un título y así debe quedar bien sea en la pantalla o en cualquier unidad de salida.

Seudocódigo

Podría surgirle la pregunta ¿cómo se representa este algoritmo en seudocódigo? Pues precisamente el algoritmo técnico (o escrito teniendo en cuenta algunas normas técnicas) es el equivalente del algoritmo solución en seudocódigo. De acuerdo a esto (y por una vez más), el seudocódigo sería:

```
Algoritmo Número_Par
Variables
        Entero : num
Inicio
        Lea num
        Si num < 0
                Escriba "El número debe ser positivo"
Sino
                Si num / 2 * 2 = num
                        Escriba "El número leído es par"
                Sino
                        Escriba "El número leído no es par"
Fin
```

Transcripción o codificación

Con el ánimo de dar ejemplos concretos, voy a mostrarle a usted, querido lector, que cuando se va a programar lo más importante no es el conocimiento de un lenguaje de programación (ya que con su uso uno se va acostumbrando a sus reglas), sino la lógica de programación que usted use para desarrollar soluciones algorítmicas encaminadas a lograr un objetivo. De acuerdo con esto, la diferencia entre un lenguaje y otro serán solo sus instrucciones debido a que la lógica seguirá siendo la misma.

Vamos pues a mostrar cómo sería el algoritmo del ejercicio escrito en términos de cuatro lenguajes: Basic, Pascal, C y Cobol. El objetivo perseguido en esta parte del libro no es que usted de una vez comience a programar utilizando un lenguaje determinado. El objetivo es que usted simplemente vea, con un ejemplo sencillo, que utilizar cualquier lenguaje de programación es fácil si tiene usted una lógica que le permita desarrollar y entender unos buenos algoritmos y además tenga una somera idea de qué es utilizar un lenguaje como tal. No voy a detenerme a explicar nada de la sintaxis de cada lenguaje, ya que no es ese el objetivo del libro. Con lo dicho anteriormente, recordemos (otra vez) el algoritmo solución original:

```
Algoritmo Número_Par
Variables
        Entero : num
Inicio
        Lea num
        Si num < 0
                Escriba "El número debe ser positivo"
Sino
                Si num / 2 * 2 = num
                        Escriba "El número leído es par"
                Sino
                        Escriba "El número leído no es par"
Fin
```

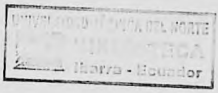
Versión en Lenguaje Basic

```
input num
if num < 0 then
        print "El número debe ser positivo"
else
        if int(num/2*2) = num then
                print "El número leído es par"
        else
                print "El número leído no es par"
```

Versión en Lenguaje Pascal

```
program numero_par;
var
        num     :       integer;
begin
        readln(num);
        if (num < 0) then
                writeln ('El número debe ser positivo');
        else
         if (num/2*2 = num) then
                writeln ('El número leído es par');
         else
                writeln ('El número leído no es par');
end.
```

Versión en Lenguaje C



```
#include <iostream.h>
void main( )
{
        int num;
        cin >> num;
        if (num < 0)
                cout << "El número debe ser positivo";
        else
                if (num/2*2 == num)
                        cout << "El número leído es par";
                else
                        cout << "El número leído no es par";
}
```

Versión en Lenguaje Cobol

```
        IDENTIFICATION DIVISION.
        PROGRAM_ID. NUMERO_PAR.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. CLON.
        OBJECT-COMPUTER. CLON.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 NUM                  PIC 99.
        PROCEDURE DIVISION.
```

```
        INICIO.
                ACCEPT NUM LINE 10 POSITION 10 NO BEEP
                IF NUM IS LESS THAN 0 THEN
                        DISPLAY "EL NÚMERO DEBE SER POSITIVO"
LINE 12 COL 10
                ELSE
                        IF NUM / 2 * 2 IS EQUAL TO NUM THEN
                                DISPLAY "EL NÚMERO LEÍDO ES PAR"
LINE 12 COL 10
                        ELSE
                                DISPLAY "EL NÚMERO LEÍDO NO ES PAR"
LINE 12 COL 10
                STOP RUN.
```

Puede usted notar que la utilización de cualquier lenguaje de programación se reduce casi a reemplazar algunas palabras clave por las que corresponden en el lenguaje, pero la lógica como tal permanece allí intacta. Por ahora solo me interesa que, con un vistazo ligero, usted solamente compare que, en el fondo, la utilización de cualquier lenguaje es lo mismo sin importar cuál sea. Siempre tenga en cuenta que lo importante, lo realmente importante, lo verdaderamente importante, es la lógica con la cual usted desarrolle sus algoritmos.

6.5. Segundo enunciado

Leer dos números enteros positivos y determinar si el último dígito de un número es igual al último dígito del otro.

Concepción del problema

Clarificación del objetivo

Primero que nada vamos a determinar cuál es el objetivo concreto de este enunciado. Según él, tendremos que leer dos números enteros (y, por lo tanto, tendremos que almacenar cada uno en una variable diferente) y tendremos que comparar mutuamente el último dígito de un número con el último dígito del otro número. En aras de que el enunciado esté absolutamente claro diremos que, para este algoritmo, será el último dígito de un número aquel que tenga el menor peso decimal. De esta forma, el último dígito del número 256 es el 6 y el último dígito del número 59 será el 9.

Ahora bien, si los dos números leídos son, por ejemplo, el 189 y el 65, entonces tendremos que comparar el 9 (último dígito del 189) con el 5 (último dígito del 65). Si fueran esos los dos números leídos, entonces tendríamos que decir

que no son iguales. Pero si los números leídos fueran el 37 y el 347, entonces al comparar el 7 del 37 con el 7 del 347 tendríamos que decir que son iguales.

Tal vez usted piense que esta es una explicación que sobra puesto que el enunciado posiblemente para usted sea muy claro, pero es muy bueno que se acostumbre, aun con enunciados muy sencillos, a clarificar el objetivo de manera que pueda usted explicarlo con absoluta certeza. Repito, no importa que el enunciado sea sencillo o parezca sencillo. Lo importante es que usted sepa claramente para dónde va para saber por dónde se va a ir.

Algoritmo

Versión informal

Algoritmo para comparar el último dígito
Inicio
    Leer un número entero y guardarlo en una variable entera
    Leer otro número entero y guardarlo en otra variable entera
    Guardar en una variable el último dígito del primer número leído
    Guardar en otra variable el último dígito del último dígito leído
    Comparar el contenido de estas dos últimas variables
    Si son iguales
        Escribir que los dos últimos dígitos son iguales
    Si no son iguales
        Escribir que los dos últimos dígitos no son iguales
Fin

Antes de desarrollar la versión técnica (o el seudocódigo), debemos pensar un momento en la forma como le vamos a decir el computador que guarde en una variable *el último dígito de cualquier número*. Para ello, nos vamos a valer de los operadores aritméticos y, de esa manera, basado en operaciones, obtener el último dígito de cualquier número (siempre que este sea positivo). Usted tal vez se preguntará por qué tenemos que valernos de operaciones para obtener el último dígito de un número y no podemos decirlo así explícitamente, sabiendo que el enunciado es tan claro. La razón es muy sencilla. El computador no fue a la escuela y por eso el solo obedece órdenes claras y "ejecutables sin razonamientos", es decir, órdenes que no involucren ningún razonamiento adicional para ser realizadas.

Basado en ello, para nosotros es muy claro hablar de "*el último dígito de un número cualquiera*", pero para el computador no. Por esta razón, vamos a utilizar las operaciones pertinentes para obtener el último dígito del número leído.

Vamos a asumir que el número leído será almacenado en una variable que se llamará *num*. Si *num* contuviera el valor 156, entonces tendríamos que obtener el dígito 6. Además, vamos a asumir que ese último dígito será almacenado en una variable llamada *ud* (como de último dígito). Entonces podríamos decir, para este ejemplo específico, que:

$$ud = num - 150$$

De nuevo, si asumimos que *num* vale 156, entonces el resultado será el 6 que estamos buscando. Vemos que 150 es un número constante, o sea, que si el número almacenado en *num* fuera 897, no nos serviría ese 150 para obtener el último dígito. Lo que por ahora sí podemos hacer es expresar ese 150 en términos de *num* aprovechando las características de la aritmética entera. El número 150 sería igual a dividir *num* entre 10 y posteriormente multiplicarlo por 10. Recuerde que en aritmética entera se generan decimales.

Por lo tanto, la expresión *num* / 10 * 10 es realizada por el computador de la siguiente forma (teniendo en cuenta la jerarquía de operadores):

- Primero se realiza la división *num* / 10. En el ejemplo hemos asumido que el valor almacenado en *num* es 156, entonces la división *num* / 10 nos da como resultado 15.
- Segundo se realiza la multiplicación de ese resultado por 10, lo cual para el ejemplo nos daría 150. Y hemos obtenido el valor que buscábamos.

Ahora vamos a reemplazar la expresión inicial por su equivalente usando la variable *num*. De esta manera tenemos:

$$ud = num - 150$$

Como 150 es lo mismo que *num / 10 * 10*, entonces:

$$ud = num - num / 10 * 10$$

Ahora realicémosle una prueba (paso a paso) asumiendo que la variable *num* tiene el valor 854.

$$ud = num - num / 10 * 10$$
$$ud = 854 - 854 / 10 * 10$$
$$ud = 854 - 85 * 10$$
$$ud = 854 - 850$$
$$ud = 4$$

Vemos que en la variable *ud* quedó almacenado el último dígito del número. Veamos otra prueba (paso a paso) asumiendo que el valor almacenado en la variable *num* es 5468.

*ud = num – num / 10 * 10*
*ud = 5468 – 5468 / 10 * 10*
*ud = 5468 – 546 * 10*
ud = 5468 – 5460
ud = 8

Pues podemos decir que hemos encontrado la forma genérica de almacenar en una variable el último dígito de un número cualquiera (que era parte de lo que estábamos buscando). No se olvide que el objetivo de este ejercicio es comparar el último dígito de cada uno de dos números y determinar si son iguales. Esto que hasta el momento se ha hecho es solo la forma de facilitar el ejercicio. Esta solución solo tiene una restricción. Vamos a asumir que el valor almacenado en *num* es –563. Realicémosle la prueba (paso a paso) con la fórmula:

*ud = num – num / 10 * 10*
*ud = - 563 – (- 563) / 10 * 10*
*ud = - 563 – (- 56) * 10*
ud = - 563 – (- 560)
ud = - 563 + 560
ud = - 3

Esto significa que, para que esta fórmula sea utilizada correctamente, tenemos que asegurarnos que el valor almacenado en la variable *num* es positivo.

Ahora sí volvamos a nuestro problema inicial para plantear el algoritmo técnico (o dicho de una vez, el seudocódigo). Recordemos que el enunciado del problema es *Leer dos números enteros positivos y determinar si el último dígito de un número es igual al último dígito del otro*. De acuerdo a esto, podríamos considerar el siguiente algoritmo como su versión técnica (no se olvide que los algoritmos expuestos aquí son la versión personal del autor). Si usted desarrolla otro algoritmo que realice lo mismo (o sea, que logre el mismo objetivo) y al hacerle la prueba de escritorio ve que realmente logra el objetivo, entonces tanto su algoritmo como el mío estarán bien así sean diferentes. Para el desarrollo del algoritmo técnico (o seudocódigo) utilizaremos las variables:

num1 en donde almacenaremos el primer número leído
num2 en donde almacenaremos el segundo número leído
ud1 en donde se almacenará el último dígito del primer número leído
ud2 en donde se almacenará el último dígito del segundo número leído

Esta vez, para facilidad del usuario de este algoritmo, le avisaremos cuándo debe entrar los números utilizando la orden *Escriba* (no se olvide que estas son órdenes que se le darían al computador).

```
Algoritmo Compara_Ult_digs
Variables
        Entero : num1, num2, ud1, ud2
Inicio
        Escriba "Digite dos números enteros"
        Lea num1, num2
Si num1 < 0
                num1 = num1 * (-1)
        Si num2 < 0
                num2 = num2 * (-1)
        ud1 = num1 - num1 / 10 * 10
        ud2 = num2 - num2 / 10 * 10
        Si ud1 = ud2
                Escriba
                "El último dígito de un número es igual al último dígito del otro"
        Sino
                Escriba
                "El último dígito de un número no es igual al último dígito del otro"
Fin_Si
Fin
```

Es de anotar que en el conjunto de instrucciones

```
Si num1 < 0
        num1 = num1 * (-1)
Si num2 < 0
        num2 = num2 * (-1)
```

lo que se busca es asegurarnos que el valor almacenado en las variables *num1* y *num2* sea positivo, precisamente para que nuestra fórmula para obtener el último dígito funcione bien.

Prueba de escritorio

Ya sabemos que la forma correcta de hacer una prueba de escritorio es realizar todas y cada una de las instrucciones "sin razonarlas", llevando cuidadosamente los cambios en la memoria y realizando lo pertinente en la pantalla. Cuando termine la prueba de escritorio, o sea, cuando el algoritmo llegue a su fin, entonces mire lo que haya quedado en pantalla y compare su coherencia con el objetivo planteado por el enunciado.

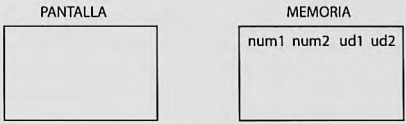
Si ve que lo que está en pantalla satisface el objetivo, entonces su algoritmo estará bien. Si no es así, entonces tendrá que revisar el algoritmo para que pueda alcanzar el objetivo propuesto.

Vamos entonces a desarrollar la prueba de escritorio de este algoritmo realizando paso a paso cada una de las instrucciones y representando lo que vaya pasando tanto en memoria como en pantalla.

Algoritmo Compara_Ult_digs

Variables

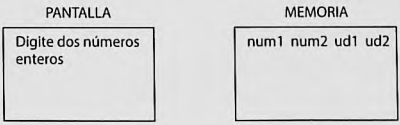
Entero : num1, num2, ud1, ud2



El algoritmo se inicia, para efectos de la prueba de escritorio, con una pantalla en blanco y unas variables ubicadas en la memoria sin ningún contenido.

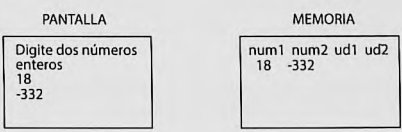
Inicio

Escriba "Digite dos números enteros "



Aparece en pantalla un título solicitando dos números enteros.

Lea num1, num2



Para efectos de la prueba de escritorio, vamos a asumir que los números leídos son 18 y –332. El computador los almacenará así: el primer valor en la variable *num1* y el segundo valor en la variable *num2* porque en ese orden está la instrucción *Lea num1, num2* que significa *Lea un número entero y guárdelo en la variable num1 y luego lea otro número entero y guárdelo en la variable num2.*

Si num1 < 0
*num1 = num1 * (-1)*

Como el contenido de la variable *num1* es igual a 18, entonces esta pregunta es Falsa, por lo tanto, pasamos a la siguiente decisión:

Si num2 < 0
*num2 = num2 * (-1)*

Como el contenido de la variable *num2* es igual a –332, entonces la pregunta *Si num2 < 0* es Verdadera. Por lo tanto, se ejecuta la orden que está allí, o sea, *num2 = num2 * (-1)*, quedando almacenado en la variable *num2* el valor 332 y anulando el valor anterior de -332. No se olvide que, cada vez que entra un nuevo valor a una variable, el valor anterior se borra.

PANTALLA

```
Digite dos números
enteros
18
-332
```

MEMORIA

```
num1  num2 ud1  ud2
 18   -332
       332
```

La siguiente instrucción es:

*ud1 = num1 – num1 / 10 * 10*

Tomando el valor almacenado en la variable *num1*, vamos a desarrollar esta expresión paso a paso para, de esta forma, saber con certeza cuánto queda almacenado en la variable *ud1*.

*ud1 = num1 – num1 / 10 * 10*
*ud1 = 18 - 18 / 10 * 10*
*ud1 = 18 – 1 * 10*
ud1 = 18 - 10
ud1 = 8

De esta manera, vemos que en la variable *ud1* queda almacenado el valor 8 (que corresponde al último dígito del valor contenido en la variable *num1*).

PANTALLA

```
Digite dos números
enteros
18
-332
```

MEMORIA

num1	num2	ud1	ud2
18	~~-332~~ 332	8	

Siguiente instrucción:

*ud2 = num2 – num2 / 10 * 10*

Conociendo que el valor almacenado en la variable *num2* es 332, entonces desarrollaremos paso a paso la expresión:

*ud2 = num2 – num2 / 10 * 10*
*ud2 = 332 – 332 / 10 * 10*
*ud2 = 332 – 33 * 10*
ud2 = 332 – 330
ud2 = 2

Así comprobamos que el valor que queda almacenado en *ud2* es 2, que corresponde al último dígito del valor almacenado en *num2*.

PANTALLA

```
Digite dos números
enteros
18
-332
```

MEMORIA

num1	num2	ud1	ud2
18	~~-332~~ 332	8	2

Note usted que mientras se han realizado operaciones en la memoria, en la pantalla no se registra ningún cambio.

Siguiente instrucción:

Si ud1 = ud2
Escriba
"El último dígito de un número es igual al último dígito del otro"

Sino
Escriba
"El último dígito de un número no es igual al último dígito del otro".

Sabiendo que la variable *ud1* contiene el valor 8 y que la variable *ud2* contiene el valor 2, internamente la decisión es *Si 8 = 2*. Como es Falso, entonces deberá hacerse lo que está por el *Sino* de la decisión, o sea, ejecutar la orden *Escriba "El último dígito de un número no es igual al último dígito del otro"*. De esta manera, obtenemos:

PANTALLA

```
Digite dos números enteros
18
-332
El último dígito de un
número no es igual al
último dígito del otro
```

MEMORIA

num1	num2	ud1	ud2
18	~~-332~~ 332	8	2

Y nuestra última instrucción (si se le puede llamar así) es:

Fin

Con lo cual damos por terminada la prueba de escritorio. Ahora sí verificamos lo que quedó en pantalla (solamente). En pantalla quedó:

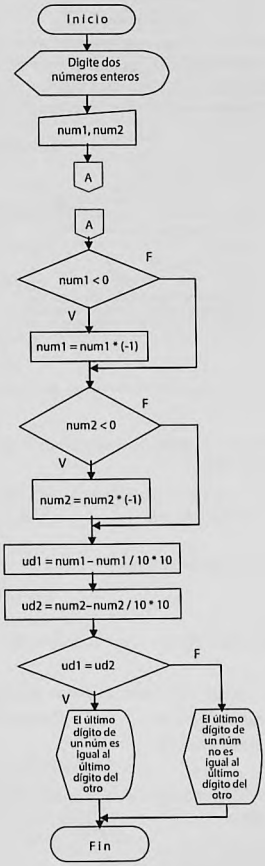
PANTALLA

```
Digite dos números
enteros
18
-332
El último dígito de un
número no es igual al
último dígito del otro
```

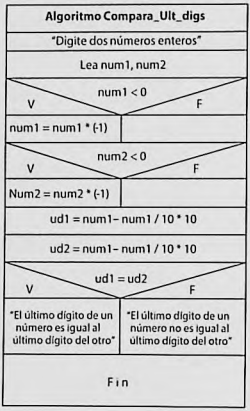
Vemos pues que el algoritmo (basado solo en lo que queda en pantalla... no se le olvide) leyó dos números enteros y pudo determinar que el último dígito de un número no es igual al último dígito del otro y como el enunciado era *Leer dos números enteros positivos y determinar si el último dígito de un número es igual al último dígito del otro*, entonces podremos decir que está bien. Ahora, para estar totalmente seguro, usted realizará la prueba de escritorio asumiendo que el valor almacenado en la variable *num1* es 459 y el valor almacenado en la variable *num2* es 6239. No olvide que la prueba de escritorio debe ser desarrollada paso a paso y ejecutada tal y como esté en el algoritmo para poder tener absoluta certeza de que el computador va a obtener los mismos resultados que nosotros obtendríamos en esa prueba. La representación gráfica nos facilita tener una comprensión mejor del algoritmo y nos permite además tenerlo escrito en términos más técnicos y entendibles a la mayoría de programadores.

Técnicas de representación

Diagrama de flujo



Diagramación rectangular estructurada



Seudocódigo

Algoritmo Compara_Ult_digs
Variables
    Entero : num1, num2, ud1, ud2
Inicio
    Escriba "Digite dos números enteros"
    Lea num1, num2
Si num1 < 0
        *num1 = num1 * (-1)*
    Si num2 < 0
        *num2 = num2 * (-1)*
    *ud1 = num1 – num1 / 10 * 10*
    *ud2 = num2 – num2 / 10 * 10*
    Si ud1 = ud2
        Escriba "El último dígito de un número es igual al último dígito del otro"
    Sino

Escriba "El último dígito de un número no es igual al último dígito del otro"

Fin

No se olvide que el seudocódigo no es más que la versión técnicamente escrita del algoritmo original, de manera que ya para nosotros es conocido. Es importante que sepa que en la práctica real se usa el seudocódigo o el diagrama de flujo o el diagrama rectangular para representar el algoritmo.

Transcripción o codificación

Lenguaje Basic

```
Print "Digita dos números enteros"
Input num1, num2
If num1 < 0 then
        num1 = num1 * (-1)
If num2 < 0 then
        num2 = num2 * (-1)
ud1 = num1 – num1 / 10 * 10
ud2 = num2 – num2 / 10 * 10
if ud1 = ud2
                Print "El último díg de un número es igual al último díg del otro"
 else
                Print "El último díg de un número no es igual al último
                díg del otro"
```

Lenguaje Pascal

```
Program Compara_Ult_digs;
Var
        num1, num2, ud1, ud2 : integer;
Begin
        Writeln ('Digite dos números enteros');
        Readln (num1, num2);
if num1 < 0 then
                num1 := num1 * (-1);
        if num2 < 0 then
                num2 := num2 * (-1);
        ud1 := num1 – num1 / 10 * 10;
```

```
        ud2 := num2 – num2 / 10 * 10;
        If ud1 = ud2 then
                Writeln ('El último díg de un número es igual al último díg del
otro');
        else
                Writeln ('El último díg de un número no es igual al último díg
del
                otro');
End.
```

Lenguaje C

```
#include <iostream.h>
void main( )
{
int num1, num2, ud1, ud2
        cout << "Digite dos números enteros";
        cin >> num1, num2;
if (num1 < 0)
                num1 = num1 * (-1);
        If (num2 < 0)
                num2 = num2 * (-1);
        ud1 = num1 – num1 / 10 * 10;
        ud2 = num2 – num2 / 10 * 10;
        If ( ud1 == ud2 )
                Cout <<"El último díg de un núm es igual al último díg del otro";
        else
                Cout << "El último díg de un núm no es igual al último
                díg del otro";
}
```

Lenguaje Cobol

```
        IDENTIFICATION DIVISION.
        PROGRAM_ID. COMPARA_ULT_DIGS.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. CLON.
        OBJECT-COMPUTER. CLON.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUM1                PIC 99.
01 NUM2                PIC 99.
01 UD1                 PIC 99.
01 UD2                 PIC 99.
PROCEDURE DIVISION.
INICIO.
        DISPLAY "DIGITE DOS NÚMEROS ENTEROS"
        LINE 10 COL 10
        ACCEPT NUM1
        LINE 12 COL 10
        ACCEPT NUM2
        LINE 14 COL 10
        IF NUM1 IS LESS THAN 0 THEN
                COMPUTE NUM1 = NUM1 * (-1)
        IF NUM2 IS LESS THAN 0 THEN
                COMPUTE NUM2 = NUM2 * (-1)
        COMPUTE UD1 = NUM1 – NUM1 / 10 * 10
        COMPUTE UD2 = NUM2 – NUM2 / 10 * 10
        IF UD1 IS EQUAL TO UD2 THEN
                DISPLAY "EL ÚLTIMO DÍGITO DE UN NÚMERO
                ES IGUAL AL ÚLTIMO DÍGITO DEL OTRO"
                LINE 20 COL 10
        ELSE
                ESCRIBA "EL ÚLTIMO DÍGITO DE UN NÚMERO
                NO ES IGUAL AL ÚLTIMO DÍGITO DEL OTRO"
LINE 20 COL 10
        STOP RUN.
```

6.6. Tercer enunciado

Leer un número entero y determinar cuántos dígitos tiene.

Concepción del problema

Clarificación del objetivo

El objetivo de este algoritmo en esencia busca solicitar un número entero y contar cuántos dígitos tiene. ¿Cómo lo vamos a lograr? Primero que nada

tengamos en cuenta que, si el número leído fuera 15, tendríamos que detectar, a través de nuestro algoritmo, que tiene 2 dígitos. Si el número leído fuera 5946, tendríamos que detectar que tiene 4 dígitos. Si el número fuera –958, tendríamos que decir que tiene 3 dígitos. Entonces, lo que tenemos que hacer es crear un procedimiento que nos permita saber cuántos dígitos tiene el número.

Algoritmo

Algoritmo para determinar la cantidad de dígitos de un número entero
Inicio
    Leer un número entero
    Guardar en una variable la cantidad de dígitos
    Mostrar el contenido de esta variable en pantalla

Fin

Ya sabemos que este es el algoritmo informal y también sabemos que no es este algoritmo el que podemos llevar, finalmente, a un computador, pues involucra una serie de razonamientos que el mismo computador no tendría. Por ello, se hace importante que nosotros transformemos este algoritmo en uno que pueda ser transcrito fácilmente a un lenguaje de programación. Para ello, lo primero que tendremos que tener en cuenta es que la parte de contar los dígitos del número requerirá de un proceso que posteriormente pueda ser entendible por un computador cuando lo hayamos codificado.

Vamos a aprovechar la aritmética entera de la cual disponemos para obtener la cantidad de dígitos. Supongamos que el número leído fuera 1546 y además supongamos que va a existir una variable llamada *Contador De Dígitos* (que abreviado sería *cd*) que va a contener la cantidad de dígitos que tenga el número. Iniciamos la variable *cd* en ceros y tomamos el número leído, al cual vamos a llamar *num* por comodidad.

num	*cd*
1546	0

Si dividimos el valor contenido en la variable *num* entre 10 y lo almacenamos de nuevo en la misma variable, entonces quedará almacenado en ella el valor 154 (por características de aritmética entera) y además vamos a incrementar en 1 el valor contenido en la variable *cd*.

num	*cd*
~~1546~~	0
154	1

Si volvemos a dividir el nuevo número almacenado en la variable *num* entre 10, obtendremos el número 15 y además volvemos a incrementar el valor de la variable *cd* en 1, entonces:

num		cd
~~1546~~		0
~~154~~		1
15	2	

Si volvemos a dividir el nuevo contenido de la variable *num* (que ahora es 15) entre 10, obtendremos el número 1 y si volvemos a incrementar el contenido de la variable *cd* en 1, obtendremos el valor 3:

num		*cd*
~~1546~~	0	
~~154~~	1	
~~15~~	2	
1		3

De la misma manera, si volvemos a dividir el contenido de la variable *num* entre 10 y volvemos a incrementar el valor de la variable *cd* en 1, entonces obtendremos:

num		*cd*
~~1546~~		0
~~154~~		1
~~15~~		2
1	3	
0		4

Ya como vemos que el valor contenido en la variable *num* ha llegado a cero y que si siguiéramos dividiendo entre 10 sucesivamente seguiría dando como resultado cero, entonces podemos detener allí esta secuencia de divisiones. Puede usted notar que, por cada vez que dividíamos entre 10 el valor contenido en la variable *num*, incrementábamos el valor de la variable *cd*. Al finalizar, cuando el contenido de la variable *num* ha llegado por primera vez a cero, el contenido de la variable *cd* ha llegado a un número que es igual a la cantidad de dígitos que originalmente tenía la variable *num*.

Entonces, si quisiéramos resumir esto que hemos hecho, tendríamos que:

```
cd = 0
mientras num sea diferente de 0
        num = num / 10
        cd = cd + 1
Fin_Mientras
```

Que no es más que lo que hemos acabado de hacer paso a paso. Este fragmento nos permite almacenar en la variable *cd* la cantidad de dígitos del número almacenado en *num*. Y ¿qué pasa si el número almacenado en la variable *num* es negativo? Pues no pasa nada. El algoritmo sigue sirviendo y entregando el resultado correcto. Por lo tanto, el algoritmo completo para determinar cuántos dígitos tiene un número que previamente fue leído expresado en términos técnicos es el siguiente:

```
Algoritmo Cuenta_Digitos
Var
        Entero : num, cd
Inicio
        Escriba "Digite un número entero"
        Lea num
        cd = o
        Mientras num < > 0
                num = num / 10
                cd = cd + 1
        Fin_mientras
        Escriba "El número digitado tiene" cd "dígitos"
Fin
```

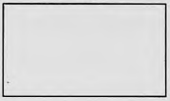
Hemos utilizado el operador < > para expresar *diferente de*, pero en los seudocódigos se puede adoptar el operador que corresponda a cualquier lenguaje. Lo importante es que se utilice un operador único y que este represente *diferente de* sin ambigüedades.

Prueba de escritorio

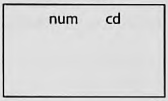
Tomemos el algoritmo resultante para realizarle la prueba de escritorio tal como lo hemos hecho hasta el momento (y es como se debe hacer), o sea, paso a paso. El algoritmo inicia su proceso teniendo en memoria dos variables (*num, cd*) y con una pantalla, teóricamente, en blanco.

Algoritmo Cuenta_Dígitos
Var
Entero : num, cd

PANTALLA



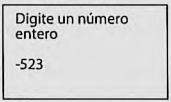
MEMORIA



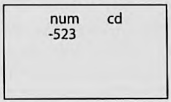
A continuación, el algoritmo da la orden de escribir un título de petición y leer un número entero y guardarlo en la variable *num* (que también obviamente es entera). Vamos a suponer que el número leído es –523.

Inicio
Escriba "Digite un número entero"
Lea num

PANTALLA



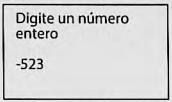
MEMORIA



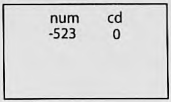
A continuación, se inicializa la variable *cd* con el valor 0:

cd = 0

PANTALLA



MEMORIA



Seguidamente viene el ciclo:

Mientras num < > 0
num = num / 10
cd = cd + 1
Fin_mientras

En el cual inicialmente se pregunta *si num< >0* (en realidad, la pregunta es *Mientras num< >0*, que para efectos de la prueba de escritorio se reemplaza brevemente por una decisión). En esta pregunta se sabe que, si la respuesta es Verdadera, entonces se deberán ejecutar las órdenes que siguen. Como en el caso inicial la variable *num* contiene el valor –523 y este valor es diferente de 0, entonces la respuesta es Verdadera, por lo tanto, entramos a realizar las dos instrucciones:

num = num / 10
cd = cd + 1

De manera que en la variable *num* queda almacenado el valor –52 y en la variable *cd* queda almacenado el valor 0.

PANTALLA

Digite un número entero.

-523

MEMORIA

num	cd
~~-523~~	0
-52	1

Como ya el valor de la variable *num* no es el mismo que el inicial, volvemos al planteamiento del ciclo y a realizar la misma pregunta, o sea, *mientras num< >0*, encontrándonos con que el valor almacenado en la variable *num* es –52 y que es realmente diferente de 0. Entonces esto quiere decir que volvemos a ejecutar las órdenes:

num = num / 10
cd = cd + 1

que conforman el cuerpo del ciclo, o sea, el conjunto de órdenes que se deben ejecutar en caso de que la condición del ciclo sea Verdadera. Al ejecutarlas, obtenemos que en la variable *num* queda almacenado el valor –5 y en la variable *cd* queda almacenado el valor 2.

PANTALLA

Digite un número entero

-523

MEMORIA

num	cd
~~-523~~	0
~~-52~~	~~1~~
-5	2

Recuerde que cada vez que es cambiado el valor de una variable, el valor anterior se pierde. Por lo tanto, en estos momentos los valores que realmente hay son: en la variable *num* es –5 y en la variable *cd* es 2.

En estas condiciones, y habiendo ejecutado el cuerpo del ciclo, volvemos a la condición que determina el fin del ciclo. Ante la pregunta *Mientras num< >0*, vemos que la variable *num* contiene el valor –5 y que este es evidentemente diferente de 0, entonces la respuesta es Verdadera, por lo tanto, volvemos a ejecutar (una vez más) las órdenes:

num = num / 10
cd = cd + 1

Con lo cual quedará almacenado en la variable *num* el valor 0 y en la variable *cd* el valor 3.

PANTALLA

Digite un número entero -523

MEMORIA

num	cd
~~-523~~	0
~~-52~~	~~1~~
-5	2
0	3

Con lo cual, después de haber ejecutado las órdenes que conformaban el cuerpo del ciclo, volvemos al inicio del mismo a verificar de nuevo la condición y vemos que esta vez, ante la pregunta de que el contenido de *num* sea diferente de 0, la respuesta es Falso. Por lo cual ya no volvemos a ejecutar las órdenes que conforman el cuerpo del ciclo, sino que pasamos directamente a ejecutar la orden que le sigue al ciclo, o sea, la que está después del finalizador *Fin_Mientras*, es decir, la orden:

Escriba "El número digitado tiene" cd "dígitos"

Con lo cual saldría en pantalla el título *El número digitado tiene 3 dígitos*, lo cual es cierto. Tenga en cuenta que, cuando una variable aparece inmersa en un título pero por fuera de las comillas dobles, esto quiere decir que saldrá en pantalla el contenido de la variable.

PANTALLA

Digite un número entero -523 El número digitado tiene 3 dígitos

MEMORIA

num	cd
~~-523~~	0
~~-52~~	~~1~~
-5	2
0	3

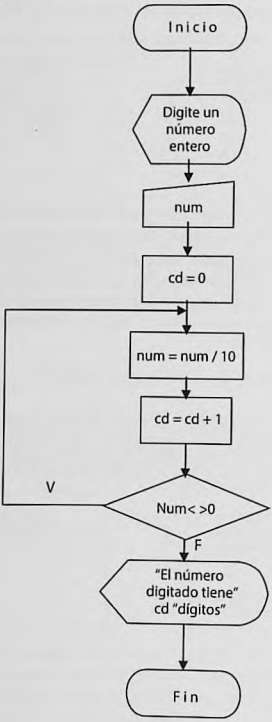
Después de lo cual lo único que nos queda es ejecutar la última orden y es finalizar el algoritmo:

Fin

Ahora bien, recuperando nuestra condición de seres humanos, podemos ver que lo que está escrito en pantalla es Verdad, es decir, si hemos digitado el número –523, el algoritmo nos ha dicho algo cierto y es que este número tiene 3 dígitos. Como el objetivo inicial era leer un número y determinar cuántos dígitos tenía y este algoritmo lo hace correctamente, entonces podemos decir que está bien.

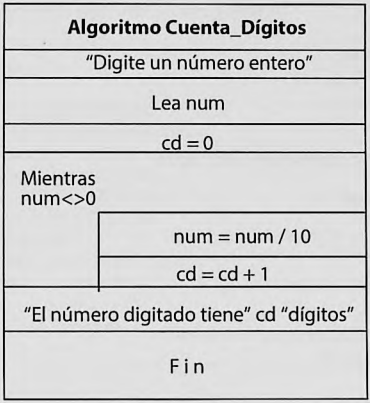
Técnicas de representación

a. Diagrama de Flujo



No olvide que lo importante es que se vea claramente cuál es el flujo de la solución.

Diagramación rectangular estructurada



Seudocódigo

```
Algoritmo Cuenta_Dígitos
Var
        Entero : num, cd
Inicio
        Escriba "Digite un número entero"
        Lea num
        cd = o
        Mientras num < > 0
                num = num / 10
                cd = cd + 1
        Fin_mientras
        Escriba "El número digitado tiene" cd "dígitos"
Fin
```

Transcripción o Codificación

Lenguaje Basic

```
Print "Digite un número entero"
Input num
cd =0
while num < > 0
        num = num / 10
        cd = cd + 1
wend
Print "El número digitado tiene", cd, "dígitos"
```

Lenguaje Pascal

```
Program Cuenta_Digitos;
Var
        num, cd : integer;
begin
        writeln('Digite un número entero');
        readln(num);
        cd = 0
        while num < > 0 do
        begin
                num := num div 10;
                cd := cd + 1;
        end;
        writeln('El número digitado tiene', cd, 'dígitos');
end.
```

Lenguaje C

```
#include <stdio.h>
void main( )
{
        int num, cd;
        cout << "Digite un número entero";
        cin >> num;
        cd = 0;
        while ( num ! = 0 )
```

```
{
        num = num / 10;
        cd ++;
}
cout << "El número digitado tiene" << cd << "dígitos";
}
```

Lenguaje Cobol

```
IDENTIFICATION DIVISION.
PROGRAM_ID. CUENTA-DIGITOS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. CLON.
OBJECT-COMPUTER. CLON.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUM          PIC 9999.
01 CD           PIC 9999.
PROCEDURE DIVISION.
INICIO.
        DISPLAY "DIGITE UN NÚMERO ENTERO" LINE 10 COL 10
        ACCEPT NUM LINE 10 COL 50
        COMPUTE CD = 0
        PERFORM CONTEO UNTIL NUM = 0.
        DISPLAY "EL NÚMERO DIGITADO TIENE"      LINE 14 COL 10
        DISPLAY CD                              LINE 14 COL 40
        DISPLAY "DÍGITOS"                       LINE 14 COL 50
        STOP RUN.
CONTEO.
        COMPUTE NUM = NUM / 10
        ADD 1 TO CD.
```

Es importante aclarar que el objetivo buscado en este capítulo es que usted tenga unos ejemplos claros de todo el proceso (al menos el que se hace en el papel) que se ha de tener en cuenta cuando se va a desarrollar un algoritmo y de él se va a derivar un programa.

También me parece muy importante recordarle que en este libro en ningún momento se busca que usted sepa de lenguajes de programación, pero lo que sí es importante, por ahora, es que usted vea que la codificación en cualquier lenguaje de programación es fácil, dado que sencillamente se trata de

reemplazar algunas palabras clave y utilizar una estructura establecida por los creadores del lenguaje. Lo realmente importante detrás de un programa es la lógica que se haya utilizado para el desarrollo del algoritmo.

En este capítulo, se ha acudido a unos lenguajes específicos que permiten mostrar fácilmente (sin requerir un entorno integrado de desarrollo) que lo que importa es la LÓGICA con que se diseñe la solución a un problema, o sea, un programa.

Capítulo 7

Decisiones

Ya sabemos que una decisión, a nivel de lógica de programación, es la escogencia de uno de entre varios caminos lógicos dependientes todos de una condición. Normalmente, algunos libros acotan en esta definición que se escoge uno de entre dos caminos lógicos y, dado que todo el proceso del computador es binario, podemos decir que esos libros también tienen la razón.

Por facilidades de representación, se han esquematizado en los algoritmos (y así mismo en los lenguajes de programación) dos estructuras de decisión que son la *Estructura Si-Entonces-Sino*, que es la que hemos estado utilizando en los ejemplos hechos hasta el momento, y la *Estructura Casos*, que nos permite realizar la escogencia de uno de entre varios ramales lógicos dependientes de una misma condición.

7.1. Estructura Si-Entonces-Sino

Esta es la estructura que hemos estado utilizando desde el comienzo de los algoritmos. Como ya se conoce, podemos decir que su utilidad, fundamentalmente, es permitir que el computador escoja uno de dos ramales lógicos dependiendo de una determinada condición. Es importante anotar que tomar una decisión, por simple que esta sea, le toma mucho tiempo al computador realizarla ya que, aunque para nosotros es muy sencillo determinar si 9 es mayor que 5, para el computador no lo es, pues debe realizar algunas operaciones para obtener la respuesta correcta.

De esta manera, es útil saber que es más eficiente un programa que tenga más decisiones que otro que tenga menos, toda vez que ambos busquen lograr el mismo objetivo. Teóricamente, no hay mucho que decir acerca de las decisiones, pero técnicamente es muy posible que usted llegue a encontrarse con algunos términos que desde ya es muy importante que conozca y que no

son más que formas de reorganización del mismo esquema de decisión que hemos utilizado hasta el momento.

7.1.1. Decisiones simples

Obedecen a la siguiente estructura:

```
Si (Condición)
        .
        .
        Instrucciones a ejecutar
        En caso de que la condición sea Verdadera
        .
        .
Sino
        .
        .
        Instrucciones a ejecutar
        En caso de que la condición sea Falsa
        .
        .
Fin_Si
```

Como puede ver, es la estructura más sencilla para una toma de decisiones. Acerca de esta estructura podemos decir que no es obligatorio que cada vez que exista un condicional *Si* tenga que existir una alternativa *Sino*, dado que no siempre es importante generar una determinada acción en el caso de que la condición sea Falsa. Normalmente, es importante delimitar hasta dónde llega toda la estructura de decisión y esa función la cumple el *Fin_Si* que aparece al final de ella. También vale la pena saber que en los lenguajes de programación estos delimitadores se pueden escribir con determinados signos establecidos por la sintaxis del mismo lenguaje.

No se olvide que en una estructura de decisión cuando se realizan las instrucciones por la parte Verdadera no se hacen las instrucciones por la parte Falsa y viceversa, es decir, cuando se realizan las instrucciones por la parte Falsa no se hacen las instrucciones por la parte Verdadera.

7.1.2. Decisiones en cascada

Este no es más que un esquema en donde el *Sino* de cada *Si* condicional da inicio a un nuevo *Si* condicional y así sucesivamente. Su esquema general es el siguiente:

```
Si Condición1
        Instrucciones a ejecutar en caso de que
        la condición1 sea Verdadera
Sino
        Si Condición2
                Instrucciones a ejecutar en caso de que
                la condición2 sea Verdadera
        Sino
                Si Condición3
                        Instrucciones a ejecutar en caso de que
                        la condición3 sea Verdadera
                Sino
                        Instrucciones a ejecutar en caso de que
                        la condición3 sea Falsa
```

Este es el esquema utilizado para el caso en el que se dan 3 condiciones en cascada, pero de acuerdo a las necesidades del algoritmo pueden ser más. Todo dependerá del objetivo que se quiera lograr. Para ilustrar un poco mejor la utilización de esta estructura, veamos un ejemplo en donde sea necesaria.

Ejemplo

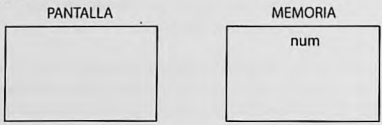
Leer un número entero y determinar si es de uno o dos o tres o cuatro dígitos. Validar que el número no sea negativo.

```
Programa Decisión_en_Cascada
Var
        Entero  :       num
Inicio
        Escriba "Por favor, digite un número entero"
        Lea num
        Si num < 0
                num = num * ( -1 )
        Si num > = 1 y num < = 9
                Escriba " El número tiene 1 dígito"
        Sino
                Si num > = 10 y num < = 99
                        Escriba "El número tiene 2 dígitos"
                Sino
                        Si num > = 100 y num < = 999
                                Escriba "El número tiene 3 dígitos"
```

```
        Sino
            Si num > = 1000 y num < = 9999
                Escriba "El número tiene 4 dígitos"
            Sino
                Escriba "El número tiene más de 4
dígitos"
Fin
```

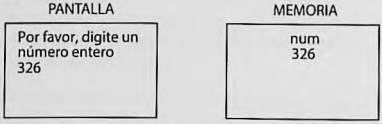
Vamos a realizarle una pequeña prueba de escritorio. En primera instancia, la memoria inicia con una sola variable entera a la que hemos llamado *num*.

Programa Decisión_en_Cascada
Var
Entero : num



Continúa el algoritmo colocando un título en pantalla y leyendo un dato entero que será almacenado en la variable *num*. Vamos a asumir que el número leído es igual a 326.

Inicio
Escriba "Por favor, digite un número entero"
Lea num



Se realiza la primera validación y es verificar si el número es negativo. De ser así, debe multiplicarse por el valor de (-1) para que se vuelva positivo. Como el valor recibido es 326 y, por lo tanto, la condición *Si num < 0* es falsa, entonces se salta la orden *num = num * (-1)*:

```
Si num < 0
        num = num * ( -1 )
```

A continuación comienza a evaluar cuántos dígitos tiene el número leído. Para ello, va realizando preguntas por rangos, sabiendo que el rango de los números de un dígito está entre 1 y 9, el rango de los números de dos dígitos está entre 10 y 99, el rango de los números de tres dígitos está entre 100 y 999, el rango de los números de 4 dígitos está entre 1000 y 9999 y sabiendo que por encima de 9999 por ahora solo nos interesa decir que hay números con más de 4 dígitos (o sea, 5 o 6 o 7 o más dígitos).

Ante la pregunta *Si num >= 1 y num <= 9* y sabiendo que la variable *num* contiene el número 326, puede decirse que se resuelve así:

Si num >= 1 y num <= 9
V y F

Vemos pues que 326 es mayor que 1 pero no es menor que 9, por lo tanto, la primera parte de la decisión es Verdadera y la segunda parte es Falsa, por lo tanto, acogiéndonos a la tabla de verdad del operador booleano **Y** vemos que *V y F* nos da Falso, por lo tanto, toda la decisión es Falsa. Por este motivo, el computador se salta la orden *Escriba "El número tiene 1 dígito"* y continúa con el *Sino* de la primera decisión.

```
Si num >= 1 y num <= 9
        Escriba "El número tiene 1 dígito"
Sino
        Si num >= 10 y num <= 99
                Escriba "El número tiene 2 dígitos"
```

Claramente la orden dice *Si num >= 10 y num <= 99* y sabiendo que la variable *num* contiene 326, entonces la decisión se convierte en:

Si 326 >= 10 y 326 <= 99
V y F

Dado que 326 es mayor que 10 y que el mismo 326 no es menor que 99 y sabiendo que están unidas las dos condiciones por un operador booleano **Y**, entonces toda la decisión es Falsa, ya que así lo dice la tabla de verdad de este operador. Por lo tanto, el computador se saltará la orden *Escriba "el número es de 3 dígitos"* y va a continuar con el *Sino* de la segunda decisión que da origen a una tercera condición.

```
        Sino
                Si num >= 100 y num <= 999
                        Escriba "El número tiene 3 dígitos"
```

En esta decisión se pregunta *Si num > = 100 y num < = 999*. Como sabemos que la variable *num* contiene 326, vemos que la decisión (internamente) se convierte en:

Si num > = 100 y num < = 999
Si 326 > = 100 y 326 < = 999
V y V

Con lo cual vemos que ambas condiciones son verdaderas, por lo tanto, toda la condición es Verdadera, debido a que según la Tabla de Verdad del operador **Y** este genera Verdadero solo si sus dos entradas son Verdaderas, o sea, solo si las condiciones que tiene a los lados son Verdaderas. Por lo tanto, como esta condición es Verdadera, entonces el computador ejecuta:

Escriba "El número tiene 3 dígitos"

PANTALLA

```
Por favor, digite un
número entero
326
El número tiene 3
dígitos
```

MEMORIA

```
num
326
```

Como ejecuta esta instrucción por ser Verdadera, la condición entonces no ejecuta ni evalúa lo que depende del *Sino* de esta condición. De tal manera que se "salta" lo siguiente:

```
Sino
        Si num > = 1000 y num < = 9999
                Escriba "El número tiene 4 dígitos"
        Sino
                Escriba "El número tiene más de 4 dígitos"
```

Llegando directamente hasta el finalizador del algoritmo, que es lo último que ejecuta:

Fin

Acogiéndonos a lo que en su momento habíamos indicado, podemos mirar ahora el área de la pantalla y verificar si lo que aparece allí es conceptualmente cierto o no. Como vemos que es cierto, entonces podemos decir que este algoritmo está bien, al menos en lo que corresponde a la evaluación de números de tres dígitos. Realice ahora usted una prueba de escritorio con este algoritmo con cada uno de los siguiente números: - 5, 6498 y 32.

Usted puede notar que la utilización de decisiones en cascada nos permite llegar de una manera sencilla y fácil a lograr un determinado objetivo en donde se involucren muchos condicionales interrelacionados.

7.1.2. Decisiones en secuencia

Este es el tipo de estructura que se utiliza cuando se deben realizar varias preguntas en donde no es importante (normalmente) el *Sino* de cada decisión. Su esquema general es el siguiente:

Si Condición1
    Instrucciones a ejecutar en caso de que
    La condición1 sea Verdadera
Si condición2
    Instrucciones a ejecutar en caso de que
    La condición2 sea Verdadera
Si condición3
    Instrucciones a ejecutar en caso de que
    La condición3 sea Verdadera
Si condición4
    Instrucciones a ejecutar en caso de que
    La condición4 sea Verdadera

No se olvide que pueden existir muchas más condiciones. El esquema aquí presentado solamente muestra la estructura general para cuando sean 4 condiciones, pero si se necesitan más simplemente se pueden utilizar y ya. Alguna de las condiciones puede tener su correspondiente *Sino*. Tenga en cuenta que si la última condición de un conjunto de decisiones en secuencia tiene *Sino*, este solo se ejecutará en caso de que la última condición sea Falsa y no tendrá nada que ver con las demás condiciones. Al igual que el ejemplo anterior, veamos un poco más clara la utilización de la estructura con un ejemplo.

Ejemplo

Leer un número entero y determinar si es positivo o negativo o si es 0.

No olvide que para todos los enunciados aquí planteados los algoritmos que se presentan son solo una versión de solución para dichos enunciados. Con esto quiero decirle una vez más que, si usted ha desarrollado por su cuenta los algoritmo aquí presentados y encuentra una solución diferente a lo que aparece en este libro, entonces no se preocupe; sencillamente, realícele una prueba de escritorio a su versión y, si cumple el objetivo, entonces estará bien,

no importa que sea diferente al algoritmo que aparezca en este libro. Esta es mi versión de la solución para este enunciado.

```
Programa Decisiones_en_Secuencia
Variables
        Entero  :       num
Inicio
        Escriba "Digite un número entero"
        Lea num
        Si num < 0
                Escriba "El número digitado es negativo"
        Si num > 0
                Escriba "El número digitado es positivo"
        Si num = 0
                Escriba "El número digitado es cero"
Fin
```

Sé que usted puede estar pensando que este algoritmo se pudo haber solucionado con decisiones en cascada, pero tenga en cuenta que la razón por la que utilizamos aquí decisiones en secuencia es que sabemos con toda seguridad que un número cualquiera podrá tener tres estados (si se le puede llamar así): que sea menor que cero o sea negativo, que sea mayor que cero o sea positivo o que sea igual a cero. Esa certeza es la que nos permite utilizar esta estructura de decisión y no la anterior.

También puede estar pensando que en vez de haber planteado el algoritmo así:

```
        Si num < 0
                Escriba "El número digitado es negativo"
        Si num > 0
                Escriba "El número digitado es positivo"
        Si num = 0
                Escriba "El número digitado es cero"
```

se pudo haber planteado así:

```
        Si num < 0
                Escriba "El número digitado es negativo"
        Si num > 0
                Escriba "El número digitado es positivo"
        Sino
                Escriba "El número digitado es cero"
```

Pero si realizáramos la prueba de escritorio con el valor –5, por ejemplo, para la variable *num* veríamos que, cuando pregunte *Si num < 0*, la respuesta sería Verdadera y, por lo tanto, ejecutaría *Escriba "El número digitado es negativo"*, dejando en pantalla el título:

El número digitado es negativo

Pero, siguiendo con la prueba de escritorio, cuando se realice la pregunta *Si num > 0*, la respuesta tendrá que ser Falsa y entonces se ejecutará el *Sino* de esta decisión, o sea, *Escriba "El número digitado es cero"*, lo cual nos dejaría en pantalla los títulos:

El número digitado es negativo
El número digitado es cero

Que son títulos contradictorios, pues un número no puede ser negativo y ser cero al mismo tiempo. Por esta razón, la buena utilización de decisiones en secuencia nos va a permitir no solo lograr los objetivos sino, además, lograrlos de una buena forma técnica.

7.1.3. Decisiones anidadas

Estas se presentan en aquellos casos en los cuales una estructura completa de decisión se encuentra dentro de otra. Su esquema general es el siguiente:

```
Si Condición_Externa
        .
        .
        Si Condición_Interna
                Instrucciones a ejecutar en caso de que
                La condición interna sea Verdadera
        Sino
                Instrucciones a ejecutar en caso de que
                La condición interna sea Falsa
        Fin_Si_Interno
        .
        .
Sino
        .
        .
```

Instrucciones a ejecutar en caso de que
La condición externa sea Falsa
.
.
Fin_Si_Externo

O también se puede encontrar de la siguiente forma:

Si Condición_Externa
.
.
Instrucciones a ejecutar en caso de que
La condición externa sea Verdadera
.
.
Sino
.
.
.
Si Condición_Interna
Instrucciones a ejecutar en caso de que
La condición interna sea Verdadera
Sino
Instrucciones a ejecutar en caso de que
La condición interna sea Falsa
Fin_Si_Interno
.
.
Fin_Si_Externo

En este caso, podemos ver que en uno de los dos ramales lógicos de una estructura de decisión completa se encuentra otra estructura de decisión completa. Los puntos suspensivos colocados en la estructura representan que pueden existir más instrucciones. Veamos su utilización con un ejemplo.

Ejemplo

Leer un número entero y determinar si es positivo o negativo. Si es positivo, determinar si tiene dos dígitos y si es negativo, determinar si tiene tres dígitos. Asumir que no puede entrar el número cero.

```
Programa Decisiones_Anidadas
Variables
        Entero  :       n
Inicio
        Escriba "Digite un número entero"
        Lea num
        Si num > 0
                Si num >= 10 y num <= 99
                        Escriba "El número es positivo y tiene dos dígitos"
                Sino
                        Escriba "El número es positivo y no tiene dos dígitos"
                Fin_Si
        Sino
                Si num >= - 999 y num <= -100
                        Escriba "El número es negativo y tiene tres dígitos"
                Sino
                        Escriba "El número es negativo y no tiene tres dígitos"
                Fin_Si
        Fin_Si
```

Realícele dos pruebas de escritorio a este algoritmo, la primera asumiendo que el número leído es –52 y la segunda asumiendo que es 1650. Tenga mucho cuidado con la ejecución de las decisiones que aquí se plantean.

7.2. Estructura casos

Esta estructura permite abreviar una serie de decisiones en cascada o en secuencia. La inmensa mayoría de lenguajes de programación tiene una instrucción equivalente para su implementación y precisamente lo que se busca con esta estructura es facilitar la toma de decisiones por parte del computador, dado que este es el proceso que más tiempo le toma. La estructura casos toma el contenido de una variable y lo evalúa acorde con unos posibles valores ejecutando lo que se le indique en cada una de las opciones.

7.2.1. Estructura casos simple

Su estructura general es la siguiente:

Evalúe (variable)

Si vale (valor_1) : Instrucciones a ejecutar en caso de que la variable sea igual a valor_1

Si vale (valor_2) : Instrucciones a ejecutar en caso de que la variable sea igual a valor_2

Si vale (valor_3) : Instrucciones a ejecutar en caso de que la variable sea igual a valor_3

. .
. .
. .

Si vale (valor_n) : Instrucciones a ejecutar en caso de que la variable sea igual a valor_n

Sino :Instrucciones a ejecutar en caso de que la variable no sea lgual a ninguno de los valores posibles (o sea valor_1, valor_2, valor_3, valor_4,....., valor_n)

Fin_Evalúe

Su forma de ejecución es muy sencilla. Al iniciar esta estructura, el computador recibe la variable con un determinado contenido (o por lo menos dentro de nuestros algoritmos nos debemos asegurar de ello), verifica si el contenido de la variable es igual a uno de los posibles valores que se hayan considerado dentro de la estructura. De ser así, ejecuta las órdenes que acompañen al valor determinado. Si el contenido de la variable no es igual a ninguno de los contenidos, entonces ejecuta las instrucciones que acompañan al *Sino* de esta estructura.

Veámoslo con un ejemplo para una mejor clarificación de su uso y al tiempo haremos una comparación con una secuencia de decisiones en cascada para que se vea su diferencia.

Ejemplo

Leer un número entero y, si es de un dígito y además es menor que 5, escribir su nombre en pantalla (el nombre del 1 es UNO, el nombre del 2 es DOS, etc.).

Versión 1.0 Algoritmo solución con decisiones

```
Programa Nom_Digito
Variables
        Entero  :       n
Inicio
        Escriba "Digite un número entero"
        Lea n
        Si n < 0
                n = n * ( -1 )
        Si n = 1
                Escriba "Uno"
        Sino
                Si n = 2
                        Escriba "Dos"
                Sino
                        Si n = 3
                                Escriba "Tres"
                        Sino
                                Si n = 4
                                        Escriba "Cuatro"
                                Sino
                                        Si n = 5
                                                Escriba "Cinco"
                                        Sino    Escriba "El número es mayor
                                                que cinco"
                                        Fin_Si
                                Fin_Si
                        Fin_Si
                Fin_Si
        Fin_Si
Fin
```

Algunas precisiones oportunas acerca de esta primera solución del problema planteado:

a. Los indicadores *Fin_Si* corresponden a cada una de las decisiones que inicialmente se colocaron. No son obligatorios pero son muy útiles cuando se escriben, sobre todo al momento de codificar el algoritmo.

b. La indentación del algoritmo (o sea, ese hecho de que cada conjunto o bloque de instrucciones comienza unas columnas más allá) es muy útil para la claridad del programa y, de manera muy especial, al igual que el numeral anterior, cuando se va a codificar el programa.

c. Esta versión incluye una serie de decisiones en cascada y, por lo tanto, realizándole una prueba de escritorio podremos notar que logra el objetivo planteado. Sin embargo, la utilización de la estructura casos nos permite escribir el mismo algoritmo de una manera más entendible.

Versión 2.0 Algoritmo solución con estructura casos

```
Programa Nom_Digito
Variables
        Entero  :       n
Inicio
        Escriba "Digite un número entero"
        Lea n
        Si n < 0
                n = n * ( -1 )
        Evalúe ( n )
                Si vale 1 :     Escriba "Uno"
                Si vale 2 :     Escriba "Dos"
                Si vale 3 :     Escriba "Tres"
                Si vale 4 :     Escriba "Cuatro"
                Si vale 5 :     Escriba "Cinco"
                Sino :  Escriba "El número es mayor que cinco"
        Fin_Evalúe
Fin
```

En esta solución, utilizando la estructura casos, usted puede notar que:

a. Este algoritmo logra el mismo objetivo que el algoritmo anterior, solo que de una manera técnica más apropiada.

b. La presentación del algoritmo es mucho más entendible y precisamente eso lo hace mucho más fácil de codificar que la anterior versión.

c. El *Sino* que aparece al final de la estructura casos se ejecuta en caso de que el contenido de la variable *n* no se igual ni a 1 ni a 2 ni a 3 ni a 4 ni a 5.

d. La indentación vuelve a ser útil y para ello quiero mostrarle dentro de este numeral cómo se verían ambas versiones sin indentación:

Versión 1.0 Algoritmo solución con decisiones sin indentación

```
Programa Nom_Digito
Variables
        Entero  :       n
Inicio
Escriba "Digite un número entero"
Lea n

Si n < 0
n = n * ( -1 )

Si n = 1
Escriba "Uno"
Sino
Si n = 2
Escriba "Dos"
Sino
Si n = 3
Escriba "Tres"
Sino
Si n = 4
Escriba "Cuatro"
Sino
Si n = 5
Escriba "Cinco"
Sino    Escriba "El número es mayor que cinco"
Fin_Si
Fin_Si
Fin_Si
Fin_Si
Fin_Si
Fin
```

Versión 2.0 Algoritmo solución con estructura casos sin indentación

```
Programa Nom_Digito
Variables
        Entero  :       n

Inicio
Escriba "Digite un número entero"
Lea n

Si n < 0
n = n * ( -1 )

Evalúe ( n )
Si vale 1 :     Escriba "Uno"
Si vale 2 :     Escriba "Dos"
Si vale 3 :     Escriba "Tres"
Si vale 4 :     Escriba "Cuatro"
Si vale 5 :     Escriba "Cinco"
Sino :  Escriba "El número es mayor que cinco"
Fin_Evalúe
Fin
```

No me podrá negar usted que los algoritmos vistos sin indentación (o sea, sin estética) son mínimamente más complejos de entenderlos que si se presentan de una manera técnica más apropiada. Recuerde que la utilidad grande de la indentación se refleja al momento de codificar los algoritmos en algún lenguaje de programación, pues allí el hecho de que su algoritmo esté bien y que la prueba de escritorio le haya arrojado unos resultados muy confiables puede llegar a truncarse por una mala codificación o, peor aún, por una mala agrupación de instrucciones. No está de más recordarle que, cuando se agrupan las instrucciones como no son, los resultados del algoritmo pueden ser completamente diferentes.

7.2.2. Estructuras casos (anidadas)

Esta estructura se utiliza cuando una de las opciones de la estructura casos general da origen a otra estructura casos y otro conjunto de instrucciones. Veamos un ejemplo de esto:

Ejemplo

Leer un entero y, si es igual a cualquier dígito comprendido entre 1 y 5, escribir su nombre. Si es igual a cinco, además de escribir su nombre, leer otro dígito y, si

este último está entre 1 y 5, escribir su componente decimal. Si entró un 3, entonces escribir "Cincuenta y Tres"; si entró un 1, entonces escribir "Cincuenta y Uno".

```
Programa Casos_Anidados
Variables
        Entero  :       num, dig
Inicio
        Escriba "Digite un número entero"
        Lea num

        Si num < 0
                num = num * ( -1 )

        Evalúe ( num )
                Si vale 1:      Escriba "Uno"
                Si vale 2:      Escriba "Dos"
                Si vale 3:      Escriba "Tres"
                Si vale 4:      Escriba "Cuatro"
                Si vale 5:      Escriba "Cinco"
                                Escriba "Digite otro número entero"
                                Lea dig
                                Evalúe ( dig )
                                        Si vale 1: Escriba "Cincuenta y Uno"
                                        Si vale 2: Escriba "Cincuenta y Dos"
                                        Si vale 3: Escriba "Cincuenta y Tres"
                                        Si vale 4: Escriba "Cincuenta y Cuatro"
                                        Si vale 5: Escriba "Cincuenta y Cinco"
                                        Sino : Escriba "El núm es mayor que 5"
                                Fin_Evalúe
                Sino :  Escriba "El número es mayor que cinco"
        Fin_Evalúe
Fin
```

Acerca de este algoritmo, es poco lo que se puede aclarar, pues como usted puede ver es bastante claro; sin embargo, debe tener en cuenta que cada *Evalúe* tiene su correspondiente *Sino* (y solo uno) y su correspondiente *Fin_Evalúe*. No se olvide que para la buena utilización de esta estructura es muy importante que usted sea una persona organizada y ordenada y verá como programar se vuelve un verdadero paseo.

7.3. Ejercicios

Algunas posibles soluciones a los siguientes ejercicios se pueden encontrar en el libro *Algoritmos* del mismo autor.

1. Leer un número entero y determinar si es un número terminado en 4.
2. Leer un número entero y determinar si tiene 3 dígitos.
3. Leer un número entero y determinar si es negativo.
4. Leer un número entero de dos dígitos y determinar a cuánto es igual la suma de sus dígitos.
5. Leer un número entero de dos dígitos y determinar si ambos dígitos son pares.
6. Leer un número entero de dos dígitos menor que 20 y determinar si es primo.
7. Leer un número entero de dos dígitos y determinar si es primo y además si es negativo.
8. Leer un número entero de dos dígitos y determinar si sus dos dígitos son primos.
9. Leer un número entero de dos dígitos y determinar si un dígito es múltiplo del otro.
10. Leer un número entero de dos dígitos y determinar si los dos dígitos son iguales.
11. Leer dos números enteros y determinar cuál es el mayor.
12. Leer dos números enteros de dos dígitos y determinar si tienen dígitos comunes.
13. Leer dos números enteros de dos dígitos y determinar si la suma de los dos números origina un número par.
14. Leer dos números enteros de dos dígitos y determinar a cuánto es igual la suma de todos los dígitos.
15. Leer un número entero de tres dígitos y determinar a cuánto es igual la suma de sus dígitos.

16. Leer un número entero de tres dígitos y determinar si al menos dos de sus tres dígitos son iguales.

17. Leer un número entero de tres dígitos y determinar en qué posición está el mayor dígito.

18. Leer un número entero de tres dígitos y determinar si algún dígito es múltiplo de los otros.

19. Leer tres números enteros y determinar cuál es el mayor. Usar solamente dos variables.

20. Leer tres números enteros y mostrarlos ascendentemente.

21. Leer tres números enteros de dos dígitos cada uno y determinar en cuál de ellos se encuentra el mayor dígito.

22. Leer un número entero de tres dígitos y determinar si el primer dígito es igual al último.

23. Leer un número entero de tres dígitos y determinar cuántos dígitos primos tiene.

24. Leer un número entero de tres dígitos y determinar cuántos dígitos pares tiene.

25. Leer un número entero de tres dígitos y determinar si alguno de sus dígitos es igual a la suma de los otros dos.

26. Leer un número entero de cuatro dígitos y determinar a cuánto es igual la suma de sus dígitos.

27. Leer un número entero de cuatro dígitos y determinar cuántos dígitos pares tiene.

28. Leer un número entero menor que 50 y positivo y determinar si es un número primo.

29. Leer un número entero de cinco dígitos y determinar si es un número capicúa. Ej. 15651, 59895.

30. Leer un número entero de cuatro dígitos y determinar si el segundo dígito es igual al penúltimo.

31. Leer un número entero y determina si es igual a 10.

32. Leer un número entero y determinar si es múltiplo de 7.

33. Leer un número entero y determinar si termina en 7.

34. Leer un número entero menor que 1000 y determinar cuántos dígitos tiene.

35. Leer un número entero de dos dígitos, guardar cada dígito en una variable diferente y luego mostrarlas en pantalla.

36. Leer un número entero de 4 dígitos y determinar si tiene más dígitos pares o impares.

37. Leer dos números enteros y determinar cuál es múltiplo de cuál.

38. Leer tres números enteros y determinar si el último dígito de los tres números es igual.

39. Leer tres números enteros y determinar si el penúltimo dígito de los tres números es igual.

40. Leer dos números enteros y, si la diferencia entre los dos es menor o igual a 10, entonces mostrar en pantalla todos los enteros comprendidos entre el menor y el mayor de los números leídos.

41. Leer dos números enteros y determinar si la diferencia entre los dos es un número primo.

42. Leer dos números enteros y determinar si la diferencia entre los dos es un número par.

43. Leer dos números enteros y determinar si la diferencia entre los dos es un número divisor exacto de alguno de los dos números.

44. Leer un número entero de 4 dígitos y determinar si el primer dígito es múltiplo de alguno de los otros dígitos.

45. Leer un número entero de 2 dígitos y, si es par, mostrar en pantalla la suma de sus dígitos; si es primo y menor que 10, mostrar en pantalla su último dígito y, si es múltiplo de 5 y menor que 30, mostrar en pantalla el primer dígito.

46. Leer un número entero de 2 dígitos y, si termina en 1, mostrar en pantalla su primer dígito; si termina en 2, mostrar en pantalla la suma de sus dígitos y, si termina en 3, mostrar en pantalla el producto de sus dos dígitos.

47. Leer dos números enteros y, si la diferencia entre los dos números es par, mostrar en pantalla la suma de los dígitos de los números; si dicha diferencia es un número primo menor que 10, entonces mostrar en pantalla el producto de los dos números y, si la diferencia entre ellos termina en 4, mostrar en pantalla todos los dígitos por separado.

48. Leer un número entero y, si es menor que 100, determinar si es primo.

49. Leer un número entero y, si es múltiplo de 4, determinar si su último dígito es primo.

50. Leer un número entero y, si es múltiplo de 4, mostrar en pantalla su mitad; si es múltiplo de 5, mostrar en pantalla su cuadrado y, si es múltiplo de 6, mostrar en pantalla su primer dígito. Asumir que el número no es mayor que 100.

Capítulo 8

Ciclos

Ya hemos utilizado no solo esta palabra, sino una estructura asociada que nos permite representar un conjunto de instrucciones que debe repetirse una cantidad determinada de veces, normalmente, dependiente de una condición. Los ciclos nos van a permitir iterar todo un proceso tantas veces como nosotros (o el usuario) lo determinemos.

8.1. Concepto general

Un ciclo puede definirse como una estructura que nos permite repetir o iterar un conjunto de instrucciones y que tiene las siguientes características:

a. El conjunto de instrucciones debe ser finito.

b. La cantidad de veces que se repita dicho conjunto de instrucciones también debe ser finita. En algunos casos, esta cantidad de veces va a depender de una condición explícita y, en otros casos, va a depender de una condición implícita. Una condición es explícita cuando depende solamente de la misma ejecución del programa sin que sea importante la participación del usuario. Asimismo, una condición es implícita cuando depende solamente de la voluntad del usuario y, por lo tanto, la cantidad de iteraciones o repeticiones del ciclo podría llegar a ser diferente cada vez, pues sería posible que cambiara con cada usuario.

c. Deben estar claramente demarcados el inicio y el fin del ciclo. En los casos en los cuales solo exista una instrucción a iterar, no serán necesarias dichas marcas.

d. Dentro de un ciclo, podrá ir cualquiera de las otras estructuras que se han estudiado incluyendo otros ciclos.

Vamos a desarrollar un ejemplo sin ciclos para notar la gran utilidad de estructurar nuestros algoritmos con ciclos.

Ejemplo

Mostrar los números del 1 al 100 de 1 en 1.

Versión ineficiente No. 1

Programa Vers_Inef_1
Inicio
    Escriba "1"
    Escriba "2"
    Escriba "3"
    Escriba "4"
    Escriba "5"
    Escriba "6"
    Escriba "7"
    Escriba "8"
    Escriba "9"
    Escriba "10"
    Escriba "11"
    Escriba "12"
    Escriba "13"
    Escriba "14"
    Escriba "15"
    Escriba "16"
    Escriba "17"
        .
        .
        .
        .
        .
    Escriba "98"
    Escriba "99"
    Escriba "100"
Fin

Como puede ver en esta versión no se han utilizado variables y los puntos suspensivos representan toda esa cantidad de instrucciones que hacen falta, pues en total serían 100 instrucciones *Escriba*. Es evidente que se logra el objetivo planteado, pero imagine que en vez de ir el enunciado hasta 100 fuera hasta 1000 o fuera hasta 10000.

Nuestro algoritmo se convertiría no solo en una cantidad ineficiente de instrucciones, sino que, además, por cada vez que existiera una modificación, prácticamente tendría que existir un algoritmo diferente, pues tendríamos que adicionarle más y más líneas de órdenes. Veamos a continuación otra forma ineficiente de solucionar este mismo problema sin utilizar ciclos.

Versión ineficiente No. 2

```
Programa Vers_Inef_2
Variables
        Entero :        N
Inicio
        N = 1
        Escriba N
        Si N < = 100
                N = N + 1
Escriba N
        Si N < = 100
                N = N + 1
        Escriba N
        Si N < = 100
                N = N + 1
        Escriba N
        Si N < = 100
                N = N + 1
        Escriba N
        Si N < = 100
                N = N + 1
        Escriba N
        Si N < = 100
                N = N + 1
        Escriba N
        Si N < = 100
                N = N + 1
```

```
Escriba N
Si N < = 100
        N = N + 1
Escriba N
Si N < = 100
        N = N + 1
Escriba N
        .
        .
        .
        .
Si N < = 100
        N = N + 1
Escriba N
Si N < = 100
        N = N + 1
Escriba N
Fin
```

Como puede ver, tendríamos que escribir 99 veces el esquema:

```
Si N < = 100
        N = N + 1
Escriba N
```

para poder lograr el objetivo, con lo cual esta segunda versión ineficiente resultaría ser mucho más larga que la anterior y, dada la gran cantidad de decisiones que debe tomar el computador, sería a su vez más ineficiente. Lo que sí podemos hacer es tomar el esquema repetitivo de esta última versión y escribirlo dentro de un ciclo que controle que se repita dicho esquema hasta cuando se hayan escrito todos los números enteros de 1 a 100 de 1 en 1.

¿Cómo lo haremos? Pues muy sencillo. Simplemente note que tenemos que decirle al computador que inicie una variable en 1 y que mientras el contenido de esta variable sea menor o igual que 100 que escriba su contenido y que lo incremente en 1. De esta manera, el contenido de la variable irá de 1 en 1 desde 1 hasta 100, escribiendo cada vez que cambie de número, es decir, cumpliendo el objetivo planteado. Note usted que acabo de decirle el algoritmo informal; ahora todo lo que tenemos que hacer es llevarlo a un algoritmo técnico para que posteriormente sea fácil codificarlo en un lenguaje de programación.

De manera que la siguiente es la versión técnica eficiente de la solución al enunciado planteado.

Versión eficiente con ciclos

```
Programa Nums_1_100
Variables
        Entero : N
Inicio
        N = 1
        Mientras N < = 100
                Escriba N
                N = N + 1
        Fin_mientras
Fin
```

Evidentemente, el algoritmo así presentado es mucho más claro. No se puede negar que tanto esta solución como las demás soluciones, a pesar de ser ineficientes, también cumplen con el objetivo planteado. Acerca de esta versión, podemos hacer algunas reflexiones:

a. Es mucho más fácil de codificar en un determinado lenguaje de programación, pues es una solución mucho más compacta.

b. Es mucho más entendible y, por lo tanto, es mucho más fácil de concebir como solución.

c. No necesita muchas líneas de código para lograr el objetivo.

d. Cuando necesiten *mostrar los números del 1 al 10000*, todo lo que tenemos que hacer es cambiar el número *100* que aparece al inicio del ciclo por el número *10000* y el mismo algoritmo funcionará bien. No tendremos necesidad de adicionar líneas de código.

e. Cuando necesiten *mostrar los números del 45 al 951*, simplemente tendremos que reemplazar la línea que dice *N = 1* por *N = 45* y, en donde aparece el número *100*, cambiarlo por el número *951* y el algoritmo logrará bien este nuevo objetivo.

f. Finalmente, la solución planteada con ciclos al enunciado *Mostrar los números del 1 al 100 de 1 en 1* se ha convertido en una solución mucho más genérica que, cambiando los valores tope del mismo ciclo, será *Mostrar todos los enteros comprendidos entre dos números asumiendo que el primer número es el menor y el segundo es el mayor.*

Con estas reflexiones, podemos justificar plenamente la utilización de ciclos dentro de un algoritmo y, en lo posible, buscar hacer un uso muy eficiente de esta estructura.

8.2. Tipos de ciclos

Solo para facilitar la escritura de algunos algoritmos y con el ánimo de que desde el balcón de la lógica de programación se puedan tener más herramientas que faciliten la estructuración de los ciclos, la mayoría de los lenguajes de programación tienen tres formas de presentación de los ciclos. Ellas son:

a. Ciclo *Mientras*
b. Ciclo *Para*
c. Ciclo *Haga Hasta*

Precisamente, vamos a revisar la estructura de construcción de cada uno de los ciclos tal como son concebidos por la mayoría de lenguajes de programación y, posteriormente, los utilizaremos para representar el mismo algoritmo con cada una de las estructuras.

8.2.1. Ciclo *Mientras*

Este es el ciclo que hemos utilizado desde que comenzamos a hablar de algoritmos. Es el esquema general de trabajo para todos los ciclos. Esto quiere decir que, si usted entiende claramente la lógica de funcionamiento de este ciclo, se le va a facilitar entender no solo los otros ciclos que aquí se explican, sino cualquier otro ciclo que encuentre en algún otro libro de lógica de programación. Es útil que sepa que este ciclo también es llamado en algunos libros el Ciclo *Mientras Que*, pero su filosofía es la misma del Ciclo *Mientras* que vamos a explicar aquí y que hemos venido utilizando.

Su estructura general es la siguiente:

```
Mientras Condición
    .
    .
    Cuerpo del Ciclo
    .
    .
Fin_Mientras
```

Su forma de ejecución (textualmente explicada) es muy sencilla: *Mientras se cumpla que la condición sea Verdadera entonces se ejecutará el Cuerpo del Ciclo.* De

manera que también podríamos decir que el cuerpo del ciclo se repetirá tantas veces como lo permita la condición o mientras dicha condición sea Verdadera. En condiciones normales la cantidad de veces que se repita el cuerpo del ciclo será siempre una cantidad finita y deberá existir, dentro del mismo cuerpo del ciclo, una o más instrucciones que nos permitan aproximarnos a la condición, o sea, que propendan porque en algún momento la condición sea Falsa.

8.2.2. Ciclo *Para*

El *Ciclo Para* tiene la siguiente estructura:

```
Para Var = tope_inicial hasta tope_final Paso Valor
        .
        .
        Cuerpo del Ciclo
        .
        .
Fin_Para
```

En este ciclo, su forma de ejecución es la siguiente: *Var* representa una variable que va a tomar valores iniciando en *tope_inicial* y terminando en *tope_final* avanzando con un *Paso de Valor*. En los casos en los que no se especifica el valor del paso, la mayoría de los lenguajes de programación asume el incremento de 1. El *Cuerpo del Ciclo* se ejecutará una vez por cada valor que tome la variable *Var*. Veamos con un ejemplo cuál sería la aplicación de este ciclo.

Ejemplo

Escribir los números impares comprendidos entre 1 y 20.

Es evidente que este ejemplo lo podemos desarrollar usando el ciclo *Mientras*, pero para efectos didácticos lo vamos a desarrollar usando el ciclo *Para*.

```
Programa Ejem_Ciclo_Para
Variables
        Entero : Num
Inicio
        Para Num = 1 hasta 20 Paso 2
                Escriba Num
        Fin_Para
Fin
```

Puede usted notar que este algoritmo es muy breve gracias a la presencia del ciclo *Para* en su contexto, ya que, si se hubiera estructurado utilizando el ciclo *Mientras*, una versión de solución hubiera podido ser la siguiente:

```
Programa Ejem_Ciclo_Mientras
Variables
        Entero : Num
Inicio
        Num = 1
        Mientras Num < = 20
                Escriba Num
                Num = Num + 2
        Fin_Mientras
Fin
```

Ambas versiones logran el mismo objetivo, lo cual significa que ambas versiones son correctas. Es importante anotar que dentro de lo normal cada ciclo siempre va a tener una variable que es la que almacena el valor de inicio del ciclo, es la que va a estar presente en la evaluación de la condición y es la que se incrementa para que en algún momento la condición sea Falsa.

Es evidente que esta variable es muy importante; por ello, este tipo de variables se ha caracterizado con el nombre de índice del *ciclo*. Podríamos decir que el índice del ciclo es la variable que permite la ejecución del cuerpo del ciclo. Un ciclo puede llegar a tener varios índices al tiempo.

Como los índices no son más que variables, entonces varios ciclos pueden tener el mismo índice siempre que se utilice este en un ciclo solo hasta cuando haya terminado la ejecución del ciclo anterior.

8.2.3. Ciclo *Haga Hasta*

Esta es otra de las formas que traen algunos de los lenguajes de programación para expresar un ciclo. Su estructura general es la siguiente:

```
Haga
        .
        .
        .
        Cuerpo del Ciclo
        .
        .
```

.
Hasta Condición

En este ciclo, el cuerpo del mismo se va a ejecutar hasta cuando se cumpla una condición; esto quiere decir que el conjunto de instrucciones que conforman el cuerpo del ciclo se va a repetir mientras la evaluación de la condición sea Falsa. Es un ciclo muy parecido al ciclo *Mientras*, con la diferencia de que en este las instrucciones se repiten mientras la condición sea falsa y no verdadera, como sería en el ciclo *Mientras*.

8.2.4. Ciclo *Haga Mientras*

Muy parecido al esquema anterior, algunos lenguajes de programación cuentan con esta otra estructura para representar un ciclo:

Haga

.
.
.
Cuerpo del Ciclo
.
.
.
Mientras Condición

Podría decirse que esta es una inversión de la estructura del ciclo *Mientras*. En este ciclo, el cuerpo del mismo se repite mientras la condición sea Verdadera y su única diferencia con el ciclo *Mientras* es que en el ciclo *Haga Mientras* primero se ejecuta el cuerpo del ciclo y luego se evalúa la condición, en cambio en el ciclo *Mientras* primero se evalúa la condición y luego se ejecuta el cuerpo del ciclo.

8.3. Ejemplos usando todas las estructuras de ciclos

8.3.1. Ejemplo 1

Leer un número entero y determinar cuántos dígitos tiene.

Es útil recordar que los datos enteros se manejan con aritmética entera, concepto que será muy útil para la concepción de este algoritmo, ya que la cantidad de dígitos que tiene un número entero es igual a la cantidad de veces que se pueda dividir el número entre 10 sin que el cociente sea cero. Entonces, lo

que vamos a hacer en este algoritmo es leer un número, inicializar una variable (que actuará como contador_de_dígitos), dividir progresivamente el número entre 10 hasta cuando sea igual a cero y, por cada vez que se divida entre 10, vamos a incrementar el contenido de la variable contador_de_dígitos en 1. De esta manera, cuando el número haya llegado a cero, tendremos en la variable contador_de_dígitos la cantidad de dígitos que originalmente tenía el número.

En caso de que el número original sea negativo también funciona esta lógica. No se olvide que siempre que vaya a desarrollar un algoritmo primero debe **clarificar el objetivo** para que sepa hacia dónde va y hasta dónde debe llegar.

Usando ciclo *Mientras*

```
Programa Ejemplo_1
Variables
        Entero : Numero, Cuenta_Digitos          // Declara Variables
Inicio
        Escriba "Digite un número entero"        // Solicita un dato entero
Lea Numero                                         //Lee un entero y lo almacena
                                                   // en la variable Numero
        Cuenta_Digitos = 0                       // Inicializa el contador en cero
        Mientras Numero < > 0          // Mientras Numero sea diferente de 0
                Numero = Numero / 10           //Divida entre 10...
                Cuenta_Digitos = Cuenta_Digitos + 1   // ...y cuente
        Fin_Mientras
        Escriba "Tiene", Cuenta_Digitos, "dígitos"   // Escriba cant de dígitos
Fin
```

Note usted que al lado derecho del algoritmo técnico está prácticamente escrito el algoritmo informal para mayor claridad en el momento que usted le desarrolle una prueba de escritorio. Igualmente, la doble barra inclinada (//) representa el inicio de un comentario. Los comentarios son textos explicativos de las órdenes o de bloques de órdenes de los algoritmos. No tienen incidencia en su ejecución. Todos los lenguajes tienen un equivalente para escribir comentarios. En nuestro caso, vamos a utilizar la doble barra inclinada que es utilizada en lenguaje C.

Usando ciclo *Para*

Este algoritmo no es "fácilmente" implementable con un ciclo *Para*. Tenga en cuenta que la utilización de uno y otro ciclo es precisamente para que se haga mucho más sencilla la implementación de su algoritmo. La razón por la cual no es fácilmente implementable con este ciclo se remite a su estructura, ya que este ciclo necesita un valor exacto de inicio, un valor exacto de finalización y un incremento y, en este caso, no es fácil encontrar estos tres valores.

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_1
Variables
        Entero: Numero, Cuenta_Digitos          // Declaración de Variables
Inicio
        Escriba "Digite un número"                  // Título de Aviso
        Lea Numero                                  // Lea un entero y guárdelo
                                                       en Numero
        Cuenta_Digitos = 0                      // Inicializa Cuenta_Digitos en 0
        Haga                                        // Inicia el ciclo
                Numero = Numero / 10                // Divida entre 10
                Cuenta_Digitos=Cuenta_Digitos+1     // y cuente
        Hasta que Numero = 0                    // Hasta que Numero sea igual a 0
        Escriba "Tiene", Cuenta_Digitos, "Dígitos"  // Escriba el resultado
Fin
```

Cabe anotar que la estructuración de un algoritmo utilizando un ciclo *Haga Hasta* implica razonar muy bien la condición del ciclo, ya que como puede verse es diferente a la condición utilizada en el ciclo *Mientras*.

Ciclo *Mientras*	*Mientras Numero < > 0*
Ciclo *Haga Hasta*	*Hasta que Numero = 0*

Normalmente, si realizamos dos soluciones para un mismo algoritmo pero en una de ellas utilizamos un ciclo *Mientras* y en otra utilizamos un ciclo *Haga Hasta*, podríamos decir que las condiciones deben ser contrarias (a nivel lógico).

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_1
Variables
        Entero : Numero, Cuenta_Digitos          // Declaración de Variables
Inicio
        Escriba "Digite un número"                        // Título de Aviso
        Lea Numero                          // Lea un entero y guárdelo en Numero
        Cuenta_Digitos = 0                  // Inicializa Cuenta_Digitos en 0
        Haga                                              // Inicia el ciclo
                Numero = Numero / 10                      // Divida entre 10
                Cuenta_Digitos = Cuenta_Digitos + 1       // y cuente
        Mientras Numero > 0                 // Mientras Numero sea mayor que 0
        Escriba "Tiene", Cuenta_Digitos, "Dígitos" // Escriba el resultado
Fin
```

Esta versión es muy aproximada a la anterior, pero la condición esta vez es diferente. Tenga en cuenta que utilizar cualquier estructura de ciclos requiere razonar muy bien la condición que se ha de usar para que, a nivel lógico, el algoritmo entregue los resultados esperados.

8.3.2. Ejemplo 2

Leer dos números enteros y mostrar todos los enteros comprendidos entre el menor y el mayor.

En este ejemplo, vamos a generar todos los números enteros comprendidos entre dos números leídos. Nótese que el algoritmo en ningún momento nos indica cuál es el mayor y cuál es el menor, lo cual significa que tenemos que averiguarlo antes de generar los números porque lo que el algoritmo sí dice es que deben escribirse **ascendentemente** (ya que reza *mostrar los enteros comprendidos entre el menor y el mayor*). Verifique si el objetivo es completamente claro para usted y de esta forma, ahora sí, puede revisar las soluciones aquí planteadas a este ejemplo.

Conceptualmente, primero vamos a detectar cuál de los dos números es el menor y cuál es el mayor. Luego haremos una variable *Auxiliar* igual al número menor y, a medida que la vayamos incrementando en 1, vamos a ir escribiendo su contenido hasta cuando esta variable alcance el valor del número mayor.

Usando ciclo *Mientras*

```
Programa Ejemplo_2
Variables
        Entero : Numero1, Numero2, Auxiliar          // Declaración de Variables
Inicio
        Escriba "Digite un Entero"          // Solicita el primer número
        Lea Numero1                         // Lo lee y lo almacena en Numero1

        Escriba "Digite otro Entero"        // Solicita el segundo número
        Lea Numero2                         // Lo lee y lo almacena en Numero2

        Si Numero1 < Numero2                // Si el primer número es el menor

                Auxiliar = Numero1                  //Haga Auxiliar igual a Numero1

                Mientras Auxiliar < = Numero2       //Mientras Auxiliar <=Numero2
                        Escriba Auxiliar            //Escriba el contenido de Auxiliar
                        Auxiliar = Auxiliar + 1     // e incremente Auxiliar
                Fin_Mientras
        Fin_Si

        Si Numero2 > Numero1                        // Si el 2º número es el menor
                Auxiliar = Numero2                  //Haga Auxiliar igual a Numero2
                Mientras Auxiliar < = Numero1       //Mientras Auxiliar <=Numero2
                        Escriba Auxiliar            //Escriba el contenido de Auxiliar
                        Auxiliar = Auxiliar + 1     // e incremente Auxiliar
                Fin_Mientras
        Fin_Si

        Si Numero1 = Numero2                        // Si los dos números son iguales
                Escriba "Los números son iguales"   // Avise que son iguales
Fin
```

Usando ciclo *Para*

```
Programa Ejemplo_2
Variables
        Entero : Num1, Num2, Aux                    // Declaración de variables
Inicio
```

```
        Escriba "Digite un Entero"          // Solicita el primero número
        Lea Num1                            // Lo lee y lo almacena en Num1

        Escriba "Digite otro Entero"        // Solicita el segundo número
        Lea Num2                            // Lo lee y lo almacena en Num2

        Si Num1 < Num2                      // Si Num1 es el menor
                // Haga que Aux vaya de Num1 a Num2 de 1 en 1
                Para Aux = Num1 hasta Num2 Paso 1
                        Escriba Auxiliar    // y escriba cada valor de Aux
        Fin_Si
        Si Num2 < Num1                      // Si Num2 es el menor
                // Haga que Aux vaya de Num2 a Num1 de 1 en 1
                Para Aux = Num2 hasta Num1 Paso 1
                        Escriba Auxiliar    // y escriba cada valor de Aux
        Fin_Si
        Si Num1 = Num2              // Si los dos números son iguales
                Escriba "Los números son iguales"   // Avise que son iguales
        Fin_Si
Fin
```

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_2
Variables
        Entero : Num1, Num2, Aux
Inicio
        Escriba "Digite un Entero"          // Solicita el primero número
        Lea Num1                            // Lo lee y lo almacena en Num1
        Escriba "Digite otro Entero"        // Solicita el segundo número
        Lea Num2                            // Lo lee y lo almacena en Num2
        Si Num1 < Num2                      // Si Num1 es el menor
                Aux = Num1                  // Inicie Aux con el valor de Num1
                Haga
                        Escriba Aux         // Escriba el contenido de Aux
                        Aux = Aux + 1       // e incremente dicho contenido
                Hasta que Aux = Num2        // Hasta que Aux alcance a Num2
        Fin_Si
```

```
    Si Num2 < Num1                          // Si Num2 es el menor
        Aux = Num2                          // Inicie Aux con el valor de Num2
        Haga
            Escriba Aux                     // Escriba el contenido de Aux
            Aux = Aux + 1                   // e incremente dicho contenido
        Hasta que Aux = Num1                // Hasta que Aux alcance a Num1
    Si Num1 = Num2                          // Si los dos números son iguales
        Escriba "Los números son iguales"   // Avise que son iguales
Fin
```

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_2
Variables
    Entero : Num1, Num2, Aux
Inicio
    Escriba "Digite un Entero"      // Solicita el primer número
    Lea Num1                        // Lo lee y lo almacena en Num1
    Escriba "Digite otro Entero"    // Solicita el 2º número
    Lea Num2                        // Lo lee y lo almacena en Num2
    Si Num1 < Num2                  // Si Num1 es el menor
        Aux = Num1                  // Inicie Aux con el valor de Num1
        Haga
            Escriba Aux             // Escriba el contenido de Aux
            Aux = Aux + 1           // e incremente dicho contenido
        Mientras Aux <= Num2        // Mientras que Aux sea <= Num2
    Fin_Si
    Si Num2 < Num1                  // Si Num2 es el menor
        Aux = Num2                  // Inicie Aux con el valor de Num2
        Haga
            Escriba Aux             // Escriba el contenido de Aux
            Aux = Aux + 1           // e incremente dicho contenido
        Mientras Aux <= Num1        // Mientras que Aux sea <= Num1
    Si Num1 = Num2                  // Si los dos números son iguales
        Escriba "Los números son iguales"   // Avise que son iguales
Fin
```

8.3.3. Ejemplo 3

Leer dos números enteros y determinar cuál de los dos tiene más dígitos.

Para la solución de este problema, podemos remitirnos al algoritmo que nos permitía saber cuántos dígitos tenía un número, ya que en el fondo este es casi lo mismo. Fundamentalmente, el objetivo de este es contar los dígitos que tiene un número, luego contar los dígitos que tiene otro número y comparar ambos resultados para decir cuál de los dos tiene más dígitos.

Usando ciclo *Mientras*

```
Programa Ejemplo_3
Variables
        Entero : Num1,          // Almacenará el primer número
                 Num2,          // Almacenará el segundo número
                 Aux1,          // Almacenará provisionalmente a Num1
                 Aux2,          // Almacenará provisionalmente a Num2
                 ContDig1,      // Almacenará cantidad de díg de Num1
                 ContDig2       // Almacenará cantidad de díg de Num2
Inicio
        Escriba "Digite un entero"      // Solicita un dato entero
        Lea Num1                        // Lo lee y lo almacena en Num1

        Escriba "Digite otro entero"    // Solicita otro dato entero
        Lea Num2                        // Lo lee y lo almacena en Num2

        Aux1 = Num1                     // Guarde en Aux1 el contenido de Num1
        ContDig1 = 0                    // Inicialice ContDig1 en 0

        Mientras Aux1 < > 0             // Mientras Aux1 sea difte. de cero
                Aux1 = Aux1 / 10                // Divida Aux1 entre 10
                ContDig1 = ContDig1 + 1         // y cuente
        Fin_Mientras
        Aux2 = Num2                     // Guarde en Aux2 el contenido de Num2
        ContDig2 = 0                    // Inicialice ContDig2 en 0

        Mientras Aux2 < > 0             // Mientras Aux2 sea difte. de cero
                Aux2 = Aux2 / 10        // Divida Aux2 entre 10
                ContDig2 = ContDig2 + 1         // y cuente
```

```
    Fin_Mientras

    Si ContDig1 > ContDig2 // Si el primer número tiene más dígitos
        Escriba Num1, "tiene más dígitos que", Num2          // Avise

    Si ContDig1 < ContDig2 // Si el segundo número tiene más dígitos
        Escriba Num2, "tiene más dígitos que", Num1          // Avise

    Si ContDig1 = ContDig2 // Si los dos números tienen la misma cantidad
                           // de dígitos, entonces avise

        Escriba Num1, "tiene la misma cantidad de dígitos que", Num2
Fin
```

Como puede ver, nos hemos basado en el algoritmo que determinaba cuántos dígitos tenía un número para desarrollar este algoritmo.

Usando ciclo *Para*

Este algoritmo no es implementable con un ciclo *Para* por las mismas razones que no era implementable el algoritmo en el cual nos basamos para desarrollar esta solución. Debo anotar que en casos muy particulares como el lenguaje C este algoritmo es implementable pero debido a que la filosofía de este lenguaje en cuanto al ciclo *Para* es un poquito diferente y además porque las "herramientas" de programación que brinda este lenguaje son un poco más flexibles que en otros lenguajes.

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_3
Variables
    Entero : Num1,          // Almacenará el primer número
             Num2,          // Almacenará el segundo número
             Aux1,          // Almacenará provisionalmente a Num1
             Aux2,          // Almacenará provisionalmente a Num2
             ContDig1,      // Almacenará la cant de dígitos de Num1
             ContDig2       // Almacenará la cant de dígitos de Num2
Inicio
    Escriba "Digite un entero"      // Solicita un dato entero
    Lea Num1                        // Lo lee y lo almacena en Num1

    Escriba "Digite otro entero"    // Solicita otro dato entero
    Lea Num2                        // Lo lee y lo almacena en Num2
```

```
Aux1 = Num1                    // Guarde en Aux1 el contenido de Num1
ContDig1 = 0                   // Inicialice ContDig1 en 0

Haga
        Aux1 = Aux1 / 10       // Divida Aux1 entre 10
        ContDig1 = ContDig1 +1 // y cuente
Hasta que Aux1 = 0             // Hasta que Aux1 sea igual a 0

Aux2 = Num2                    // Guarde en Aux2 el contenido de Num2
ContDig2 = 0                   // Inicialice ContDig2 en 0
Haga
        Aux2 = Aux2 / 10       // Divida Aux2 entre 10
        ContDig2 = ContDig2 +1 // y cuente
Hasta que Aux2 = 0             // Hasta que Aux2 sea igual a 0

Si ContDig1 > ContDig2 // Si el primer número tiene más dígitos
Escriba Num1, "tiene más dígitos que", Num2                    // Avise
Si ContDig1 < ContDig2 // Si el segundo número tiene más dígitos
        Escriba Num2, "tiene más dígitos que", Num1            // Avise

Si ContDig1 = ContDig2 // Si los dos números tienen la misma
                       // cantidad de dígitos
        Escriba Num1, "tiene la misma cantidad de dígitos que", Num2  // Avise
Fin
```

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_3
Variables
    Entero : Num1,          // Almacenará el primer número
             Num2,          // Almacenará el segundo número
             Aux1,          // Almacenará provisionalmente a Num1
             Aux2,          // Almacenará provisionalmente a Num2
             ContDig1,      // Almacenará la cant de dígitos de Num1
             ContDig2       // Almacenará la cant de dígitos de Num2
Inicio
    Escriba "Digite un entero"     // Solicita un dato entero
    Lea Num1                       // Lo lee y lo almacena en Num1

    Escriba "Digite otro entero"   // Solicita otro dato entero
```

```
Lea Num2                        // Lo lee y lo almacena en Num2

Aux1 = Num1                     // Guarda en Aux1 el contenido de Num1
ContDig1 = 0                    // Inicialice ContDig1 en 0

Haga
        Aux1 = Aux1 / 10        // Divida Aux1 entre 10
        ContDig1 = ContDig1 + 1 // y cuente
Mientras Aux1 < > 0             // Mientras Aux1 sea diferente de cero

Aux2 = Num2                     // Guarda en Aux2 el contenido de Num2
ContDig2 = 0                    // Inicialice ContDig2 en cero

Haga
        Aux2 = Aux2 / 10        // Divida Aux2 entre 10
        ContDig2 = ContDig2 + 1 // y cuente
Mientras Aux2 < > 0             // mientras Aux2 sea diferente de cero

Si ContDig1 > ContDig2 // Si el primer número tiene más dígitos
        Escriba Num1, "tiene más dígitos que", Num2          // Avise

Si ContDig1 < ContDig2 // Si el segundo número tiene más dígitos
        Escriba Num2, "tiene más dígitos que", Num1          // Avise

Si ContDig1 = ContDig2 // Si los 2 núms tienen la misma cantidad
                       // de dígitos
        Escriba Num1, "tiene la misma cantidad de dígitos que", Num2  // Avise
Fin
```

8.3.4. Ejemplo 4

Leer números enteros hasta que digiten 0 y determinar a cuánto es igual el promedio de los números leídos que hayan sido positivos.

De nuevo, y al igual que en todos los ejercicios, vamos a clarificar el objetivo. Necesitamos leer números enteros y a medida que vamos leyendo estos números los vamos contando y los vamos acumulando en sendas variables. Cuando digiten el número 0, eso significará que en ese momento debemos obtener el resultado entero de dividir el resultado de haber acumulado todos los números entre el resultado de haberlos contado, es decir, lo que vamos a dividir es la sumatoria de números entre la cantidad de números y ese resultado es lo que debemos mostrar en pantalla porque es el dato que nos están solicitando.

Usando ciclo *Mientras*

```
Programa Ejemplo_4
Variables
    Entero : Num,                   // Almacenará cada uno de los
                                    // números leídos
             Acum,                  // Almacenará la suma de los números
                                    // leídos diferentes de cero
             Cont                   // Almacenará la cantidad de números
                                    // leídos diferentes de cero
    Real :   Promedio               // Almacenará el resultado de dividir la
                                    // sumatoria de números entre la
                                    // cantidad de números
Inicio
    Acum = 0                        // Inicialice el acumulador en cero
    Cont = 0                  // Inicialice el contador en cero

    Escriba "Digite enteros y finalice con 0" // Aviso para solicitar los núms
    Lea Num                         // Lea el primer número
    Mientras Num < > 0              // Mientras los números que entren
                                    // sean diferentes de cero
        Si Num > 0                  // Si el último número leído es positivo
            Acum = Acum + Num       // Acumule el último número leído
            Cont = Cont + 1         // y cuéntelo
        Fin_Si

        Lea Num                     // Lea un nuevo número
    Fin_Mientras                    // Fin del ciclo

    Promedio = Acum / Cont// Calcule el promedio
    Escriba "El promedio es", Promedio  // Muestre el promedio en pantalla
Fin
```

Hemos utilizado la primera orden *Lea Num* para recibir el primer número y con esto poder entrar al ciclo.

Usando ciclo *Para*

Si se supiera con exactitud cuántos números se van a leer, entonces sí sería implementable este algoritmo con el ciclo *Para*, pero como el algoritmo dice

que *Hasta que digiten 0*, entonces este algoritmo no es implementable con un ciclo *Para*.

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_4
Variables
        Entero : Num,              //Almacenará cada uno de los
//números leídos
                        Acum,  //Almacenará la suma de los números
//leídos diferentes de cero
                        Cont    //Almacenará la cantidad de números
//leídos diferentes de cero
        Real :          Promedio        //Almacenará el resultado de dividir la
//sumatoria de números entre la
//cantidad de números
Inicio
        Acum = 0                        //Inicialice el acumulador en cero
        Cont = 0                        //Inicialice el contador en cero
Escriba "Digite enteros y finalice con 0" //Aviso para solicitar los núms
        Lea Num                                 //Lea el primer número
        Haga
                Acum = Acum + Num       //Acumule el último número leído
                Cont = Cont + 1         //Cuéntelo
                Lea Num                 //Lea el siguiente número
        Hasta que Num = 0               //Hasta que el último número leído sea
//igual a 0
        Promedio = Acum / Cont//Calcule el promedio
        Escriba "El promedio es", Promedio      //Muestre el promedio en
pantalla
Fin
```

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_4
Variables
        Entero : Num,              //Almacenará cada uno de los
```

```
// números leídos
                    Acum,          // Almacenará la suma de los números
// leídos diferentes de cero
                    Cont           // Almacenará la cantidad de números
// leídos diferentes de cero
    Real:           Promedio       // Almacenará el resultado de dividir la
// sumatoria de números entre la
// cantidad de números
Inicio
    Acum = 0                       // Inicialice el acumulador en cero
    Cont = 0                       // Inicialice el contador en cero
Escriba "Digite enteros y finalice con 0" // Aviso para solicitar los núms
    Lea Num                        // Lea el primer número
    Haga
        Acum = Acum + Num          // Acumule el último número
leído
        Cont = Cont + 1            // Cuéntelo
        Lea Num                    // Lea el siguiente número
    Mientras Num < > 0             // Mientras el último número leído sea
// diferente de 0
    Promedio = Acum / Cont// Calcule el promedio
    Escriba "El promedio es", Promedio   // Muestre el promedio en
pantalla
Fin
```

8.3.5. Ejemplo 5

Leer un número entero y calcular su factorial.

Primero que nada vamos a ver qué es el factorial de un número. Se define como factorial de un número N cualquiera el resultado de multiplicar sucesivamente todos los enteros comprendidos entre 1 y ese número N. No se aplica esta definición cuando dicho número N es negativo y, en caso de que ese número N sea 0, se asume que el factorial de 0 es 1. De esta forma, el factorial de 6 es el resultado de multiplicar 1 * 2 * 3 * 4 * 5 * 6, lo cual nos da como resultado 720; igualmente, el factorial de 3 es igual a multiplicar 1 * 2 * 3, que es igual a 6; el factorial de 0 es 1 y el factorial de –8 no está definido o mejor no está incluido en la definición.

Ya con esta definición lo que tendremos que hacer es implementarla a nivel de un algoritmo. Para ello, primero que nada vamos a leer dicho número N y validando que dicho número sea positivo, utilizando otra variable a manera de contador, vamos a generar la secuencia de números enteros comprendida entre 1 y N, solo que a medida que la vayamos generando vamos a ir multiplicando los números generados, resultado que será almacenado en una variable *Facto* que será la que al final se mostrará en pantalla para cumplir con el objetivo del enunciado.

Usando ciclo *Mientras*

```
Programa Ejemplo_5
Variables
        Entero : N,             // Almacenará el número leído que es el
                                // número al cual se le quiere calcular el
                                // factorial
                 Cont,          // Esta variable es la que nos va a permitir
                                // generar la secuencia de todos los enteros
                                // comprendidos entre 1 y el número leído (a
                                // manera de contador)
                 Facto          // Almacenará el resultado del factorial
Inicio
        Escriba "Digite un número entero"       // Solicita un número entero
        Lea N                                   // Lo lee y lo almacena en N

        Si N < 0                // Si el número es negativo entonces avisa
                Escriba "El factorial no está definido para números negativos"

        Facto = 1                               // Inicializa el factorial en 1
        Cont = 1                                // Inicializa el contador en 1
        Mientras Cont < = N             // Mientras el contador sea menor que el
                                        // número leído
                Facto = Facto * Cont    // Multiplique Facto por cada valor
                                        // que tome Cont
                Cont = Cont + 1         // Incremente el valor de Cont
        Fin_Mientras                    // Fin del ciclo

        Escriba "El factorial de", N, "es", Facto       // Escriba el resultado
Fin
```

Usando ciclo *Para*

```
Programa Ejemplo_5
Variables
        Entero : N,             // Almacenará el número leído que es el
                                // número al cual se le quiere calcular el
                                // factorial
                Cont,           // Esta variable es la que nos va a permitir
                                // generar la secuencia de todos los enteros
                                // comprendidos entre 1 y el número leído (a
                                // manera de contador)
                Facto           // Almacenará el resultado del factorial
Inicio
        Escriba "Digite un número entero"       // Solicita un número entero
        Lea N                                   // Lo lee y lo almacena en N

        Si N < 0                // Si el número es negativo entonces avisa
                Escriba "El factorial no está definido para números negativos"

        Facto = 1                       // Inicializa el factorial en 1

        Para Cont = 1 hasta N (Paso 1)  // Genera la secuencia de núms enteros
                                        // desde 1 hasta el número leído de 1 en 1
                                        // y cada valor lo va almacenando en Cont
                Facto = Facto * Cont    // Multiplique Facto por valor
                                        // que tome Cont
        Fin_Para                        // Fin Ciclo Para
        Escriba "El factorial de", N, "es", Facto // Escriba el factorial
Fin
```

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_5
Variables
        Entero : N,             // Almacenará el número leído que es el
                                // número al cual se le quiere calcular el
                                // factorial
                Cont,           // Esta variable es la que nos va a permitir
                                // generar la secuencia de todos los enteros
```

```
                                    // comprendidos entre 1 y el número leído (a
                                    // manera de contador)
                          Facto     // Almacenará el resultado del factorial
Inicio
    Escriba "Digite un número entero"       // Solicita un número entero
    Lea N                                   // Lo lee y lo almacena en N

    Si N < 0        // Si el número es negativo entonces avisa
        Escriba "El factorial no está definido para números negativos"

    Facto = 1                   // Inicializa el factorial en 1
    Cont = 1                    // Inicializa el contador en 1
    Haga
        Facto = Facto * Cont    // Multiplique Facto por cada valor
                                // que vaya tomando Cont
        Cont = Cont + 1         // Incremente el valor de Cont
    Hasta que Cont > N          // Hasta que el valor almacenado en Cont sea
                                // mayor que el valor almacenado en N

    Escriba "El factorial de", N, "es", Facto       // Escriba el resultado
Fin
```

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_5
Variables
    Entero : N,             // Almacenará el número leído que es el
                            // número al cual se le quiere calcular el
                            // factorial
                  Cont,     // Esta variable es la que nos va a permitir
                            // generar la secuencia de todos los enteros
                            // comprendidos entre 1 y el número leído (a
                            // manera de contador)
                  Facto     // Almacenará el resultado del factorial
Inicio
    Escriba "Digite un número entero"       // Solicita un número entero
    Lea N                                   // Lo lee y lo almacena en N
```

```
    Si N < 0          // Si el número es negativo entonces avisa
        Escriba "El factorial no está definido para números negativos"

    Facto = 1         // Inicializa el factorial en 1
    Cont = 1          // Inicializa el contador en 1
Haga
        Facto = Facto * Cont    // Multiplique Facto por cada valor
                                // que vaya tomando Cont
        Cont = Cont + 1         // Incremente el valor de Cont
    Mientras Cont < = N   // Mientras que el valor almacenado en Cont
                          // sea menor o igual al valor almacenado en N
    Escriba "El factorial de", N, "es", Facto   // Escriba el resultado
Fin
```

8.3.6. Ejemplo 6

Leer un número (asumir que es una base) y leer otro número (asumir que es un exponente) y elevar dicha base a dicho exponente.

Al igual que en todos los casos anteriores, lo que necesitamos esta vez es leer dos números (asumiremos que el primero es una base y que el segundo es un exponente). Se trata de elevar dicha base a dicho exponente. Recordemos pues que si la base fuera 5 y el exponente fuera 4 entonces tendríamos que calcular a cuánto es igual 5^4, o sea, 5 * 5 * 5 * 5, que quiere decir multiplicar el número 5 cuatro veces.

Por lo tanto, necesitaremos que una variable actúe como contador para que nos permita controlar las veces que la base se va a multiplicar. La restricción fundamental que hemos de tener en cuenta es que el exponente sea positivo, ya que b n es diferente de b –n.

Usando ciclo Mientras

```
Programa Ejemplo_6
Variables
    Entero: Base,       // Almacenará el número que se tomará como
                        // base
            Exp,        // Almacenará el número que se tomará como
                        // exponente
            Aux,        // Variable que va a almacenar
```

```
                                    // progresivamente todos los valores
                                    // comprendidos entre 1 y el valor del
                                    // exponente
                          Resul     // Variable que almacenará el resultado final de
                                    // haber multiplicado la base tantas veces como
                                    // lo diga el exponente
Inicio
Escriba "Digite una base"           // Solicita una base
    Lea Base                        // y la lee

    Escriba "Digite un exponente"   // Solicita un exponente
    Lea Exp                         // y lo lee

    Si Exp < 0                      // En caso de que el exponente sea negativo
        Escriba "Exponente Errado"  // Avisa
    Sino                            // Si el exponente es cero o es positivo
        Resul = 1                   // Inicializa Resul en 1
        Aux = 1                     // Inicializa la auxiliar en 1
        Mientras Aux <= Exp         // Mientras contenido de variable auxiliar
                                    // sea menor o igual que el contenido de la
                                    // variable que almacena el exponente
            Resul = Resul * Base    // Multiplica
            Aux = Aux + 1           // e incremente el valor de la auxiliar
        Fin_Mientras
        Escriba "Resultado =", Resul  // Muestra el resultado
Fin
```

Usando ciclo *Para*

```
Programa Ejemplo_6
Variables
    Entero : Base,          // Almacenará el número que se tomará como
                            // base
             Exp,           // Almacenará el número que se tomará como
                            // exponente
             Aux,           // Variable que va a almacenar
                            // progresivamente todos los valores
```

```
                                    // comprendidos entre 1 y el valor del
                                    // exponente
                            Resul   // Variable que almacenará el resultado final de
                                    // haber multiplicado la base tantas veces como
                                    // lo diga el exponente
Inicio
        Escriba "Digite una base"           // Solicita una base
        Lea Base                            // y la lee

        Escriba "Digite un exponente"       // Solicita un exponente
        Lea Exp                     // y lo lee

        Si Exp < 0                  // En caso de que el exponente sea negativo
                Escriba "Exponente Errado"  // Avisa
        Sino                        // Si el exponente es cero o es positivo
                Resul = 1                   // Inicializa Resul en 1
                // Inicia un ciclo desde 1 hasta el valor
                // que tiene almacenado la variable Exp
                // guardando cada valor en la variable Aux

                Para Aux = 1 hasta Exp ( Paso 1)
                        Resul = Resul * Base    // Por cada valor de Aux multiplica
                                                // lo que contenga la variable Resul por lo
                                                // que almacena la variable Base
                Fin_Para

                Escriba "Resultado =", Resul    // Muestra en pantalla el resultado
Fin
```

Usando ciclo *Haga Hasta*

```
Programa Ejemplo_6
Variables
        Entero : Base,          // Almacenará el número que se tomará como
                                // base
                 Exp,           // Almacenará el número que se tomará como
                                // exponente
                 Aux,           // Variable que va a almacenar
                                // progresivamente todos los valores
```

```
                                    // comprendidos entre 1 y el valor del
                                    // exponente
                          Resul     // Variable que almacenará el resultado final de
                                    // haber multiplicado la base tantas veces como
                                    // lo diga el exponente
Inicio
    Escriba "Digite una base"               // Solicita una base
    Lea Base                                // y la lee
    Escriba "Digite un exponente"           // Solicita un exponente
    Lea Exp                         // y lo lee
    Si Exp < 0                  // En caso de que el exponente sea negativo
        Escriba "Exponente Errado"          // Avisa
    Sino                                // Si el exponente es cero o es positivo
        Resul = 1                       // Inicializa Resul en 1
        Aux = 1                         // Inicializa la auxiliar en 1
        Haga
            Resul = Resul * Base        // Multiplica
            Aux = Aux + 1               // e incrementa el valor de Aux
        Hasta que Aux > Exp     // Hasta que Aux sea mayor que Exp
        Escriba "Resultado =", Resul    // Escriba el resultado final
Fin
```

Usando ciclo *Haga Mientras*

```
Programa Ejemplo_6
Variables
    Entero : Base,          // Almacenará el número que se tomará como
                            // base
             Exp,           // Almacenará el número que se tomará como
                            // exponente
             Aux,           // Variable que va a almacenar
                            // progresivamente todos los valores
                            // comprendidos entre 1 y el valor del
                            // exponente
             Resul          // Variable que almacenará el resultado final de
```

```
                                    // haber multiplicado la base tantas veces como
                                    // lo diga el exponente
Inicio
    Escriba "Digite una base"               // Solicita una base
    Lea Base                                // y la lee

    Escriba "Digite un exponente"           // Solicita un exponente
    Lea Exp                         // y lo lee

    Si Exp < 0              // En caso de que el exponente sea negativo
        Escriba "Exponente Errado"      // Avisa
    Sino                    // Si el exponente es cero o es positivo
        Resul = 1                       // Inicializa Resul en 1
        Aux = 1                         // Inicializa la auxiliar en 1
        Haga
            Resul = Resul * Base        // Multiplica
            Aux = Aux + 1               // e incrementa el valor de Aux
        Mientras Aux <= Exp     // Mientras que Aux sea <= que Exp

        Escriba "Resultado =", Resul    // Escriba el resultado final
Fin
```

A esta altura del libro, usted podrá ver la gran utilidad que tiene la implementación de ciclos en nuestros algoritmos dado que ellos facilitan no solo la comprensión de la solución, sino que además nos permiten obtener unos algoritmos muy flexibles a los cambios, pues con solo cambiar el valor de inicio del ciclo o el valor final del ciclo o el incremento del ciclo, con eso es suficiente para que ese mismo ciclo pueda cumplir de una manera eficiente otros objetivos.

8.4. Ciclos anidados

Una variante muy interesante en cuanto a la concepción de ciclos corresponde a los ciclos anidados. Se define como ciclos anidados la estructura en la cual un ciclo está dentro de otro (completamente). Esto significa que, si se tratara de una estructura *Mientras* anidada, su esquema sería el siguiente:

```
Mientras Condición1
    .
    .
```

```
        Mientras Condición2
                .
                .
                Cuerpo del ciclo más interno
                .
                .
        Fin_Mientras Interno
        .
        .
Fin_Mientras Externo
```

Su forma de ejecución es muy sencilla: mientras sea verdadera la *condición1*, el ciclo externo va a ejecutar el cuerpo del ciclo dentro del cual se encuentra un ciclo interno que está regido por la *condición2*, de tal manera que se ejecutará completamente el ciclo interno mientras la *condición2* sea verdadera. Cuando esta *condición2* sea falsa, se ejecutarán el resto de instrucciones del ciclo externo y se volverá a la *condición1* para evaluar si aún es verdadera para volver a entrar en el cuerpo de dicho ciclo. Veamos esta teoría aplicada con un ejemplo al cual le haremos su respectiva prueba de escritorio.

8.4.1. Ejemplo 1

Leer números hasta que digiten 0 y a cada valor leído calcularle su factorial.

Clarificación del objetivo

No se olvide que la solución presentada en este libro es solo una de las posibles soluciones a cada uno de los enunciados planteados como objetivos. Si usted ha desarrollado soluciones para estos algoritmos y encuentra que son diferentes no se preocupe. Realícele pruebas de escritorio y si cumplen con el objetivo entonces su algoritmo estará tan bien como el mío.

Para solucionar este algoritmo, vamos a tener en cuenta la solución ya presentada para calcular el factorial de un número. En esencia, nuestro algoritmo va a leer números hasta que digiten cero y, mientras esta condición sea verdadera, se calculará el factorial (o sea, se aplicará el algoritmo ya resuelto de cálculo del factorial de un número) a cada uno de los números leídos.

No se olvide que el factorial de un entero es el resultado de multiplicar sucesivamente todos los enteros comprendidos entre 1 y el número dado. No está definido el factorial para números negativos y el factorial de 0 es 1.

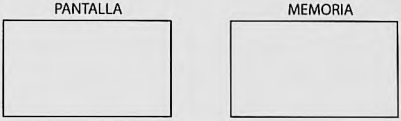
Algoritmo

```
Programa Ciclos_Anidados_1
Variables
        Entero : Num,           // Almacenará cada uno de los números
                                // diferentes de cero que sean digitados
                 Facto,         // Almacenará el factorial de cada uno de los
                                // números digitados
                 Cont           // Almacenará la secuencia de enteros
                                // comprendidos entre 1 y el último número
                                // leído, toda vez que este último número sea
                                // positivo
Inicio
        Escriba "Digite números y finalice con 0" // Solicitar números
        Lea Num                         // Lea el primer dato

        Mientras Num < > 0              // Mientras el dato leído sea diferente de 0
                Si Num < 0              // Si el dato leído es negativo avise
                        Escriba "No está definido el factorial para núms negativos"

                Sino                    // Sino
                        Facto = 1       // Inicialice la variable Facto en 1
                        // Con un ciclo Para almacene sucesivamente en la
                        // variable Cont todos los enteros comprendidos entre 1
                        // y el número leído

                        Para Cont = 1 hasta Num (paso 1)
                        Facto = Facto * Cont    // Multiplique progresivamente
                                                // cada uno de los valores
                                                // enteros comprendidos entre 1
                                                // y el número leído
                        Escriba "Factorial de", Num, "=", Facto
                        // Escriba resultado
                Fin_Si
                Lea Num                         // Lea el siguiente número
        Fin_Mientras                            // Fin Ciclo Mientras
Fin
```

Note usted que la utilización de ciclos anidados no tiene restricciones en cuanto a los ciclos que se utilicen. En este ejemplo, vemos como dentro de un ciclo *Mientras* puede ir sin ningún problema un ciclo *Para*. En general, usted en un algoritmo puede combinar cualquier tipo de ciclos.

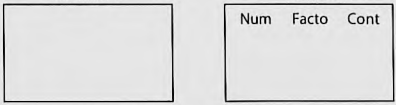
Prueba de escritorio

De nuevo y tal como lo hicimos en ejemplos anteriores, vamos a desarrollar una prueba de escritorio (paso a paso) con el objetivo de entender claramente el concepto de los ciclos anidados. Inicialmente, tendremos las mismas dos partes para la prueba de escritorio que hemos tenido en todas ellas: la pantalla y la memoria.



Comenzamos la prueba de escritorio con la declaración de las variables en memoria.

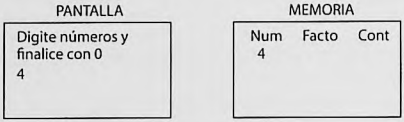
```
Programa Ciclos_Anidados_1
Variables
    Entero : Num,          // Almacenará cada uno de los números
                           // diferentes de cero que sean digitados
             Facto,        // Almacenará el factorial de cada uno de los
                           // números digitados
             Cont          // Almacenará la secuencia de enteros
                           // comprendidos entre 1 y el último número
                           // leído, toda vez que este último número sea
                           // positivo
```



En la parte de ejecución, nuestro algoritmo comienza avisándole al usuario para que ingrese unos datos y sepa cómo finalizar. A continuación, lee el

primer entero y lo almacena en la variable *Num*. Vamos a asumir que el primer entero digitado es un 4.

```
Inicio
    Escriba "Digite números y finalice con 0"      // Solicita números
    Lea Num                                        // Lea el primer dato
```



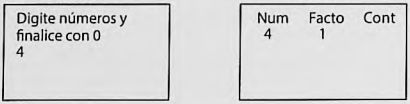
Como primera medida, con el número leído se verifica que sea diferente de cero y, como esta proposición es verdadera, se procede a entrar en el ciclo.

```
    Mientras Num < > 0      // Mientras el dato leído sea diferente de 0
```

Se verifica igualmente si el número leído es menor que cero (o sea si es negativo). Como el número 4 no es negativo, entonces se procede a realizar el conjunto de instrucciones que se encuentran por el lado del *Sino*.

```
        Si Num < 0
            Escriba "No está definido factorial para núms negativos"
        Sino
            Facto = 1
```

De acuerdo a esto, inicializamos la variable *Facto* con el valor de 1 para iniciar el ciclo *Para* que es el que realmente nos va a permitir calcular el factorial.



El ciclo *Para* planteado tiene como índice a la variable *Cont* y le asigna inicialmente el valor de 1. El contenido de esta variable deberá llegar hasta el valor actual de *Num* (o sea, 4) incrementándola cada vez de 1 en 1.

PANTALLA

```
Digite números y
finalice con 0
4
```

MEMORIA

Num	Facto	Cont
4	1	1

Para Cont = 1 hasta Num (paso 1)
*Facto = Facto * Cont*

Y por cada valor que vaya tomando la variable *Cont* vamos realizando la operación asignación *Facto = Facto * Cont*. De esta manera, las asignaciones sucesivas serán:

Cuando *Cont* vale 1, entonces *Facto* vale 1.

PANTALLA

```
Digite números y
finalice con 0
4
```

MEMORIA

Num	Facto	Cont
4	~~1~~	1
	1	

Al incrementar el valor de *Cont* en 1, entonces *Cont* queda valiendo 2 y, por lo tanto, *Facto* es igual a 2.

PANTALLA

```
Digite números y
finalice con 0
4
```

MEMORIA

Num	Facto	Cont
4	~~1~~	~~1~~
4	~~1~~	2
	2	

Al incrementar el valor de *Cont* en 1, entonces *Cont* queda ahora valiendo 3 y, por lo tanto, *Facto* es igual a 6 puesto que *Facto = Facto * Cont*.

PANTALLA

```
Digite números y
finalice con 0
4
```

MEMORIA

Num	Facto	Cont
4	~~1~~	~~1~~
4	~~1~~	~~2~~
4	~~2~~	3

Al incrementar el valor de *Cont* (de nuevo) en 1, su actual contenido pasa a ser 4. Recuerde usted que el ciclo decía inicialmente *Para Cont = 1 hasta Num*, eso significa que como *Num* vale 4 y en este momento *Cont* también vale 4, entonces esta es la última iteración del ciclo *Para* planteado, o sea, que es la última vez que hacemos *Facto = Facto * Cont*. Con lo que acabo de decir pasaríamos a la siguiente instrucción dentro del ciclo *Mientras* externo. Es importante que note que hemos desarrollado el ciclo *Para* y que, aunque ya casi terminamos de ejecutarlo, no hemos terminado la ejecución del ciclo externo *Mientras*.

PANTALLA

```
Digite números y
finalice con 0
4
```

MEMORIA

Num	Facto	Cont
4	~~1~~	~~1~~
4	~~1~~	~~2~~
4	~~2~~	~~3~~
4	6	4
4	24	

De esta forma, se ejecutaría la siguiente orden que se encuentra después del ciclo. En las órdenes *Escriba* no se olvide que en donde aparezcan nombres de variables se coloca el contenido actual de ellas:

Escriba "El factorial de", Num, "es", Facto
Fin_Si

PANTALLA

```
Digite números y
finalice con 0
4
El factorial de 4 es 24
```

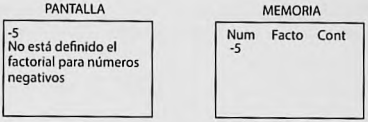
MEMORIA

Num	Facto	Cont
4	~~1~~	~~1~~
4	~~1~~	~~2~~
4	~~2~~	~~3~~
4	6	4
4	24	

Resultado que efectivamente es cierto, con lo cual terminamos la decisión pasando a la siguiente instrucción que se encuentra después de ella y que corresponde a leer otro entero. Luego de lo cual volveríamos a evaluar la condición del ciclo *Mientras* para saber si ya se ha terminado o no.

Lea Num
Fin_Mientras

Y para el caso de la prueba de escritorio que estamos desarrollando, supongamos que el nuevo valor leído es –5 que, por la orden dada, quedaría almacenado en la variable *Num*.



He borrado los contenidos sucesivos de las variables *Facto* y *Cont* solo por hacer más clara la prueba de escritorio a partir del momento en que entra un nuevo valor. Esto no tiene ninguna implicación en el desarrollo de ESTA prueba de escritorio, ya que al leer un nuevo valor (según el algoritmo) se repite todo el proceso con él. Tenga usted mucho cuidado cuando vaya a borrar deliberadamente contenidos de variables. Le sugiero que mejor nunca lo haga. En este libro se hace por cuestión de espacio, no por otra razón, pero admito que no es lo más conveniente.

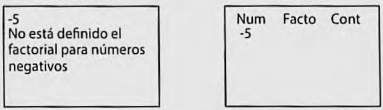
Con este valor, volvemos a evaluar la condición que aparece en el inicio del ciclo, o sea,

Mientras Num < > 0

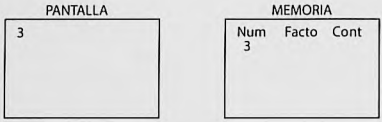
Como vemos que el valor almacenado en la variable *Num* es diferente de cero (porque es igual a –5), entonces volvemos a ejecutar el cuerpo del ciclo. Lo primero que encontramos es una decisión acerca de si el número es negativo (o sea, menor que cero). Como puede verse, el contenido de la variable *Num* en este momento es –5, lo cual indica que la decisión es verdadera.

Si Num < 0
Escriba "No está definido factorial para núms negativos"

Por lo tanto, se ejecuta la orden *Escriba* que se encuentra por el lado verdadero de la decisión y se ignora el correspondiente *Sino*.



De esta manera, la ejecución del algoritmo pasará a la siguiente instrucción que se encuentra después del final de la decisión (o sea, inmediatamente después del *Fin_Si*) y que es *Lea Num*, o sea, que lea otro número entero. Esta vez supongamos que el número leído es 3.



De esta forma, con el nuevo valor almacenado en la variable *Num*, volvemos a evaluar la condición del ciclo, o sea,

Mientras Num < > 0

Como el valor actual de *Num* es 3, entonces volvemos a entrar al ciclo (externo). Note usted que aún no nos hemos salido del ciclo externo y que, si el número leído es positivo, tendremos que entrar también al ciclo interno.

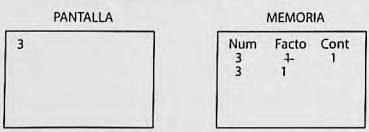
Si Num < 0
Escriba "No está definido factorial para núms negativos"

La primera decisión que se encuentra es si el número es negativo. Como vemos que *Num* almacena un número 3 y el 3 no es negativo, entonces se ejecutarán las órdenes que se encuentran por el *Sino* de la decisión.

Sino
Facto = 1

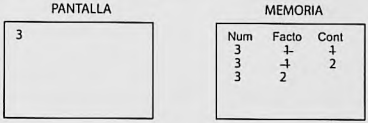
Ya sabemos que inicializamos la variable *Facto* con el valor de 1 y así mismo hacemos con la variable *Cont*, que según el ciclo *Para* planteado irá desde 1 hasta el valor actual de *Num* (o sea, hasta 3) de 1 en 1 para realizar progresivamente la operación *Facto = Facto * Cont*.

Para Cont = 1 hasta Num (paso 1)
*Facto = Facto * Cont*

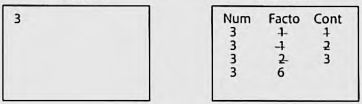


Nos quedamos en este ciclo *Para* realizando las iteraciones que sean necesarias hasta cuando *Cont* llegue al valor actual de *Num*. De esta manera,

incrementamos el valor de *Cont* en 1 y queda valiendo 2, razón por la cual la operación *Facto = Facto * Cont* es igual a 2.



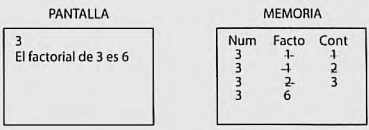
Al incrementar el valor de *Cont* en 1 (continuando con la ejecución del ciclo *Para*), obtenemos en la variable *Cont* el valor 3, que es el tope al cual debía llegar puesto que el ciclo estaba estructurado para que *Cont* fuera desde 1 hasta el valor actual de *Num* que es 3. Por lo tanto, esta es la última iteración del ciclo *Para* (interno).



Como ya se llegó al tope planteado por el ciclo para entonces pasamos a la siguiente instrucción, o sea,

Escriba "El factorial de", Num, "es", Facto

Con lo cual aparecería en pantalla:



Lo cual vemos que es cierto, pues el factorial de 3 realmente es 6. Con esta orden, llegamos al final de la decisión, o sea, al *Fin_Si*, pero no hemos terminado la prueba de escritorio pues después de este *Fin_Si* ejecutamos la orden *Lea Num*, para lo cual vamos a asumir que esta vez digitan el número 0.

PANTALLA
3
El factorial de 3 es 6
0

MEMORIA		
Num	Facto	Cont
3	1	1
3	1	2

Con lo cual volvemos a evaluar la condición que se encuentra en el ciclo *Mientras* y que dice:

Mientras Num < > 0

Como vemos que esta vez el valor almacenado en *Num* es 0 (o sea, que no es diferente de cero), entonces esto quiere decir que la condición es falsa y, por lo tanto, nos salimos hasta la siguiente instrucción después del *Fin_Mientras*, que es:

Fin

Y que representa que hemos llegado al final de esta prueba de escritorio. Vemos que los resultados obtenidos en pantalla en las diferentes ejecuciones fueron:

Digite números y finalice con 0
4
El factorial de 4 es 24
-5
No está definido el factorial para números negativos
3
El factorial de 3 es 6
0

Realizando manualmente las operaciones, podemos ver que el factorial de 4 realmente es 24, que no está definido el factorial para números negativos (como por ejemplo el –5), que el factorial de 3 es 6 y que cuando se digitó cero se finalizó tal como se indicó al inicio del algoritmo. Con esto podemos decir que el algoritmo planteado aquí es correcto. No se olvide que cuando un ciclo está dentro de otro su orden de ejecución es muy importante para que realmente se obtengan los resultados esperados.

8.4.2. Ejemplo 2

Mostrar las tablas de multiplicar del 1 al 3.

Clarificación del objetivo

Todos hemos conocido desde nuestra primaria las tablas de multiplicar. El objetivo de este algoritmo es mostrar en pantalla la tabla del 1, la tabla del 2 y la tabla del 3 tal como nos las enseñaron, o sea:

```
1 x 1 =  1
1 x 2 =  2
1 x 3 =  3
1 x 4 =  4
1 x 5 =  5
1 x 6 =  6
1 x 7 =  7
1 x 8 =  8
1 x 9 =  9
1 x 10 = 10

2 x 1 =  2
2 x 2 =  4
2 x 3 =  6
2 x 4 =  8
2 x 5 =  10
   .
   .
   .
```

El objetivo es, básicamente, mostrar en pantalla las tablas tal como se han mostrado aquí. Para ello, nos vamos a valer de dos ciclos anidados. El ciclo externo nos va a permitir controlar que solo salga hasta la tabla del 3, es decir, que el índice del ciclo externo va a ir de 1 a 3, y el ciclo interno es el que nos va a permitir que se generen los números del 1 al 10 para que al realizar la multiplicación entre el índice del ciclo externo y el índice del ciclo interno obtengamos los resultados esperados. Para facilidad de una posterior prueba de escritorio, vamos a utilizar ciclos, dado que en este caso todos los topes son concretos.

De nuevo, se trata solo de tener un ciclo externo con una variable que va a ir desde 1 hasta 3 y otro ciclo interno con una variable que va a ir desde 1 hasta 10. Dentro del ciclo interno estará ubicada una orden que permita mostrar en pantalla lo solicitado.

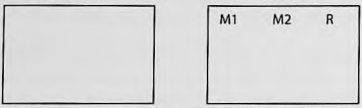
Algoritmo

```
Programa Ciclos_Anidados_2
Variables
        Entero : M1,        // Índice del ciclo externo, almacenará el valor del
                            //multiplicando
                 M2,        // Índice del ciclo interno, almacenará el valor del
                            // multiplicador
                 R          // Almacena el resultado de multiplicar cada vez el
                            // multiplicando por el Multiplicador
Inicio
        Escriba "Tablas de Multiplicar del 1 al 3" // Anuncia lo que va a escribir

        // Ciclo externo cuya variable va desde 1 hasta 3
        Para M1 = 1 hasta 3 (Paso 1)
                // Ciclo interno cuya variable va desde 1 hasta 10
                Para M2 = 1 hasta 10 (Paso 1)
                        R = M1 * M2             // Resultado de cada Multiplicación
                        Escriba M1, "x", M2, "=", R// Muestra el resultado
                Fin_Para                        // Fin del ciclo interno
        Fin_Para                                // Fin del ciclo externo
Fin                                             // Fin del Algoritmo
```

Prueba de escritorio

Teniendo nuestro algoritmo ya planteado, vamos a desarrollarle una prueba de escritorio completa para acercarnos un poco más al concepto de ciclos anidados. Primero que nada, ubicamos en memoria tres variables dado que así es como comienza el algoritmo.

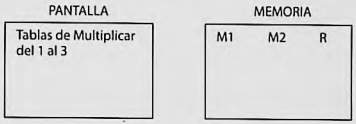
```
Programa Ciclos_Anidados_2
Variables
        Entero : M1, M2, R
```



Aparece en pantalla un aviso que anuncia qué es lo que se va a mostrar a continuación.

Inicio

Escriba "Tablas de Multiplicar del 1 al 3"



El ciclo externo involucra una variable *M1* que va a tomar valores comenzando en 1 y terminando en 3 (inclusive) de 1 en 1. Así mismo, el ciclo interno incluye una variable *M2* que va a tomar valores comenzando en 1 y terminando en 10 (inclusive) de 1 en 1. Dentro del ciclo más interno ejecutará el resultado de multiplicar *M1* por *M2* y lo almacenará en la variable *R* mostrando todo en pantalla.

Para M1 = 1 hasta 3 (Paso 1)
Para M2 = 1 hasta 10 (Paso 1)
*R = M1 * M2*
Escriba M1, "x", M2, "=", R

De esta manera, mientras *M1* toma el valor de 1, *M2* tomará valores entre 1 y 10 (sin que cambie para nada el valor de *M1*). Entonces, cuando *M2* valga 1 la ejecución será:

Cuando *M2* tome el valor de 2 (manteniéndose el valor de 1 en *M1*):

Cuando *M2* tome el valor de 3 (manteniéndose el valor de 1 en *M1*):

PLANTILLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3

Cuando *M2* tome el valor de 4 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4

Cuando *M2* tome el valor de 5 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5

Cuando *M2* tome el valor de 6 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6

Cuando *M2* tome el valor de 7 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7

Cuando *M2* tome el valor de 8 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8

Cuando *M2* tome el valor de 9 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8
1	9	9

Cuando *M2* tome el valor de 10 (manteniéndose el valor de 1 en *M1*):

PANTALLA

```
Tablas de Multiplicar
del 1 al 3

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
```

MEMORIA

M1	M2	R
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8
1	9	9
1	10	10

Como la variable *M2* ya llegó a su tope (que era 10), entonces nos salimos del ciclo interno (que incluye sus instrucciones) y volvemos al ciclo externo, o sea, a incrementar la variable *M1* en 1, de manera que quedaría con el valor de 2.

Como esta variable aún no ha llegado a su tope, entonces volvemos a entrar al cuerpo del ciclo externo que incluye un ciclo interno en donde una variable *M1* va a almacenar enteros empezando en 1 y terminando en 10 y ejecutándose (dentro de este ciclo interno) las órdenes de multiplicar y mostrar los resultados en pantalla. De esta forma, *M1* toma el valor de 2 mientras *M2* va desde 1 hasta 10.

Cuando *M2* tome el valor de 1 (manteniéndose el valor de 2 en *M1*):

PANTALLA

```
.
.
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10

2 x 1 = 1
```

MEMORIA

M1	M2	R
.	.	.
.	.	.
1	7	7
1	8	8
1	9	9
1	10	10
2	1	1

Cuando *M2* tome el valor de 2 (manteniéndose el valor de 2 en *M1*):

PANTALLA

```
.
.
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10

2 x 1 = 2
2 x 2 = 4
```

MEMORIA

M1	M2	R
.	.	.
.	.	.
1	7	7
1	8	8
1	9	9
1	10	10
2	1	2
2	2	4

Así iremos sucesivamente manteniendo el valor que almacena *M1* pero incrementando el valor de *M2* progresivamente a medida que vamos haciendo la multiplicación y vamos mostrando los resultados. De esta forma, al finalizar la ejecución del ciclo interno, tendremos tanto en memoria como en pantalla:

Igualmente, puede usted realizar la prueba de escritorio en donde *M1* se incrementa en 1 quedando con el valor de 3 y *M2* vuelve a tomar valores desde 1 hasta 10 generando la tabla de multiplicar del 10. Luego de esto, *M2* llega a su tope, tal como sucedió en dos oportunidades más, solo que esta vez *M1* también llega a su tope, por lo cual nos salimos del ciclo externo y ejecutamos la orden que se encuentra inmediatamente después del *Fin_Ciclo* correspondiente. Esa orden no es más que el *Fin* del algoritmo. Al final, obtendremos el resultado esperado, o sea, tener en pantalla las tablas de multiplicar del 1

al 3. Si hubiéramos querido tener todas las tablas de multiplicar, todo lo que tendríamos que hacer es cambiar la orden

Para M1 = 1 hasta 3 (Paso 1)

por

Para M1 = 1 hasta 10 (Paso 1)

Y nuestro algoritmo quedaría generando todas las tablas de multiplicar.

8.4.3. Ejemplo 3

Leer un número entero y mostrar todos los enteros comprendidos entre 1 y cada uno de los dígitos del número leído.

Clarificación del objetivo

Antes de plantearle una solución a este enunciado, no se olvide que los algoritmos presentados en este libro son solo una versión de solución a los problemas aquí planteados. Esto quiere decir que si usted desarrolla un algoritmo que no es igual a los presentados en este libro pero al realizarle la correspondiente prueba de escritorio usted ve que también cumple con el objetivo, entonces esto quiere decir que su algoritmo está bien, sin importar que sea diferente al presentado en este libro. Es muy importante que usted tenga esto en cuenta, dado que muchas personas me preguntan lo mismo. Por ahora, no se olvide que lo que importa es lograr el objetivo planteado utilizando un camino algorítmico.

En este ejercicio, vamos a leer un número entero. Si es negativo, lo multiplicamos por (-1) para facilitar las operaciones y las validaciones. Recordemos que la fórmula

$$Dig = num - num / 10 * 10$$

va a almacenar en la variable *Dig* el último dígito de un número entero. Si por ejemplo *num* es igual a 1543, entonces:

$$Dig = num - num / 10 * 10$$
$$Dig = 1543 - 1543 / 10 * 10$$
$$Dig = 1543 - 154 * 10$$
$$Dig = 1543 - 1540$$
$$Dig = 3$$

Que corresponde al último dígito del valor tomado como ejemplo. Entonces lo que vamos a hacer es que dentro de un ciclo vamos a ir obteniendo el último dígito del número y vamos a ir dividiendo progresivamente el número entre 10. De esta manera, habremos obtenido todos los dígitos por separado. Al tiempo que obtenemos cada dígito, a través de otro ciclo, vamos generando los números comprendidos entre 1 y el dígito obtenido. Todos los resultados se irán escribiendo en pantalla.

Algoritmo

```
Programa Ciclos_Anidados_3
Variables
    Entero : Num,                  // Almacenará el número leído
             Dig,                  // Almacenará cada uno de los dígitos
                                   // que tenga el número leído
             Aux                   // Almacenará cada uno de los enteros
                                   // comprendidos entre 1 y cada uno de
                                   // los dígitos del número leído
Inicio
    Escriba "Digite un número entero"   // Solicita un número entero, lo lee
    Lea Num                        // y lo almacena en la variable Num

    Si Num < 0                     // Si el número es negativo
        Num = Num * ( -1 )         // lo volvemos positivo para facilitar los
                                   // cálculos

    Mientras Num < > 0             // Mientras el contenido del número
                                   // leído sea diferente de cero
        Dig = Num – Num / 10 * 10  // Almacene el último dígito en la
                                   // variable Dig
        Para Aux = 1 hasta Dig (Paso 1) // Genere la secuencia de enteros
                                        // comprendidos entre 1 y el dígito
                                        // obtenido y cada valor
                                        // almacénelo en
                                        // la variable Aux
            Escriba Aux            // Muestre cada uno de los valores que
                                   // tome la variable Aux

        Fin_Para
```

```
        Num = Num / 10          // Divida el número original entre 10
                                // (para "quitarle" aritméticamente el
                                // último dígito)
    Fin_Mientras                    // Fin del ciclo inicial
Fin                                 // Fin del Algoritmo
```

Prueba de escritorio

Nuestra prueba de escritorio comienza con la declaración de tres variables en pantalla, tal como se ha hecho en los otros ejercicios.

```
Programa Ciclos_Anidados_3
Variables
    Entero : Num,
             Dig,
             Aux
```

PANTALLA

```
.
.
.
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

MEMORIA

M1	M2	R
~~1~~	~~10~~	~~10~~
2	1	2
2	2	4
2	3	6
2	4	8
2	5	~~10~~
2	6	~~12~~
2	~~7~~	~~14~~
2	8	~~16~~
2	9	~~18~~
2	10	20

```
Inicio
    Escriba "Digite un número entero"
    Lea Num
```

Se solicita un dato entero, se lee y se almacena en la variable *Num*. Para efectos de nuestra prueba de escritorio, vamos a asumir que el número digitado es el 354.

PANTALLA

MEMORIA

Num	Dig	Aux

A continuación, se pregunta si el contenido de la variable *Num* es negativo y entonces se multiplica por –1 para facilitar los cálculos. Como en este caso el contenido de la variable *Num* es positivo, entonces la decisión es Falsa, por lo tanto, nos saltamos la instrucción a ejecutar en caso de que la decisión fuera Verdadera. Continuamos con el ciclo *Mientras* que está a continuación:

```
Si Num < 0
        Num = Num * (-1)
```

El ciclo comienza estableciendo una condición para entrar en él: mientras el contenido de la variable *Num* sea diferente de cero. Como en este caso el valor de la variable *Num* es 354, entonces la condición es Verdadera, por lo tanto, entramos al ciclo planteado:

```
Mientras Num < > 0
```

Lo primero que vamos a realizar es la operación:

```
Dig = Num – Num / 10 * 10
```

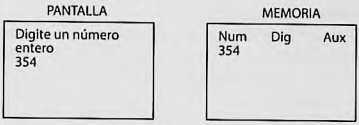
Según la cual se obtiene el último dígito del contenido de la variable *Num*. De esta forma, como *Num* vale 354, entonces:

```
Dig = Num – Num / 10 * 10
Dig = 354 – 354 / 10 * 10
Dig = 354 – 35 * 10
Dig = 354 – 350
Dig = 4
```

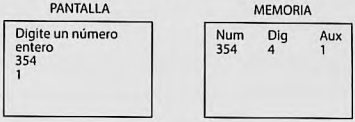
Y efectivamente obtenemos el último dígito del contenido de la variable *Num*. A continuación, siguiendo con nuestro algoritmo, generaremos un ciclo en donde la variable *Aux* va a tomar valores desde 1 hasta el valor actual de *Dig* (o sea, 4) incrementándola de 1 en 1, ciclo dentro del cual se irá escribiendo progresivamente el valor almacenado en la variable *Aux*:

```
Para Aux = 1 hasta Dig (Paso 1)
        Escriba Aux
Fin_Para
```

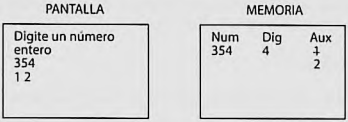
De esta forma, cuando *Aux* tenga el valor de 1, entonces se escribirá:



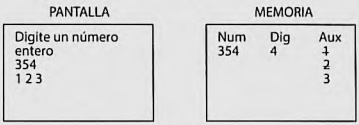
Cuando *Aux* tenga el valor de 2, entonces se escribirá:



Cuando *Aux* tenga el valor de 3, entonces se escribirá:



Cuando *Aux* tenga el valor de 4, habrá llegado al tope planteado, pues el ciclo inicialmente decía que *Aux* tomaría valores entre 1 y el valor actual de *Dig* que es 4, luego esta sería su última iteración, por lo cual escribirá:



Como ya se finalizó el ciclo *Para* planteado, entonces continuamos con la instrucción que se encuentra después del *Fin_Para*:

Num = Num / 10

Sabiendo que el contenido de la variable *Num* es 354, el resultado de esta expresión sería:

Num = Num / 10
Num = 354 / 10
Num = 35

PANTALLA

```
Digite un número
entero
354
1 2 3 4
```

MEMORIA

Num	Dig	Aux
354	4	1
		2
		3
		4

Instrucción con la cual termina el ciclo, pues encontramos a continuación el *Fin_Mientras* correspondiente.

Fin_Mientras

De tal manera que volvemos a la condición del ciclo *Mientras* para evaluarla y saber si se continúa ejecutando el cuerpo del ciclo o se finaliza definitivamente la ejecución del mismo. Al volver al ciclo que decía:

Mientras Num < > 0

Como el contenido actual de la variable *Num* es 35, entonces la condición es Verdadera, por lo tanto, volvemos a ejecutar el cuerpo del ciclo. Por lo tanto, ejecutamos la orden:

*Dig = Num – Num / 10 * 10*

Cuyo desarrollo sería:

*Dig = Num – Num / 10 * 10*
*Dig = 35 – 35 / 10 * 10*
*Dig = 35 – 3 * 10*
Dig = 35 – 30
Dig = 5

Dejando almacenado en la variable *Dig* el valor 5:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
```

MEMORIA

Num	Dig	Aux
354	4	1
35		2
		3
		4

A continuación, vamos a generar un ciclo, utilizando la variable *Aux* como índice, para que tome valores entre 1 y el valor actual de *Dig* (que es 5) con incrementos de 1. De esta manera, cuando *Aux* valga 1, entonces en pantalla se escribirá su contenido:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1
```

MEMORIA

Num	Dig	Aux
~~354~~	4	1
35	5	

Cuando *Aux* valga 2, entonces en pantalla se escribirá:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
35	5	2

Cuando *Aux* valga 3, entonces en pantalla se escribirá:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
35	5	~~2~~
		3

Cuando *Aux* valga 4, entonces en pantalla se escribirá:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
35	5	~~2~~
		~~3~~
		4

Cuando *Aux* valga 5, entonces habrá llegado a su tope, pues los valores tomados por la variable *Aux* llegarían hasta el valor actual de la variable *Dig*. De manera que esta sería su última iteración y en pantalla se escribirá:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4 5
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
35	5	2
		3
		4
		5

Al haber terminado el ciclo *Para*, continuamos de nuevo con la instrucción que se encuentra después de su *Fin_Para* respectivo, que es una asignación cuyo desarrollo sería:

Num = Num / 10
Num = 35 / 10
Num = 3

Quedando en memoria almacenado en la variable Num el valor 3.

PANTALLA

```
Digite un número
entero
354
1 2 3 4
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
~~35~~	5	~~2~~

Con lo cual, y dado que encontramos el fin del ciclo *Mientras*, debemos volver a este para evaluar si su condición sigue siendo Verdadera.

Mientras Num < > 0

Como en este momento el contenido de la variable *Num* es Verdadero, dado que Num almacena un número 3, entonces volvemos a ejecutar una vez más el cuerpo del ciclo que inicia con una instrucción de asignación cuyo desarrollo es:

*Dig = Num – Num / 10 * 10*
*Dig = 3 – 3 / 10 * 10*
*Dig = 3 – 0 * 10*
Dig = 3 – 0
Dig = 3

Quedando almacenado en la variable *Dig* (en memoria obviamente) el valor 3:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4 5
```

MEMORIA

Num	Dig	Aux
~~354~~	4	
~~35~~	~~5~~	
3	3	

A continuación generamos, utilizando un ciclo *Para*, una secuencia de enteros comprendidos entre 1 y el valor actual de *Dig* (que es 3) con incrementos de 1 escribiendo cada contenido en la pantalla. Así, cuando *Aux* valga 1, entonces se escribirá en pantalla:

```
Para Aux = 1 hasta Dig (Paso 1)
        Escriba Aux
Fin_Para
```

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4 5
1
```

MEMORIA

Num	Dig	Aux
~~354~~	4	1
~~35~~	~~5~~	
3	3	

Cuando *Aux* valga 2, entonces se escribirá en pantalla:

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4 5
1 2
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
~~35~~	~~5~~	2
3	3	

Cuando *Aux* valga 3, entonces se escribirá en pantalla el valor 3 y habrá finalizado el ciclo, pues este debería ir desde 1 hasta el valor actual de *Dig* que es 3.

PANTALLA

```
Digite un número
entero
354
1 2 3 4
1 2 3 4 5
1 2 3
```

MEMORIA

Num	Dig	Aux
~~354~~	4	~~1~~
~~35~~	~~5~~	~~2~~
3	3	3

23. Leer un número entero y determinar si la suma de sus dígitos es también un número primo.

24. Leer un número entero y determinar a cuánto es igual al suma de sus dígitos pares.

25. Leer un número entero y determinar a cuánto es igual el promedio entero de sus dígitos.

26. Leer un número entero y determinar cuál es el mayor de sus dígitos.

27. Leer 2 números enteros y determinar cuál de los dos tiene mayor cantidad de dígitos.

28. Leer 2 números enteros y determinar cuál de los dos tiene mayor cantidad de dígitos primos.

29. Leer un número entero y determinar a cuánto es igual el primero de sus dígitos.

30. Leer un número entero y mostrar todos sus componentes numéricos, o sea, aquellos para quienes él sea un múltiplo.

31. Leer números hasta que digiten 0 y determinar a cuánto es igual el promedio de los números terminados en 5.

32. Leer números hasta que digiten 0 y determinar a cuánto es igual el promedio entero de los número primos leídos.

33. Si 32768 es el tope superior para los números entero cortos, determinar cuál es el número primo más cercano por debajo de él.

34. Generar los números del 1 al 10 utilizando un ciclo que vaya de 10 a 1.

35. Leer dos números enteros y determinar a cuánto es igual el producto mutuo del primer dígito de cada uno.

36. Mostrar en pantalla la tabla de multiplicar del número 5.

37. Generar todas las tablas de multiplicar del 1 al 10.

38. Leer un número entero y mostrar en pantalla su tabla de multiplicar.

39. Se define la serie de Fibonacci como la serie que comienza con los dígitos 1 y 0 y va sumando progresivamente los dos últimos elementos de la serie, así:

0 1 1 2 3 5 8 13 21 34.....

Utilizando el concepto de ciclo, generar la serie de Fibonacci hasta llegar o sobrepasar el número 10000.

40. Leer un número de dos dígitos y determinar si pertenece a la serie de Fibonacci.

41. Determinar a cuánto es igual la suma de los elementos de la serie de Fibonacci entre 0 y 100.

42. Determinar a cuánto es igual el promedio entero de los elementos de la serie de Fibonacci entre 0 y 1000.

43. Determinar cuántos elementos de la serie de Fibonacci se encuentran entre 1000 y 2000.

44. Leer un número y calcularle su factorial.

45. Leer un número y calcularle el factorial a todos los enteros comprendidos entre 1 y el número leído.

46. Leer un número entero y calcular el promedio entero de los factoriales de los enteros comprendidos entre 1 y el número leído.

47. Leer un número entero y calcular a cuánto es igual la sumatoria de todos los factoriales de los números comprendidos entre 1 y el número leído.

48. Utilizando ciclos anidados, generar las siguientes parejas de enteros:

0	1
1	1
2	2
3	2
4	3
5	3
6	4
7	4
8	5
9	5

49. Utilizando ciclos anidados, generar las siguientes ternas de números:

1	1	1
2	1	2
3	1	3
4	2	1
5	2	2
6	2	3
7	3	1
8	3	2
9	3	3

50. Utilizando ciclos anidados, generar las siguientes parejas de números:

0	1
1	1
2	1
3	1
4	2
5	2
6	2
7	2

Capítulo 9

Arreglos

9.1. Concepto general

Un arreglo es un conjunto de variables en donde cada una de ellas puede ser referenciada utilizando su posición relativa, es decir, su ubicación en relación con el primer elemento de dicho conjunto. Estoy seguro que más de una vez usted habrá notado que el cajero de un banco, para referirse a una de las personas que hacen cola para hacer una consignación, puede decir "Venga la quinta persona" y habrá notado como una persona que no estaba de primera pasa a ser atendida por él. Varios razonamientos se involucran inconscientemente en esta situación:

- En primera instancia, sabemos que la "quinta persona" se ubica contando a partir de la primera. Esto es algo muy obvio, pero para el tema que nos ocupa en el momento es importantísimo.
- Normalmente, todas las personas, sin equivocación, voltean a mirar a una sola persona que es, con toda seguridad, la que ocupa la quinta posición en la fila.
- El "quinto puesto" en la cola tiene que estar ocupado por alguien. No puede estar vacío, pues esa tuvo que haber sido una de las razones para que el cajero se refiriera a dicha persona con tanta seguridad.
- La cola de personas no es infinita. Tiene una cantidad determinada de clientes.
- Delante de la quinta persona debe haber cuatro personas y después de ella puede haber muchas o ninguna persona.

- Si existen varias colas, solo atenderá el llamado del cajero la "quinta persona" de la cola a la que él se refiera.

- Si en algún momento otro cajero dijera "Vea, todas las personas de esa cola pásense a esta caja", entonces todo el conjunto de personas atendería el llamado de ese otro cajero, lo cual significaría que en cualquier momento ese conjunto de personas puede ser manejado como una sola unidad.

Estos razonamientos, vuelvo a repetir, son muy obvios. En nuestro caso, nos van a permitir caracterizar algunos elementos técnicos en referencia a los arreglos partiendo de unos conceptos muy simplificados y "domésticos".

Cuando yo estaba en la escuela desatendía la costumbre de llevar todos los días los zapatos embolados y, por lo tanto, casi todas las mañanas escuchaba como, desde la Dirección y mientras todos estábamos perfectamente formados, la directora decía: "Allá, el tercer niño de la quinta fila, acérquese a la oficina de la Dirección". Siempre, en ese momento, todos comenzaban a contar las filas y luego, cuando habían ubicado la "quinta fila", comenzaban a buscar cuál era el "tercer niño". Obviamente era yo. Si la directora solo hubiera dicho alguna vez "Venga el tercer niño para acá", se habría generado una confusión que me habría permitido pasar inadvertido, pues, como había varias filas, entonces muchos "terceros niños" se hubieran sentido aludidos. Sin embargo, en esta anécdota, que cuento con algo de vergüenza, también se involucran varios razonamientos:

- Es evidente que todos los niños estaban formados en varias filas o, para mejor decirlo, existían varias filas en donde cada fila estaba formada por varios niños.

- Cualquier niño era fácilmente ubicable diciendo sencillamente en qué fila y en qué posición dentro de esa fila se encontraba. Y dicha ubicación no daba espacio a la duda, ya que cada uno tenía una única posición dentro de la formación.

- Las filas se contaban partiendo de la primera en adelante y la posición de los niños dentro de cada fila se contaba partiendo del primer niño en adelante.

- La escuela tenía una cantidad finita de niños que formaban todas las mañanas, o sea, que tanto la cantidad de filas como la cantidad de niños en cada fila era finita.

- Antes de la "quinta fila" existían cuatro filas más y después de la "quinta fila" podían existir más filas o ninguna fila. Antes del tercer niño existían

dos niños más y después del tercer niño (en este caso) existían muchos niños más, aunque pudiera no haber existido ninguno.

- Si en algún momento la directora dijera "Todos los niños fórmense en el patio de allá", entonces podría manejar todo el grupo de niños como una sola unidad.

- Cuando se referían a una fila determinada, esta debía existir (con toda seguridad). Cuando se referían a un niño determinado en una fila determinada, y valga mucho esa redundancia, este también debería existir.

- En algún momento, la directora de la escuela pudo haber dicho "La tercera fila fórmese en ese lado de allá" y solo la tercera fila se habría movido, quedando el resto de la formación en su estado original.

Qué pasa si frente a un colegio cuyo edificio tiene varios pisos y cada piso tiene varios salones una persona se para y dice: "Necesito al quinto estudiante". Seguramente, todos en el colegio se van a preguntar a partir de dónde comienzan a contar para saber quién es el quinto estudiante y lo más posible es que nadie va a salir. Pero si esa persona se para frente al colegio y dice "Necesito al quinto estudiante que está sentado en la tercera fila del cuarto salón del sexto piso de este colegio", entonces con seguridad alguien va a salir porque fue ubicado apropiadamente.

Hemos tenido tres ejemplos: una cola en un banco, una formación en filas de los niños de una escuela y la ubicación de un estudiante en un colegio. En cada caso, podemos ver cómo para ubicar efectivamente a una persona necesitamos utilizar referencias diferentes. Esto quiere decir:

a. Para ubicar a una persona dentro de una cola, todo lo que tenemos que hacer es utilizar UNA y solo UNA referencia. Es claro pues referirnos a la TERCERA persona o a la QUINTA persona dentro de una cola, pues no habría duda en su ubicación.

b. Para ubicar a un niño dentro de la población estudiantil de una escuela, cuando esta está formada en el patio, necesitamos utilizar DOS y solo DOS referencias. De manera que, al referirnos al TERCER niño de la QUINTA fila, estamos determinando exactamente la posición incuestionable de un determinado niño.

c. Para ubicar a un estudiante en un colegio cuyo edificio tiene varios pisos y cada piso tiene varios salones, necesitamos entonces utilizar varias referencias. Cuando se dice "Necesito al QUINTO estudiante que está sentado

en la TERCERA fila del CUARTO salón del SEXTO piso", nos estamos refiriendo exactamente a una persona de manera incuestionable.

Es curioso pensar que esta persona pudo haber sido la misma, solo que para referirnos a ella tuvimos que ubicarla exactamente de manera diferente en cada una de las situaciones. En los tres casos, las características son similares aun a pesar de que no es igual la distribución de los elementos (o personas para este caso).

9.2. Índices

9.2.1. Definición

Se conocen como índices todas aquellas variables que nos sirven para ubicar perfectamente un elemento dentro de un arreglo. De esta forma, en el ejemplo de la cola de personas en vez de decir "la quinta persona" podríamos haber dicho la "persona No. 5" o, para decirlo de una manera más técnica, la persona sub 5. En el ejemplo del "tercer niño de la quinta fila", podríamos haber dicho también el niño No. 3 de la fila No. 5; igualmente con el ánimo de decirlo de una manera más técnica, el "niño sub 3 de la fila sub 2". Cada dato numérico utilizado para ubicar estos elementos dentro de un arreglo se conoce como ***índice***.

Cuando el índice es reemplazado por una variable cuyo contenido es el dato entero que necesitamos, entonces esta variable se conoce como ***subíndice***. De esta forma, si en el ejemplo de la cola tenemos una variable que se llama *Num* y *Num* contiene el número 5, entonces podríamos habernos referido a la "quinta persona" como la "Persona sub *Num*" (eso sí, teniendo la seguridad de que *Num* contiene solo el número 5). Luego de esta manera, si la cola tiene 10 personas, entonces un algoritmo de atención para las 10 personas sería:

```
Para Num = 1 hasta 10
        Atienda a Persona sub Num
```

O escrito de una forma más resumida, y de paso más técnica, podríamos decir que:

```
Para Num = 1 hasta 10
        Atienda a Persona ( Num )
```

Y de esta manera, sabiendo que la variable *Num* puede tomar valores desde 1 hasta 10, pero que cada vez que tome un valor atiende a una determinada persona, se podrá concluir que:

Cuando Num valga 1
atenderá a la Persona (1) que corresponde a la primera persona
Cuando Num valga 2
atenderá a la Persona (2) que corresponde a la segunda persona
Cuando Num valga 3
atenderá a la Persona (3) que corresponde a la tercera persona
Cuando Num valga 4
atenderá a la Persona (4) que corresponde a la cuarta persona
Cuando Num valga 5
atenderá a la Persona (5) que corresponde a la quinta persona
Cuando Num valga 6
atenderá a la Persona (6) que corresponde a la sexta persona
Cuando Num valga 7
atenderá a la Persona (7) que corresponde a la séptima persona
Cuando Num valga 8
atenderá a la Persona (8) que corresponde a la octava persona
Cuando Num valga 9
atenderá a la Persona (9) que corresponde a la novena persona
Cuando Num valga 10
atenderá a la Persona (10) que corresponde a la décima persona

Como puede ver, el manejo de la cola, sin importar cuántas personas tenga, se va a reducir al manejo de un ciclo con una variable importantísima que actuará como subíndice. Igualmente, supongamos que en la escuela existían solo 5 cursos y que cada curso tenía 10 estudiantes. Para referenciar a cada fila, vamos a utilizar la variable *Fila* y para referenciar a cada niño dentro de la fila, vamos a utilizar la variable *Pos* como para hacer referencia a la posición en la cual se encuentre. Igualmente, vamos a asumir que Niño(*Fila*)(*Pos*) representará al niño que se encuentre en la fila *Fila* y dentro de ella en la posición *Pos*.

De manera que, si queremos revisar la presentación de cada niño, podríamos utilizar el siguiente fragmento de algoritmo:

Para Fila = 1 hasta 5
Para Pos = 1 hasta 10
Revise al Niño (Fila) (Pos)

Con lo cual, si le hacemos una pequeña "prueba de escritorio" a este algoritmo, obtendremos que inicialmente *Fila* tendrá el valor de 1. Mientras mantiene este valor, *Pos* va a tomar valores entre 1 y 10. De esta forma:

Cuando Pos valga 1
se revisará al Niño (1) (1) o sea el primer niño de la primera fila
Cuando Pos valga 2
se revisará al Niño (1) (2) o sea el segundo niño de la primera fila
Cuando Pos valga 3
se revisará al Niño (1) (3) o sea el tercer niño de la primera fila
Cuando Pos valga 4
se revisará al Niño (1) (4) o sea el cuarto niño de la primera fila
Cuando Pos valga 5
se revisará al Niño (1) (5) o sea el quinto niño de la primera fila
Cuando Pos valga 6
se revisará al Niño (1) (6) o sea el sexto niño de la primera fila
Cuando Pos valga 7
se revisará al Niño (1) (7) o sea el séptimo niño de la primera fila
Cuando Pos valga 8
se revisará al Niño (1) (8) o sea el octavo niño de la primera fila
Cuando Pos valga 9
se revisará al Niño (1) (9) o sea el noveno niño de la primera fila
Cuando Pos valga 10
se revisará al Niño (1) (10) o sea el décimo niño de la primera fila

Al llegar a esta punto, la variable *Fila* se incrementará en 1 y entonces, mientras esta variable tiene el valor de 2, la variable *Pos* tendrá valores desde 1 hasta 10, luego:

Cuando Pos valga 1
se revisará al Niño (2) (1) o sea el primer niño de la segunda fila
Cuando Pos valga 2
se revisará al Niño (2) (2) o sea el segundo niño de la segunda fila
Cuando Pos valga 3
se revisará al Niño (2) (3) o sea el tercer niño de la segunda fila

. . .
. . .
. . .

Cuando Pos valga 9

se revisará al Niño (2) (9) o sea el noveno niño de la segunda fila

Cuando Pos valga 10

se revisará al Niño (2) (10) o sea el décimo niño de la segunda fila

Cuando se llegue a este punto, entonces la variable *Fila* se incrementará en 1 y entonces, mientras esta variable tiene el valor de 3, la variable *Pos* tendrá valores desde 1 hasta 10, luego:

Cuando Pos valga 1

se revisará al Niño (3) (1) o sea el primer niño de la tercera fila

Cuando Pos valga 2

se revisará al Niño (3) (2) o sea el segundo niño de la tercera fila

Cuando Pos valga 3

se revisará al Niño (3) (3) o sea el tercer niño de la tercera fila

. . .
. . .
. . .

Cuando Pos valga 9

se revisará al Niño (3) (9) o sea el noveno niño de la tercera fila

Cuando Pos valga 10

se revisará al Niño (3) (10) o sea el décimo niño de la tercera fila

De nuevo, en este momento, se incrementaría el contenido de la variable *Fila* en 1 y, por lo tanto, almacenaría el valor 4. De manera que, mientras esta variable sea igual a 4, la variable *Pos* tomará valores desde 1 hasta 10, por lo tanto:

Cuando Pos valga 1

se revisará al Niño (4) (1) o sea el primer niño de la cuarta fila

Cuando Pos valga 2

se revisará al Niño (4) (2) o sea el segundo niño de la cuarta fila

Cuando Pos valga 3

se revisará al Niño (4) (3) o sea el tercer niño de la cuarta fila

. . .
. . .
. . .

Cuando Pos valga 9

se revisará al Niño (4) (9) o sea el noveno niño de la cuarta fila

Cuando Pos valga 10

se revisará al Niño (4) (10) o sea el décimo niño de la cuarta fila

Finalmente, mientras la variable *Fila* vale 5, la variable *Pos* tomará valores desde 1 hasta 10 y por lo tanto:

Cuando Pos valga 1

se revisará al Niño (5) (1) o sea el primer niño de la quinta fila

Cuando Pos valga 2

se revisará al Niño (5) (2) o sea el segundo niño de la quinta fila

Cuando Pos valga 3

se revisará al Niño (5) (3) o sea el tercer niño de la quinta fila

Cuando Pos valga 4

se revisará al Niño (5) (3) o sea el tercer niño de la quinta fila

. . .
. . .
. . .

Cuando Pos valga 9

se revisará al Niño (5) (9) o sea el noveno niño de la quinta fila

Cuando Pos valga 10

se revisará al Niño (5) (10) o sea el décimo niño de la quinta fila que correspondería dentro del ejemplo al último niño de la escuela.

9.2.2. Características

Fundamentalmente, los índices se conciben como números estrictamente enteros debido a que referenciar una posición dentro de un arreglo siempre se dará en términos enteros. Cuando usted hace una cola en un banco, usted puede quedar de primero, de segundo, de tercero, etc. o lo que es lo mismo puede quedar de 1, de 2, de 3, etc. Lo que no se puede concebir es que usted quede de 1.4 o de 4.6 dentro de la cola. Por esta razón es que se han conceptualizado los índices como datos estrictamente enteros.

Asimismo, debido a que normalmente los datos de un arreglo se manejan como un conjunto, se acostumbra, por facilidad y flexibilidad de los algoritmos, a trabajar los índices a través de variables de tipo entero. Facilitan las expresiones en donde se involucren los elementos de los arreglos y flexibilizan

los algoritmos, debido a que cambiar el tope final de un ciclo, cuyos valores sirven para que se generen las posiciones correspondientes de un arreglo, es suficiente para que el mismo algoritmo sirva para un arreglo con otras dimensiones.

Tal como se ha visto es muy normal, también por conveniencia técnica, que se utilicen activamente ciclos para facilitar el manejo de esas variables que actuarán como índices.

9.3. Vectores

9.3.1. Características

Un vector es un arreglo en donde la ubicación *exacta* de cada uno de sus elementos necesita solamente la utilización de un subíndice. Tal es el ejemplo de una cola de personas en donde cada una de ellas se puede ubicar *exactamente* con un solo numerito: "Venga la quinta persona", "Acérquese la tercera persona", etc. He resaltado en letra cursiva la palabra *exacta* y *exactamente* porque ellas representan la diferencia con los otros tipos de arreglos.

Un vector siempre tendrá:

Tipo.- Por lo que se ha dicho, en un vector, los datos que se han de almacenar siempre serán del mismo tipo, por lo cual es supremamente importante especificar de qué tipo van a ser los datos almacenados en él. No se olvide que los tipos de datos estándar son *entero*, *real* y *carácter*, cada uno con características propias que fueron explicadas en los primeros capítulos. En esta parte es importante anotar que el conjunto de datos almacenado en un vector siempre será *homogéneo*, o sea, que todos los datos son del mismo tipo.

Nombre.- Sabiendo que todos los datos almacenados en un vector van a pertenecer a un mismo arreglo, entonces dicho arreglo deberá tener un nombre ajustado a las mismas reglas con que se le colocan los nombres a las variables convencionales. No ha de olvidarse que un arreglo, en el fondo, es una variable dividida en varios pedacitos donde cada pedacito puede almacenar un dato diferente en su contenido mas no en su tipo.

Dimensión.- Se refiere a la cantidad de elementos que van a ser utilizados en el vector. Dimensionar un vector significa definir cuántos elementos se van a utilizar. En este aspecto, debemos conocer lo mejor posible el objetivo porque, cuando el algoritmo se haya convertido en programa y se encuentre en ejecución, no podremos cambiar la cantidad de elementos de que consta dicho

vector. Esa tal vez será una de las desventajas de la utilización de los arreglos en general y es que su dimensionamiento nos va a arriesgar a subdimensionar o a sobredimensionar, es decir, a definir más campos de memoria o menos campos de memoria de los que necesitemos.

Tamaño.- El tamaño es la cantidad total de campos de memoria que van a ser utilizados. En el caso de los vectores, no existe ninguna diferencia entre la dimensión y el tamaño, diferencia que se hará muy clara cuando expliquemos las matrices.

Destinación.- Es muy importante, tal como lo hicimos con el objetivo, que sepamos cuál va a ser la utilización del vector que vayamos a usar. Así como para nosotros en algún algoritmo era claro que la variable *Cont_Par* era la que iba a contener la cantidad de números pares que se generarán en algún proceso, así también va a ser muy importante que sepa con claridad cuál va a ser el uso de un determinado vector. De esta manera, siempre que hagamos referencia al vector sabremos a qué conjunto de datos nos estamos refiriendo.

Índice.- Siempre que vamos a usar un vector es natural pensar que necesitaremos una variable de tipo entero que será utilizada a manera de subíndice, o sea, que es la que nos va a almacenar cada una de las posiciones posibles que tenga el vector. Dicha variable solo tiene que cumplir con el requisito de ser una variable entera. Es importante que recuerde que, por ser el subíndice una variable, esta, dentro del programa, podrá tener todos los usos que le queramos dar dentro del contexto del mismo algoritmo, además de ser utilizada como subíndice.

Justificación.- Se justifica la utilización de vectores cuando se dan algunas de las siguientes razones:

1. En algún momento se necesita manejar una cantidad de datos como todo un conjunto.

2. Se necesitan almacenar datos que posteriormente se van a volver a utilizar.

3. Se necesitan realizar cálculos de manera que los resultados progresivos se vayan a necesitar más adelante.

4. Se necesita realizar un determinado proceso con un conjunto de datos, "al tiempo".

5. Se necesitan realizar cálculos tan complejos que resulte más óptimo almacenar los resultados provisionales que volverlos a calcular.

6. En general, cada vez que necesitemos hacer operaciones con conjuntos de datos.

7. Siempre que se necesite desarrollar algoritmos con conjuntos de datos cuya cantidad pueda en algún momento, no dentro de una misma ejecución, cambiar.

Vamos a desarrollar un algoritmo ineficiente, sin tener en cuenta el concepto de vectores, para que posteriormente usted note la diferencia y entienda la gran utilidad que tienen estos dentro de las aplicaciones de programación.

9.3.2. Ejemplo ineficiente sin vectores No. 1

Leer 10 números enteros y determinar cuál es el promedio entero de dichos números.

Clarificación del objetivo

Como vamos a desarrollar una solución sin tener en cuenta el concepto de vectores, entonces lo primero que vemos fácilmente es que necesitamos, por lo menos, 10 variables que nos permitan almacenar cada uno de los números leídos. Igualmente, necesitaremos otra variable que nos permita almacenar el promedio entero de dichos números. En el contexto del algoritmo en sí, lo que vamos a hacer es que vamos a leer 10 números enteros; cada uno se almacenará en una variable diferente y, cuando se hayan leído todos los números, los sumaremos y dividiremos entre 10 dicha suma, almacenando el resultado final en la variable destinada para tal fin y escribiendo su contenido en pantalla.

Algoritmo

```
Programa Ejem_Inef_sin_Vect_1
Var
        Entero : a,     // Almacenará el primer número leído
                 b,     // Almacenará el segundo número leído
                 c,     // Almacenará el tercer número leído
                 d,     // Almacenará el cuarto número leído
                 e,     // Almacenará el quinto número leído
                 f,     // Almacenará el sexto número leído
                 g,     // Almacenará el séptimo número leído
                 h,     // Almacenará el octavo número leído
                 i,     // Almacenará el noveno número leído
                 j,     // Almacenará el décimo número leído
```

```
                    Prom          //Almacenará el promedio entero de todos
                                  // los números leídos
Inicio
    Escriba "Digite 10 números enteros"   // Solicite 10 números enteros
    Lea a                                 // y léalos
    Lea b
    Lea c
    Lea d
    Lea e
    Lea f
    Lea g
    Lea h
    Lea i
    Lea j
    Prom = 0                      // Inicialice la variable Prom en Ceros
    Prom = Prom + a               // Sume progresivamente cada uno de los
                                  // valores leídos
    Prom = Prom + b
    Prom = Prom + c
    Prom = Prom + d
    Prom = Prom + e
    Prom = Prom + f
    Prom = Prom + g
    Prom = Prom + h
    Prom = Prom + i
    Prom = Prom + j
    Prom = Prom / 10              // Calcule el promedio de los números
                                  // leídos
    Escriba Prom                  // Escriba dicho promedio en pantalla
Fin
```

También hubiéramos podido escribir este algoritmo de la siguiente forma:

```
Programa Ejem_Inef_sin_Vect_1
Var
    Entero : a,                   // Almacenará el primer número leído
             b,                   // Almacenará el segundo número leído
```

```
        c,          // Almacenará el tercer número leído
        d,          // Almacenará el cuarto número leído
        e,          // Almacenará el quinto número leído
        f,          // Almacenará el sexto número leído
        g,          // Almacenará el séptimo número leído
        h,          // Almacenará el octavo número leído
        i,          // Almacenará el noveno número leído
        j,          // Almacenará el décimo número leído
        Prom        // Almacenará el promedio entero de todos
                    // los números leídos
Inicio
    Escriba "Digite 10 números enteros" // Avise que va a leer 10
                                        // números enteros y léalos
    Lea a, b, c, d, e, f, g, h, i, j
    Prom = ( a + b + c + d + e + f + g + i + j + k) / 10   // Calcule el promedio
    Escriba Prom                        // Escríbalo en pantalla
Fin
```

Resulta ser tan simplificada la prueba de escritorio de este algoritmo que a simple vista podemos ver que está bien, es decir, vemos que cumple plenamente con el objetivo planteado. Sin embargo, a pesar de que eso es verdad, este algoritmo tiene como desventaja principal el hecho de que solamente sirve para calcular el promedio de 10 números enteros digitados por el usuario. Usted tal vez dirá que ese precisamente era el objetivo a cumplir, pero, sabiendo la gran utilidad que tiene el cálculo de un promedio, en cualquier momento sería muy útil tener un algoritmo más flexible, o sea, que permitiera calcular el promedio de 10 datos o de 25 datos o de cualquier cantidad de datos.

Si se quisiera ajustar este algoritmo para que permita leer 15 números enteros y calcular su promedio, entonces tendríamos que aumentar otras cinco variables y ajustar los cálculos correspondientes en lo que se refiere al promedio. En ese caso, el cambio puede no ser mucho, pero qué tal que se quiera lograr el mismo objetivo pero con 1000 datos enteros. Entonces allí sí tendríamos un verdadero problema porque a nuestra solución inicial tendríamos que adicionarle 990 variables más y hacer los ajustes correspondientes para que en los cálculos el algoritmo tuviera en cuenta el resto de variables. Como puede ver, este algoritmo, a pesar de que cumple con el objetivo, es demasiado rígido y, por lo tanto, es muy poco flexible, lo cual lo hacer ser un algoritmo altamente ineficiente.

Entonces, ¿cómo haríamos para que además de cumplirse el objetivo se logre obtener un algoritmo eficiente...?

```
Programa Ejem_Efic_con_Vect_1
Variables
        Entero : Vector ( 10 ),          // Almacenará los 10 datos enteros que
                                         // se lean
                 Indice,                 // Servirá como variable subíndice
                 Promedio                // Almacenará el promedio entero de
                                         // los números leídos
Inicio
        Escriba "Digite 10 números enteros"      // Avisa que va a leer 10 enteros
        Para Indice = 1 hasta 10         // y los lee
                Lea Vector ( Indice )
        Fin_Para
        Promedio = 0                             // Inicializa el promedio en cero
        Para Indice = 1 hasta 10 // Acumula provisionalmente todos los
                                 // valores en la variable Promedio
                Prom = Prom + Vector ( Indice )
        Fin_Para
        Prom = Prom / 10                 // Calcula el promedio como tal
        Escriba "El promedio entero es", Prom    //Muestra el promedio en pantalla
Fin
```

Usted podrá notar que la estructura de este algoritmo es la misma pero es verdaderamente más eficiente que la versión anterior debido a que, si queremos que este algoritmo sirva para calcular el promedio entero de 1000 números, todo lo que tenemos que hacer es cambiar el número 10 por el número 1000 tantas veces como aparezca. Sin embargo, podemos hacerlo todavía más eficiente de la siguiente forma:

```
Programa Ejem_Efic_con_Vect_1
Variables
        Entero : Vector ( 10 ),          // Almacenará los 10 datos enteros que
                                         // se lean
```

```
                    Indice,         // Servirá como variable subíndice
                    Promedio,       // Almacenará el promedio entero de
                                    // los números leídos
                    Tope            // Almacenará la cantidad de números
                                    // a leer
Inicio
    Tope = 10
    Escriba "Digite", Tope, "números enteros" // Avisa que leerá 10 enteros

    Para Indice = 1 hasta Tope              // y los lee
        Lea Vector ( Indice )
    Fin_Para

    Promedio = 0                            // Inicializa el promedio en cero
    Para Indice = 1 hasta Tope      // Acumula provisionalmente todos los
                                    // valores en la variable Promedio
        Prom = Prom + Vector ( Indice )
    Fin_Para

    Prom = Prom / Tope                      // Calcula el promedio como tal
    Escriba "El promedio entero es", Prom   // Muestra el promedio en pantalla
Fin
```

Ahora note usted que, para hacer que esta nueva versión del mismo algoritmo nos permita leer 1000 números enteros y calcularles su promedio, entonces solo tendremos que cambiar la instrucción

```
Tope = 10
```

por

```
Tope = 1000
```

Y será suficiente para que nuestro algoritmo haya quedado en condiciones de cumplir el mismo objetivo pero con una cantidad diferente de datos. Esa es la esencia de la utilización de los vectores (y en general de los arreglos) que nos permiten desarrollar unos algoritmos altamente flexibles.

En este último algoritmo, sé que usted puede pensar que siendo así sería mucho más fácil que el mismo usuario digitara la cantidad de datos a leer. Pero tenga en cuenta que, como la declaración de variables es lo primero que se hace en el

algoritmo, entonces de todas maneras tendríamos que dimensionar un vector de una cantidad grande de posiciones enteras (por ejemplo, un vector de 50000 posiciones), con lo cual es posible que estemos dentro de las necesidades del usuario. Sin embargo, dimensionar un vector tan grande nos obliga a pensar que, si el usuario solo va a necesitar 10 posiciones, entonces, de alguna manera, se van a desperdiciar las 49990 demás posiciones y si el usuario, por casualidad, necesita más de 50000 datos, entonces este vector no nos va a servir.

Esta es la desventaja que en renglones anteriores mencionábamos acerca de los arreglos en general y es que, como hay que dimensionarlos al momento de su declaración, corremos el riesgo de separar mucha más memoria de la que necesitamos o mucha menos. Ahora sí, con esta conceptualización, veamos un ejemplo eficiente de utilización de vectores.

9.3.3. Ejemplo con vectores No. 1

Desarrollar un programa que permita leer 10 números enteros y determinar en qué posición entró el número mayor.

Clarificación del objetivo

Fundamentalmente, el objetivo de este algoritmo radica en tener 10 números almacenados en memoria y determinar, asumiendo el orden de lectura, en qué posición está el mayor de los números leídos. Debe tenerse en cuenta que, para saber cuál es el dato mayor de un conjunto de datos, debemos tener dicho conjunto completo. Por ejemplo, para que usted sepa cuál es el más alto de su familia, entonces tendrá que haber conocido a toda su familia... ¿cierto? Por esa misma razón, en este caso, "conocer" significa almacenar los datos, por lo cual, para poder determinar cuál es el mayor, primero los debemos almacenar y tenerlos de manera que puedan ser manejables. Precisamente esto es lo que justifica la utilización de un vector en este ejercicio.

Algoritmo

Nunca olvide que los algoritmos que se presentan en este libro no son más que una versión solución planteada por el autor. Recuerde que, si usted desarrolla un algoritmo que cumpla el mismo objetivo de alguno de los algoritmos aquí presentado y nota que es diferente al que aparezca en este libro, eso no quiere decir que su algoritmo esté mal o que el algoritmo de este libro esté mal. Sencillamente que son dos algoritmos diferentes que cumplen el mismo objetivo y que eso es completamente normal.

```
Programa Posic_Mayor
Variables
        Entero : Vector ( 10 ),          // Almacenará los 10 números enteros
                                         // que se van a leer
                 Indice,                 // Servirá como variable subíndice
                 Pos_May                 // Almacenará la posición del número
                                         // mayor que se vaya encontrando
                                         // provisionalmente y del número mayor
                                         // absoluto cuando se haya finalizado el
                                         // ciclo de búsqueda del mayor
Inicio
        Escriba "Digite 10 números enteros"   // Avisa que va a leer 10 números
                                                 // enteros
        Para Indice = 1 hasta 10 // Genera un ciclo apoyado en la
                                 // variable índice que tomará valores
                                 // desde 1 hasta 10
                Lea Vector ( Indice )    // Lea un entero y guárdelo en el vector
                                         // en la posición que sea igual al valor
                                         // almacenado en la variable Indice
        Fin_Para                         // Fin del ciclo
        Pos_May = 1                      // Se inicializa la variable Pos_May con
                                         // el valor 1 para asumir,
                                         // provisionalmente, que el mayor se
                                         // encuentra en la primera posición. La
                                         // intención es ir comparando
                                         // progresivamente con este dato cada
                                         // uno de los restantes datos que se
                                         // encuentran almacenados en el vector
        Para Indice = 2 hasta 10 // Genera un ciclo desde 2 hasta 10
                                 // para comparar contra el primer dato
                                 // el resto de datos
                Si Vector (Indice) > Vector (Pos_May)
                // Si el dato almacenado en el vector en
                // la posición donde vaya el índice en el
                // momento es mayor que el dato que
                // provisionalmente es el mayor
```

```
                    Pos_May = Indice // eso quiere decir que el mayor que
                                     // estaba en la posición que decía la
                                     // variable Pos_May ya no es el mayor
                                     // y que el nuevo mayor está en la
                                     // posición en donde está el índice, por
                                     // lo cual la variable Pos_May debe ser
                                     // igual al contenido de la variable
                                     // Indice
            Fin_Si                   // Fin de la decisión
        Fin_Para                     // Fin del ciclo

                                     // Al final escribirá el valor solicitado y
                                     // es la posición en la cual está el
                                     // número mayor de entre los números
                                     // leídos. Esa posición está almacenada
                                     // en la variable Pos_May
    Escriba "El número mayor está en la posición", Pos_May
Fin                                  // Fin del algoritmo
```

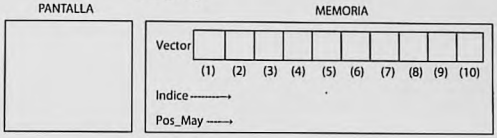
Prueba de escritorio

Vamos a desarrollar una prueba paso a paso tal como la haría el computador internamente.

Como es natural pensar, lo primero que se hace en este algoritmo es declarar las variables, o sea, separar el espacio de memoria que se va a utilizar. Por tal motivo, separamos en memoria espacio para 10 datos enteros que serán manejados a través del nombre de un vector y sus respectivas posiciones, espacio para una variable que se llamará *Indice* y otra que se llamará *Pos_May* y que almacenará la posición en la que se encuentre el número mayor.

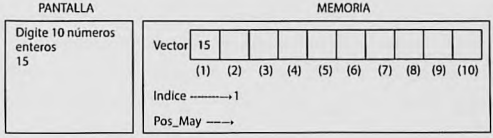
Para facilitar la distribución estética de las variables, vamos a desarrollar la prueba de escritorio colocando los diferentes valores de cada variable horizontalmente.

Programa Posic_Mayor
Variables
Entero : Vector (10),
Indice,
Pos_May

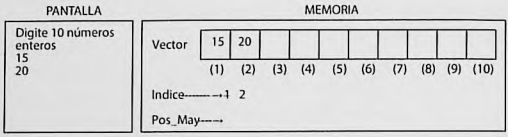


Nuestro algoritmo comienza anunciando en pantalla lo que se dispone a leer y, por supuesto, leyéndolo.

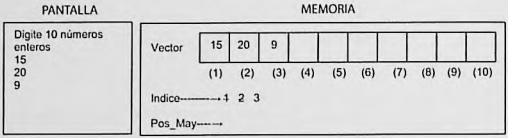
Inicio
Escriba "Digite 10 números enteros"
Para Indice = 1 hasta 10
Lea Vector (Indice)
Fin_Para



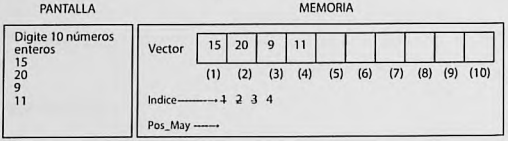
Se colocó en pantalla el título correspondiente, se inició el ciclo comenzando con la variable *Indice* en 1 (hemos de llevar hasta 10 el contenido de esta variable) y se recibió el primer número (supongamos que es 15). Como el contenido de la variable *Indice* es igual a 1 y la orden es *Lea Vector (Indice)*, o sea, *Lea un dato entero y guárdelo en Vector en la posición Indice*, entonces el dato recibido se almacena en *Vector (1)*. Como se encuentra el fin del ciclo (*Fin_Para*), entonces se regresa a incrementar en 1 el valor de la variable *Indice*, o sea, que su contenido es igual a 2. Razón por la cual al leer el siguiente número este quedará almacenado en el vector *Vector* en la posición 2. No se olvide que para efectos de la prueba de escritorio vamos a asumir algunos valores. Suponga-mos pues que el siguiente valor leído es igual a 20.



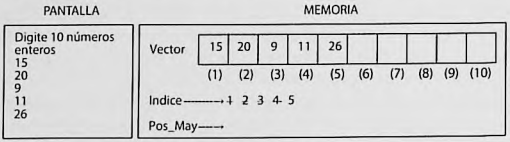
Cuando *Indice* valga 3 y asumiendo que el valor leído sea 9, entonces:



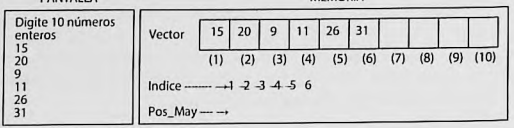
Cuando *Indice* valga 4 y asumiendo que el valor leído sea 11, entonces:



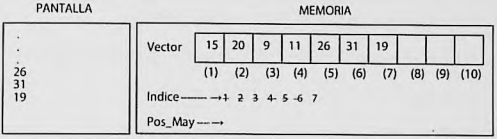
Cuando *Indice* valga 5 y asumiendo que el valor leído sea 26, entonces:



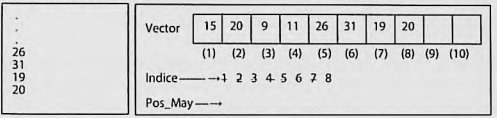
Cuando *Indice* valga 6 y asumiendo que el valor leído sea 31, entonces este valor se almacenará en el vector en la posición 6:



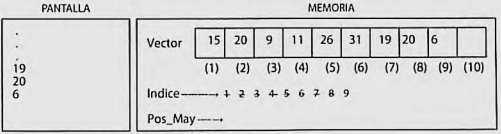
Cuando *Indice* valga 7 y asumiendo que el valor leído sea 19, entonces este valor se almacenará en el vector en la posición 7:



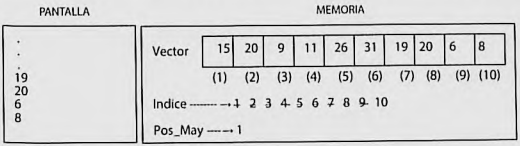
Cuando *Indice* valga 8 y asumiendo que el valor leído sea 20, entonces este valor se almacenará en el vector en la posición 8:



Cuando *Indice* valga 9 y asumiendo que el valor leído sea 6, entonces este valor se almacenará en el vector en la posición 9:



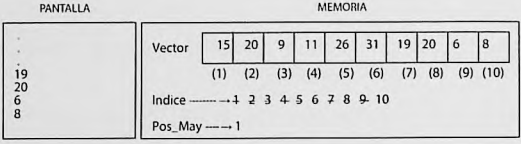
Y cuando la variable *Indice* valga 10, asumiendo que el valor leído sea 8, entonces este valor se almacenará en el vector en la posición 10. Además, como la variable *Indice* llega al tope planteado inicialmente, entonces la ejecución continúa con el resto de instrucciones, obviamente saliéndose del ciclo.



Como puede ver, de esta manera queda "cargado" con datos el vector y todo lo que tuvimos que hacer fue utilizar apropiadamente un ciclo para que a través de una variable se controlaran las posiciones en donde progresivamente se iban a ir almacenando cada uno de los valores digitados.

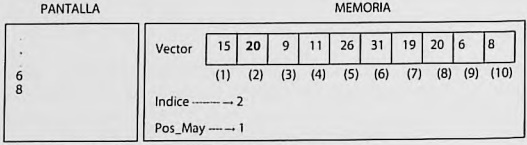
El algoritmo continúa inicializando la variable *Pos_May* con el valor de 1.

Pos_May = 1



Luego continúa realizando un ciclo *Para* utilizando la variable *Indice* que va a tomar valores desde 2 hasta 10 y que va a servir como referencia para comparar desde el elemento *Sub 2* hasta el elemento *Sub 10* del vector con el primer elemento.

```
Para Indice = 2 hasta 10
        Si Vector (Indice) > Vector (Pos_May)
                Pos_May = Indice
        Fin_Si
Fin_Para
```



Cuando la variable *Indice* valga 2, entonces la pregunta

Si Vector (Indice) > Vector (Pos_May)

se traducirá en

Si Vector (2) > Vector (1)

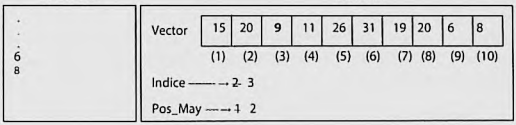
Dado que la variable *Pos_May* vale 1. Como *Vector (2)* es igual a 20 y *Vector (1)* es igual a 15, entonces internamente la pregunta se convierte en:

Si 20 > 15

Lo cual es Verdadero, por lo tanto, se ejecuta la orden:

Pos_May = Indice

Con lo cual la variable *Pos_May* queda con el valor 2. Como a continuación encontramos tanto el fin de la decisión como el fin del ciclo, entonces esto nos indica que debemos incrementar el contenido de la variable *Indice* en 1, con lo cual queda en dicha variable el valor 3.



Vuelve a hacerse la pregunta:

Si Vector (Indice) > Vector (Pos_May)

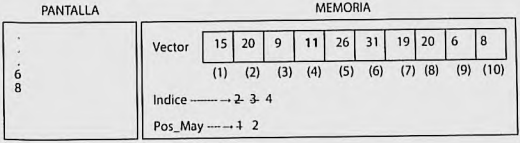
Como la variable *Indice* es igual a 3 y la variable *Pos_May* es igual a 2, entonces esta pregunta se convierte en:

Si Vector (3) > Vector (2)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 9 > 20

Como es falso, y dado que a continuación de esta decisión sigue el fin de ella y el fin del ciclo, entonces volvemos a incrementar el valor de la variable *Indice* en 1.



De la misma manera, volvemos a desarrollar la decisión:

Si Vector (Indice) > Vector (Pos_May)

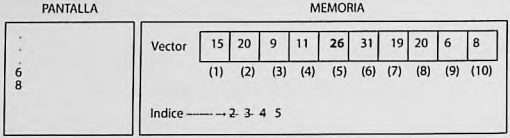
Como la variable *Indice* es igual a 4 y la variable *Pos_May* es igual a 2, entonces esta pregunta se convierte en:

Si Vector (4) > Vector (2)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 11 > 20

Note usted, a esta altura de la prueba de escritorio, que este algoritmo va comparando cada número con el último que haya encontrado como mayor. Como esta última decisión también es Falsa, y luego de haber encontrado el fin de la decisión y el fin el ciclo, entonces volvemos a incrementar en 1 el valor almacenado en la variable Indice.



Volvemos a hacer la pregunta:

Si Vector (Indice) > Vector (Pos_May)

Como la variable *Indice* es igual a 5 y la variable *Pos_May* es igual a 2, entonces esta pregunta se convierte en:

Si Vector (5) > Vector (2)

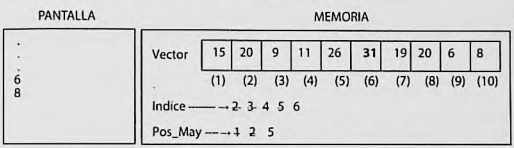
Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 26 > 20

Como es Verdadero, entonces se ejecuta la asignación:

Pos_May = Indice

Con lo cual en la variable Pos_May queda almacenado el valor 5. Como luego de esta asignación encontramos el fin de la decisión y el fin del ciclo. entonces incrementamos de nuevo el valor de la variable Indice en 1.



Volvemos a ejecutar la decisión:

Si Vector (Indice) > Vector (Pos_May)

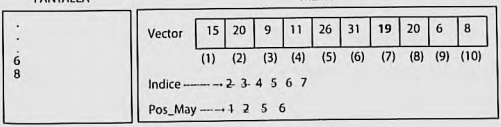
Como la variable *Indice* es igual a 6 y la variable *Pos_May* es igual a 5, entonces esta pregunta se convierte en:

Si Vector (6) > Vector (5)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 31 > 26

Como vemos que es Verdadero, entonces almacenamos en la variable Pos_ May el contenido almacenado en la variable Indice. Volvemos entonces a incrementar el contenido de la variable Indice en 1.



Ejecutamos la decisión:

Si Vector (Indice) > Vector (Pos_May)

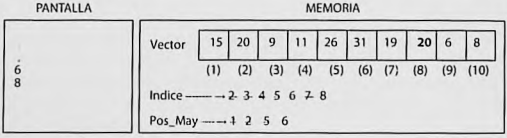
Como la variable *Indice* es igual a 7 y la variable *Pos_May* es igual a 6, entonces esta pregunta se convierte en:

Si Vector (7) > Vector (6)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 19 > 31

Como es Falso, entonces volvemos a incrementar el valor almacenado en la variable Indice.



Ejecutamos de nuevo la decisión:

Si Vector (Indice) > Vector (Pos_May)

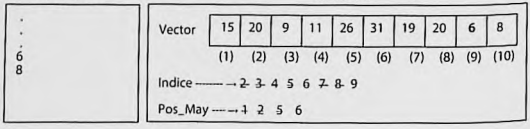
Como la variable *Indice* es igual a 8 y la variable *Pos_May* es igual a 6, entonces esta pregunta se convierte en:

Si Vector (8) > Vector (6)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 20 > 31

Igualmente al caso anterior, esta decisión es Falsa, razón por la cual, y luego de haber encontrado el fin de la decisión y el fin del ciclo, incrementamos el contenido de la variable Indice.



Se realiza una vez más la decisión:

Si Vector (Indice) > Vector (Pos_May)

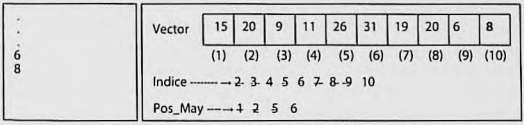
Como la variable *Indice* es igual a 9 y la variable *Pos_May* es igual a 6, entonces esta pregunta se convierte en:

Si Vector (9) > Vector (6)

Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 2 > 31

Como es Falsa la respuesta, entonces incrementamos de nuevo el contenido de la variable Indice.



Se ejecuta de nuevo la decisión:

Si Vector (Indice) > Vector (Pos_May)

Como la variable *Indice* es igual a 10 y la variable *Pos_May* es igual a 6, entonces esta pregunta se convierte en:

Si Vector (10) > Vector (6)

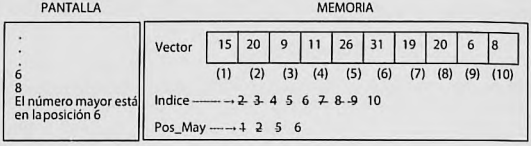
Por los valores almacenados en el vector vemos que la pregunta se convierte en:

Si 8 > 31

Cuya respuesta también es Falsa. Con esto hemos llegado al final del ciclo, pues inicialmente el ciclo planteaba la generación de números entre 2 y el número 10 para la variable Indice. Como esta variable ya vale 10 y se ha ejecutado con este valor el ciclo, entonces continuamos con la instrucción que se encuentra después del *Fin_Para* que representa el fin del ciclo y que es la instrucción que nos muestra el resultado solicitado en pantalla.

Escriba "El número mayor está en la posición", Pos_May

Con lo cual se escribiría en pantalla lo que está entre comillas dobles seguido por el contenido de la variable *Pos_May*.



Por último, lo que encontramos es el fin del algoritmo:

Fin

Hemos terminado la prueba de escritorio y solo nos queda determinar si el algoritmo cumplió su objetivo inicial, que era leer 10 números enteros y determinar en qué posición estaba el mayor de los números leídos. Vemos que, efectivamente, el número 31 es el mayor de los números que se encuentran almacenados en el vector y que corresponden a los números leídos. Con esto podemos concluir que el algoritmo está correcto y que es solución al objetivo planteado inicialmente.

¿Y qué tal que se quisiera decir, además de la posición del mayor, cuál es el número mayor...? Entonces el algoritmo que originalmente es así:

```
Programa Posic_Mayor
Variables
        Entero : Vector ( 10 ),
                         Indice,
                         Pos_May
Inicio
        Escriba "Digite 10 números enteros"

Para Indice = 1 hasta 10
                Lea Vector ( Indice )
        Fin_Para
        Pos_May = 1
        Para Indice = 2 hasta 10
                Si Vector (Indice) > Vector (Pos_May)
```

```
                    Pos_May = Indice
                Fin_Si
        Fin_Para

        Escriba "El número mayor está en la posición", Pos_May
Fin
```

Deberá quedar así (he resaltado los cambios para que se noten más):

```
Programa Posic_Mayor
Variables
        Entero : Vector ( 10 ),
                 Indice,
                 Pos_May,
                 Auxiliar        //Variable que almacenará el número
                                 // que se vaya encontrando como
                                 // mayor
Inicio
        Escriba "Digite 10 números enteros"

Para Indice = 1 hasta 10
                Lea Vector ( Indice )
        Fin_Para

        Pos_May = 1
        Auxiliar = Vector ( 1 )     // Almacena el contenido que hay en la
                                    // primera posición del vector

        Para Indice = 2 hasta 10
                Si Vector (Indice) > Vector (Pos_May)
                        Pos_May = Indice
                        Auxiliar = Vector ( Indice )
                        // Guarda provisionalmente en la
                        // variable Auxiliar el valor almacenado
                        // en el Vector en la posición Indice
                Fin_Si
        Fin_Para
```

```
        Escriba "El número mayor es", Auxiliar, "y está en la posición", Pos_May
                                                    // Muestra el número mayor y la
                                                    // posición en la cual se encuentra
Fin
```

¿Y si el objetivo hubiera sido determinar si el número mayor de 10 números leídos es par...? Entonces los cambios también serían muy sencillos. La última línea de escritura en donde está:

```
        Escriba "El número mayor es", Auxiliar, "y está en la posición", Pos_May

                                                    // posición en la cual se encuentra
```

se reemplazaría por:

```
Si Auxiliar / 2 * 2 = Auxiliar
        Escriba "El número mayor es", Auxiliar, "y es par"
                                                    // Muestra el número mayor y la
                                                    // posición en la cual se encuentra
Sino
        Escriba "El número mayor es igual a", Auxiliar, "y no es par"
```

Como puede ver, realizando algunos leves cambios en un algoritmo se pueden obtener otros y, por lo tanto, lograr otros objetivos.

9.3.4. Ejemplo con vectores No. 2

Leer 10 números enteros. Luego leer 1 número entero y determinar si este último número está entre los 10 primeros números leídos.

Clarificación del objetivo

De acuerdo al enunciado, primero vamos a leer 10 datos enteros y los vamos a almacenar en un vector (obviamente de 10 posiciones). Luego de tener almacenados los 10 enteros, vamos a leer otro número que será almacenado en otra variable. Se tratará entonces de preguntar si el contenido de la última variable es igual al contenido de alguna de las posiciones del vector. Si la respuesta a dicha pregunta es Verdadera, entonces deberemos avisar por pantalla y asimismo en caso de que sea Falsa.

Algoritmo

No olvide que cada algoritmo presentado aquí es apenas una de las posibles soluciones que pueda tenerse para alcanzar el objetivo.

```
Programa Búsqueda
Variables
        Entero : V ( 10 ),      // Almacenará los 10 números que el
                                // usuario va a digitar
                 Ind,           // Esta es la variable que nos va a
                                // servir como subíndice del vector
                 Num,           // Almacenará el número que se va a
                                // buscar
                 S              // El contenido de esta variable nos va
                                // a indicar si el número se encontró o
                                // no
Inicio
        Escriba "Digite 10 números enteros"     // Solicita los 10 números enteros
        Para Ind = 1 hasta 10 Paso 1    // Y los lee almacenando cada uno en
                                        // una posición diferente del vector
                Lea V ( Ind )
        Fin_Para
        Escriba "Ahora digite un número"  // Solicita el número a buscar
        Lea Num                           // y lo lee almacenándolo en la variable
                                          // Num
        S = 0                             // Inicializa la variable "misteriosa" en 0
        Para Ind = 1 hasta 10 Paso 1      // Recorre el vector desde la primera
                                          // posición hasta la última preguntando
                                          // si el contenido de cada posición es
                                          // igual al contenido de la variable Num
                Si V ( Ind ) = Num
                        Escriba "El dato", Num, "está en los 10 núms digitados"
                        S = 1
                Fin_Si                    // Si dicha respuesta es Verdadera,
                                          // entonces lo muestra en pantalla y
```

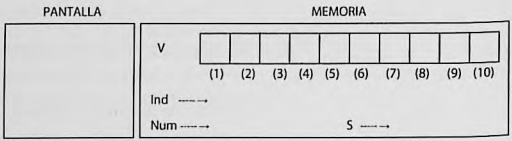
```
                                    // almacena en la variable "misteriosa"
                                    // el valor 1
    Fin_Para

                                    // Cuando haya terminado el ciclo,
                                    // pregunta por el valor almacenado en
                                    // la variable S. Si este valor aún es
                                    // cero, quiere decir que no encontró el
                                    // número buscado y por lo tanto lo
                                    // avisa en pantalla
    Si S = 0
        Escriba "El número", Num, "no está en los 10 números digitados"
    Fin_Si
Fin
```

Puede usted notar el uso de la variable *S*. Esta variable, a lo largo de todo el algoritmo, toma solo dos valores 1 o 0. Esto representa que la variable es utilizada a manera de interruptor (o también llamada *switche*). Esta forma de manejo de variables nos permite conocer respuestas que de otra forma sería muy difícil saberlas a nivel algorítmico.

Prueba de escritorio

Vamos a desarrollar paso a paso esta prueba no sin antes recomendarle que observe, en el desarrollo de la misma, el recorrido gráfico. La prueba, obviamente, inicia con la declaración de variables en memoria.

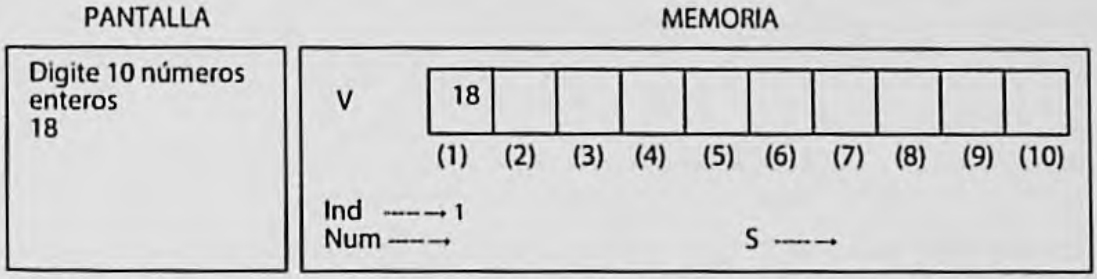
```
Programa Búsqueda
Variables
    Entero : V ( 10 ),
             Ind,
             Num
             S
```



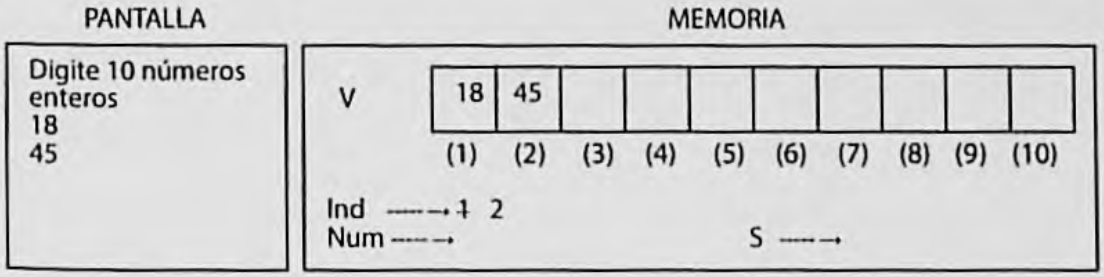
Lo primero que se hará en el algoritmo será "cargar" el vector con datos. Para ello, en pantalla saldrá un título, a continuación se leerán los datos y cada dato numérico se almacenará en cada una de las "casillas" del vector.

```
Inicio
        Escriba "Digite 10 números enteros"
        Para Ind = 1 hasta 10 Paso 1
                Lea V ( Ind )
        Fin_Para
```

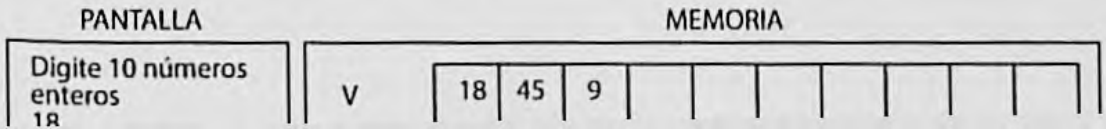
Se coloca el título en pantalla. Se inicia la variable *Ind* en 1 (e iremos incrementándola de 1 en 1 hasta llegar a 10) y a continuación se lee un entero y se almacena en el vector *V* en la posición 1. Supongamos que el número leído es 18, entonces:



A continuación, se incrementa el contenido de la variable *Ind* en 1 y se lee un nuevo dato almacenándose en el vector *V* en la posición 2. Supongamos que el número leído sea 45, entonces:



Incrementamos el contenido de la variable *Ind* en 1 y volvemos a leer otro dato almacenándolo en el vector *V* en la posición 3. Asumamos que el valor leído es 9.



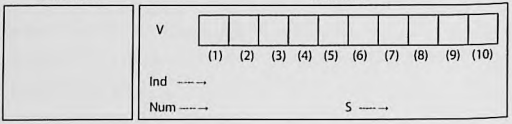
```
                                    // almacena en la variable "misteriosa"
                                    // el valor 1
    Fin_Para

                                    // Cuando haya terminado el ciclo,
                                    // pregunta por el valor almacenado en
                                    // la variable S. Si este valor aún es
                                    // cero, quiere decir que no encontró el
                                    // número buscado y por lo tanto lo
                                    // avisa en pantalla
    Si S = 0
            Escriba "El número", Num, "no está en los 10 números digitados"
    Fin_Si
Fin
```

Puede usted notar el uso de la variable *S*. Esta variable, a lo largo de todo el algoritmo, toma solo dos valores 1 o 0. Esto representa que la variable es utilizada a manera de interruptor (o también llamada *switche*). Esta forma de manejo de variables nos permite conocer respuestas que de otra forma sería muy difícil saberlas a nivel algorítmico.

Prueba de escritorio

Vamos a desarrollar paso a paso esta prueba no sin antes recomendarle que observe, en el desarrollo de la misma, el recorrido gráfico. La prueba, obviamente, inicia con la declaración de variables en memoria.

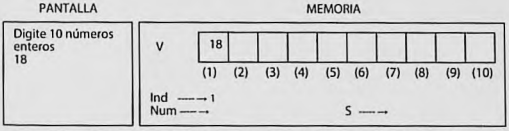
```
Programa Búsqueda
Variables
    Entero : V ( 10 ),
             Ind,
             Num
             S
```



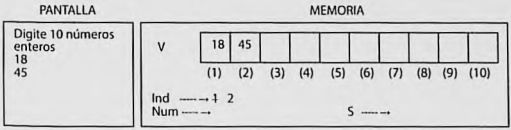
Lo primero que se hará en el algoritmo será "cargar" el vector con datos. Para ello, en pantalla saldrá un título, a continuación se leerán los datos y cada dato numérico se almacenará en cada una de las "casillas" del vector.

```
Inicio
        Escriba "Digite 10 números enteros"
        Para Ind = 1 hasta 10 Paso 1
                Lea V (Ind)
        Fin_Para
```

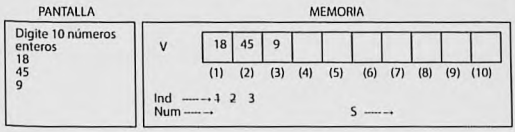
Se coloca el título en pantalla. Se inicia la variable *Ind* en 1 (e iremos incrementándola de 1 en 1 hasta llegar a 10) y a continuación se lee un entero y se almacena en el vector *V* en la posición 1. Supongamos que el número leído es 18, entonces:



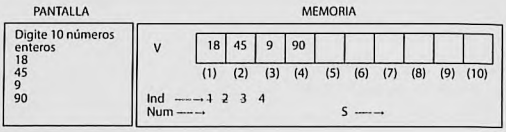
A continuación, se incrementa el contenido de la variable *Ind* en 1 y se lee un nuevo dato almacenándose en el vector *V* en la posición 2. Supongamos que el número leído sea 45, entonces:



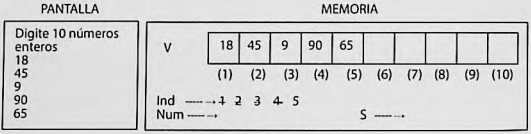
Incrementamos el contenido de la variable *Ind* en 1 y volvemos a leer otro dato almacenándolo en el vector *V* en la posición 3. Asumamos que el valor leído es 9.



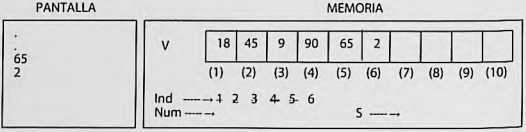
Incrementamos el contenido de la variable *Ind* en 1 y volvemos a leer otro dato almacenándolo en el vector *V* en la posición 4. Asumamos que el valor leído es 90.



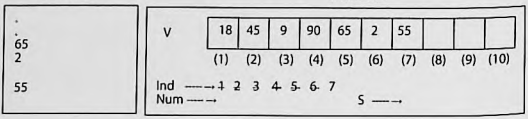
Volvemos a incrementar en 1 el contenido de la variable *Ind* y leemos un dato entero que quedará almacenado en el vector *V* en la posición 4. Asumamos que el valor leído es 65.



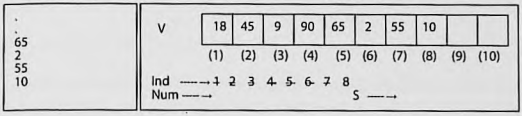
Se incrementa el contenido de la variable *Ind*, la cual queda con el valor 6, y se lee un nuevo dato que se almacenará en el vector *V* en la posición 6. Asumamos que el valor leído es 2.



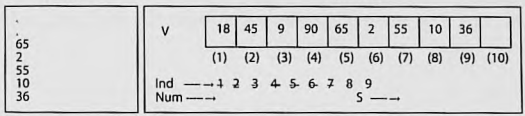
Incrementamos de nuevo el contenido de *Ind* en 1 y leemos un nuevo dato.



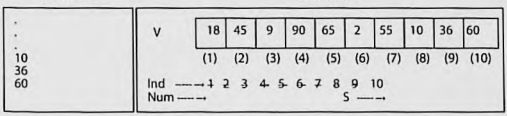
Volvemos a incrementar el contenido de la variable *Ind* y volvemos a leer un nuevo dato que quedará almacenado en el vector *V* en la posición 8.



Incrementamos de nuevo el contenido de la variable *Ind* en 1 y leemos un dato entero que quedará almacenado en el vector *V* en la posición 9. Asumamos que el valor leído es 36.

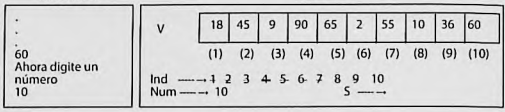


Por último, incrementamos el contenido de la variable *Ind* y llegamos al tope (10). Leemos el último dato que va a ser almacenado en el vector *V* en la posición 10.



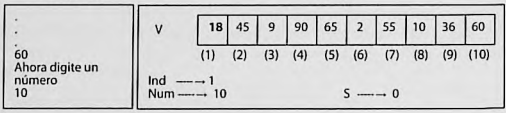
Como hemos llegado al tope del ciclo, entonces continuamos con las instrucciones que se encuentran después del *Fin_Para* correspondiente al ciclo que acabamos de terminar. Estas instrucciones son una orden de escritura para solicitar un número y la lectura de un número entero que se ha de almacenar en la variable *Num*. Supongamos que ese número leído es el número 10.

Escriba "Ahora digite un número"
Lea Num



Inicializamos la variable *S* en 0 y generamos un ciclo que va desde 1 hasta 10 sobre la variable *Ind*, ciclo que nos servirá para comparar el contenido de la variable *Num* (que es el número que queremos buscar) con cada uno de los números almacenados en el vector *V*.

```
S = 0
Para Ind = 1 hasta 10 Paso 1
        Si V ( Ind ) = Num
            Escriba "El número", Num, "si está en los 10 números digitados"
                S = 1
        Fin_Si
Fin_Para
```



De esta forma, cuando el contenido de la variable *Ind* sea 1, la decisión

```
Si V ( Ind ) = Num
```

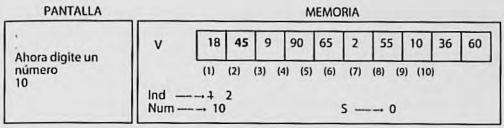
se convertirá en

```
Si V ( 1 ) = Num
```

y como el vector *V* en la posición 1 almacena el número 18 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

```
Si 18 = 10
```

Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



De esta forma, cuando el contenido de la variable *Ind* sea 2 la decisión

Si V (Ind) = Num

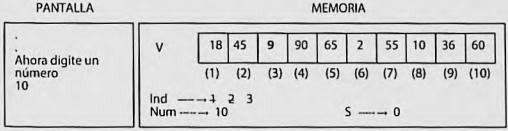
se convertirá en

Si V (2) = Num

y como el vector *V* en la posición 2 almacena el número 45 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

Si 45 = 10

Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



De esta forma, cuando el contenido de la variable *Ind* sea 3, la decisión

Si V (Ind) = Num

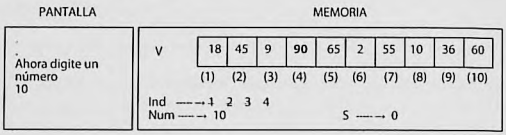
se convertirá en

Si V (3) = Num

y como el vector *V* en la posición 3 almacena el número 9 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

Si 9 = 10

Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



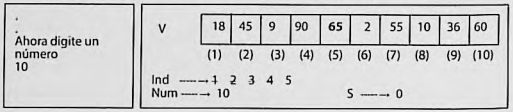
Cuando el contenido de la variable *Ind* sea 4, la decisión se convertirá en

Si V (Ind) = Num
Si V (4) = Num

y como el vector *V* en la posición 4 almacena el número 90 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

Si 90 = 10

Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



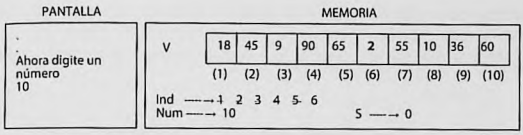
Cuando el contenido de la variable *Ind* sea 5, la decisión se convertirá en

Si V (Ind) = Num
Si V (5) = Num

y como el vector *V* en la posición 5 almacena el número 65 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

Si 65 = 10

Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



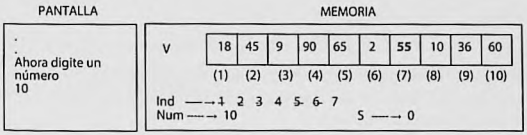
Cuando el contenido de la variable *Ind* sea 6, la decisión se convertirá en

Si V (Ind) = Num
Si V (6) = Num

y como el vector *V* en la posición 6 almacena el número 2 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en

Si 2 = 10

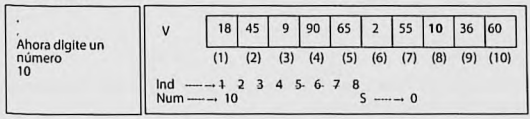
Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



Cuando el contenido de la variable *Ind* sea 7, la decisión se convertirá en

Si V (Ind) = Num
Si V (7) = Num

y como el vector *V* en la posición 7 almacena el número 55 y el contenido de la variable *Num* es 10, entonces la decisión internamente, en últimas, se convierte en *Si 55 = 10.* Como la respuesta es Falso, entonces, luego de encontrar el *Fin_Si* y el *Fin_Para*, volvemos a incrementar el contenido de la variable *Ind* en 1 y volvemos a hacer la pregunta correspondiente.



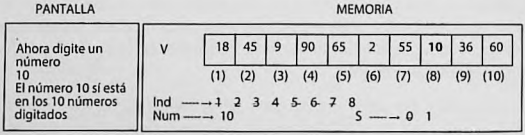
Cuando el contenido de *Ind* sea 8, entonces la decisión *Si V (Ind) = Num* se convierte en *Si V (8) = Num* y haciendo uso de sus respectivos contenidos la misma decisión representa

Si 10 = 10

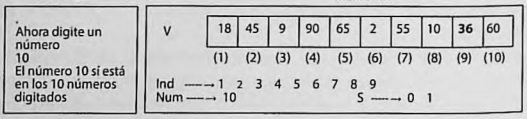
Como la respuesta es Verdadero, entonces se ejecutan las órdenes que están dentro de la parte verdadera de la decisión, o sea, las siguientes dos instrucciones:

Escriba "El número", Num, "si está en los 10 números digitados"
S = 1

Por lo tanto, se coloca el título en pantalla y se cambia el valor de *S* a 1.



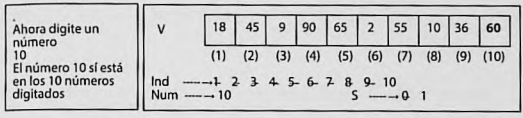
Encontramos de nuevo el *Fin_Si* y el *Fin_Para* correspondiente, con lo cual volvemos de todas formas a incrementar en 1 el contenido de la variable *Ind* y volvemos a hacer la pregunta.



Cuando el contenido de *Ind* sea igual a 9, entonces la decisión *Si V (Ind) = Num* se convierte en *Si V (9) = Num* y como *V* (9) es igual a 36 y *Num* es igual a 10, entonces la decisión finalmente es:

Si 36 = 10

Como la respuesta es Falso, entonces volvemos a incrementar (por última vez) el contenido de la variable *Ind.*



Cuando el contenido de *Ind* sea igual a 10, entonces la decisión *Si V (Ind) = Num* se convierte en *Si V (10) = Num* y como *V* (10) es igual a 36 y *Num* es igual a 10, entonces la decisión finalmente es:

Si 60 = 10

Como la respuesta es Falso y ya llegamos al tope del ciclo, entonces seguimos con la instrucción que está después del *Fin_Para* correspondiente.

Si S = 0
Escriba "El número", Num, "no está en los 10 números digitados"
Fin_Si

Podemos ver que el contenido actual de la variable *S* es 1, por lo tanto, la pregunta *Si S = 0* es Falsa y, por lo tanto, pasamos a la instrucción que está después del *Fin_Si* y es el *Fin* del algoritmo.

Fin

Puede usted notar que el algoritmo falla en el hecho de que, a pesar de que se encuentre el valor buscado, él sigue buscando sabiendo que el objetivo era sencillamente decir si el dato buscado está o no. Para ello, le sugiero que le haga una prueba de escritorio a la siguiente versión solución de este mismo problema.

```
Programa Búsqueda
Variables
    Entero : V ( 10 ),      // Almacenará los 10 números que el
                            // usuario va a digitar
             Ind,           // Esta es la variable que nos va a
                            // servir como subíndice del vector
             Num,           // Almacenará el número que se va a
                            // buscar
```

```
                    S                   // El contenido de esta variable nos va
                                        // a indicar si el número se encontró o
                                        // no.
Inicio
    Escriba "Digite 10 números enteros"     // Solicita los 10 números enteros
    Para Ind = 1 hasta 10 Paso 1        // Y los lee almacenando cada uno en
                                        // una posición diferente del vector
        Lea V ( Ind )
    Fin_Para
    Escriba "Ahora digite un número"        // Solicita el número a buscar
    Lea Num                 // y lo lee almacenándolo en la variable Num
    S = 0                               // Inicializa la variable "misteriosa" en 0
    Ind = 1
    Mientras Ind <= 10 Y S = 0
        Si V ( Ind ) = Num
            Escriba "El número", Num,
            "si está en los 10 números leídos"
            S = 1
        Fin_Si
    Fin_Mientras
    Si S = 0
        Escriba "El número buscado no está"
    Fin_Si
Fin
```

La parte resaltada busca condicionar el ciclo a que la variable *Ind* no haya llegado hasta 10 y, al mismo tiempo, a que no se haya encontrado el dato buscado, pues esa es la única forma de que *S* mantenga el valor 0, pues apenas se encuentre el contenido de la variable *S* cambiará a 1. Usted tal vez se preguntará: ¿y qué pasa si se quisiera implementar el mismo control pero esta vez utilizando el ciclo *Para*...? Entonces, basados en el ciclo *Para*, que era:

```
Para Ind = 1 hasta 10 Paso 1
    Si V ( Ind ) = Num
```

```
                Escriba "El número", Num, "si está en los 10 números digitados"
                S = 1
            Fin_Si
        Fin_Para
```

Con el control que se propone quedaría:

```
    Para Ind = 1 hasta 10 Paso 1
        Si V ( Ind ) = Num
                Escriba "El número", Num, "si está en los 10 números digitados"
                S = 1
                Ind = 11
            Fin_Si
        Fin_Para
```

Siendo su único cambio la línea u orden que se encuentra en negrilla. Sugiero que le realice una buena prueba de escritorio para que vea la diferencia en la ejecución de este algoritmo incluyendo el control que se le ha incorporado.

9.3.5. Ejemplo con vectores No. 3

Leer 10 números enteros. Luego, leer 1 número entero y determinar cuántas veces está entre los 10 primeros; además decir en qué posiciones está.

Clarificación del objetivo

Como usted puede notar, el algoritmo anterior se podrá tomar como base para desarrollar otro algoritmo que nos permita lograr este objetivo. En esencia, lo que se busca es leer 10 números enteros, que obviamente serán almacenados en un vector, y luego leer otro número entero con el objetivo de buscarlo entre los 10 números leídos y determinar no solo la cantidad de veces que está repetido, sino también en qué posiciones se encuentra. Esto quiere decir que si el vector se "cargara" con los siguientes números

Vector V	15	20	16	24	23	16	19	16	25	82
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

y el número leído fuera 16, entonces en pantalla deberá aparecer:

```
El número 16 se encuentra en
las siguientes posiciones
3
6
8
En total está 3 veces
```

Algoritmo

Es obvio pensar en la gran similitud que tendrá este algoritmo con el anterior. Solo por ejercicio, vamos a desarrollarlo usando solamente ciclos *Mientras*.

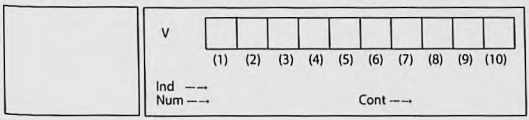
```
Programa Busqueda_Detallada
Variables
    Entero : V ( 10 ), // Almacenará los 10 números leídos
                Ind,    // Servirá como índice de los ciclos para
                        // manejar las diferentes posiciones del vector
                Num,    // Almacenará el número a buscar
                Cont    // Almacenará la cantidad de veces que esté el
                        // número en el vector y servirá para determinar
                        // si el número estaba o no
Inicio
    Escriba "Digite 10 números enteros" // Solicita los 10 números
    Ind = 1                 // Inicializa la variable Ind en 1 para hacer
                            // referencia a la primer posición
    Mientras Ind <= 10      // Mientras no se haya llegado a la última
                            // posición del vector
        Lea V ( Ind )       // Lea un dato entero y guárdelo en la posición
                            // Ind del vector V
        Ind = Ind + 1       // Pase a la siguiente posición del vector
    Fin_Mientras            // Fin del Ciclo
    Escriba "Digite un número entero" // Solicita el número a buscar
    Lea Num                 // Lo lee y lo almacena en la variable Num
    Cont = 0        // Inicializa el contador en ceros
```

```
Ind = 1          // Inicializa la variable Ind en 1 para hacer
                 // referencia a la primera posición

Mientras Ind <= 10          // Mientras no se haya llegado a la última
                            // posición en el vector

    Si Num = V ( Ind )      // Si el número buscado es igual al número
                            // almacenado en posición Ind del vector V
                            // Escriba en dónde lo encontró
        Escriba "El número", Num, "está en la posición", Ind
        Cont = Cont + 1     // Incremente el contador pues lo ha
                            // encontrado una vez ( mas )
    Fin_Si                  // Fin de la decisión

    Ind = Ind + 1           // Pase a la siguiente posición en el vector
Fin_Mientras                // Fin del Ciclo

Si Cont = 0                 // Si el contador es cero o sea si en ningún
                            // momento lo encontró entonces avise
    Escriba "El número no se encuentra"
Sino                // Si el número estaba entonces muestre en
                    // pantalla cuántas veces estaba
    Escriba "En total está", Cont, "veces"
Fin_Si
Fin                 // Fin del algoritmo
```

Prueba de escritorio

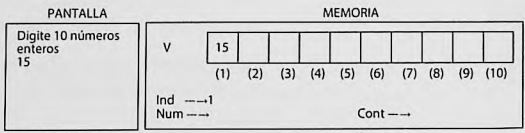
De nuevo vamos a desarrollar una prueba de escritorio paso a paso apoyándonos en el soporte gráfico para conceptualizar mejor el flujo del algoritmo. Primero que nada declaramos en memoria las variables que se van a necesitar.

```
Programa Busqueda_Detallada
Variables
    Entero : V ( 10 ), Ind, Num, Cont
```

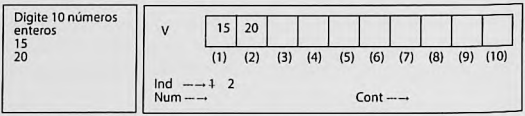


Seguidamente, se inicia el algoritmo escribiendo en pantalla un aviso solicitando que se digiten 10 números. Luego se inicia la variable *Ind* en 1 y mientras esta variable no haya llegado a 10 (o sea, mientras no se haya llegado a la última posición del vector), se va a leer un dato y se almacenará en el vector *V* en la posición *Ind* y luego se va a incrementar *Ind* en 1.

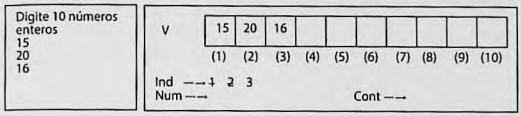
```
Inicio
    Escriba "Digite 10 números enteros"
    Ind = 1
    Mientras Ind <= 10
        Lea V ( Ind )
        Ind = Ind + 1
    Fin_Mientras
```



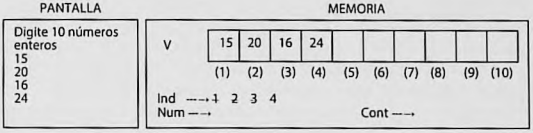
Cuando *Ind* valga 1, entonces el dato se almacenará en el vector *V* en la posición 1. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



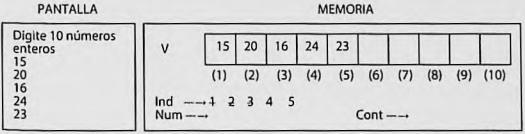
Cuando *Ind* valga 2, entonces el dato se almacenará en el vector *V* en la posición 2. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



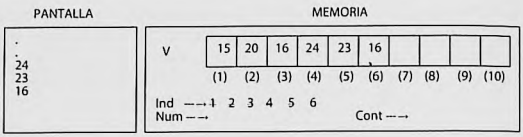
Cuando *Ind* valga 3, entonces el dato se almacenará en el vector *V* en la posición 3. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10.* Como es Verdadero, entonces volvemos a leer otro dato.



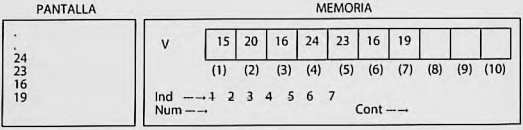
Cuando *Ind* valga 4, entonces el dato se almacenará en el vector *V* en la posición 4. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10.* Como es Verdadero, entonces volvemos a leer otro dato.



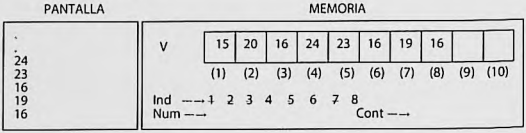
Cuando *Ind* valga 5, entonces el dato se almacenará en el vector *V* en la posición 5. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10.* Como es Verdadero, entonces volvemos a leer otro dato.



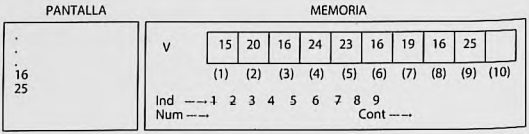
Cuando *Ind* valga 6, entonces el dato se almacenará en el vector *V* en la posición 6. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



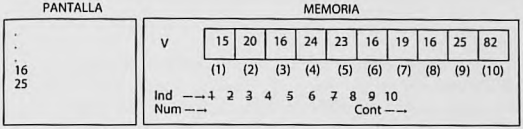
Cuando *Ind* valga 7, entonces el dato se almacenará en el vector *V* en la posición 7. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



Cuando *Ind* valga 8, entonces el dato se almacenará en el vector *V* en la posición 8. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



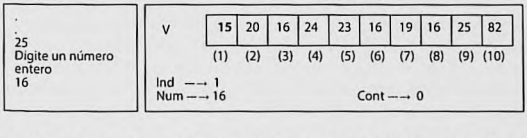
Cuando *Ind* valga 9, entonces el dato se almacenará en el vector *V* en la posición 9. Luego de esto, se incrementará en 1 el contenido de la variable *Ind* y se verifica si todavía *Ind es menor o igual que 10*. Como es Verdadero, entonces volvemos a leer otro dato.



Y cuando *Ind* valga 10, entonces el dato se almacenará en el vector *V* en la posición 10. Como este valor era el tope del ciclo, entonces este se finaliza y se continúa con la instrucción que se encuentra después de su respectivo *Fin_Para*. Continuando con la instrucción correspondiente, aparecerá en pantalla un título solicitando un número y seguidamente se deberá leer un número entero que ha de quedar almacenado en la variable *Num*. Vamos pues a asumir que el número leído para ser buscado es el número 16.

Escriba "Digite un número entero"

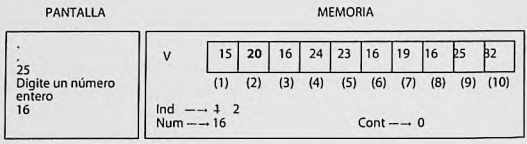
Lea Num



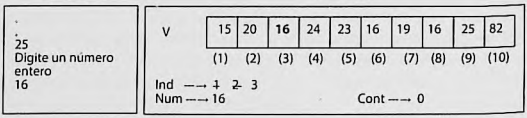
Luego de leído el número a buscar, iniciamos la variable *Cont* con el valor 0, dado que esta es la variable que nos va a permitir contar cuántas veces está el número buscado entre los 10 números leídos. Igualmente, se inicializa (de nuevo) la variable *Ind* en 1 para hacer referencia a la primera posición del vector y con ella misma controlar el recorrido a lo largo de las diferentes posiciones del mismo. Seguidamente, se inicia un ciclo *Mientras*, en donde su condición establece *Mientras no se haya llegado a la última posición del vector* (o como quien dice *Mientras Ind <= 0*). En este ciclo, se va a evaluar si el contenido del vector *V* en la posición *Ind* es igual al número que estamos buscando. De ser así, entonces se escribirá en pantalla que el número que buscamos está en la posición que esté almacenada en la variable *Ind* y, a continuación, se incrementará en 1 el contenido de la variable *Cont*, que es la que va a almacenar la

cantidad de veces que se encuentre el número en el vector. De otra parte, sea Verdadera o Falsa la pregunta *Si Num = V (Ind)*, se incrementará de todas maneras el contenido de la variable *Ind* para pasar a evaluar el valor almacenado en la siguiente "casilla" del vector. Así se seguirá progresivamente hasta llegar a la última posición, tal como lo indica la condición del ciclo.

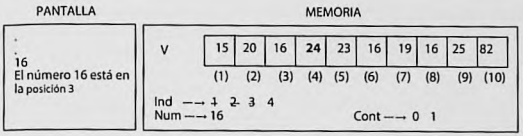
```
Cont = 0
Ind = 1
Mientras Ind <= 10
        Si Num = V ( Ind )
                Escriba "El número", Num, "está en la posición", Ind
                Cont = Cont + 1
        Fin_Si
        Ind = Ind + 1
Fin_Mientras
```



Cuando *Ind* valga 1, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (1)* y como el contenido del vector V en la posición 1 es 15, entonces la decisión se convierte en *Si 16 = 15*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



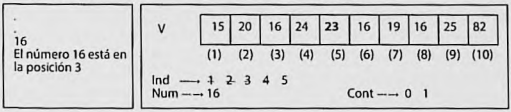
Cuando *Ind* valga 2, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (2)* y como el contenido del vector V en la posición 2 es 20, entonces la decisión se convierte en *Si 16 = 20*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



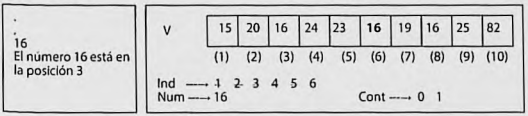
Cuando *Ind* valga 3, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (3)* y como el contenido del vector V en la posición 3 es 16, entonces la decisión se convierte en *Si 16 = 16*, lo cual es Verdadero. Por lo tanto, ejecutamos las órdenes:

Escriba "El número", Num, "está en la posición", Ind
Cont = Cont + 1

O sea, que en pantalla aparece el aviso correspondiente diciendo que el número buscado está en la posición que es igual al contenido de la variable *Ind* (que en este momento vale 3) y se incrementa en 1 el contenido de la variable *Cont*. Posteriormente, se incrementará en 1 el contenido de la misma variable *Ind*.

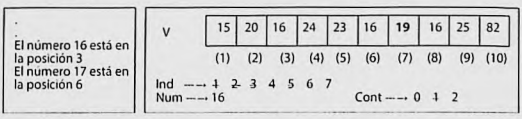


Cuando *Ind* valga 4, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (4)* y como el contenido del vector V en la posición 4 es 24, entonces la decisión se convierte en *Si 16 = 24*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



Cuando *Ind* valga 5, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (5)* y como el contenido del vector V en la posición 5 es 23, entonces la decisión

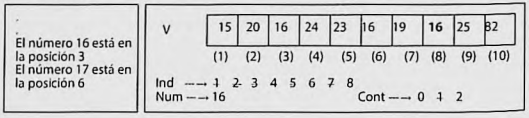
se convierte en *Si 16 = 23*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



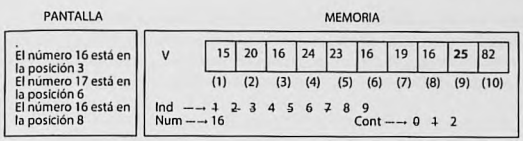
Cuando *Ind* valga 6, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (6)* y como el contenido del vector V en la posición 6 es 16, entonces la decisión se convierte en *Si 16 = 16*, lo cual es Verdadero. Por lo tanto, se ejecutan las órdenes:

Escriba "El número", Num, "está en la posición", Ind
Cont = Cont + 1

Es decir, se escribe en pantalla que el valor almacenado en la variable *Num* (que es el dato que estamos buscando) está en la posición 6 (que es igual al contenido de la variable *Ind*) y luego se incrementará el contenido de la variable *Cont* (que es la que va a almacenar la cantidad de veces que se encuentra el dato buscado). Después de esto, se incrementará en 1 el contenido de la variable *Ind* y se volverá a evaluar la condición del ciclo para repetir el mismo proceso pero con el dato que se encuentra enseguida de la posición actual.



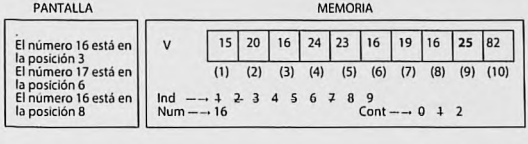
Cuando *Ind* valga 7, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (7)* y como el contenido del vector V en la posición 7 es 19, entonces la decisión se convierte en *Si 16 = 19*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



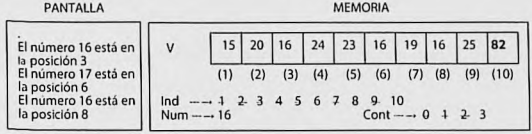
Cuando *Ind* valga 8, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (8)* y como el contenido del vector V en la posición 8 es 16, entonces la decisión se convierte en *Si 16 = 16*, lo cual es Verdadero. Por lo tanto, se ejecutan las órdenes:

Escriba "El número", Num, "está en la posición", Ind
Cont = Cont + 1

Es decir, se escribe en pantalla que el valor almacenado en la variable *Num* (que es el dato que estamos buscando) está en la posición 8 (que es igual al contenido de la variable *Ind*) y luego se incrementará el contenido de la variable *Cont* (que es la que va a almacenar la cantidad de veces que se encuentra el dato buscado). Después de esto, se incrementará en 1 el contenido de la variable *Ind* y se volverá a evaluar la condición del ciclo para repetir el mismo proceso pero con el dato que se encuentra enseguida de la posición actual.



Cuando *Ind* valga 9, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (9)* y como el contenido del vector V en la posición 9 es 25, entonces la decisión se convierte en *Si 16 = 25*, lo cual es Falso. Entonces se incrementa en 1 el contenido de la variable *Ind* y se vuelve a evaluar la condición del ciclo, pues se ha encontrado el correspondiente *Fin_Mientras*.



Cuando *Ind* valga 10, la decisión *Si Num = V (Ind)* se convierte en *Si Num = V (10)* y como el contenido del vector V en la posición 10 es 82, entonces la decisión se convierte en *Si 16 = 82*, lo cual es Falso. En este punto finaliza el ciclo, dado que se alcanza el último valor planteado al inicio del mismo.

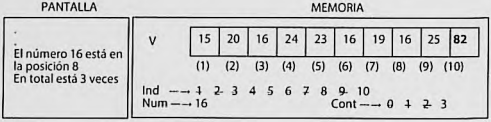
Luego de que se ha realizado este ciclo, se pasa a la decisión:

```
Si Cont = 0
        Escriba "El número no se encuentra"
Sino
        Escriba "En total está", Cont, "veces"
Fin_Si
```

Con la cual podemos saber si el número buscado fue encontrado o no. Este resultado depende del contenido de la variable *Cont*, pues si esta variable contiene un cero es porque el número buscado no estuvo presente ni siquiera una sola vez. Si el contenido de esta variable es diferente de cero, entonces quiere decir que el algoritmo encontró el número buscado al menos una vez. En este caso, como el contenido de la variable *Cont* es 3, entonces se ejecuta:

```
Sino
        Escriba "En total está", Cont, "veces"
```

Con lo cual aparecería en pantalla:



Lo cual, por las características específicas de la prueba de escritorio, es verdad. Luego de esto, encontramos el fin del algoritmo.

Fin

Y con ello termina el mismo. Es momento entonces de evaluar los resultados y compararlos con el objetivo propuesto. Vemos que, si entre los dígitos

15	20	16	24	23	16	19	16	25	82

queremos buscar el número 16, encontramos que este está 3 veces. También vemos que la primera ocurrencia de este número se da en la posición 3, la segunda ocurrencia se da en la posición 6 y la tercera ocurrencia se da en la posición 8. Como esto precisamente coincide con los resultados entregados por el algoritmo, entonces podemos garantizar que este algoritmo está bien.

Ejemplo con vectores No. 4

"Cargar" dos vectores cada uno con 5 datos enteros y determinar si los datos almacenados en ambos vectores son exactamente los mismos tanto en contenido como en posición.

Clarificación del objetivo

Se trata entonces de comparar dos conjuntos de datos y determinar si ambos conjuntos son exactamente iguales. Para ello, lo que vamos a hacer será leer 5 datos enteros y los almacenaremos en un vector; luego leeremos otros 5 datos enteros y los almacenaremos en otro vector. Luego implementaremos un proceso de comparación que nos permite determinar si ambos vectores son exactamente iguales. Tenga en cuenta que la comparación entre vectores no se puede hacer como se hace con las variables. Si usted tiene una variable *X* con un contenido igual a 8

$$X = 8$$

y tiene una variable *Y* con un contenido igual a 8

$$Y = 8$$

entonces en cualquier momento usted puede preguntar

Si $X = Y$

Y el computador entenderá que deberá comparar los contenidos de dichas variables dado que dichos contenidos son únicos. Pero si se tiene el vector *V1* con los siguientes contenidos

Vector V1	5	8	9	12	18
	(1)	(2)	(3)	(4)	(5)

Y tenemos el vector *V2* con los siguientes contenidos

Vector V2	6	9	11	5	8
	(1)	(2)	(3)	(4)	(5)

No podremos realizar la pregunta

Si V1 = V2

Dado que, como cada uno es un vector y no contiene datos únicos, entonces realmente no sabría el computador (cuando este algoritmo se convierta en un programa) qué es lo que realmente tiene que comparar.

Precisamente, para implementar dicha comparación, lo que vamos a hacer será comparar una a una y, análogamente, cada contenido de los vectores. Por lo dicho aquí, tendremos que recorrer los vectores, posición a posición, valiéndonos de un ciclo que irá desde la primera hasta la última posición (en este caso, de 1 hasta 5). Nuestra pregunta será:

Si V1 (Ind) = V2 (Ind)

Siendo *Ind* la variable que vamos a utilizar para referenciar el subíndice de posición de cada uno de los datos. Antes de iniciar con este ciclo de comparación, asignaremos a una variable el valor 0 y esta variable actuará como un interruptor, es decir, cambiará su valor solo si se encuentra que, en algún momento, la decisión planteada es Falsa. Al finalizar el ciclo, veremos cuál fue el valor de la variable interruptor y con ello podremos determinar si los dos vectores eran iguales o no. Debemos tener en cuenta que el proceso de comparación de dos vectores es diferente al proceso de comparación de dos variables, dado que los vectores son conjuntos de variables y las variables solas no.

Algoritmo

No olvide lo que a lo largo de este libro le he recordado: es posible que usted desarrolle un algoritmo que logre este mismo objetivo; si no es igual al que aquí se expone, no se preocupe. Todo lo que tiene que hacer es verificar si cumple con el objetivo o no realizándole una buena prueba de escritorio. Si su solución cumple con el objetivo, entonces estará bien y si la solución expuesta en este libro cumple con el objetivo, entonces también estará bien. Siendo así, ambas soluciones serán correctas.

Programa Compara_Vectores
Variables
Entero : V1 (5), // Vector en donde se almacenará el primer
// conjunto de 5 datos enteros
V2 (5), // Vector en donde se almacenará el segundo
// conjunto de 5 datos enteros

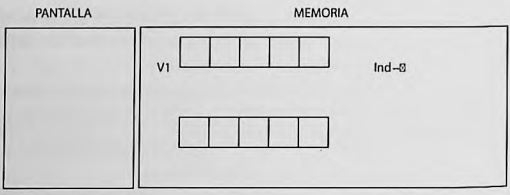
```
                Ind,        // Variable que servirá como subíndice
                S           // Variable que permitirá determinar si el
                                // contenido de los dos vectores era
                                    // exactamente igual o no

Inicio
    Escriba "Digite el primer conjunto de 5 enteros"
    Para Ind = 1 hasta 5            // Se solicitan 5 números enteros y se leen
        Lea V1 ( Ind )              // almacenando cada uno en una posición
    Fin_Para                    // diferente del vector V1

    Escriba "Digite el segundo conjunto de 5 enteros"

    Para Ind = 1 hasta 5            // Se solicitan los otros 5 números enteros
        Lea V2 ( Ind )              // y se leen almacenando cada uno en una
    Fin_Para                    // posición diferente del vector V2

    S = 0                       // Se inicializa la variable S en 0
    Ind = 1                     // Se inicia la variable Ind en 1 para hacer
                                // referencia al 1º elemento de los vectores
                                // e iniciar allí la comparación

    Mientras Ind <= 5 Y S = 0   // Mientras no llegue al final del vector
                                    // (o sea a su última posición) y
                                    mientras la
                                // variable S siga con el valor 0 (o sea mientras
                                // los vectores sigan iguales en sus contenidos)
        Si V1 ( Ind ) != V2 ( Ind ) // Si el contenido del 1º vector en una
                                    // posición es igual al contenido
                                    // del 2º vector en la misma posición
            S = 1                   // Cambie el contenido S a 1
        Fin_S                   // Fin de la decisión
        Ind = Ind + 1           // Pase a la siguiente posición del vector
    Fin_Mientras                // Fin del Ciclo

    Si S = 0                    // Si esta variable aún permanece con el valor 0
                                // quiere decir que en ningún momento se le
                                // asignó el valor 1 o sea que en ningún
                                // momento el contenido de los vectores fue
```

```
                                    // diferente por lo tanto los vectores son
                                    // exactamente iguales
                        Escriba "El contenido de los dos vectores es exactamente igual"
        Sino                        // Si la variable S vale 1 entonces quiere decir
                                    // que en algún momento el contenido de los
                                    // vectores fueron diferente por lo tanto no son
                                    // exactamente iguales
                        Escriba "El contenido de los dos vectores es diferente"
        Fin_Si                      // Fin de la decisión
Fin                                 // Fin del ciclo
```

Prueba de escritorio

Realice un seguimiento de esta prueba paso a paso tanto en lo que corresponde al manejo de la pantalla como en lo que corresponde al manejo de las variables en la memoria. Lo primero que hacemos es declarar las variables correspondientes en memoria.

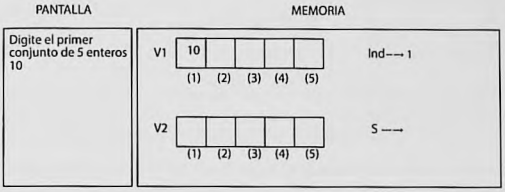
```
Programa Compara_Vectores
Variables
        Entero: V1 (5),
                V2 (5),
                Ind,
                S
```



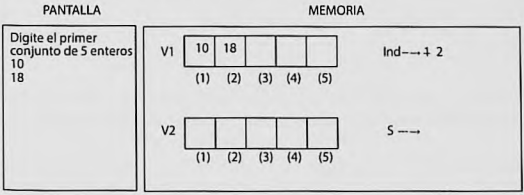
A continuación, solicitamos 5 datos enteros y los leemos valiéndonos de un ciclo que, utilizando la variable *Ind*, vaya desde 1 hasta 5 para referenciar cada una de las posiciones dentro del vector. Con este ciclo queda "cargado" el vector *V1*.

Inicio

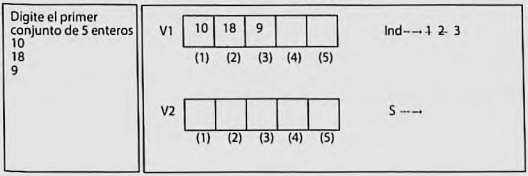
Escriba "Digite el primer conjunto de 5 enteros"
Para Ind = 1 hasta 5
Lea V1 (Ind)
Fin_Para



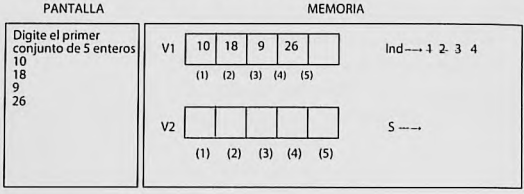
Cuando la variable *Ind* almacene el número 1, y asumiendo que el número leído sea el 10, dicho número quedará almacenado en el vector *V1* en la posición 1. Se incrementa entonces el valor de la variable *Ind* en 1.



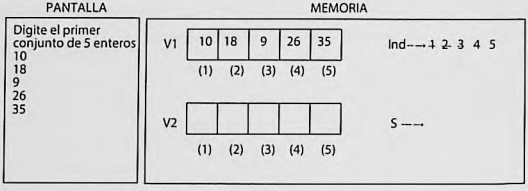
Cuando la variable *Ind* almacene el número 2, y asumiendo que el número leído sea el 18, dicho número quedará almacenado en el vector *V1* en la posición 2. Se incrementa entonces el valor de la variable *Ind* en 1.



Cuando la variable *Ind* almacene el número 3, y asumiendo que el número leído sea el 9, dicho número quedará almacenado en el vector *V1* en la posición 3. Se incrementa entonces el valor de la variable *Ind* en 1.



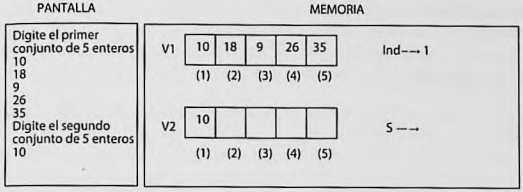
Cuando la variable *Ind* almacene el número 4, y asumiendo que el número leído sea el 9, dicho número quedará almacenado en el vector *V1* en la posición 4. Se incrementa entonces el valor de la variable *Ind* en 1.



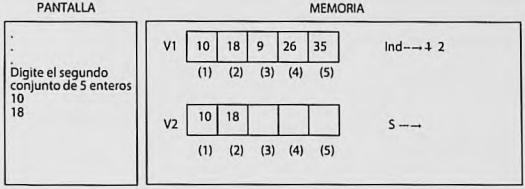
Y cuando la variable *Ind* almacene el número 5, y asumiendo que el número leído sea el 35, dicho número quedará almacenado en el vector *V1* en la posición 5. Como este es el tope hasta donde debía llegar la variable *Ind*, entonces se finaliza este ciclo y se continúa con la instrucción que se encuentra después del respectivo *Fin_Para*.

Terminada esta parte en donde se leen 5 datos enteros y se almacenan en el primer vector, pasamos a la segunda parte, en donde solicitamos otros 5 datos enteros pero esta vez los vamos a almacenar en el segundo vector (que hemos llamado *V2*). Volvemos a generar los números del 1 al 5 utilizando la variable *Ind* como índice y valiéndonos de un ciclo.

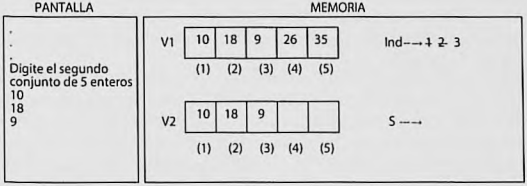
```
Escriba "Digite el segundo conjunto de 5 enteros"
    Para Ind = 1 hasta 5
        Lea V2 ( Ind )
    Fin_Para
```



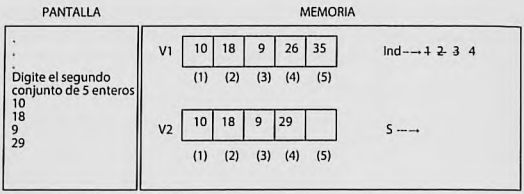
Cuando la variable *Ind* almacene el número 1, y asumiendo que el número leído sea el 10, dicho número quedará almacenado en el vector *V2* en la posición 1. Se incrementa entonces el valor de la variable *Ind* en 1.



Cuando la variable *Ind* almacene el número 2, y asumiendo que el número leído sea el 18, dicho número quedará almacenado en el vector *V2* en la posición 2. Se incrementa entonces el valor de la variable *Ind* en 1.



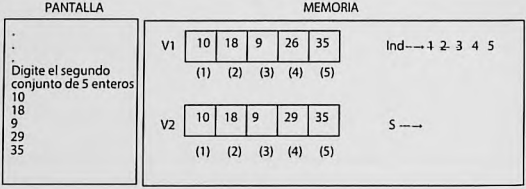
Cuando la variable *Ind* almacene el número 3 y asumiendo que el número leído sea el 9, dicho número quedará almacenado en el vector *V2* en la posición 3. Se incrementa entonces el valor de la variable *Ind* en 1.



S = 0

Ind = 1

Cuando la variable *Ind* almacene el número 4 y asumiendo que el número leído sea el 29, dicho número quedará almacenado en el vector *V2* en la posición 4. Se incrementa entonces el valor de la variable *Ind* en 1.



Y cuando la variable *Ind* almacene el número 5 y asumiendo que el número leído sea el 35, dicho número quedará almacenado en el vector *V2* en la posición 5. Con esto terminará este segundo ciclo, dado que el tope del mismo era 5 y ya se llegó a este valor.

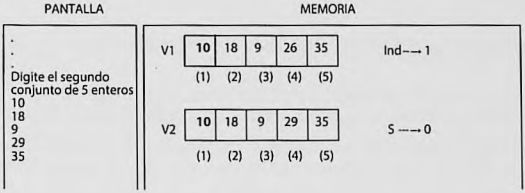
Con esto finalizamos la "carga" de los vectores, con lo cual ya podemos entrar a determinar si son exactamente iguales o no. Es natural pensar en este momento que los dos vectores no son iguales y por ello pareciera ser inoficioso este algoritmo, pero por ahora nos interesa cumplir con el objetivo, es decir, demostrar que a través de este algoritmo un computador puede determinar si un conjunto de datos (almacenado en un vector) es igual a otro conjunto de datos o no. Por tal motivo, nuestro algoritmo continúa almacenando en las variables *S* e *Ind* los valores 0 y 1 respectivamente.

Luego se plantea un ciclo que debe permanecer mientras el contenido de la variable *Ind* sea menor o igual que 5 (o sea, mientras no se haya llegado a la

posición del último dato en los vectores) y mientras la variable *S* siga almacenando el número 0. Dentro de este ciclo, se preguntará si el contenido del vector *V1* en la posición *Ind* es diferente al valor almacenado en el vector *V2* en la misma posición (para cualquier valor que almacene la variable *Ind*).

En el instante en que esto sea Verdadero, entonces se cambiará el dato almacenado en la variable *S*, que estaba en 0, por un 1 y con esto se "abortaría" el ciclo que se está ejecutando, dado que la condición dice *Mientras Ind <= 5 Y S = 0*. Sea pues Verdadera o Falsa la decisión, se le adicionará 1 al valor almacenado en la variable *Ind* para poder hacer referencia a la siguiente posición dentro de los vectores.

```
S = 0
Ind = 1
Mientras Ind <= 5 Y S = 0
        Si V1 ( Ind ) != V2 ( Ind )
                S = 1
        Fin_Si
        Ind = Ind + 1
Fin_Mientras
```



Cuando la variable *Ind* valga 1, la decisión *Si V1 (Ind) != V2 (Ind)* se convierte en

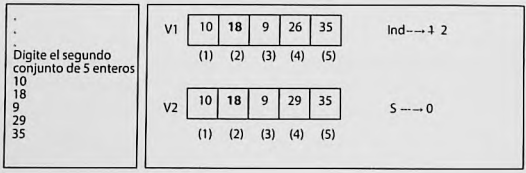
Si V1 (1) != V2 (1)

Es de anotar que el signo != lo estamos utilizando en este libro para representar *Diferente de* tal como se usa en el lenguaje C. El lector podrá utilizar cualquier otra equivalencia para este operador relacional (tales como < > o sencillamente *diferente de)*. Como el contenido del vector *V1* en la posición 1 es 10 y el contenido del vector *V2* en la posición 1 es 10, entonces la decisión se transforma, internamente, en:

Si 10 != 10

Es decir, si 10 es diferente de 10. Como la respuesta a esta decisión es Falso, entonces continuamos con la instrucción que se encuentra después del respectivo *Fin_Si* y que corresponde a incrementar en 1 el valor almacenado en la variable *Ind*, quedando con el valor *2*.

Como seguidamente encontramos el *Fin* del ciclo, entonces volvemos a evaluar la condición de dicho ciclo para entrar de nuevo al cuerpo del mismo, debido a que el contenido de la variable *Ind* es menor que 5 y también debido a que el contenido de la variable *S* sigue siendo 0.



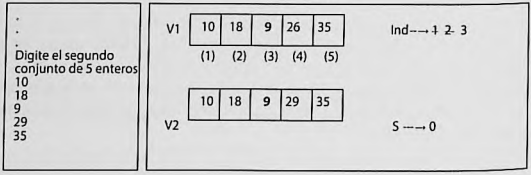
Cuando la variable *Ind* valga 2, la decisión *Si V1 (Ind) != V2 (Ind)* se convierte en:

Si V1 (2) ! = V2 (2)

Como el contenido del vector *V1* en la posición 2 es 18 y el contenido del vector *V2* en la posición 2 es 18, la instrucción se convierte internamente en:

Si 18 ! = 18

Como la respuesta a esta decisión es Falso, entonces ejecutamos la orden que está después del *Fin_Si*, o sea, incrementamos en 1 el contenido de la variable *Ind*, que ahora quedará con el valor 3.



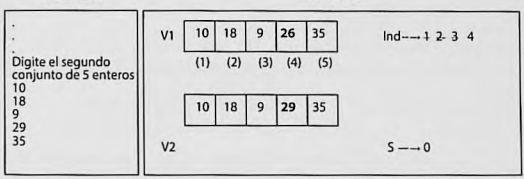
Cuando la variable *Ind* valga 3, la decisión *Si V1 (Ind) != V2 (Ind)* se convierte en:

Si V1 (3) != V2 (3)

Como el contenido del vector *V1* en la posición 3 es 9 y el contenido del vector *V2* en la posición 3 es 9, la instrucción se convierte internamente en:

Si 9 != 9

Como la respuesta a esta decisión es Falso, entonces ejecutamos la orden que está después del *Fin_Si*, o sea, incrementamos en 1 el contenido de la variable *Ind*, que ahora quedará con el valor 4.



Cuando la variable *Ind* valga 4, la decisión *Si V1 (Ind) != V2 (Ind)* se convierte en:

Si V1 (4) != V2 (4)

Como el contenido del vector *V1* en la posición 4 es 26 y el contenido del vector *V2* en la posición 4 es 29, la instrucción se convierte internamente en:

Si 26 != 29

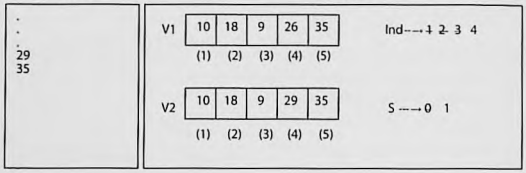
Como la respuesta a esta decisión es Verdadero, entonces se ejecuta la orden:

S = 1

Con lo cual se modifica el contenido de la variable *S*. Seguidamente, ejecutamos la orden que incrementa en 1 el contenido de la variable *Ind*. Volvemos, entonces, a evaluar la condición del ciclo y vemos que *Ind* todavía es menor que 5 pero esta vez *S* ya no es igual a 0, por lo tanto, la primera parte de la condición es Verdadera, pero la segunda parte de la condición es Falsa. Como

están unidas por un operador **Y**, entonces, basados en su tabla de verdad, vemos que *Verdadero* ***Y*** *Falso* nos da como resultado total *Falso.*

Por lo tanto, esto indica que finalizamos la ejecución del cuerpo del ciclo y pasamos a la instrucción que está después del correspondiente *Fin_Mientras.*



Con las variables en memoria, tal como están aquí, se realiza la siguiente decisión:

Si S = 0

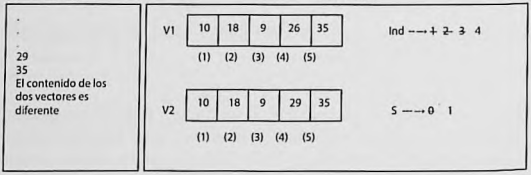
Escriba "El contenido de los dos vectores es exactamente igual"

Sino

Escriba "El contenido de los dos vectores es diferente"

Fin_Si

Se pregunta si el contenido de la variable *S* es igual a 0. Como vemos, dicho contenido es 1, por lo tanto, la respuesta a dicha decisión es Falso. Se ejecuta la instrucción que está seguidamente al Sino de esta decisión. Por lo tanto, saldrá en pantalla:



Lo cual es Verdadero, dado que realmente los dos vectores no son exactamente iguales. Después de esto llegamos al fin del algoritmo.

Fin

Es importante que note usted el papel que desempeña en este algoritmo la variable *S*, que es la que finalmente nos permite determinar si los dos vectores son iguales (en contenido) o no. Como se puede notar, el resultado del algoritmo es cierto y por ahora podemos decir que, en el caso en el cual los vectores tengan contenidos diferentes, este algoritmo lo determinará acertadamente. Ahora realícele una prueba de escritorio a este algoritmo asumiendo que el contenido de los dos vectores es exactamente igual para saber si el algoritmo en ese caso también lo determina. Si es así, entonces podremos decir que el algoritmo aquí presentado está bien.

9.4. Ejercicios

1. Leer 10 enteros, almacenarlos en un vector y determinar en qué posición del vector está el mayor número leído.

2. Leer 10 enteros, almacenarlos en un vector y determinar en qué posición del vector está el mayor número par leído.

3. Leer 10 enteros, almacenarlos en un vector y determinar en qué posición del vector está el mayor número primo leído.

4. Cargar un vector de 10 posiciones con los 10 primeros elementos de la serie de Fibonacci y mostrarlo en pantalla.

5. Almacenar en un vector de 10 posiciones los 10 números primos comprendidos entre 100 y 300. Luego mostrarlos en pantalla.

6. Leer dos números enteros y almacenar en un vector los 10 primeros números primos comprendidos entre el menor y el mayor. Luego mostrarlos en pantalla.

7. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones se encuentra el número mayor.

8. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones se encuentran los números terminados en 4.

9. Leer 10 números enteros, almacenarlos en un vector y determinar cuántas veces está repetido el mayor.

10. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones se encuentran los números con más de 3 dígitos.

11. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números tienen, de los almacenados allí, menos de 3 dígitos.

12. Leer 10 números enteros, almacenarlos en un vector y determinar a cuánto es igual el promedio entero de los datos del vector.

13. Leer 10 números enteros, almacenarlos en un vector y determinar si el promedio entero de estos datos está almacenado en el vector.

14. Leer 10 números enteros, almacenarlos en un vector y determinar cuántas veces se repite el promedio entero de los datos dentro del vector.

15. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos datos almacenados son múltiplos de 3.

16. Leer 10 números enteros, almacenarlos en un vector y determinar cuáles son los datos almacenados múltiplos de 3.

17. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números negativos hay.

18. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones están los números positivos.

19. Leer 10 números enteros, almacenarlos en un vector y determinar cuál es el número menor.

20. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posición está el menor número primo.

21. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posición está el número cuya suma de dígitos sea la mayor.

22. Leer 10 números enteros, almacenarlos en un vector y determinar cuáles son los números múltiplos de 5 y en qué posiciones están.

23. Leer 10 números enteros, almacenarlos en un vector y determinar si existe al menos un número repetido.

24. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posición está el número con más dígitos.

25. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos de los números leídos son números primos terminados en 3.

26. Leer 10 números enteros, almacenarlos en un vector y calcularle el factorial a cada uno de los números leídos almacenándolos en otro vector.

27. Leer 10 números enteros, almacenarlos en un vector y determinar a cuánto es igual el promedio entero de los factoriales de cada uno de los números leídos.

28. Leer 10 números enteros, almacenarlos en un vector y mostrar en pantalla todos los enteros comprendidos entre 1 y cada uno de los números almacenados en el vector.

29. Leer 10 números enteros, almacenarlos en un vector y mostrar en pantalla todos los enteros comprendidos entre 1 y cada uno de los dígitos de cada uno de los números almacenados en el vector.

30. Leer 10 números enteros, almacenarlos en un vector. Luego leer un entero y determinar si este último entero se encuentra entre los 10 valores almacenados en el vector.

31. Leer 10 números enteros, almacenarlos en un vector. Luego leer un entero y determinar cuántos divisores exactos tiene este último número entre los valores almacenados en el vector.

32. Leer 10 números enteros, almacenarlos en un vector. Luego leer un entero y determinar cuántos números de los almacenados en el vector terminan en el mismo dígito que el último valor leído.

33. Leer 10 números enteros, almacenarlos en un vector y determinar a cuánto es igual la suma de los dígitos pares de cada uno de los números leídos.

34. Leer 10 números enteros, almacenarlos en un vector y determinar cuántas veces en el vector se encuentra el dígito 2. No se olvide que el dígito 2 puede estar varias veces en un mismo número.

35. Leer 10 números enteros, almacenarlos en un vector y determinar si el promedio entero de dichos números es un número primo.

36. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos dígitos primos hay en los números leídos.

37. Leer 10 números enteros, almacenarlos en un vector y determinar a cuántos es igual el cuadrado de cada uno de los números leídos.

38. Leer 10 números enteros, almacenarlos en un vector y determinar si la semisuma entre el valor mayor y el valor menor es un número primo.

39. Leer 10 números enteros, almacenarlos en un vector y determinar si la semisuma entre el valor mayor y el valor menor es un número par.

40. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números de los almacenados en dicho vector terminan en 15.

41. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números de los almacenados en dicho vector comienzan con 3.

42. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números con cantidad par de dígitos pares hay almacenados en dicho vector.

43. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones se encuentra el número con mayor cantidad de dígitos primos.

44. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos de los números almacenados en dicho vector pertenecen a los 100 primeros elementos de la serie de Fibonacci.

45. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números de los almacenados en dicho vector comienzan por 34.

46. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números de los almacenados en dicho vector son primos y comienzan por 5.

47. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posiciones se encuentran los números múltiplos de 10. No utilizar el número 10 en ninguna operación.

48. Leer 10 números enteros, almacenarlos en un vector y determinar en qué posición se encuentra el número primo con mayor cantidad de dígitos pares.

49. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números terminan en dígito primo.

50. Leer 10 números enteros, almacenarlos en un vector y determinar cuántos números de los almacenados en dicho vector comienzan en dígito primo.

Capítulo 10

Matrices

10.1. Definición

Una matriz es un conjunto de datos organizados en forma de filas y columnas en donde para referenciar cada dato necesitaremos establecer claramente en qué fila y en qué columna se encuentra. Tomemos la siguiente matriz de datos enteros como ejemplo.

Nro. de fila ↓ (1) (2) (3) (4) (5) (6) ← Nro. de columna

(1)	10	32	-61	24	19	66
(2)	25	36	2	12	100	1
(3)	15	20	26	30	35	18
(4)	10	21	23	26	85	17

La anterior es, pues, una matriz de números enteros cuyos datos están organizados en 4 filas y 6 columnas. De esta manera, para ubicar exactamente un dato, solo tendremos que referenciar su posición en cuanto a filas y columnas y sabremos exactamente a qué dato nos estamos refiriendo. Note usted que se necesita tanto de la ubicación exacta de la fila como de la ubicación de la columna para que no exista ninguna duda en cuanto al dato al cual nos estemos refiriendo. Por ejemplo, ¿cuál es el dato que queda en la tercera fila...? Usted seguramente dirá que en la tercera fila están los datos 15, 20, 26, 30, 35 y 18. De manera que no podría preguntarse de manera individual por **el dato** de la tercera fila, sino por **los datos** de la tercera fila.

De acuerdo a esto, la posición de un dato en una matriz siempre estará determinada por el número de su fila y el número de su columna. Así, por ejemplo, ¿cuál es el dato que se encuentra en la fila 3 columna 2? Pues el número 20.

Nro. de fila →

	(1)	***(2)***	(3)	(4)	(5)	(6) ← Nro. de columna
(1)	10	32	-61	24	19	66
(2)	25	36	2	12	100	1
(3)	15	***20***	26	30	35	18
(4)	10	21	23	26	85	17

¿Cuál es el dato que se encuentra en la fila 2 columna 4? Respuesta: el número 12.

Nro. de fila →

	(1)	(2)	(3)	***(4)***	(5)	(6) ← Nro. de columna
(1)	10	32	-61	24	19	66
(2)	25	36	2	***12***	100	1
(3)	15	20	26	30	35	18
(4)	10	21	23	26	85	17

¿Cuál es el dato que se encuentra en la fila 1 columna 6? Respuesta: el número 66.

Nro. de fila →

	(1)	(2)	(3)	(4)	(5)	***(6)*** ← Nro. de columna
(1)	10	32	-61	24	19	***66***
(2)	25	36	2	12	100	1
(3)	15	20	26	30	35	18
(4)	10	21	23	26	85	17

¿Cuál es el dato que se encuentra en la fila 5 columna 3? Respuesta: no existe fila 5. Para cada una de las preguntas, usted puede notar que existe una única respuesta, pues se ha dado exactamente la posición en fila y columna de cada uno de los datos solicitados. Si quisiéramos mostrar en pantalla el contenido de la primera casillita de la primera fila de esta matriz y asumiendo que toda la matriz se llama *M*, entonces simplemente diríamos:

Escriba M (1 , 1)

Esto significaría que escriba en pantalla el contenido de la matriz *M* en la fila 1 columna 1. Siempre que se use esta notación, el primer número representará el número de la fila y el segundo representará el número de la columna. Ejecutada esta orden, saldrá en pantalla el número 10. Asimismo, si se da la orden:

Escriba M (3, 4)

estaremos mostrando en pantalla el número que se encuentra en la matriz M en la fila 3 columna 4, o sea, que nos referimos al número 30. De esta manera, podremos referenciar todos y cada uno de los datos almacenados en la matriz. Si quisiéramos escribir todos los datos almacenados en la segunda fila, entonces diríamos:

Escriba M (2, 1), M (2, 2), M (2, 3), M (2, 4), M (2, 5), M (2, 6)

Con lo cual saldrían en pantalla los siguientes datos:

25 36 2 12 100 1

Note usted que en la orden

Escriba M (2, 1), M (2, 2), M (2, 3), M (2, 4), M (2, 5), M (2, 6)

el valor que referencia la fila se mantiene constante mientras que el valor que referencia la columna varía desde la primera hasta la última (en este caso, la primera columna es 1 y la última es 6). De tal manera que, si quisiéramos incorporar variables adicionales para simplificar la escritura de los datos de la segunda fila, podríamos decir:

```
.
.
Para Col = 1 hasta 6
        Escriba M ( 2, Col )
Fin_Para
.
.
```

O escrito con un ciclo *Mientras* sería:

```
.
.
Col = 1
Mientras Col < = 6
        Escriba M ( 2, Col )
        Col = Col + 1
Fin_Mientras
.
.
```

He colocado unos puntos suspensivos para indicar que estos conjuntos de instrucciones son parte de un algoritmo. Note usted que, utilizando apropiadamente un ciclo, su variable índice nos va a servir para referenciar progresivamente las columnas de la matriz y permitir que en pantalla aparezcan todos los números de la segunda fila.

Si quisiéramos mostrar en pantalla todos los números de todas las filas, tendríamos dos formas. La primera e ineficiente que sería:

```
Escriba M ( 1, 1 ), M ( 1, 2 ), M ( 1, 3 ), M ( 1, 4 ), M ( 1, 5 ), M ( 1, 6 )
Escriba M ( 2, 1 ), M ( 2, 2 ), M ( 2, 3 ), M ( 2, 4 ), M ( 2, 5 ), M ( 2, 6 )
Escriba M ( 3, 1 ), M ( 3, 2 ), M ( 3, 3 ), M ( 3, 4 ), M ( 3, 5 ), M ( 3, 6 )
Escriba M ( 4, 1 ), M ( 4, 2 ), M ( 4, 3 ), M ( 4, 4 ), M ( 4, 5 ), M ( 4, 6 )
```

Puede usted notar que, al mostrar los datos por fila, mientras se mantiene constante el valor de referencia de una fila se avanza en las columnas desde 1 hasta 6. Por lo tanto, en una segunda versión ineficiente de este conjunto de instrucciones podríamos escribir lo mismo de la siguiente forma:

```
.
.
Para Col = 1 hasta 6
        Escriba M ( 1, Col )
Fin_Para
Para Col = 1 hasta 6
        Escriba M ( 2, Col )
Fin_Para
Para Col = 1 hasta 6
        Escriba M ( 3, Col )
Fin_Para
```

```
Para Col = 1 hasta 6
        Escriba M ( 4, Col )
Fin_Para
.
.
```

O si quisiéramos escribir los datos, columna a columna, entonces se podría utilizar:

```
.
.
Para Col = 1 hasta 6
        Escriba M ( 1, Col ), M ( 2, Col), M ( 3, Col ), M ( 4, Col )
Fin_Para
.
.
```

Ya puede usted notar que en cualquiera de los dos casos se repiten instrucciones o conjuntos de instrucciones, lo que hace suponer que con dos ciclos anidados (o sea, adicionando un ciclo interno a la estructura planteada) podremos recorrer todos los datos de la matriz mostrándolos en pantalla.

De acuerdo a esto, podremos recorrer la matriz en mención utilizando los dos ciclos siguientes:

```
.
.
Para Fil = 1 hasta 4
        Para Col = 1 hasta 6
                Escriba M ( Fil , Col )
        Fin_Para
Fin_Para
.
.
```

O también se hubiera podido plantear de la siguiente forma (utilizando ciclos *Mientras* anidados):

```
.
.
Fil = 1
Mientras Fil < = 4
        Col = 1
        Mientras Col < = 6
                Escriba M ( Fil , Col )
                Col = Col + 1
        Fin_Mientras
        Fil = Fil + 1
Fin_Mientras
.
.
```

Deberá el lector desarrollar una prueba de escritorio de cada uno de estos conjuntos de instrucciones con el ánimo de que pueda conceptualizar de una manera más clara el funcionamiento estos ciclos anidados en unión con la idea de una matriz.

En estos ejemplos, hemos asumido que tenemos una matriz de 4 filas y 6 columnas con datos. ¿Qué sucede si en vez de querer mostrar los datos en pantalla quisiéramos leer datos y almacenarlos en la matriz? Entonces lo que tendríamos que hacer es cambiar la orden

Escriba M (Fil , Col)

por *Lea M (Fil , Col)*

De esta manera cuando, por ejemplo, la variable *Fil* almacene el valor 2 y la variable *Col* almacene el valor 3, la orden *Escriba M (Fil , Col)* se entenderá como *Lea un entero y guárdelo en la matriz M en la Fila Fil (que vale 2) Columna Col (que vale 3)*, o sea, en la posición (2 , 3), sabiendo que el primer entero siempre representa la fila y el segundo entero siempre representa la columna.

De esta forma, si utilizamos el siguiente conjunto de instrucciones:

```
.
.
Fil = 1
Mientras Fil < = 4
```

```
        Col = 1
        Mientras Col <= 6
                Lea M ( Fil , Col )
                Col = Col + 1
        Fin_Mientras
        Fil = Fil + 1
Fin_Mientras
.
.
```

estaremos leyendo 24 enteros y cada uno de ellos irá quedando almacenado en una "casillita" independiente dentro de la matriz en el orden de las filas, es decir, primero llena todas las "casillas" de una fila, luego continúa con la siguiente fila hasta llegar a la cuarta fila que es la última (considerando que seguimos hablando de la misma matriz 4x6 con la que hemos construido todos los ejemplos hasta el momento).

10.2. Características de una matriz

Siempre que vayamos a utilizar una matriz es importante que tengamos en cuenta que los datos en ella están organizados en forma de filas y columnas. Esto es importante porque ello precisamente es lo que justifica la utilización del concepto de matriz dentro de un algoritmo. Una matriz tendrá las siguientes características:

Nombre.- Toda matriz, por ser **un** conjunto de variables, deberá tener un identificador que permita referirse a ella en cualquier momento dentro del algoritmo. Dicho nombre se ajusta a todas las normas y reglas que existen para cualquier variable. Dentro de las particularidades del nombre de una matriz, es importante que este sea altamente mnemónico, o sea, que fácilmente haga referencia a su contenido. Esto será muy útil al utilizar la matriz en un determinado algoritmo.

Tipo.- Este representa cuál es el tipo de dato que se va a almacenar en cada una de las casillitas de la matriz. Es importante anotar que tanto una matriz como un vector son conjuntos de datos homogéneos; esto quiere decir que todos los datos almacenados en ellos deben ser del mismo tipo. De tal forma que usted no podrá pensar en que tiene una matriz de 8 filas por 5 columnas y que en las primeras cuatro filas va a almacenar datos de tipo entero y en las otras cuatro va a almacenar datos de tipo real.

Dimensión.- Se refiere específicamente a la cantidad de filas y columnas que va a tener la matriz. Es muy útil que tenga en cuenta que "dimensionar" una matriz no es más que determinar claramente cuántas filas y cuántas columnas va a tener antes de comenzar cualquier orden. También es bueno que recuerde que el hecho de que se dimensione una matriz no implica la obligatoriedad de utilizar todas las filas o todas las columnas de ella. Por ejemplo, si usted dimensiona una matriz de 1000 filas por 1000 columnas, puede utilizar dentro de un algoritmo las primeras 15 filas y 12 columnas. Esta es precisamente una de las grandes desventajas de los arreglos, dado que no siempre vamos a saber con certeza cuántas filas y cuántas columnas vamos a necesitar y más de una vez vamos a pecar por exceso, debido a que es posible que dimensionemos más columnas o más filas de las que realmente necesitábamos, o por defecto, pues también nos pueden hacer falta y haber dimensionado menos de las que eran.

Tamaño.- Se refiera al resultado de multiplicar la cantidad total de filas por la cantidad total de columnas de una matriz. De esta manera, una matriz de 4 filas por 6 columnas tendrá un tamaño de 24 posiciones. Se denomina tamaño relativo la cantidad total de posiciones que se usan de una matriz. Por ejemplo, si se tiene una matriz 6x9 (recuerde que el primer número representa la cantidad de filas y el segundo, la cantidad de columnas) y de ella usted solo utiliza las 4 primeras filas y las 7 primeras columnas, entonces su tamaño absoluto es 6x9, o sea, 54 posiciones y su tamaño relativo es 4x7, o sea, 28 posiciones. Es obvio pensar que el tamaño relativo es menor o igual que el tamaño absoluto, nunca es mayor.

Destinación.- Es muy importante que cuando utilice una matriz dentro de un algoritmo sepa por qué la necesita y tenga sentido incorporar en él un conjunto de campos de memoria con formato de filas y columnas.

10.3. Ejemplo con matrices No. 1

Enunciado

Leer una matriz 3x4 y determinar en qué posición está el mayor número par.

Clarificación del objetivo

Fundamentalmente, se trata de leer 12 datos enteros, almacenarlos en una matriz que tenga 3 filas y 4 columnas y buscar en qué posición está el mayor número par. Recordemos entonces que un número par es aquel que es divisible exactamente entre 2 y siendo N cualquier número *Si N / 2 * 2 = N*, entonces N es un número par asumiendo obviamente que N es además un entero para que al realizar las correspondientes operaciones se aplique aritmética entera y no se generen decimales.

¿Cómo vamos a determinar cuál es el mayor número par? Recorremos una a una las posiciones dentro de la matriz y vamos preguntando en cada una de ellas si su contenido es par, o sea, si se cumple con la decisión *Si N / 2 * 2 = N*, tomando como N cualquier número almacenado en la matriz (o sea, que N es un *M (i , j)* para cualquier valor de *i* comprendido entre 1 y el tope máximo de las filas y para cualquier valor de *j* comprendido entre 1 y el tope máximo de las columnas).

En caso de que algún contenido de la matriz sea par, entonces lo comparamos con una variable que va a almacenar el mayor número par y si dicho contenido es mayor que el que previamente se había almacenado en esa variable, entonces se cambia el dato almacenado en ella por el nuevo valor mayor que se acaba de encontrar. Al tiempo que seleccionamos el mayor par que encontremos, vamos almacenando su posición en dos variables, debido a que tenemos que almacenar tanto la columna como la fila del número que encontremos.

Finalizado esto, mostramos en pantalla lo que nos solicitan, que es la posición del mayor número par y de paso podemos decir cuál era ese número encontrado. La variable que va a almacenar el mayor número par ha de ser inicializada en un número muy pequeño para garantizar la efectividad del algoritmo. Pudiera ser inicializada en cero si tenemos la garantía de que todos los datos leídos van a ser positivos. Para efectos de garantizar la efectividad de este algoritmo, vamos a inicializar dicha variable con el número –30000, un número verdaderamente pequeño.

Algoritmo

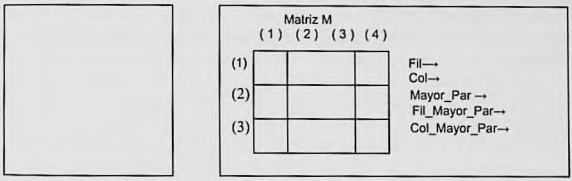
```
Algoritmo Ejem_Matriz_1
Variables
        Entero : M ( 3 , 4 ),          // Almacenará los 12 datos leídos
                 Fil,                  // Se utilizará como índice para las filas
                 Col,                  // Se utilizará como índice para las columnas
                 Mayor_Par,            // Almacenará el mayor número par hallado
                 Fil_Mayor_Par,        // Almacenará la fila en donde se encuentra el
                                       // mayor número par hallado
                 Col_Mayor_Par,        // Almacenará la columna en donde se
                                       // encuentra el mayor número par hallado
Inicio
        Escriba "Digite 12 números enteros"   // Solicita los 12 números
        Para Fil = 1 hasta 3          //y los almacena en una matriz de 3 filas por
                Para Col = 1 hasta 4  //4 columnas, dejando cada número en cada
                        Lea M ( Fil , Col ) // una de las posiciones de la matriz
```

```
            Fin_Para
        Fin_Para
        Mayor_Par = -30000          //Se inicializa la variable que va a almacenar
                                    //el mayor número par en un número muy muy
                                    //pequeño
        Para Fil = 1 hasta 3        //Con estos dos ciclos se recorre la matriz (por
                                    //filas) preguntando en cada posición si su
                                    //contenido es par y además si es mayor que
                                    //el último mayor número par encontrado
            Para Col = 1 hasta 4
            Si M ( Fil , Col ) / 2 * 2 = M ( Fil , Col ) Y M ( Fil , Col ) > Mayor_Par
                Mayor_Par = M (Fil , Col )      //En caso de que
                Fil_Mayor_Par = Fil             //sea Verdadera se
                Col_Mayor_Par = Col             //almacena tanto el
                                                //número encontrado
                                                //como la fila y la
                                                //columna en donde se
                                                //encontró
                Fin_Si
            Fin_Para
        Fin_Para
        Si Mayor_Par = -30000       //Si esta variable aún almacena –30000 ello
                                    //querrá decir que la matriz no tenía números
                                    //pares
            Escriba "No existen números pares en la matriz"
        Sino                        //Si esta variable tiene un valor diferente se
                                    //mostrará en pantalla no solo el número sino
                                    //la posición en donde se encontró
            Escriba "Mayor par ", Mayor_Par, "y está en fila", Fil, "columna", Col
        Fin_Si
Fin
```

Prueba de escritorio

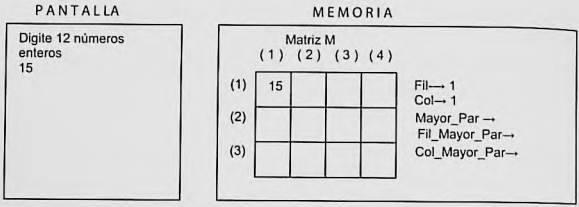
Lo primero que haremos será declarar en memoria las variables que van a ser necesarias para el desarrollo de este algoritmo.

```
Algoritmo Matrices_1
Variables
        Entero : M ( 3 , 4 ),
                 Fil,
                 Col,
                 Mayor_Par,
                 Fil_Mayor_Par,
                 Col_Mayor_Par,
```

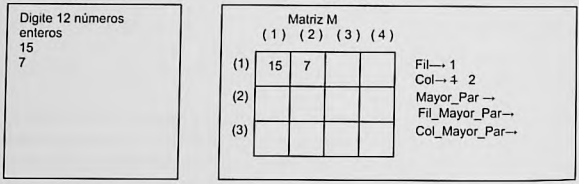


A continuación, solicitamos los datos del ejercicio y procedemos a leerlos almacenándolos uno a uno en la matriz destinada para ello. Este almacenamiento se hace en el siguiente orden: se inicia un ciclo con la variable *Fil* que tomará valores entre 1 y 3 para hacer referencia a cada una de las filas del vector, dentro de este ciclo se tiene otro ciclo en donde la variable *Col* tomará valores entre 1 y 4 para hacer referencia a cada una de las posiciones que existen en cada fila de la matriz, o sea, como quien dice, para hacer referencia a cada una de las columnas de la matriz. Dentro de estos dos ciclos se ejecutará la orden *Lea M (Fil , Col)* que significará *Lea un entero y almacénelo en la matriz M en la fila Fil columna Col.*

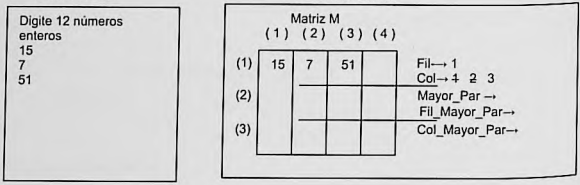
```
Inicio
        Escriba "Digite 12 números enteros"
        Para Fil = 1 hasta 3
                Para Col = 1 hasta 4
                        Lea M ( Fil , Col )
                Fin_Para
        Fin_Para
```



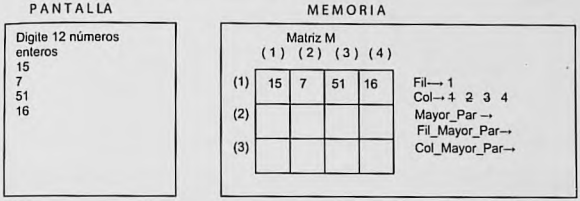
Se inicia la variable *Fil* con el valor 1 para comenzar el ciclo externo. Se inicia igualmente el ciclo interno, entonces cuando *Col* valga 1, se leerá un entero (supongamos que es el número 15) y se almacenará en la matriz M en la fila 1 columna 1.



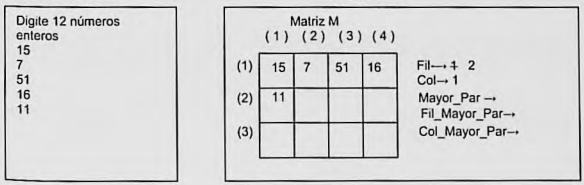
Cuando *Col* valga 2, se leerá otro entero (supongamos que es el número 7) y se almacenará en la matriz M en la fila 1 columna 2. Recuerde que estamos ejecutando el ciclo interno, pues solo volvemos al ciclo externo cuando el interno haya terminado.



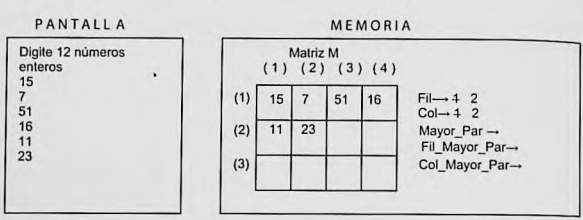
Cuando *Col* valga 3, se leerá otro entero (supongamos que es el número 51) y se almacenará en la matriz M en la fila 1 columna 3.



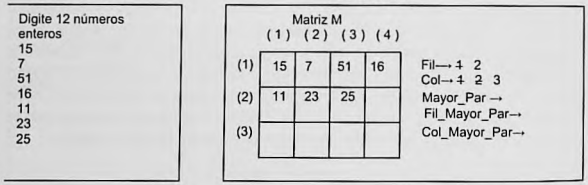
Cuando *Col* valga 4, se leerá otro entero (supongamos que es el número 16) y se almacenará en la matriz M en la fila 1 columna 4. Con esto finalizaríamos el almacenamiento de datos en una fila (por eso es que se dice que estamos recorriendo la matriz por filas). Como ya llegamos al tope final del ciclo interno pues este iba a generar números entre 1 y 4 utilizando la variable *Col* como índice, entonces nos salimos del ciclo interno y vamos al ciclo externo e incrementamos en 1 el contenido de la variable *Fil* (o sea, que esta queda con el valor 2), con lo cual comenzaríamos a llenar la segunda fila de la matriz.



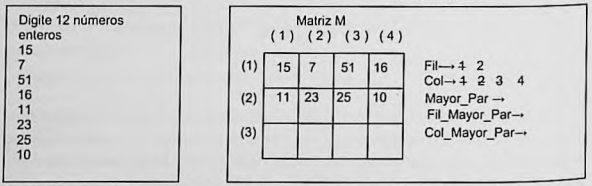
Se incremente en 1 el contenido de la variable *Fil* y se reinicia el ciclo interno, o sea, que la variable *Col* comienza de nuevo con el valor 1. De esta manera, al ejecutar la orden de lectura y asumiendo que el valor leído fuera el número 11, este se almacenaría en la matriz M en la fila 2 columna 1. Luego de esto, incrementamos en 1 el contenido de la variable *Col*.



Cuando la variable *Col* sea igual a 2, entonces se leerá un dato entero (suponga-mos que es el número 23) y se almacenará en la matriz M en la fila 2 columna 2.



Cuando la variable *Col* sea igual a 3, entonces se leerá un dato entero (suponga-mos que es el número 25) y se almacenará en la matriz M en la fila 2 columna 3.



Cuando la variable *Col* sea igual a 4, entonces se leerá un dato entero (supon-gamos que es el número 10) y se almacenará en la matriz M en la fila 2 columna 4. En este momento, se habrá llegado al tope del ciclo interno y se vuelve al ciclo externo para incrementar en 1 el contenido de la variable *Fil*. Volvemos entonces a iniciar el ciclo interno, o sea, que la variable *Col* volverá a tomar valores desde 1 hasta 4.

PANTALLA

```
Digite 12 números
enteros
15
7
51
16
11
23
25
10
9
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	15	7	51	16
(2)	11	23	25	10
(3)	9			

Fil→ 1 2 3
Col→ 1
Mayor_Par →
Fil_Mayor_Par→
Col_Mayor_Par→

Cuando *Col* valga 1, entonces se leerá un dato entero (supongamos que es 9) y se almacenará en la matriz M en la fila 3 columna 1. Incrementamos en 1 el contenido de la variable *Col* y volvemos a leer un dato.

PANTALLA

```
.
.
25
10
9
38
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	15	7	51	16
(2)	11	23	25	10
(3)	9	38		

Fil→ 1 2 3
Col→ 1 2
Mayor_Par →
Fil_Mayor_Par→
Col_Mayor_Par→

Cuando Col valga 2, entonces se leerá un dato entero (supongamos que es 38) y se almacenará en la matriz M en la fila 3 columna 2. Incrementamos en 1 el contenido de la variable *Col* y volvemos a leer un dato.

PANTALLA

```
.
.
25
10
9
38
54
```

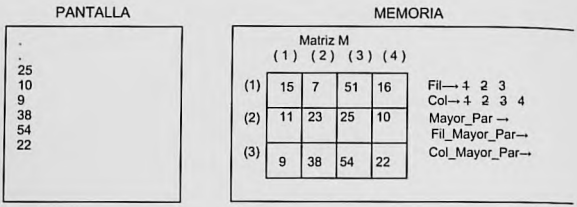
MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	15	7	51	16
(2)	11	23	25	10
(3)	9	38	54	

Fil→ 1 2 3
Col→ 1 2 3
Mayor_Par →
Fil_Mayor_Par→
Col_Mayor_Par→

Cuando *Col* valga 3, entonces se leerá un dato entero (supongamos que es 54) y se almacenará en la matriz M en la fila 3 columna 3. Incrementamos en 1 el contenido de la variable *Col* y volvemos a leer un dato.



Finalmente, cuando *Col* valga 4, entonces se almacenará el dato leído que asumimos es el número 22. En ese instante, terminaremos el ciclo interno, pues hemos llegado al tope de este ciclo y también hemos llegado al tope del ciclo externo, razón por la cual continuamos con la instrucción que está después del ciclo Para externo.

Inicializamos la variable *Mayor_Par* con el número –30000, dado que en este número se va a almacenar el mayor número par almacenado en la matriz; entonces lo iniciamos con un valor muy pequeño para poder realizar unas comparaciones efectivas. Para ello, debemos tener la certeza de que el número –30000 no fue digitado entre los números leídos. Vamos a asumir que en ningún momento y bajo ninguna prueba de escritorio se leerá el número –30000 como parte de los números leídos.

Seguidamente, daremos inicio a un ciclo, utilizando la variable *Fil* como índice, que va a tomar valores entre 1 y 3 (de 1 en 1), y dentro de este ciclo generamos otro ciclo utilizando la variable *Col* como índice, que va a tomar valores entre 1 y 4. De esta manera, establecemos los topes de las variables que van a permitir que se recorra la matriz por filas. En cada posición vamos a preguntar si su contenido es par y si además es mayor que el último número par encontrado. De ser así, almacenamos no solo el número encontrado, sino la fila y la columna en donde se encuentra dicho número, es decir, su posición exacta.

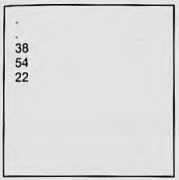
```
Mayor_Par = -30000
Para Fil = 1 hasta 3
 Para Col = 1 hasta 4
        Si M ( Fil , Col ) / 2 * 2 = M ( Fil , Col ) Y M ( Fil , Col ) > Mayor_Par
                                    Mayor_Par = M (Fil , Col )
                                    Fil_Mayor_Par = Fil
                                    Col_Mayor_Par = Col
```

Fin_Si

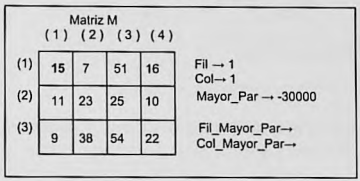
Fin_Para

Fin_Para

PANTALLA



MEMORIA



Cuando la variable *Fil* valga 1 y la variable *Col* valga 1, la decisión

$$Si\ M(Fil, Col)/2 * 2 = M(Fil, Col)\ Y\ M(Fil, Col) > Mayor\_Par$$

se convertirá en

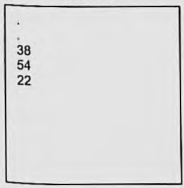
$$Si\ M(1,1)/2 * 2 = M(1,1)\ \mathbf{Y}\ M(1,1) > Mayor\_Par$$

Que a su vez es

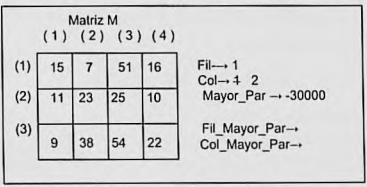
$$Si\ 15/2 * 2 = 15\ \mathbf{Y}\ 15 > -30000$$

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 15 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.

PANTALLA



MEMORIA



Cuando la variable *Fil* valga 1 y la variable *Col* valga 2, la decisión

$$Si\ M(Fil, Col)/2*2 = M(Fil, Col)\ \mathbf{Y}\ M(Fil, Col) > Mayor\_Par$$

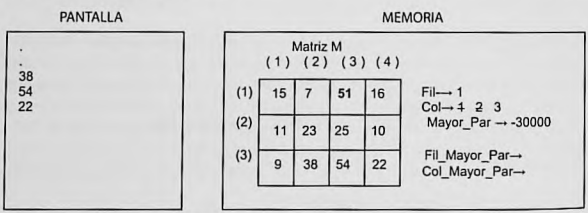
se convertirá en

$$Si\ M(1,2)/2*2 = M(1,2)\ \mathbf{Y}\ M(1,2) > Mayor\_Par$$

Que a su vez es

$$Si\ 7/2*2 = 7\ \mathbf{Y}\ 7 > -30000$$

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 7 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.



Cuando la variable *Fil* valga 1 y la variable *Col* valga 3, la decisión

$$Si\ M(Fil, Col)/2*2 = M(Fil, Col)\ \mathbf{Y}\ M(Fil, Col) > Mayor\_Par$$

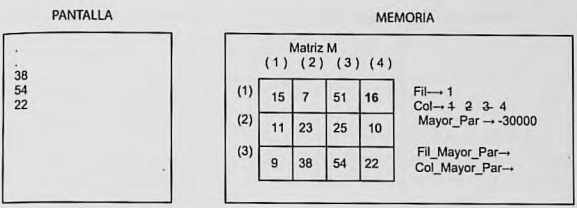
se convertirá en

$$Si\ M(1,3)/2*2 = M(1,3)\ \mathbf{Y}\ M(1,3) > Mayor\_Par$$

Que a su vez es

$$Si\ 51/2*2 = 51\ \mathbf{Y}\ 51 > -30000$$

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 7 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.



Cuando la variable *Fil* valga 1 y la variable *Col* valga 4, la decisión

$$Si\ M(Fil, Col)/2*2 = M(Fil, Col)\ \mathbf{Y}\ M(Fil, Col) > Mayor\_Par$$

se convertirá en

$$Si\ M(1,4)/2*2 = M(1,4)\ \mathbf{Y}\ M(1,4) > Mayor\_Par$$

Que a su vez es

$$Si\ 16/2*2 = 16\ \mathbf{Y}\ 16 > -30000$$

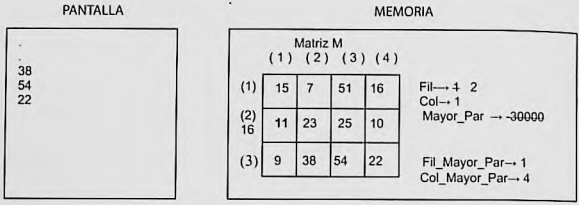
Podemos notar entonces que el número 16 es par y además es mayor que el número –30000 (tomando como base la recta de números enteros), por lo tanto, como ambas condiciones son verdaderas, entonces toda la decisión es verdadera, dado que están unidas a través de un operador booleano **Y**. Por lo tanto, ejecutamos las órdenes:

$$Mayor\_Par = M(Fil, Col)$$
$$Fil\_Mayor\_Par = Fil$$
$$Col\_Mayor\_Par = Col$$

Que, tomando los valores correspondientes de *Fil* y *Col*, se convertirán en:

$$Mayor\_Par = 16$$
$$Fil\_Mayor\_Par = 1$$
$$Col\_Mayor\_Par = 4$$

En este instante, hemos llegado al tope de *Col* (que era 4), por lo tanto, nos salimos al ciclo externo e incrementamos en 1 el contenido de la variable *Fil* para volver a reiniciar el ciclo interno desde 1 hasta 4.



Cuando la variable *Fil* valga 2 y la variable *Col* valga 1, la decisión

$$Si\ M(Fil, Col)/2 * 2 = M(Fil, Col)\ Y\ M(Fil, Col) > Mayor\_Par$$

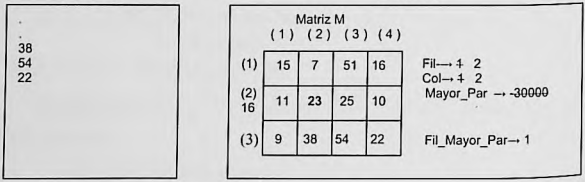
se convertirá en

$$Si\ M(2, 1)/2 * 2 = M(2, 1)\ \mathbf{Y}\ M(2, 1) > Mayor\_Par$$

Que a su vez es

$$Si\ 11/2 * 2 = 11\ \mathbf{Y}\ 11 > 16$$

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 11 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.



Cuando la variable *Fil* valga 2 y la variable *Col* valga 2, la decisión

$$Si\ M(Fil, Col)/2 * 2 = M(Fil, Col)\ Y\ M(Fil, Col) > Mayor\_Par$$

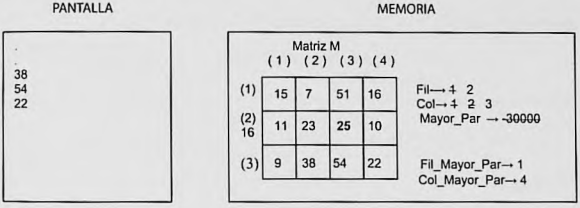
se convertirá en

*Si M (2 , 2) / 2 * 2 = M (2 , 2) Y M (2 , 2) > Mayor_Par*

Que a su vez es

*Si 23 / 2 * 2 = 23 Y 23 > 16*

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 23 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.



Cuando la variable *Fil* valga 2 y la variable *Col* valga 3, la decisión

*Si M (Fil , Col) / 2 * 2 = M (Fil , Col) Y M (Fil , Col) > Mayor_Par*

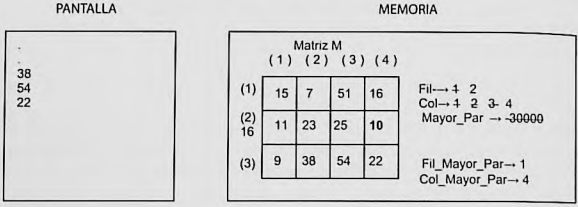
se convertirá en

*Si M (2 , 3) / 2 * 2 = M (2 , 3) Y M (2 , 3) > Mayor_Par*

Que a su vez es

*Si 25 / 2 * 2 = 25 Y 25 > 16*

Vemos que tanto la primera decisión como la segunda son Falsas, dado que 25 no es par y además no es mayor que –30000. Por lo tanto, incrementamos en 1 el contenido de la variable *Col* que corresponde al índice del ciclo interno y volvemos a tomar la decisión.



Cuando la variable *Fil* valga 2 y la variable *Col* valga 4, la decisión

$$Si\ M(Fil, Col)/2*2 = M(Fil, Col)\ \mathbf{Y}\ M(Fil, Col) > Mayor\_Par$$

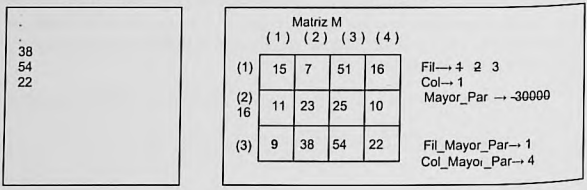
se convertirá en

$$Si\ M(2,4)/2*2 = M(2,4)\ \mathbf{Y}\ M(2,4) > Mayor\_Par$$

Que a su vez es

$$Si\ 10/2*2 = 10\ \mathbf{Y}\ 10 > 16$$

Podemos notar que el número 10 es par, o sea, que la primera decisión es verdadera, pero la segunda decisión es Falsa, dado que el número 10 no es mayor que el número 16, por lo tanto, como están unidas con un operador booleano **Y** y según este para que toda la condición sea Verdadera tendrían que ser Verdaderas sus partes, entonces toda la condición es Falsa. Por lo tanto, nos salimos del ciclo interno, debido a que ya llegamos a su tope, y volvemos al ciclo externo a incrementar en 1 el contenido de la variable *Fil*. Con esto volvemos a entrar y ejecutar el ciclo interno iniciando la variable *Col* en 1 y llevándola de 1 en 1 hasta 4.



Cuando la variable *Fil* valga 3 y la variable *Col* valga 1, la decisión

*Si M (Fil , Col) / 2 * 2 = M (Fil , Col) Y M (Fil , Col) > Mayor_Par*

se convertirá en

*Si M (3 , 1) / 2 * 2 = M (3 , 1) Y M (3 , 1) > Mayor_Par*

Que a su vez es

*Si 9 / 2 * 2 = 9 Y 9 > 16*

Podemos notar que el número 9 no es par ni tampoco es mayor que 16, por lo tanto, toda la condición es Falsa. Incrementamos entonces en 1 el contenido de la variable *Col*.

PANTALLA

```
.
.
38
54
22
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	15	7	51	16
(2)	11	23	25	10
(3)	9	38	54	22

16

Fil→ ~~1~~ 2 3
Col→ ~~1~~ 2
Mayor_Par → ~~-30000~~

Fil_Mayor_Par→ 1
Col_Mayor_Par→ 4

Cuando la variable *Fil* valga 3 y la variable *Col* valga 2, la decisión

*Si M (Fil , Col) / 2 * 2 = M (Fil , Col) Y M (Fil , Col) > Mayor_Par*

se convertirá en

*Si M (3 , 2) / 2 * 2 = M (3 , 2) Y M (3 , 2) > Mayor_Par*

Que a su vez es

*Si 38 / 2 * 2 = 38 Y 38 > 16*

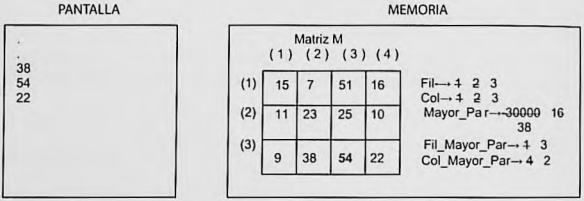
Podemos notar que el número 38 es par y a la vez es mayor que el número 16, por lo tanto, ejecutamos las órdenes:

Mayor_Par = M (Fil , Col)
Fil_Mayor_Par = Fil
Col_Mayor_Par = Col

Que tomando sus correspondientes valores se convierten en:

Mayor_Par = 38
Fil_Mayor_Par = 3
Col_Mayor_Par = 2

Y volvemos a incrementar en 1 el contenido de la variable *Col*.



Cuando la variable *Fil* valga 3 y la variable *Col* valga 3, la decisión

*Si M (Fil , Col) / 2 * 2 = M (Fil , Col) Y M (Fil , Col) > Mayor_Par*

se convertirá en

*Si M (3 , 3) / 2 * 2 = M (3 , 3) Y M (3 , 3) > Mayor_Par*

Que a su vez es

*Si 54 / 2 * 2 = 54 Y 54 > 38*

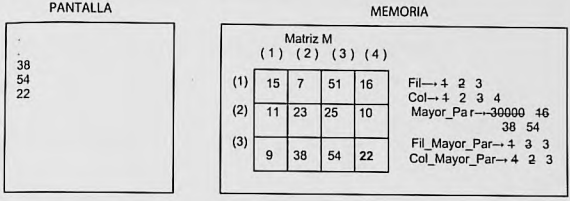
Como el número 54 es par y además es mayor que 38, entonces se ejecutan las órdenes:

Mayor_Par = M (Fil , Col)
Fil_Mayor_Par = Fil
Col_Mayor_Par = Col

Que tomando sus correspondientes valores se convierten en:

Mayor_Par = 54
Fil_Mayor_Par = 3
Col_Mayor_Par = 3

Y volvemos a incrementar en 1 el contenido de la variable *Col*.



Cuando la variable *Fil* valga 3 y la variable *Col* valga 4, la decisión

*Si M (Fil , Col) / 2 * 2 = M (Fil , Col) Y M (Fil , Col) > Mayor_Par*

se convertirá en

*Si M (3 , 4) / 2 * 2 = M (3 , 4) Y M (3 , 4) > Mayor_Par*

Que a su vez es

*Si 22 / 2 * 2 = 22 Y 22 > 54*

La primera decisión es verdadera, debido a que el número 22 es par, pero la segunda condición es falsa, pues 22 no es mayor que 54; entonces, debido a la presencia del operador booleano Y, toda la expresión es Falsa. Con esto hemos llegado al tope del ciclo interno y también hemos llegado al tope del ciclo externo; entonces, a continuación, nos salimos de ambos ciclos y ejecutamos la orden que se encuentra después del *Fin_Para* del ciclo externo.

Preguntamos entonces por el contenido de la variable *Mayor_Par*, debido a que, si esta variable todavía almacena el número –30000 (y teniendo nosotros toda la seguridad de que este número en ningún momento fue digitado), esto querrá decir que en toda la matriz no había números pares y, por lo tanto, así lo avisaremos al usuario del programa. En este caso, esta decisión es Falsa, debido a que el contenido actual de la variable *Mayor_Par* es 54, entonces se ejecuta el correspondiente Sino de esta pregunta. Por tanto, escribiríamos en pantalla la información solicitada desde el principio tomando los últimos valores de las variables correspondientes.

```
Si Mayor_Par = -30000
        Escriba "No existen números pares en la matriz"
Sino
        Escriba "Mayor par es", Mayor_Par, "y está en fila", Fil, "columna",
Col
Fin_Si
```

PANTALLA

```
38
54
22

El mayor par es 54 y está
en la fila 3 columna 3
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	15	7	51	16
(2)	11	23	25	10
(3)	9	38	54	22

Fil→ 1 2 3
Col→ 1 2 3 4
Mayor_Par→ ~~-30000~~ 16 38 54
Fil_Mayor_Par→ 1 3 3
Col_Mayor_Par→ 4 2 3

Finalmente, encontramos el fin del algoritmo y con ello termina nuestra prueba de escritorio.

Fin

Ahora podemos ver que si los datos leídos hubieran sido los que aparecen en la matriz. El algoritmo, efectivamente, hubiera detectado el mayor par y la posición en la cual se encontraba, pues el número 54 es el mayor par (de este grupo de números) y realmente se encuentra en la fila 3 columna 3 de esta matriz. Por lo tanto, podemos decir que este algoritmo está correcto.

10.4. Ejemplo con matrices No. 2

Enunciado

Leer una matriz 4x4 y determinar a cuánto es igual la suma de los elementos que se encuentran en su diagonal.

Clarificación del objetivo

Se define como *matriz cuadrada* toda aquella matriz en donde la cantidad de filas es igual a la cantidad de columnas. De define como *diagonal* de una matriz todos los elementos que se encuentran en posiciones en donde la fila es igual a la columna. Por ejemplo, si tenemos la siguiente matriz M de 4x4 (o sea, de 4 filas y 4 columnas)

Filas.....> \ Columnas.....>	(1)	(2)	(3)	(4)
(1)	18	16	25	24
(2)	15	54	65	12
(3)	54	45	58	21
(4)	45	41	74	32

Los números 18, 54, 58, 32 corresponden a los que están ubicados en la diagonal de esta matriz. ¿Por qué? Pues sencillamente porque son los números ubicados en las posiciones en donde la fila es igual a la columna. El número 18 está en la fila 1 columna 1, el número 54 está en la fila 2 columna 2, el número 58 está en la fila 3 columna 3 y el número 32 está en la fila 4 columna 4. Por esta razón es que se hablaba al principio de las matrices cuadradas, dado que son las que realmente tienen definida la diagonal.

Filas.....> \ Columnas.....>	(1)	(2)	(3)	(4)
(1)	**18**	16	25	24
(2)	15	**54**	65	12
(3)	54	45	**58**	21
(4)	45	41	74	**32**

De acuerdo a lo dicho anteriormente, nuestro algoritmo busca realmente leer todos los datos de una matriz de datos enteros de 4 filas por 4 columnas y sumar solo los elementos que se encuentran en la diagonal. Luego lo que primero vamos a hacer será leer todos y cada uno de los datos que han de quedar almacenados en la matriz generando dos ciclos anidados: uno externo que permita referenciar las filas y otro interno que permita referenciar las columnas (dentro de cada fila). Cuando ya la matriz esté completamente llena de datos, o sea, cuando ambos ciclos hayan terminado, entonces procederemos a generar un ciclo con una sola variable que vaya desde 1 hasta 4 y con esa misma variable vamos a referenciar filas y columnas al mismo tiempo. De manera que si esa es la variable *Ind*, entonces cada elemento a sumar (en un acumulador por supuesto) será el elemento M (*Ind, Ind*), entonces con ello estaremos referenciando solamente a los elementos de la diagonal.

Cuando se haya realizado el recorrido, entonces todo lo que tenemos que hacer es mostrar en pantalla el valor acumulado en la variable correspondiente y con ello habremos cumplido con nuestro objetivo.

Algoritmo

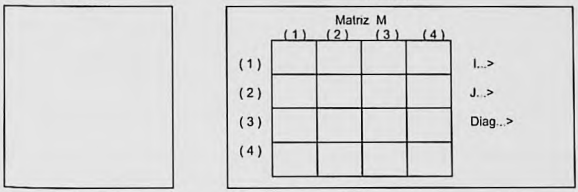
```
Programa Suma_Diagonal
Variables
        Entero : M ( 4, 4 ),          // Matriz en donde se almacenarán los
                                      // datos leídos
                 I,                   // Índice de referencia
                 J,                   // Índice de referencia
                 Diag                 // Variable que almacenará el resultado
                                      // de sumar todos los elementos de la
                                      // diagonal
Inicio
        Escriba "Digite 16 números enteros"   // Solicita los datos a leer
        I = 1                                 // Inicializa la variable I en el
                                              // índice de la primera fila
        Mientras I <= 4                       // Mientras no haya llegado a la
                                              // última fila
                J = 1                         // Comienza en la primera
                                              // posición dentro de la fila
                Mientras J <= 4               // Mientras no haya llegado a la
                                              // última columna dentro de la fila
                        Lea M ( I, J )        // Lea un entero y almacénelo
                                              // en la matriz M Columna I Fila J
                        J = J + 1             // Pase a la siguiente columna dentro de
                                              // la fila
                Fin_Mientras                  // Fin del ciclo interno
                I = I + 1                     // Pase a la siguiente fila
        Fin_Mientras                          // Fin del ciclo externo
        Diag = 0                              // Inicialice la variable Diag en 0
        I = 1                                 // Inicialice la variable I en 1
        Mientras I <= 4                       // Mientras no haya llegado a la
                                              // última posición
```

```
        Diag = Diag + M (I, I)        // Acumule en la variable Diag
                                      // la suma de los elementos de
                                      // la diagonal
        I = I + 1                     // Pase a la siguiente posición
    Fin_Mientras                      // Fin del ciclo de recorrido
                                      // Muestre el resultado solicitado
                                      // en pantalla
    Escriba "La suma de los elementos de la diagonal es igual a", Diag
Fin                                   // Fin del algoritmo
```

Prueba de escritorio

Al igual que todos los algoritmos, lo primero que hacemos es declarar en memoria todas las variables que vamos a necesitar.

```
Programa Suma_Diagonal
Variables
    Entero : M (4, 4), I, J, Diag
```

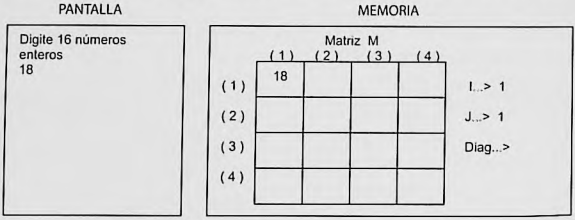


Una vez declaradas en memoria las variables que necesitamos en nuestro algoritmo, siguen las instrucciones:

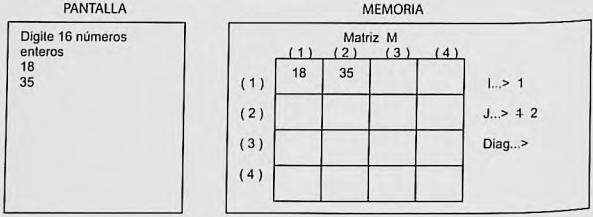
```
Inicio
    Escriba "Digite 16 números enteros"
    I = 1
    Mientras I <= 4
        J = 1
        Mientras J <= 4
            Lea M (I, J)
            J = J + 1
```

```
        Fin_Mientras
        I = I + 1
    Fin_Mientras
```

Según las cuales estaríamos leyendo 16 números enteros y los estaríamos almacenando en cada una de las posiciones de la matriz. Iniciamos, pues, la variable I en 1 y, mientras el contenido de esta variable sea menor o igual que 4, vamos a desarrollar el conjunto de instrucciones que se encuentran dentro de este ciclo. Este conjunto de instrucciones comienza asignando 1 a la variable J y, mientras el contenido de esta variable sea menor o igual que 4, se deberá leer un dato entero y almacenarse en la matriz M en posición I columna J y luego se deberá incrementar el contenido de la variable J en 1. Cuando se termine este ciclo interno, se deberá incrementar el contenido de la variable I y se deberá volver a evaluar la condición del ciclo externo. De esta manera, cuando la variable I valga 1 y la variable J valga 1, se leerá un dato entero. Supongamos que es el número 18, luego de lo cual incrementaremos el valor de J en 1.



Incrementamos en 1 el contenido de la variable J y volvemos a evaluar la condición mientras J sea menor o igual que 4. Como es Verdadero, entonces volvemos a leer otro dato y lo almacenamos en la posición correspondiente. Supongamos que el otro dato leído sea el número 35.



No se olvide que cada dato se almacenará en la fila I columna J. Volvemos a incrementar en 1 el valor almacenado en J y volvemos a evaluar la condición de este ciclo interno. Como vemos que aún es Verdadera, debido a que el valor almacenado en J sigue siendo menor o igual que 4, entonces volvemos a leer otro dato y lo almacenamos en la fila 1 columna 3.

PANTALLA

```
Digite 16 números
enteros
18
35
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10		I...> 1
(2)					J...> 4 2 3
(3)					Diag...>
(4)					

De nuevo, incrementamos por última vez el contenido de la variable J y leemos un dato para ser almacenado en la fila 1 columna 4. Supongamos que el número leído es el 13.

PANTALLA

```
Digite 16 números
enteros
18
35
10
13
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1
(2)					J...> 4 2 3 4
(3)					Diag...>
(4)					

En este instante, al volver a incrementar en 1 el contenido de la variable J, vemos que ya no se cumple la condición de que siga siendo menor o igual que 4, por lo tanto, nos salimos de este ciclo interno para ejecutar la instrucción que sigue después de su correspondiente *Fin_Mientras*.

Dicha instrucción representa incrementar en 1 el contenido de la variable I, por lo tanto, realizamos este incremento y volvemos a evaluar la condición de dicho ciclo externo. Como vemos que es Verdadera, dado que el contenido de la variable I aún es menor o igual que 4, entonces volvemos a inicializar la variable J en 1 para entrar en su correspondiente ciclo. Cuando esta variable valga 1 (de nuevo), entonces se leerá un dato entero y se almacenará en la

fila 2 columna 1, debido a que estos son los valores de I y J respectivamente. Supongamos que el nuevo dato leído sea 23.

PANTALLA

```
Digite 16 números
enteros
18
35
10
13
23
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2
(2)	23				J...> 1
(3)					Diag...>
(4)					

Incrementamos el valor almacenado en J en 1 y evaluamos la condición de este ciclo. Como es Verdadera, entonces volvemos a leer otro dato (supongamos que es el número 8) y lo almacenamos en la fila 2 columna 2.

PANTALLA

```
Digite 16 números
enteros
18
35
10
13
23
8
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2
(2)	23	8			J...> 1 2
(3)					Diag...>
(4)					

De nuevo incrementamos en 1 el contenido de la variable J y volvemos a evaluar la condición. Como dicho contenido sigue siendo menor que 4, entonces volvemos a leer otro dato y lo almacenamos en la posición correspondiente.

PANTALLA

```
Digite 16 números
enteros
18
35
10
13
23
8
11
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2
(2)	23	8	11		J...> 1 2 3
(3)					Diag...>
(4)					

Luego incrementamos de nuevo el contenido de la variable J en 1 y realizamos una lectura de un dato que quedaría almacenado en la fila 2 columna 4. Supongamos que se lee un 45.

PANTALLA

```
Digite 16 números
enteros
18
35
10
13
23
8
11
45
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)				
(4)				

I...> 1 2

J...> 1 2 3 4

Diag...>

En este momento, al volver a incrementar en 1 el contenido de la variable J, vemos que ya no cumple la condición de seguir siendo menor o igual que 4 (pues valdría 5) y entonces nos salimos de este ciclo interno. Incrementamos en 1 el contenido de la variable I y volvemos a evaluar la condición del ciclo externo. Como dicho contenido es menor o igual que 4, entonces entramos al ciclo interno, es decir, volvemos a inicializar la variable J en 1 y como dicho valor es menor o igual que 4, entonces leemos un dato entero y lo almacenamos en la fila 3 columna 1.

PANTALLA

```
.
.
.
45
11
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11			
(4)				

I...> 1 2 3

J...> 1

Diag...>

Incrementamos el contenido de la variable J en 1 y evaluamos la condición del ciclo interno. Como el valor almacenado en J es menor o igual que 4, volvemos a leer un dato (supongamos que es el número 88) y lo almacenamos en la fila 3 columna 3.

PANTALLA

```
.
.
.
45
11
88
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2 3
(2)	23	8	11	45	J...> 1 2
(3)	11	88			Diag...>
(4)					

De nuevo incrementamos en 1 el contenido de la variable J y al evaluar la condición del ciclo interno vemos que sigue siendo menor que 4, por lo tanto, leemos otro dato y lo almacenamos en la posición correspondiente.

PANTALLA

```
.
.
.
45
11
88
16
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2 3
(2)	23	8	11	45	J...> 1 2 3
(3)	11	88	16		Diag...>
(4)					

Se incrementa el contenido de la variable J y se evalúa la condición del ciclo interno. Como es Verdadera, entonces leemos otro dato y lo almacenamos en la matriz M en la fila 3 columna 3.

PANTALLA

```
.
.
.
45
11
88
16
51
13
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2 3 4
(2)	23	8	11	45	J...> 1
(3)	11	88	16	51	Diag...>
(4)					

Al volver a incrementar en 1 el contenido de la variable J, vemos que la condición deja de ser Verdadera y, por lo tanto, nos salimos de este ciclo interno e

incrementamos en 1 el contenido de la variable I, después de lo cual evaluamos la condición del ciclo externo. Como el valor almacenado en la variable I sigue siendo Verdadero, entonces volvemos a inicializar la variable J en 1 y volvemos a leer un dato entero que, para este caso, vamos a asumir que es el número 13 y lo almacenamos en la fila 4 columna 1.

PANTALLA

```
.
.
.
45
11
88
16
51
13
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13			

I...> 1 2 3 4

J...> 1

Diag...>

Realizamos las mismas operaciones para llenar la cuarta fila de manera que se obtenga llenar la matriz completamente.

PANTALLA

```
.
.
.
51
13
22
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13	22		

I...> 1 2 3 4

J...> 1 2

Diag...>

PANTALLA

```
.
.
.
51
13
22
57
35
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13	22	57	35

I...> 1 2 3 4

J...> 1 2 3 4

Diag...>

PANTALLA

.
.
.
51
13
22
57
35

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13	22	57	

I...> 1 2 3 4

J...> 1 2 3 4

Diag...>

Teniendo ya la matriz llena, vemos que, al incrementar en 1 el contenido de la variable J, esta ya no es menor o igual que 4, razón por la cual nos salimos al ciclo externo e incrementamos de nuevo el valor almacenado en la variable I, con lo cual vemos igualmente que dicho valor no es menor o igual que 4 por lo cual nos salimos también de este ciclo externo. El conjunto de instrucciones que sigue nos va a permitir realizar la suma de los elementos que se encuentran ubicados en la diagonal de la matriz. Inicializamos la variable *Diag* en 0 y la variable I en 1. A continuación evaluamos la condición y vemos efectivamente que I es menor o igual que 4, por lo tanto, realizamos la operación de suma *Diag = Diag + M (I, I)* e incrementamos en 1 el contenido de la variable I.

```
Diag = 0
I = 1
Mientras I <= 4
        Diag = Diag + M (I, I)
        I = I + 1
Fin_Mientras
```

PANTALLA

.
.
.
51
13
22
57
35

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13	22	57	35

I...> 1 2 3 4

J...> 1 2 3 4

Diag...>

Como la variable I almacena el valor 1 entonces la expresión *Diag = Diag + M(I,I)* se convierte en *Diag = Diag + M(1, 1)*

Como el valor almacenado en la fila 1 columna 1 de la matriz M es 18, entonces finalmente la expresión se convierte en *Diag = Diag + 18*

Seguidamente incrementamos en 1 el contenido de la variable I y evaluamos la condición de este ciclo. Vemos que el valor almacenado en I es menor o igual que 4, por lo tanto, volvemos a resolver la expresión *Diag = Diag + M(I, I)*.

PANTALLA

```
.
.
.
22
57
35
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1
(2)	23	8	11	45	J...>
(3)	11	88	16	51	Diag...> 0 18
(4)	13	22	57	35	

Como la variable I almacena el valor 1, entonces la expresión *Diag = Diag + M(I,I)* se convierte en *Diag = Diag + M(2, 2)*

Como el valor almacenado en la fila 2 columna 2 de la matriz

M es 8, entonces finalmente la expresión se convierte en *Diag = Diag + 18* quedando almacenado en la variable *Diag* el número 26.

Seguidamente incrementamos en 1 el contenido de la variable I y evaluamos la condición de este ciclo. Vemos que el valor almacenado en I es menor o igual que 4, por lo tanto, volvemos a resolver la expresión *Diag = Diag + M(I, I)*.

PANTALLA

```
.
.
.
22
57
35
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)	
(1)	18	35	10	14	I...> 1 2
(2)	23	8	11	45	J...>
(3)	11	88	16	51	Diag...> 0 18 26
(4)	13	22	57	35	

Como la variable I almacena el valor 1, entonces la expresión *Diag = Diag + M (I, I)* se convierte en *Diag = Diag + M (3, 3)*

Como el valor almacenado en la fila 3 columna 3 de la matriz M es 8, entonces finalmente la expresión se convierte en *Diag = Diag + 16* quedando almacenado en la variable *Diag* el número 42.

Seguidamente incrementamos en 1 el contenido de la variable I y evaluamos la condición de este ciclo. Vemos que el valor almacenado en I es menor o igual que 4, por lo tanto, volvemos a resolver la expresión *Diag = Diag + M (I, I)*.

PANTALLA

```
.
.
.
22
57
35
```

MEMORIA

Matriz M

	(1)	(2)	(3)	(4)
(1)	18	35	10	14
(2)	23	8	11	45
(3)	11	88	16	51
(4)	13	22	57	35

I...> 1 2 3

J...>

Diag...> 0 18 26 42

Como la variable I almacena el valor 4, entonces la expresión *Diag = Diag + M (I, I)* se convierte en *Diag = Diag + M (4, 4)*

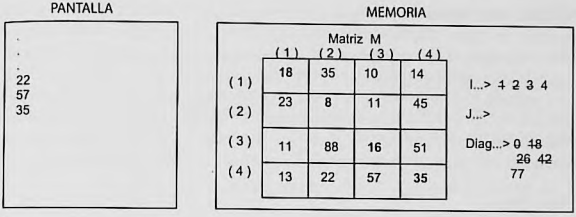
Como el valor almacenado en la fila 4 columna 4 de la matriz

M es 8, entonces finalmente la expresión se convierte en *Diag = Diag + 35* quedando almacenado en la variable *Diag* el número 77.

Volvemos a incrementar el contenido de la variable I y vemos que ya no se cumple la condición de que dicho contenido sea menor o igual que 4, razón por la cual nos salimos de este ciclo y pasamos a la instrucción:

Escriba "La suma de los elementos de la diagonal es igual a", Diag

Con la cual saldrá en pantalla:



Después de lo cual encontramos el fin del algoritmo.

Fin

Al revisar los resultados vemos que efectivamente el número mostrado en pantalla al final es igual a la suma de los datos que quedaron ubicados en la diagonal de la matriz, por lo tanto, podemos dar fe de que este algoritmo cumple con el objetivo planteado que era *Leer una matriz 4x4 y determinar a cuánto es igual la suma de los elementos que se encuentran en su diagonal.*

10.5. Ejemplo con matrices No. 3

Enunciado

Leer una matriz 4x3 y determinar cuántas veces se repite el mayor de los números almacenados en ella.

Clarificación del objetivo

Ya sabemos que deben leerse 12 números enteros e irse almacenando uno a uno en las diferentes posiciones de la matriz. Para ello, y tal como lo hemos hecho en los casos anteriores, utilizamos dos ciclos anidados que nos permitan recorrer por filas la matriz al tiempo que vamos leyendo datos enteros y los vamos almacenando en cada una de sus posiciones.

Luego de tener "cargada" la matriz, buscamos cuál es el mayor dato almacenado en ella y, por lo tanto, para ello, utilizamos una variable que inicializamos en un valor muy pequeño y contra ella vamos comparando uno a uno los datos almacenados en la matriz. Cada vez que encontremos que un dato es mayor que el número almacenado en la matriz, entonces deberemos guardar en dicha variable ese último número mayor encontrado. Este proceso lo haremos al tiempo que recorremos toda la matriz desde la primera fila hasta la última fila posición a posición.

Cuando ya se haya recorrido toda la matriz y tengamos almacenado el número mayor, entonces volveremos a recorrer la matriz para poder contar cuántas veces se repite dicho número en ella, evaluando si cada dato de la matriz es igual al dato almacenado en la variable que nos va a guardar el mayor número encontrado. Es importante que tenga en cuenta que este proceso solo se podrá realizar cuando se tenga completamente identificado el número mayor. Quiero decir con esto que no podemos contar la cantidad de veces que se encuentra el mayor al tiempo que lo buscamos.

Algoritmo

```
Programa Cuenta_Mayor
Variables
        Entero : M ( 4, 3 ),          // Matriz que almacenará los números
                                      // leídos
                 I,                   // Índice de referencia
                 J,                   // Índice de referencia
                 May_Num,             // Variable que almacenará el mayor de
                                      // los números almacenados en la
                                      // matriz
                 Cont_May             // Variable que almacenará la cantidad
                                      // de veces que se encuentra el mayor
                                      // valor en la matriz
Inicio
        Escriba "Digite 12 números enteros"   // Solicita los datos que va a leer
        Para I = 1 hasta 4                    // Con este índice se van a
                                              // referenciar las filas
                Para J = 1 hasta 3            // Con este índice se van a
                                              // referenciar las columnas
                        Lea M ( I, J )        // Lea un dato entero y almacénelo
                                              // en la matriz M en la fila I
                                              // columna J
                Fin_Para                      // Fin del ciclo interno
        Fin_Para                              // Fin del ciclo externo
        May_Num = -30000                      // Inicializamos esta variable en un
                                              // valor muy muy pequeño
```

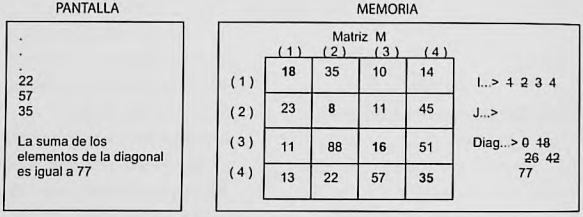
```
Para I = 1 hasta 4                    // Desde la primera hasta la
                                      // última fila
    Para J = 1 hasta 3                // Desde la primera hasta la
                                      // última posición en cada fila
        Si M ( I, J ) > May_Num       // Si algún valor en la matriz es
                                      //mayor que el valor almacenado
// en la variable May_Num
            May_Num = M ( I, J )      // Entonces ese es el
                                      // nuevo valor mayor
        Fin_Si                        // Fin de la decisión
    Fin_Para                          // Fin del ciclo interno
Fin_Para                              // Fin del ciclo externo
Cont_May = 0                          //Inicializamos el contador en ceros
Para I = 1 hasta 4                    //Índice que va a referenciar desde
                                      // la primera hasta la cuarta fila
    Para J = 1 hasta 3                //Índice que va a referenciar desde
                                      // la primera hasta la tercera fila
        Si M ( I, J ) = May_Num       // Si algún valor en la matriz es
                                      // igual al número
                                      // que se encontró como mayor
            Cont_May = Cont_May + 1   //  entonces
cuéntelo
        Fin_Si                        // Fin de la decisión
    Fin_Para                          // Fin del ciclo interno
Fin_Para                              // Fin del ciclo externo
                                      //Mostrar la información solicitada
Escriba "El núm mayor es", May_Num. "y se encuentra", Cont_May, "veces"
Fin
```

Basado en el enunciado, no era obligatorio mostrar cuál era el número mayor que se había encontrado, pero teniéndolo a la mano no está de más mostrarlo. Lo que sí es importante mostrar en pantalla, dado que es la información que se solicita en el enunciado, es la cantidad de veces que está el número mayor en esta matriz.

Prueba de escritorio

Como en todos los algoritmos, lo primero que hacemos es declarar en memoria las variables que vamos a necesitar.

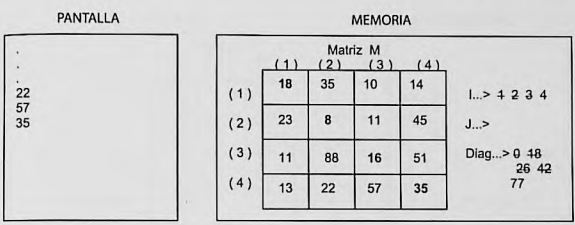
```
Programa Cuenta_Mayor
Variables
        Entero : M ( 4, 3 ),
                 I,
                 J,
                 May_Num,
                 Cont_May
```



Solicitamos 12 números enteros y los vamos leyendo valiéndonos de dos ciclos anidados que nos permiten recorrer, por filas, la matriz al tiempo que vamos almacenando cada dato en cada una de las posiciones de la matriz. Este ciclo involucra una variable que nos permite referenciar las filas y otra que nos permite referenciar las columnas. Como la variable que referenciará las filas es el índice del ciclo externo y la variable que referenciará las columnas es el índice del ciclo interno, entonces nuestro recorrido llenará inicialmente toda la primera fila, luego llenará toda la segunda fila, luego llenará toda la tercera fila y por último llenará toda la cuarta fila con los datos que progresivamente vaya digitando el usuario.

Como en los ejercicios anteriores se ha hecho una prueba de escritorio al conjunto de instrucciones que nos permiten llenar de datos enteros la matriz, vamos a "saltarnos" esta parte del algoritmo, debido a que ya tenemos la certeza de que sí funciona y de que sí nos permite llenar la matriz con datos.

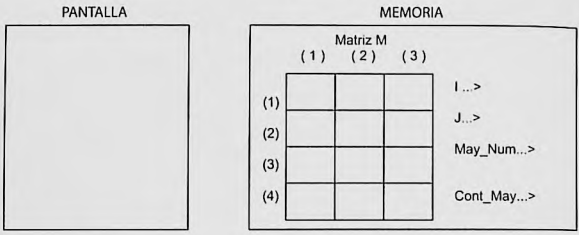
```
Inicio
        Escriba "Digite 12 números enteros"
        Para I = 1 hasta 4
                Para J = 1 hasta 3
                        Lea M ( I, J )
                Fin_Para
        Fin_Para
```



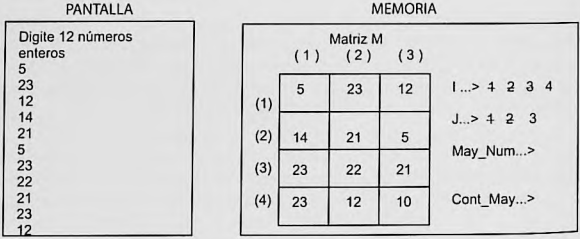
Continuando con nuestro algoritmo, inicializamos la variable *May_Num* con un número muy pequeño (para este caso, hemos escogido el número –30000). Luego de esto, vamos a generar, valiéndonos de un ciclo externo, los números del 1 al 4 que se irán almacenando progresivamente en la variable I y que servirá para referenciar la posición de las filas. Dentro de este ciclo, generaremos otro ciclo que utilizará a la variable J como índice, la cual tomará valores entre 1 y 3 y servirá para referenciar las columnas dentro de cada fila.

Dentro del ciclo interno, preguntaremos si cada uno de los datos almacenados en la matriz M en la fila I columna J es mayor que el valor que esté almacenado en la variable *May_Num*. Si es así, entonces dicho valor deberá quedar almacenado en la variable *May_Num*. De cualquier forma, después se deberá incrementar en 1 el contenido de J para repetir el proceso.

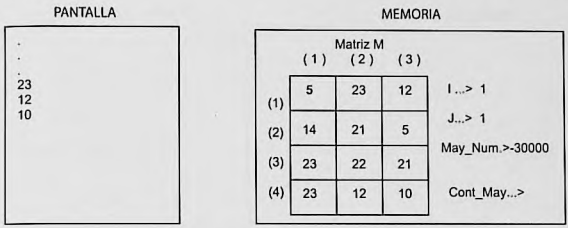
```
May_Num = -30000
Para I = 1 hasta 4
        Para J = 1 hasta 3
                Si M (I, J) > May_Num
                        May_Num = M (I, J)
                Fin_Si
        Fin_Para
Fin_Para
```



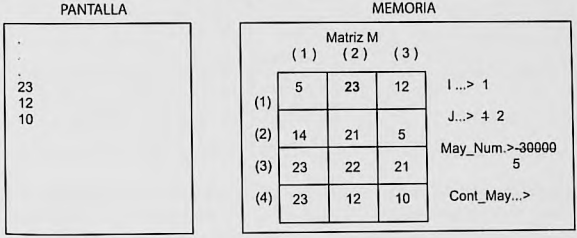
Cuando la variable I vale 1 y la variable J vale 1, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (1, 1) > May_Num*, o sea, *Si 5 > -30000.* Como es Verdadero, entonces ejecutamos la orden *May_Num = M (I, J)*, que se convierte en *May_Num = 5.* Incrementamos en 1 el valor almacenado en la variable J y como su nuevo valor sigue siendo menor o igual que 3, entonces volvemos a evaluar la decisión.



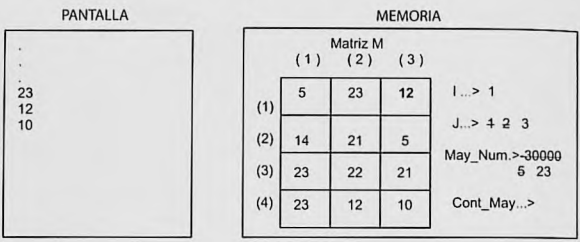
Cuando la variable I vale 1 y la variable J vale 2, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (1, 2) > May_Num*, o sea, *Si 23 > 5.* Como es Verdadero, entonces ejecutamos la orden *May_Num = M (I, J)*, que se convierte en *May_Num = 23.* Seguidamente incrementamos en 1 el valor almacenado en la variable J y como su nuevo valor sigue siendo menor o igual que 3, entonces volvemos a evaluar la decisión.



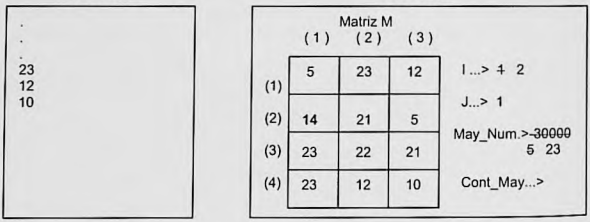
Cuando la variable I vale 1 y la variable J vale 3, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (1, 3) > May_Num*, o sea, *Si 12 > 23.* Como es Falso, entonces nos saltamos la orden de asignación y como ya la variable J tiene el valor 3 (que era su tope), volvemos al ciclo externo e incrementamos en 1 el valor almacenado en la variable I. Como esta variable no ha llegado aún a su tope, entonces volvemos a asignarle a la variable J el valor 1 y vamos a generar de nuevo el ciclo interno con valores para J desde 1 hasta 3.



Cuando la variable I vale 2 y la variable J vale 1, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (2, 1) > May_Num*, o sea, *Si 14 > 23.* Como es Falso, entonces incrementamos en 1 el contenido de la variable J.



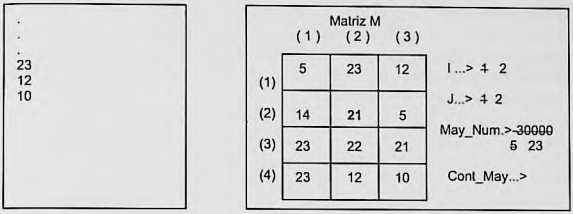
Cuando la variable I vale 2 y la variable J vale 2, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (2, 2) > May_Num*, o sea, *Si 21 > 23.* Como es Falso, entonces incrementamos en 1 el contenido de la variable J.



Cuando la variable I vale 2 y la variable J vale 3, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (2, 3) > May_Num*, o sea, *Si 5 > 23.* Como es Falso, entonces incrementamos en 1 el contenido de la variable J. En este instante, la variable J ha llegado a su tope, razón por la cual nos salimos del ciclo interno y vamos al ciclo externo a incrementar en 1 el contenido de la variable I. Como dicho contenido todavía es menor o igual que 4, entonces volvemos a iniciar el ciclo interno asignándole valores a J desde 1 hasta 3.

PANTALLA

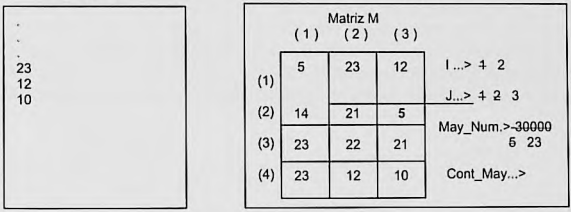
MEMORIA



Cuando la variable I vale 3 y la variable J vale 1, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (3, 1) > May_Num*, o sea, *Si 23 > 23.* Como es Falso, entonces incrementamos en 1 el contenido de la variable J.

PANTALLA

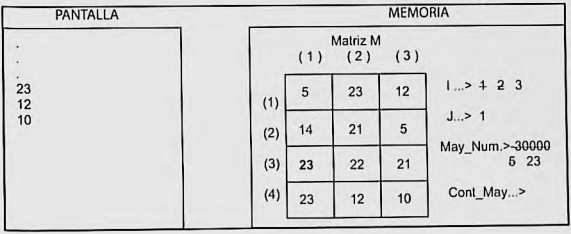
MEMORIA



Cuando la variable I vale 3 y la variable J vale 2, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (3, 2) > May_Num*, o sea, *Si 22 > 23.* Como es Falso, entonces incrementamos en 1 el contenido de la variable J.

PANTALLA

MEMORIA



Cuando la variable I vale 3 y la variable J vale 3, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (3, 3) > May_Num*, o sea, *Si 21 > 23.* Como es Falso, entonces volvemos al ciclo externo debido a que el índice del ciclo interno ya llegó a su tope. Por tanto, incrementamos en 1 el contenido de la variable I y volvemos a generar un ciclo desde 1 hasta 3 para referenciar las posiciones dentro de la cuarta fila.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 1 2 3

J...> 1 2

May_Num.>-30000
5 23

Cont_May...>

Cuando la variable I vale 4 y la variable J vale 1, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (4, 1) > May_Num*, o sea, *Si 23 > 23.* Como es Falso, entonces volvemos a incrementar en 1 el contenido de la variable J.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

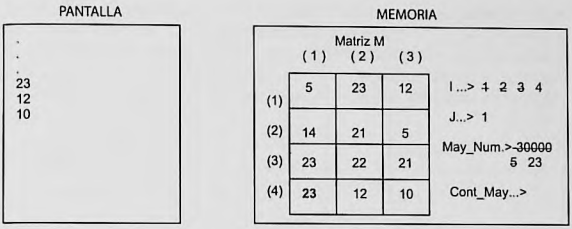
I ...> 1 2 3

J...> 1 2 3

May_Num.>-30000
5 23

Cont_May...>

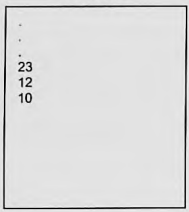
Cuando la variable I vale 4 y la variable J vale 2, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (4, 2) > May_Num*, o sea, *Si 12 > 23.* Como es Falso, entonces volvemos a incrementar en 1 el contenido de la variable J.



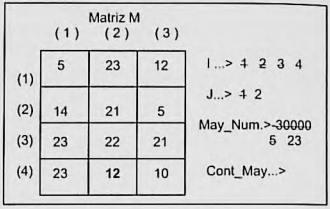
Cuando la variable I vale 4 y la variable J vale 3, entonces la decisión *Si M (I, J) > May_Num* se convierte en *Si M (4, 3) > May_Num*, o sea *Si 10 > 23.* Como es Falso, nos salimos del ciclo interno debido a que la variable J ya llegó a su tope y nos salimos a su ciclo externo, debido a que la variable I también ya llegó a su tope. Continuando con nuestro algoritmo y sabiendo que ya tenemos almacenado en la variable *May_Num* el mayor valor contenido en la matriz (que es igual al número 23), entonces procedemos a buscar cuántas veces está este valor en la misma matriz. Para ello, vamos a inicializar un contador en ceros y de nuevo vamos a recorrer la matriz por filas, preguntando en cada uno de los datos contenidos en ella si es igual al mayor valor obtenido.

```
Cont_May = 0
Para I = 1 hasta 4
        Para J = 1 hasta 3
                Si M ( I, J ) = May_Num
                        Cont_May = Cont_May + 1
                Fin_Si
        Fin_Para
Fin_Para
```

PANTALLA

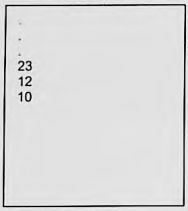


MEMORIA

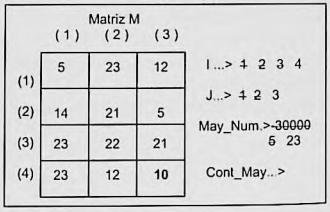


Cuando la variable I valga 1 y J valga 1, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (1, 1) = May_Num*, lo cual significa *Si 5 = 23*, lo cual es Falso. Por tanto, incrementamos en 1 el valor almacenado en la variable J.

PANTALLA

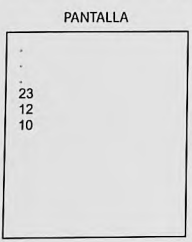


MEMORIA

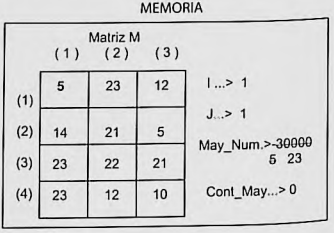


Cuando la variable I valga 1 y J valga 2, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (1, 2) = May_Num*, lo cual significa *Si 23 = 23*, lo cual es Verdadero. Por lo tanto, incrementamos en 1 el contenido de la variable *Cont_May* y volvemos a incrementar en 1 el contenido de la variable J.

PANTALLA



MEMORIA



Cuando la variable I valga 1 y J valga 3, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (1, 3) = May_Num*, lo cual significa *Si 12 = 23*, lo cual es Falso. Por tanto, nos salimos del ciclo interno debido a que la variable J ya llegó a su tope. Incrementamos de nuevo el contenido de la variable I y volvemos a generar el ciclo interno desde 1 hasta 3 utilizando al variable J como índice.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 1

J...> 1 2

May_Num.>-30000
5 23

Cont_May...> 0

Cuando la variable I valga 2 y J valga 1, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (2, 1) = May_Num*, lo cual significa *Si 14 = 23*, lo cual es Falso. Por lo tanto, incrementamos en 1 el valor almacenado en la variable J.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 1

J...> 1 2 3

May_Num.>-30000
5 23

Cont_May...> 0 1

Cuando la variable I valga 2 y J valga 2, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (2, 2) = May_Num*, lo cual significa *Si 21 = 23*, lo cual es Falso. Por lo tanto, incrementamos en 1 el valor almacenado en la variable J.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 4 2

J ...> 1

May_Num.>-30000
5 23

Cont_May...> 0 1

Cuando la variable I valga 2 y J valga 3, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (2, 3) = May_Num*, lo cual significa *Si 5 = 23*, lo cual es Falso. Por lo tanto, nos salimos del ciclo interno debido a que de nuevo la variable J llegó a su tope. Incrementamos de nuevo el valor almacenado en I y volvemos a entrar al ciclo interno.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 4 2 3

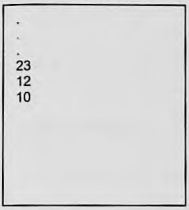
J ...> 1

May_Num.>-30000
5 23

Cont_May...> 0 1

Cuando la variable I valga 3 y J valga 1, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (3, 1) = May_Num*, lo cual significa *Si 23 = 23*, lo cual es Verdadero, razón por la cual se incrementa en 1 el valor almacenado en *Cont_May*. Volvemos pues a incrementar en 1 el contenido de J.

PANTALLA



MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 1 2 3

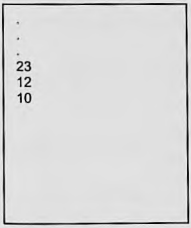
J...> 1 2

May_Num.>-30000
5 23

Cont_May...> 0 1 2

Cuando la variable I valga 3 y J valga 2, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (3, 2) = May_Num*, lo cual significa *Si 22 = 23*, lo cual es Falso. Entonces simplemente incrementamos en 1 el valor almacenado en J.

PANTALLA



MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	23	12	10

I ...> 1 2 3

J...> 1 2 3

May_Num.>-30000
5 23

Cont_May...> 0 1 2

Cuando la variable I valga 3 y J valga 3, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (3, 3) = May_Num*, lo cual significa *Si 21 = 23*, lo cual es Falso. Entonces nos salimos del ciclo interno, pues J volvió a llegar a su tope. Seguidamente incrementamos en 1 el valor almacenado en I y volvemos a generar el ciclo interno.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4)	**23**	12	10

I ...> ~~1~~ 2 ~~3~~ ~~4~~

J...> 1

May_Num.> ~~30000~~ ~~5~~ 23

Cont_May...> 0 ~~1~~ 2

Cuando la variable I valga 4 y J valga 1, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (4, 1) = May_Num*, lo cual significa *Si 23 = 23*, lo cual es Verdadero, por lo tanto, incrementamos en 1 el valor almacenado en la variable *Cont_May*.

PANTALLA

```
.
.
.
23
12
10
```

MEMORIA

Matriz M

	(1)	(2)	(3)
(1)	5	23	12
(2)	14	21	5
(3)	23	22	21
(4) 3	23	**12**	10

I ...> ~~1~~ 2 ~~3~~ ~~4~~

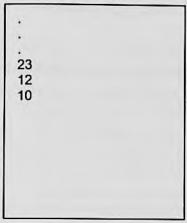
J...> ~~1~~ 2

May_Num.> ~~30000~~ ~~5~~ 23

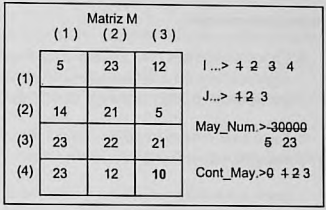
Cont_May.> 0 ~~1~~ 2

Cuando la variable I valga 4 y J valga 2, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (4, 2) = May_Num*, lo cual significa *Si 12 = 23*, lo cual es Falso, por lo tanto, incrementamos en 1 el valor almacenado en J.

PANTALLA



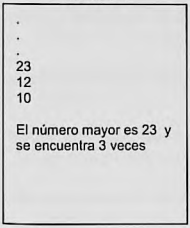
MEMORIA



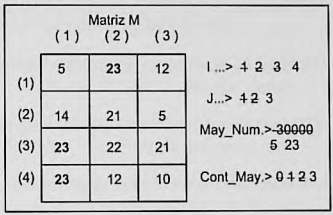
Cuando la variable I valga 4 y J valga 3, la decisión *Si M (I, J) = May_Num* se convertirá en *Si M (4, 3) = May_Num*, lo cual significa *Si 10 = 23*, que es Falso, por lo tanto, nos salimos tanto del ciclo interno como del ciclo externo, debido a que ambas variables ya llegaron a sus correspondientes topes. Finalmente, la última orden muestra en pantalla el valor solicitado:

Escriba "El número mayor es", May_Num. "y se encuentra", Cont_May, "veces"

PANTALLA



MEMORIA



Con lo cual solo quedaría "ejecutar" el fin del algoritmo.

Fin

Vemos pues que realmente, de los números leídos, el número 23 es el mayor y está 3 veces en la matriz, con lo cual podemos decir que este algoritmo sí cumple con el objetivo planteado inicialmente.

10.6. Ejercicios

Notas Aclaratorias:

a. En los siguientes enunciados cuando se diga *Leer una matriz mxn entera* significa leer *mxn* datos enteros y almacenarlos en *m* filas y *n* columnas para cualquier valor positivo de *m* y de *n*.

b. Cuando el enunciado diga *Posición Exacta* se refiere a la fila y a la columna del dato especificado.

1. Leer una matriz 4x4 entera y determinar en qué fila y en qué columna se encuentra el número mayor.

2. Leer una matriz 4x4 entera y determinar cuántas veces se repite en ella el número mayor.

3. Leer una matriz 3x4 entera y determinar en qué posiciones exactas se encuentran los números pares.

4. Leer una matriz 4x3 entera y determinar en qué posiciones exactas se encuentran los números primos.

5. Leer una matriz 4x3 entera, calcular la suma de los elementos de cada fila y determinar cuál es la fila que tiene la mayor suma.

6. Leer una matriz 4x4 entera y calcular el promedio de los números mayores de cada fila.

7. Leer una matriz 4x4 entera y determinar en qué posiciones están los enteros terminados en 0.

8. Leer una matriz 4x4 entera y determinar cuántos enteros terminados en 0 hay almacenados en ella.

9. Leer una matriz 3x4 entera y determinar cuántos de los números almacenados son primos y terminan en 3.

10. Leer una matriz 5x3 entera y determinar en qué fila está el mayor número primo.

11. Leer una matriz 5x3 entera y determinar en qué columna está el menor número par.

12. Leer una matriz 5x5 entera y determinar en qué fila está el mayor número terminado en 6.

13. Leer una matriz 5x3 entera y determinar en qué columna está el mayor número que comienza con el dígito 4.

14. Leer una matriz 5x5 entera y determinar cuántos números almacenados en ella tienen más de 3 dígitos.

15. Leer una matriz 5x4 entera y determinar cuántos números almacenados en ella terminan en 34.

16. Leer una matriz 5x4 entera y determinar cuántos números almacenados en ella tienen un solo dígito.

17. Leer una matriz 5x4 entera y determinar cuántos múltiplos de 5 hay almacenados en ella.

18. Leer una matriz 5x5 entera y determinar en qué posición exacta se encuentra el mayor múltiplo de 8.

19. Leer dos matrices 4x5 enteras y determinar si sus contenidos son exactamente iguales.

20. Leer dos matrices 4x5 enteras, luego leer un entero y determinar si cada uno de los elementos de una de las matrices es igual a cada uno de los elementos de la otra matriz multiplicado por el entero leído.

21. Leer dos matrices 4x5 enteras y determinar cuántos datos tienen en común.

22. Leer dos matrices 4x5 enteras y determinar si el número mayor almacenado en la primera está en la segunda.

23. Leer dos matrices 4x5 enteras y determinar si el número mayor de una de las matrices es igual al número mayor de la otra matriz.

24. Leer dos matrices 4x5 enteras y determinar si el mayor número primo de una de las matrices también se encuentra en la otra matriz.

25. Leer dos matrices 4x5 enteras y determinar si el mayor número primo de una de las matrices es también el mayor número primo de la otra matriz.

26. Leer dos matrices 4x5 enteras y determinar si la cantidad de números pares almacenados en una matriz es igual a la cantidad de números pares almacenados en la otra matriz.

27. Leer dos matrices 4x5 enteras y determinar si la cantidad de números primos almacenados en una matriz es igual a la cantidad de números primos almacenados en la otra matriz.

28. Leer una matriz 4x6 entera y determinar en qué posiciones se encuentran los números cuyo penúltimo dígito sea el 5.

29. Leer una matriz 4x6 entera y determinar si alguno de sus números está repetido al menos 3 veces.

30. Leer una matriz 4x6 entera y determinar cuántas veces está en ella el número menor.

31. Leer una matriz 4x6 entera y determinar en qué posiciones están los menores por fila.

32. Leer una matriz 4x6 entera y determinar en qué posiciones están los menores primos por fila.

33. Leer una matriz 4x6 entera y determinar en qué posiciones están los menores pares por fila.

34. Leer una matriz 4x6 entera y determinar cuántos de los números almacenados en ella pertenecen a los 100 primeros elementos de la serie de Fibonacci.

35. Leer dos matrices 4x6 enteras y determinar cuál es el mayor dato almacenado en ella que pertenezca a la serie de Fibonacci.

36. Leer dos matrices 4x6 enteras y determinar si el mayor número almacenado en una de ellas que pertenezca a la serie de Fibonacci es igual al mayor número almacenado en la otra matriz que pertenezca a la serie de Fibonacci.

37. Leer dos matrices 4x6 enteras y determinar si el número mayor de una matriz se encuentra en la misma posición exacta en la otra matriz.

38. Leer dos matrices 4x6 enteras y determinar si el mayor número primo de una matriz está repetido en la otra matriz.

39. Leer dos matrices 4x6 enteras y determinar si el promedio de las "esquinas" de una matriz es igual al promedio de las "esquinas" de la otra matriz.

40. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los elementos de la diagonal de una matriz es igual al promedio de los elementos de la diagonal de la otra matriz.

41. Leer dos matrices 5x5 enteras y determinar si el promedio entero de todos los elementos que no están en la diagonal de una matriz es igual al promedio entero de todos los elementos que no están en la diagonal de la otra matriz.

42. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los números primos de una matriz se encuentra almacenado en la otra matriz.

43. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los números pares de una matriz es igual al promedio de los números pares de la otra matriz.

44. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los números terminados en 4 de una matriz se encuentra al menos 3 veces en la otra matriz.

45. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los números mayores de cada fila de una matriz es igual al promedio de los números mayores de cada fila de la otra matriz.

46. Leer dos matrices 5x5 enteras y determinar si el promedio entero de los números menores por cada fila de una matriz corresponde a alguno de los datos almacenados en las "esquinas" de la otra matriz.

47. Leer dos matrices 5x5 enteras y determinar si el promedio de los mayores números primos por cada fila de una matriz es igual al promedio de los mayores números primos por cada columna de la otra matriz.

48. Leer dos matrices 5x5 entera y determinar si el promedio de los mayores elementos que pertenecen a la serie de Fibonacci de cada fila de una matriz es igual al promedio de los mayores elementos que pertenecen a la serie de Fibonacci de cada fila de la otra matriz.

49. Leer una matriz 3x3 entera y determinar si el promedio de todos los datos almacenados en ella se encuentra también almacenado.

50. Leer una matriz 5x5 y determinar si el promedio de los elementos que se encuentran en su diagonal está almacenado en ella. Mostrar en pantalla en qué posiciones exactas se encuentra dicho dato.

Capítulo 11

Funciones

11.1. Concepto general

Ningún concepto a nivel de la programación es más importante que el concepto de *función*. Sin temor a equivocarme, puedo garantizarle que la función es lo que podríamos llamar la gran "*vedette*" de la programación. Esto por ahora no será muy significativo para usted, pues tendrá que depurar muy bien el concepto de función como tal para que vea qué tan sencillo se vuelve programar basados en esta poderosísima herramienta.

Antes de entrar a definir qué es una función, quiero exponerle un breve texto para que lo lea detenidamente:

Receta para preparar unos huevos fritos

Ingredientes:

- *2 huevos crudos*
- *¼ de cucharadita de sal*
- *Una cucharadita de aceite*

Preparación:

Colóquese una cacerola a calentar en medio. Échese la cucharadita de aceite hasta cuando esté bien caliente. Quiébrense los huevos y vacíese su contenido en la cacerola. Esperar hasta cuando la clara esté bien blanca. Echar por encima del huevo de manera que quede repartida uniformemente la sal que se incluyó en los ingredientes.

Resultado:

Unos ricos huevos en cacerola que pueden ser acompañados con un pan.

Estoy absolutamente seguro de que en este momento del libro usted estará un poco desconcertado pues no sabrá por qué he comenzado este capítulo explicando algo que cada uno de nosotros de alguna manera sabe, como es preparar unos huevos fritos. Antes de entrar un poco más en materia, no está de más reescribir la receta anterior de la siguiente forma:

Receta para preparar unos huevos fritos

Ingredientes:

- *2 huevos crudos*
- *¼ de cucharadita de sal*
- *Una cucharadita de aceite*

Preparación:

Colóquese una cacerola a calentar en medio.

Échese la cucharadita de aceite hasta cuando esté bien caliente.

Quiébrense los huevos y vacíese su contenido en la cacerola.

Esperar hasta cuando la clara esté bien blanca.

Echar por encima del huevo de forma que quede repartida uniformemente la sal que se incluyó en los ingredientes.

Resultado:

Unos ricos huevos en cacerola que pueden ser acompañados con un pan.

Solo es un pequeño cambio como para que de alguna manera se vaya asociando con lo que hasta el momento hemos visto. De acuerdo a todo lo anterior, entonces tenemos la *receta para preparar unos huevos fritos* o, dicho mejor en nuestros términos, *el algoritmo para preparar unos huevos fritos.*

Acerca de esta receta (o algoritmo) vamos a plantear algunas precisiones:

a. La receta tiene un nombre específico que, en condiciones normales, no lo tiene ninguna otra receta. Esto quiere decir algo que, aunque parece redundante, será muy importante plantearlo desde ahora y es que la única receta que debe llamarse *receta para preparar unos huevos fritos* es la que nos enseña a preparar unos huevos fritos. Obvio, ¿cierto...?

b. Para la preparación efectiva de esta receta, podemos ver que se necesitan unos ingredientes sin los cuales sería imposible realizarla, por lo menos, tal como está explicada en el texto de la misma. Estos ingredientes serán diferentes por cada receta, aunque puedan existir recetas que tengan los mismos ingredientes y su diferencia radicará en el texto de la misma, o sea, en su preparación real. Es importante anotar que los ingredientes tienen unas características en tamaño, en peso y en medida. Igualmente, es útil anotar que los ingredientes no se explican detalladamente; por ejemplo: se supone, en esta receta, que estamos hablando de huevos de gallina.

c. La preparación no es más que un conjunto de pasos secuenciales y ordenados que nos permiten lograr el objetivo inicial (que en este caso era *preparar unos huevos fritos*). Estos pasos no se pueden alterar ya que hacerlo no nos garantizaría que el resultado final fuera el esperado.

d. El resultado final de la receta debe lograr el objetivo inicial planteado y ese resultado final es el que puede ser utilizado para todo aquello en lo cual consideremos que puede sernos útil.

Bien, de acuerdo a lo dicho anteriormente, nuestra receta también podríamos haberla escrito de la siguiente forma para estar acorde con las normas algorítmicas de seudocódigo planteadas a lo largo de este libro.

Receta para preparar unos huevos fritos ***(2 huevos crudos, ¼ de cucharadita de sal, una cucharadita de aceite)***

```
Inicio
        Colocar cacerola a calentar en medio
        Echar la cucharadita de aceite
        Mientras el aceite no esté bien caliente
                Esperar
        Fin_Mientras
        Quebrar los huevos (uno a uno)
        Vaciar su contenido en la cacerola
        Mientras la clara no esté bien blanca
                Esperar
        Fin_Mientras
        Echar la sal por encima del huevo
        Verificar que quede repartida uniformemente
Fin
```

Resultado:

Unos ricos huevos en cacerola que pueden ser acompañados con un pan.

Note usted que, vista así, la receta inicial puede ser vista, ahora sí oficialmente, como un algoritmo. Un algoritmo que tiene unas características:

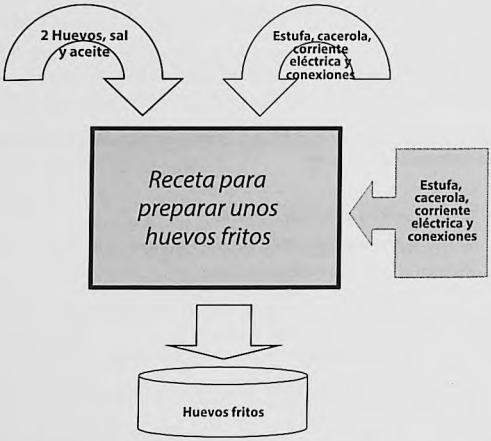
a. Tiene un nombre específico y único.

b. Tiene unos ingredientes que son los que permiten que la receta se realice.

c. Tiene un conjunto de pasos que son la receta en sí.

d. Tiene un resultado final que es el objetivo de la receta.

Siendo así, podríamos representar también este algoritmo con el siguiente diagrama de bloques:



Basados en lo concluido con esta receta, podemos escribir la siguiente definición:

FUNCIÓN.- Conjunto de órdenes que tiene las siguientes características:

- Permite lograr un objetivo.
- Tiene un nombre único identificativo.
- Puede necesitar parámetros para lograr dicho objetivo.
- Nos puede retornar un resultado que deberá concordar con el objetivo propuesto.
- Puede ser manejado como una sola unidad.

Conociendo usted la estética y características de este libro, sé que debe estarse preguntando por qué esta definición, que pareciera ser una más en la teoría de la programación, se ha colocado con tanta rimbombancia y de manera tan destacadamente notoria.

No es un "decoradito" que le coloqué al libro para que usted no se me fuera a aburrir. No. Es precisamente la definición más importante que tiene la programación. pues no en vano le dije al inicio de este capítulo que el concepto de función se podía considerar como la gran "*vedette*" de la programación... Voy a explicarle el porqué.

11.2. Problemas reales de la programación

Cuando se desarrolla un programa, el programador se enfrenta a varios problemas entre los cuales se encuentran los errores que en uno de los capítulos anteriores le expliqué. Ya hablábamos pues de los errores de sintaxis y los errores de precaución. En ese momento le decía yo que los errores más difíciles de encontrar eran los errores lógicos y que dichos errores solo se podían enfrentar desarrollando una excelente prueba de escritorio. Sin embargo, no es fácil concebir una prueba de escritorio cuando hemos desarrollado un algoritmo que tiene más de 1000 líneas... (¡¡¡ojo!!!, más de mil líneas), pues de manera muy fácil podemos "perdernos" en la ejecución de dicha prueba o sencillamente hacerla mal.

También cabe anotar que, cuando se desarrolla un programa extenso, muchas veces nos encontramos con que parte de él puede sernos útil en otro programa, pero con frecuencia lo que nos toca hacer es volver a copiar las líneas que creemos pueden tener cabida en el otro algoritmo. De esta manera, podemos resumir los problemas reales de la programación en tres tópicos:

a. Necesidad de simplificar la idea general para lograr un objetivo

Cuando usted se enfrenta a un objetivo, cualquiera que este sea, no sabemos en primera instancia cuántas líneas (u órdenes) va a tener nuestro algoritmo.

Cuando hemos escrito un algoritmo que, teóricamente, suponemos logrará dicho objetivo, muchas veces nos encontramos con la gran sorpresa de que hemos desarrollado una solución muchísimo más larga de lo que esperábamos. Basado en esto, concebir una sola idea que sea solución para lograr un objetivo es más complejo que concebir la misma idea pero fragmentada o dividida en pedacitos funcionales, dado que cuando algo complejo se subdivide en fracciones simples y no se pierde ninguna característica inicial entonces eso complejo se convierte en algo igualmente simple.

b. Simplificación de la prueba de escritorio y, por lo tanto, detección muy sencilla de errores

Dado el cuestionamiento anterior, podemos ver que, si tenemos un algoritmo muy extenso, entonces nuestra prueba de escritorio también será igualmente extensa y el riesgo de que nos quede mal hecho o de que omitamos pasos o de que cometamos algún error en la ejecución simulada es muy alto y, por lo tanto, en la medida en que un algoritmo es extenso, se reduce la confiabilidad de la prueba de escritorio no porque la teoría falle, sino porque en la realidad no es tan fácil realizarle pruebas de escritorio a algoritmos de muchas órdenes y a la vez de muchas variables.

c. Reutilización del código fuente

Siempre que desarrollamos un algoritmo, nos encontramos con que parte de lo que hicimos nos puede servir para otro algoritmo, pero casi siempre nos toca volver a copiar lo que hicimos y, de esa manera, incorporarlo en el nuevo algoritmo. Una de las necesidades más grandes que se comenzaron a sentir fue la necesidad de poder utilizar en posteriores programas lo que en un momento dado se estuviera haciendo, sin necesidad de hacerle ningún cambio. Esta necesidad es lo que ha llevado a desarrollar la teoría de la programación buscando nuevas y mejores formas de reutilizar el código fuente, o sea, lo que nosotros escribimos como solución y que hemos llamado algoritmos computacionales.

Estos tres problemas son los que llevaron a los diseñadores de lenguajes de programación a construir una "célula" fundamental, un núcleo de trabajo que permitiera a los programadores superar los tres grandes problemas que tiene la programación (o por lo menos facilitarle el camino para ello). Esta célula es lo que se llama **función**. Es importante anotar que en algunos lenguajes de programación tiene otros nombres pero la esencia es la misma y no es más que un conjunto de instrucciones que llevan un nombre único, que puede recibir parámetros (o ingredientes) para lograr su objetivo y que nos puede retornar un valor.

11.3. Macro algoritmo

Este concepto ha permitido colocar a las funciones en el puesto que les corresponde, pues con él podemos desarrollar algoritmos en donde, basados en funciones, llegamos claramente a:

a. Simplificar el algoritmo.

b. Simplificar la prueba de escritorio y, por lo tanto, encontrar fácilmente errores lógicos.

c. Reutilizar el código fuente.

¿Qué es pues el macro algoritmo? Es un algoritmo dividido en unidades funcionales en donde cada una de ellas logra un pequeño objetivo dentro de toda la solución y en conjunto dichas unidades logran el objetivo general. Vamos a verlo con un ejemplo:

Ejemplo.- Almacenar 10 números enteros en un vector y mostrar cuál es el mayor de los números leídos.

Ajustándonos a lo que hasta antes de este capítulo se había explicado, un posible algoritmo solución a este objetivo podría ser el siguiente:

```
Algoritmo Buscar_Mayor
Variables
        Entero :        V( 10 ),        // Almacenará los 10 datos enteros
                        Ind,            // Servirá como Índice del Vector
                        Mayor           // Almacenará el mayor de los números
                                        // leídos
Inicio
        Escriba "Digite 10 números enteros"     // Solicita los 10 datos enteros
        Ind = 1                                 // Comienza en la primera posición
        Mientras Ind < = 10                     // Mientras no haya llegado a la
                                                // última posición del vector
                Lea V ( Ind )                   // Lea un entero y guárdelo en la
                                                // posición Ind del Vector
                Ind = Ind + 1                   // Incremente la posición
        Fin_Mientras                            // Fin del ciclo
        Mayor = V ( 1 )                         // Inicialice la variable Mayor
```

```
                                        // con el contenido del Vector
                                        // en la primera posición
    Ind = 2                             // Comience en la segunda posición
    Mientras Id < = 10                  // Mientras no haya llegado a la
                                        // última posición del vector
        Si V ( Ind ) > Mayor            // Si el contenido del vector en la
                                        // posición Ind es mayor que el
                                        // contenido de la variable Mayor
            Mayor = V ( Ind )           // Entonces V ( Ind ) será el nuevo
                                        // mayor
        Fin_Si                          // Fin de la decisión
    Ind = Ind + 1                       // Pase a la siguiente posición
                                        // del vector
    Fin_Mientras                        // Fin del ciclo
    Escriba "El mayor número leído es", Mayor   // Muestre el resultado
Fin                                     // Fin del algoritmo
```

Usted podrá notar que este es un algoritmo común y corriente. Ahora bien, vamos a suponer tres cosas que nos serán muy útiles en este ejercicio:

1. Supongamos que este es un algoritmo muy extenso.
2. Supongamos que este es un algoritmo muy complejo.
3. Supongamos que este es un algoritmo exageradamente útil.

Note usted que este algoritmo de ejemplo, al cual es necesario que le hagamos las suposiciones aquí expuestas, se divide fundamentalmente en tres partes: en la primera parte se leen 10 datos enteros y se almacena cada uno en un vector, en la segunda parte se busca cuál es el número mayor de entre los números leídos y en la tercera parte se muestra el resultado final. Sé que coincidimos en el análisis de este algoritmo y en su subdivisión en estas tres partes tan lógicas. Bueno, pues precisamente ese es el macro algoritmo. Escribámoslo técnicamente:

```
Macro Algoritmo Buscar_Mayor
Inicio
    Lectura_de_Datos
    Búsqueda_del_Mayor
    Muestra_de_Resultado
Fin
```

Es evidente que nuestro algoritmo se simplificó muchísimo (y mucho más si partimos de las suposiciones hechas inicialmente). Ahora bien, cabría la pregunta *¿qué contiene cada uno de estos procesos?* Pues veamos cómo quedaría el algoritmo completo.

Algoritmo Buscar_mayor

Variables

Entero: *V(10),*
Ind,
Mayor

Función Principal

Inicio
Lectura_De_Datos
Búsqueda_del_Mayor
Muestra_de_Resultado
Fin

Función Lectura_De_Datos

Inicio
Escriba "Digite 10 números enteros"
Ind = 1
Mientras Ind < = 10
Lea V(Ind)
Ind = Ind + 1
Fin_Mientras
Fin

Función Búsqueda_del_Mayor

Inicio
Mayor = V(1)
Ind = 2
Mientras Id < = 10
Si V(Ind) > Mayor
Mayor = V(Ind)
Fin_Si
Ind = Ind + 1
Fin_Mientras
Fin

Función Muestra_del_Resultado

Inicio

Escriba "El mayor número leído es", Mayor

Fin

Estoy seguro de que usted debe estar pensando que el algoritmo no se acortó, sino que se alargó y puede que tenga razón si lo miramos en la cantidad de líneas que tienen las dos versiones del mismo algoritmo. Quiero exponerle algunas reflexiones acerca de esta versión del algoritmo:

a. Note que existe una función principal cuyo objetivo es permitir la coordinación de los llamados a las demás funciones.

b. También notará que este algoritmo está organizado en unas unidades funcionales donde cada una cumple un pequeño objetivo frente al objetivo general. Como puede ver usted, la función principal se encarga de llamar a las demás funciones: la ***Función Lectura_De_Datos*** cumple con el objetivo de leer los datos y almacenarlos en el vector, la ***Función Búsqueda_del_Mayor*** cumple con el objetivo de encontrar cuál es el mayor de los números almacenados en el vector y la función ***Muestra_del_Resultado*** se encarga de entregarnos el valor solicitado.

c. Esta forma de escribir el programa nos permite (mirando la función principal) entender mucho más fácil la idea general de solución de este algoritmo y, por lo tanto, eso nos simplifica el camino en el momento en el que, de alguna forma, le queramos hacer ajustes, mejoras o correcciones.

d. Realizarle una prueba de escritorio al algoritmo inicial (que no se olvide que estamos suponiendo que es un algoritmo extenso y muy complejo) se reduce ahora a realizar tres pruebas de escritorio sencillas.

La primera prueba será a la ***Función Lectura_De_Datos***, que como ya sabemos busca como objetivo leer 10 datos enteros y almacenarlos en un vector. Si al realizar dicha prueba vemos que SOLO esta función cumple su pequeño objetivo, entonces no tendremos que preocuparnos más de ella.

La segunda prueba de escritorio será a la ***Función Búsqueda_del_Mayor***, cuyo objetivo es encontrar el mayor número almacenado en un vector de 10 posiciones enteras. Si al realizar esta prueba vemos que la función cumple con este objetivo, entonces no tendremos que preocuparnos más por ella.

La tercera prueba (y en este caso la más sencilla) será a la ***Función Muestra_del_Resultado***, cuyo objetivo será mostrar en pantalla un dato entero almacenado

en una variable llamada *Mayor*. Si al realizar la prueba de escritorio vemos que realmente se cumple el objetivo, entonces no tendremos que preocuparnos más ni por la función ni por el objetivo al igual que en todos los casos.

a. Como puede notar, si al realizarle independientemente las respectivas pruebas de escritorio a estas tres funciones cada una cumple su objetivo (independientemente), entonces ya habremos desarrollado el algoritmo completo, pues la unión de las tres es lo que nos permite lograr el objetivo general del mismo.

b. Si alguien nos dijera que nuestro algoritmo no está funcionando bien, o sea, que no cumple con el objetivo general de encontrar el número mayor de entre los 10 números leídos, entonces esto querrá decir, sin lugar a dudas, que el error tiene que estar en la función que se encarga de seleccionar el mayor, dado que este proceso se realiza solo en esta función y no en ninguna de las otras dos.

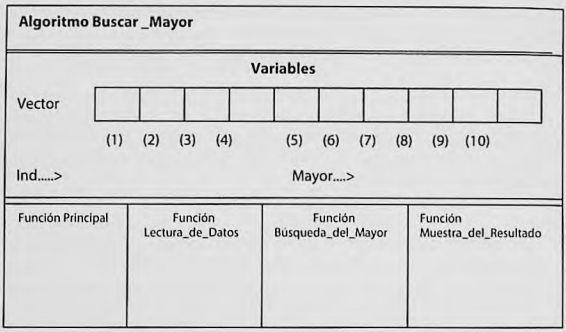
c. Si nos encontramos con que al ejecutar el algoritmo al final nos muestra en pantalla que el número mayor es un número que estamos seguros que no estaba entre los número leídos originalmente, entonces esto querrá decir que el error con toda seguridad está en la lectura de datos y que, seguramente, los datos que se están almacenando en el vector no corresponden a los datos leídos.

d. Si vemos que no nos muestra ningún resultado nuestro algoritmo al momento de ejecutarlo, entonces esto quiere decir que el error tiene que estar en la ***Función Muestra_de_Resultado***.

e. Si todas las pruebas de escritorio han sido exitosas, entonces la función principal estará bien, pues es la que llama a las otras tres funciones, y todo lo que tendremos que revisar es el orden de los llamados.

No podemos desconocer todas las ventajas que surgen cuando utilizamos el concepto de función para simplificar la solución de un problema y hacerlo mucho más sencillo en su concepción y su posterior ejecución. Es importante anotar que cuando se habla aquí de ejecución se hace referencia al caso en el cual el algoritmo se encuentre debidamente codificado y transcrito en un lenguaje de programación.

Utilizando un diagrama de bloques, podríamos representar el algoritmo planteado de la siguiente manera:



Como se puede notar, en un programa hecho con funciones el esquema gráfico para el desarrollo de la prueba de escritorio cambia un poquito. Sé que usted se estará preguntando qué podría ir dentro del cajoncito de cada función. Precisamente vamos a entrar en un concepto muy interesante que nos va a permitir hacer mucho más flexible y mucho más sencilla la programación.

11.4. Variables globales y variables locales

Hasta el momento, hemos hablado de variables en general y sencillamente las hemos definido al inicio de cada algoritmo y comenzamos a trabajar con ellas y listo. Ahora vamos a tecnificar un poco más la definición de las variables para que podamos llegar a construir funciones realmente flexibles y reutilizables.

Una variable global es aquella que se declara por fuera de cualquier función y, por lo tanto, puede ser reconocida en cualquier función. En el ejemplo que tratamos en este momento podemos ver que las variables Vector (10), *Ind* y Mayor son variables globales porque están declaradas por fuera de cualquier función (incluyendo la función principal). Por eso, estas variables las utilizamos indiscriminadamente en cualquiera de las funciones.

Una variable local es aquella que se declara dentro de alguna función y solo puede ser reconocida en esa función. En el ejemplo aún no hemos comenzado a utilizar variables locales. Esta definición implica que una función puede tener internamente declarada una variable con un nombre determinado y otra función también puede tener internamente declarada otra variable con el

mismo nombre y no habrá ningún error, ya que cada variable es diferente y es reconocida en la función en donde se declare. Eso es como María la de los Díaz y María la de los Pérez.

Supongo que usted hasta este momento del libro ya ha encontrado que es muy importante ver lo que se explica con un ejemplo. Por eso vamos a desarrollar un ejemplo sencillo en el cual se va a hacer necesario que usted se imagine que es un algoritmo muy complejo y difícil ya que solo así le encontrará, por ahora, sentido a muchos de los conceptos nuevos que aquí le presento.

11.5. Ejemplo

Leer dos números enteros y mostrar el resultado de su suma.

Clarificación del objetivo

Como puede ver el objetivo es muy sencillo y bien podría solucionarse de la siguiente forma:

```
Algoritmo Suma_Sencilla
Variables
        Entero :        Num1,
                        Num2,
                        Res
Inicio
        Escriba "Digite dos datos enteros"
        Lea Num1, Num2
        Resultado = Num1 + Num2
        Escriba Resultado
Fin
```

De resolverlo así, pues, no tendría mucho sentido extender esta explicación, pero lo importante es que por ahora usted suponga que este es un algoritmo complejo, pues basado en su desarrollo sobre funciones vamos a explicar qué son las variables globales y locales. Vamos pues a desarrollar este mismo algoritmo (o sea, a leer dos números enteros y mostrar su suma) pero utilizando funciones.

Algoritmo

```
Algoritmo Suma_con_Funciones
```

Función Principal

```
Variables
        Entero :                    // Estas son variables locales para la
                                    // Función Principal lo cual quiere decir
                                    // que pueden ser utilizadas solo dentro
                                    // de ella pues solo allí serán
                                    // reconocidas
Num1,  // Almacenará uno de los números
                    Num2,  // Almacenará el otro de los números
                    Res    // Almacenará el resultado de la suma
Inicio
        Escriba "Digite dos números enteros"    // Solicita los dos enteros
        Lea Num1, Num2                          // y los lee
        Res = Sumar ( Num1, Num2 )              // El contenido de la variable Res
                                    // será igual a lo que retorne la función
                                    // Sumar a la cual se le han enviado
                                    // dos parámetros (como quien dice dos
                                    // ingredientes) que deberá sumarlos
        Escriba Res                 // Escriba el resultado de la suma
Fin                                 // Fin de la función principal
```

Función Sumar (Entero a, Entero b) *// Función Sumar que para lograr*
// su objetivo siempre va a
// necesitar dos datos enteros

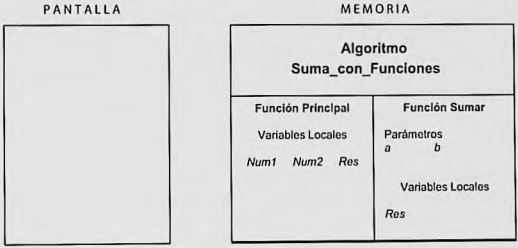
```
Variables
        Entero: Res             // Variable que almacenará la suma de
                                // los dos datos (ingredientes) recibidos
                                // por la función Sumar
Inicio
        Res = a + b             // Se le asigna a la variables Res
                                // la suma del contenido de a más el
                                // contenido de b
        Retorne ( Res )                 // Se le ordena a la función que
                                        // devuelva el valor almacenado
                                        // en la variable local Res
Fin                                     // Fin de la función Sumar
```

Sé que aún puede tener la inquietud del porqué he complicado tanto un algoritmo tan sencillo pero no se preocupe, siga leyendo y verá que el objetivo es puramente pedagógico y busca precisamente que usted tenga muy claro cada uno de los elementos que intervienen en una función y el manejo de variables. También estoy seguro de que en este instante usted se sentirá como si no supiera absolutamente nada de programación y como si todo lo que hasta el momento ha estudiado se hubiera desvanecido. No se preocupe, a continuación vamos a desarrollar (paso a paso) la prueba de escritorio detallada y notará que lo que aquí presentamos en este momento aparentemente confuso no es más que el pináculo de aplicación de todo lo que hasta el momento hemos visto.

Prueba de escritorio

Como puede ver, nuestro algoritmo no tiene variables globales debido a que todas las variables que se involucran en él están declaradas dentro de alguna función (no se olvide que la función principal no es más que otra función con la única diferencia de que es la primera que se va a ejecutar). Vamos a tener pues en la memoria dos funciones: la **Función Principal** y la **Función Sumar**, cada una con sus respectivas variables.

Algoritmo Suma_con_Funciones



Como puede ver, ya nuestra memoria no es, como hasta el momento había sucedido, un espacio solo para unas variables y no más. Ahora la memoria se convierte en el área en donde convergen funciones, variables locales y variables globales (si las hay) y nuestro objetivo con las pruebas de escritorio es manejar la memoria tal como lo haría el computador.

Aunque haciendo honor a esta última frase que acabo de escribir, debo admitir que inicialmente en la memoria del computador solo existe la función principal y que las demás se van colocando en ella (junto con sus variables) en la medida

en que se vayan llamando. Iniciamos pues nuestro algoritmo solicitando dos números enteros y almacenándolos el primero en la variable *Num1* y el segundo en la variable *Num2*. Vamos a suponer que dichos números son el 15 y el 18.

Función Principal

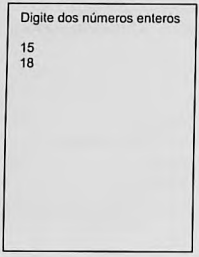
Variables

Entero : Num1,
Num2,
Res

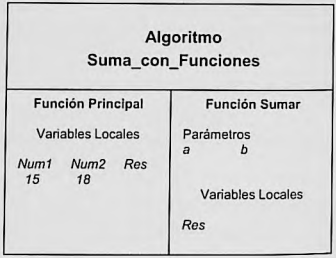
Inicio

Escriba "Digite dos números enteros"
Lea Num1, Num2

PANTALLA



MEMORIA



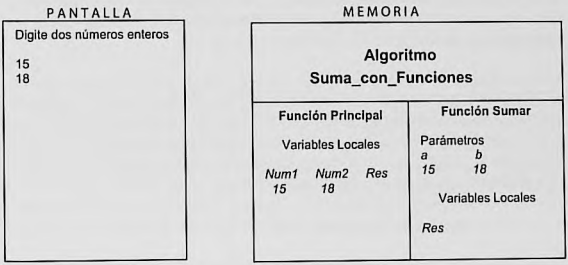
Como puede notarse y tal como se había explicado, el primer número (15) quedó almacenado en la variable Num1 y el segundo número (18) quedó almacenado en la variable Num2. A continuación, en nuestro algoritmo aparece la orden según la cual en la variable Res se almacenará el valor que retorne la función Sumar, a la cual se le envían como parámetros (o ingredientes para que logre su objetivo) los contenidos de cada una de las variables Num1 y Num2.

Res = Sumar (Num1, Num2)

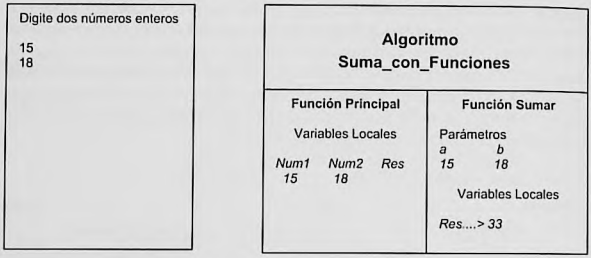
Al hacer el llamado a la función Sumar, entonces se activa esta función y se le envían como parámetros los contenidos de las variables Num1 y Num2 que se

almacenan respectivamente en las variables a y b. En estos momentos se suspende momentáneamente la ejecución de la Función Principal debido a que como la variable Res es igual al valor que retorne la Función Sumar entonces tendremos primero que ejecutarla antes de continuar. Realmente no es que se suspenda, sino que la variable Res de la función principal queda a la espera de lo que le retorne la Función Sumar. Entonces, para ejecutar la Función Sumar primero almacenamos en la variable a el valor que tenía la variable Num1 y en la variable b el valor que tenía la variable Num2.

```
Función Sumar (Entero a, Entero b)
Variables
        Entero : Res
Inicio
        Res = a + b
        Retorne ( Res )
Fin
```



Se puede notar que en la función Sumar la primera orden es almacenar en la variable Res el resultado de sumar los contenidos de las variables a y b. Tenga en cuenta que como a almacena el valor original de Num1 y b almacena el valor original de Num2, entonces ejecutar la suma a + b es igual que ejecutar la suma Num1 + Num2. Luego la orden Res = a + b se convierte en Res = 15 + 18, lo cual indica que en la variable Res queda almacenado el valor 33.



Seguidamente encontramos en la función Sumar la orden Retorne (Res), lo cual indica que la función Sumar va a devolver el valor almacenado en Res, que es 33. Allí, en el momento de retornar un valor, es donde termina la función Sumar. Tenga en cuenta que la variable Res que aparece en la Función Principal es diferente a la variable Res que aparece en la Función Sumar, debido a que cada una es variable local para su respectiva función (no se olvide que eso es como María la de los Díaz y María la de los Pérez).

Ahora sí podemos volver a la Función Principal pues ya la Función Sumar ha devuelto un valor. ¿A quién se lo devuelve? ¿Se acuerda usted que suspendimos la prueba de escritorio de la Función Principal pues se necesitaba saber cuál era el valor que se iba a almacenar en la variable Res (local para la función principal)? Pues bien, ahora es el momento de retomar esa orden en donde se había hecho la suspensión y era la orden:

Res = Sumar (Num1, Num2)

Como la Función Sumar al habérsele enviado los parámetros Num1 y Num2 (o sea, 15 y 18 respectivamente) retornó el valor 33, entonces internamente esta orden se convierte en:

Res = 33

Que no es más que una orden sencilla de asignación (no se olvide que ese cambio se hace internamente). Ya con esto podemos ver entonces que en la variable *Res* de la *Función Principal* queda almacenado el valor *33*. Y como la orden que sigue a continuación dentro de la *Función Principal* es mostrar en pantalla el valor almacenado en la variable *Res*

Escriba Res

Entonces nuestra prueba de escritorio termina realizándolo así:

PANTALLA

Digite dos números enteros

15
18

33

MEMORIA

Algoritmo
Suma_con_Funciones

Función Principal	**Función Sumar**
Variables Locales *Num1* *Num2* *Res* *15* *18* *33*	Parámetros *a* *b* *15* *18* Variables Locales *Res....> 33*

Quedando pendiente solamente dar por terminada esta prueba de escritorio, pues hemos llegado al fin de la Función Principal.

Fin

Ahora sí podemos ver en la pantalla que el valor final es igual a la suma de los dos números iniciales y precisamente ese era el objetivo: Leer dos números enteros y mostrar su suma. Sé que a esta altura usted debe estar pensando que no se justificaba hacer tanto para lograr un objetivo tan sencillo (y tiene toda la razón). No se olvide que el objetivo de este ejemplo es solamente didáctico, pues ahora sí vamos a ver un ejemplo que va a tener mucha utilidad, pero antes quiero plantearle algunas reflexiones acerca de este ejemplo.

a. No existen para el caso presentado variables globales.

b. Se utilizan variables locales en cada una de las funciones.

c. Una de las funciones (la *Función Sumar*) necesita unos parámetros que son como los ingredientes con los cuales esa función puede lograr su objetivo. Recuerde que dicho objetivo era retornar el resultado de sumar dos números; entonces lo que necesita para poder cumplirlo son los dos números.

d. Cuando una función retorna un valor, entonces allí termina la función.

e. Una función (en este caso la *Función Principal*) puede llamar a otra (en este caso la *Función Sumar*).

f. Cualquier función puede llamar a otra función; solo se debe tener en cuenta que si una función A llama a una función B, entonces dicha función B no debe llamar explícitamente a la función A ya que con su retorno es suficiente.

g. Una función A puede llamar a una función B y esa función B puede llamar a una función C y así sucesivamente. Cada retorno regresará el control del programa a la función llamadora.

h. Un algoritmo puede tener muchas funciones. Todo dependerá del objetivo que se quiera lograr y del conocimiento de esta técnica por parte del programador.

i. El fin de la función principal es el fin de todo el algoritmo.

Ahora sí, basados en todo lo que hasta ahora hemos visto, vamos a ver un ejemplo que le brinde a usted un panorama más práctico no sin desconocer la importancia de los ejemplos didácticos que van hasta el momento.

11.5.1. Ejemplo No. 2

Leer un número y si es un número par, calcular su factorial, mostrarlo en pantalla y determinar si también es par.

Clarificación del objetivo

Como ya podemos suponer, el objetivo de este algoritmo es leer un número entero y sencillamente calcularle su factorial en el caso de que dicho número sea par. Primero que nada tendremos que determinar si el número es par. Para ello, recordemos que un número es par si al dividirlo entre 2 y volverlo a multiplicar por 2 nos da como resultado el mismo número. Este razonamiento está basado en las características y propiedades de la aritmética entera, según la cual ninguna operación genera decimales. Siendo así, entonces podremos saber que el número 8 es par si cumple con la condición establecida:

$$8 = 8 / 2 * 2$$

Resolviendo esta expresión por jerarquía de operadores sabemos que primero que nada se haría la división y luego se realizaría la multiplicación, luego:

$$8 = 8 / 2 * 2$$
$$8 = 4 * 2$$
$$8 = 8$$

Al ver que desarrollando esta expresión obtenemos el mismo valor inicial, entonces podemos implementar un algoritmo que concluya que el número 8 es par. Sin embargo, hagamos la prueba con un número impar para que usted vea qué pasa:

$$9 = 9 / 2 * 2$$
$$9 = 4 * 2$$
$$9 = 8$$

Vemos que el resultado de la expresión, basados en la aritmética entera, no es igual al valor inicial planteado, por lo tanto, nuestro algoritmo podría concluir que 9 no es par.

Siguiendo con la clarificación del objetivo, recordemos que el factorial de un número es la multiplicación sucesiva de todos los enteros comprendidos entre 1 y dicho número, partiendo de que dicho número sea positivo. No está definido el factorial para los números negativos y el factorial de 0 es 1. De esta forma, el factorial de 4 es igual al resultado de multiplicar:

$$1 \times 2 \times 3 \times 4 = 24$$

El factorial de -6 no está definido y, como dice la definición, el factorial de 0 es 1.

Con estos conceptos ya podemos saber que lo que tenemos que organizar es un algoritmo que nos permita leer un número entero, verificar si es un número par y si lo es, entonces calcularle su factorial, al cual también se le debe verificar si es un número par.

Algoritmo

Ya esta vez no vamos a mostrar un algoritmo y no más, tal como en la mayoría de las veces. Esta vez vamos a explicar el porqué de la solución final. Primero que nada vamos a construir una función que nos permita recibir, como parámetro, un número entero y determinar si es par o impar. Ese *determinar* lo haremos de la siguiente forma: la función retornará un número 1 si el número es par y retornará un número 0 si el número es impar. De esta manera, una solución a esta función podría ser.

```
Función Es_Par ( Entero n )          // Función Es_Par que recibe como
                                     // parámetro un valor entero
Inicio
    Si n / 2 * 2 = n                 // Pregunta si el valor recibido es par
        Retorne ( 1 )                // entonces que retorne un 1
    Sino                             // Sino
        Retorne ( 0 )                // que retorne un 0
    Fin_Si                           // Fin de la decisión
Fin                                  // Fin de la función
```

Usted tal vez se preguntará a quién se le retorna ese 1 o ese 0... Pues muy sencillo, se le retorna a la función que llame a esta función porque es claro que, para que esta función se ejecute, otra función la tiene que llamar.

Vamos a hacerle una pequeña prueba de escritorio a esta función. Suponga-mos que llamamos a la *Función Es_Par* y se le envía el valor 5, o sea, que el llamado es de la siguiente forma:

```
Es_Par ( 5 )
```

No se olvide que una función se llama escribiendo sencillamente su nombre. La palabra función se ha colocado en este libro solo para fines didácticos, pues es muy importante que a lo largo de él usted vaya reconociendo las verdadera unidades fundamentales de trabajo de un algoritmo.

Siendo ese el llamado, entonces la variable *n* se cargará con el número *5* e internamente nuestra función preguntará:

```
Si 5 / 2 * 2 = 5
```

Lo cual es *Falso*, dado que *5 / 2 * 2* es igual a *4*, entonces la función retornará un *0* tal como la habíamos planteado, pues se había dicho que debería retornar 0 en caso de que el número no fuera par.

Ahora vamos a probar en el caso de que el llamado a la *Función Es_Par* sea el siguiente:

```
Es_Par ( 8 )
```

Entonces el parámetro (ingrediente) llamado *n* se cargará con el valor *8* y, por lo tanto, la función tomará la siguiente decisión:

```
Si 8 / 2 * 2 = 8
```

Lo cual es *Verdadero*, pues *8 / 2 * 2* es igual a *8* por propiedades de la aritmética entera. Con esto podemos garantizar que la *Función Es_Par* nos permite determinar si un número es par o no. En caso de que el número que le enviemos como parámetro sea par, esta función retornará 1 y en caso de que dicho parámetro no sea par, entonces nos retornará un 0. Con esto ya no tenemos que preocuparnos por la decisión de si un número es par o no.

Ahora vamos a construir una función a la cual le enviemos un parámetro entero y ella nos retorne el factorial de ese número. Esta función nos retornará un número entero positivo en caso de que el parámetro que le enviemos sea igualmente positivo, nos retornará un número 1 en caso de que el parámetro que le enviemos sea un 0 y nos retornará un número –1 en caso de que le enviemos un valor negativo. Como ya sabemos que no existen factoriales negativos, eso querrá decir que, si al llamar la función ella nos retorna un –1, es porque el parámetro que le habíamos enviado era negativo y con ello podremos detectarlo. Una posible solución a esta función podría ser:

```
Función Factorial ( Entero n )          // Función Factorial que recibirá como
                                        // parámetro un entero
Variables Locales                       // Declaración de las variables locales
        Entero : Facto,                 // Almacenará el valor del factorial
                 Cont                   // Permitirá generar los números desde 1
                                        // hasta el parámetro recibido que es a
                                        // quien se le debe calcular el factorial
Inicio
        Si n < 0                        // Si el parámetro recibido es negativo
                Retorne ( -1 )          // Retorna un –1 que luego será interpretado
                                        // desde la función llamadora
        Facto = 1                       // Inicie la variable Facto en 1
        Cont = 1                        // Inicie la variable Cont en 1
        Mientras Con < = n              // Mientras Cont no haya llegado al
                                        // número recibido como parámetro
                Facto = Facto * Cont    // Multiplique sucesivamente todos los
                                        // enteros
Cont = Cont + 1                         // comprendidos entre 1 y el número
                                        // recibido como parámetro
        Fin_Mientras                    // Fin del ciclo Mientras
        Retorne ( Facto )               // Retorne el valor almacenado en la
                                        // variable Facto
Fin                                     // Fin de la función
```

```
Función Es_Par ( Entero n )          // Función Es_Par que recibe como
                                     // parámetro un valor entero
Inicio
    Si n / 2 * 2 = n                 // Pregunta si el valor recibido es par
        Retorne ( 1 )                // entonces que retorne un 1
    Sino                             // Sino
        Retorne ( 0 )                // que retorne un 0
    Fin_Si                           // Fin de la decisión
Fin                                  // Fin de la función
```

Usted tal vez se preguntará a quién se le retorna ese 1 o ese 0... Pues muy sencillo, se le retorna a la función que llame a esta función porque es claro que, para que esta función se ejecute, otra función la tiene que llamar.

Vamos a hacerle una pequeña prueba de escritorio a esta función. Suponga-mos que llamamos a la *Función Es_Par* y se le envía el valor 5, o sea, que el llamado es de la siguiente forma:

```
Es_Par ( 5 )
```

No se olvide que una función se llama escribiendo sencillamente su nombre. La palabra función se ha colocado en este libro solo para fines didácticos, pues es muy importante que a lo largo de él usted vaya reconociendo las verdadera unidades fundamentales de trabajo de un algoritmo.

Siendo ese el llamado, entonces la variable *n* se cargará con el número *5* e internamente nuestra función preguntará:

```
Si 5 / 2 * 2 = 5
```

Lo cual es *Falso*, dado que *5 / 2 * 2* es igual a *4*, entonces la función retornará un *0* tal como la habíamos planteado, pues se había dicho que debería retornar 0 en caso de que el número no fuera par.

Ahora vamos a probar en el caso de que el llamado a la *Función Es_Par* sea el siguiente:

```
Es_Par ( 8 )
```

Entonces el parámetro (ingrediente) llamado *n* se cargará con el valor *8* y, por lo tanto, la función tomará la siguiente decisión:

```
Si 8 / 2 * 2 = 8
```

Lo cual es *Verdadero*, pues *8 / 2 * 2* es igual a *8* por propiedades de la aritmética entera. Con esto podemos garantizar que la *Función Es_Par* nos permite determinar si un número es par o no. En caso de que el número que le enviemos como parámetro sea par, esta función retornará 1 y en caso de que dicho parámetro no sea par, entonces nos retornará un 0. Con esto ya no tenemos que preocuparnos por la decisión de si un número es par o no.

Ahora vamos a construir una función a la cual le enviemos un parámetro entero y ella nos retorne el factorial de ese número. Esta función nos retornará un número entero positivo en caso de que el parámetro que le enviemos sea igualmente positivo, nos retornará un número 1 en caso de que el parámetro que le enviemos sea un 0 y nos retornará un número –1 en caso de que le enviemos un valor negativo. Como ya sabemos que no existen factoriales negativos, eso querrá decir que, si al llamar la función ella nos retorna un –1, es porque el parámetro que le habíamos enviado era negativo y con ello podremos detectarlo. Una posible solución a esta función podría ser:

```
Función Factorial ( Entero n )          // Función Factorial que recibirá como
                                        // parámetro un entero
Variables Locales                       // Declaración de las variables locales
        Entero : Facto,          // Almacenará el valor del factorial
                 Cont            // Permitirá generar los números desde 1
                                 // hasta el parámetro recibido que es a
                                 // quien se le debe calcular el factorial
Inicio
        Si n < 0                 // Si el parámetro recibido es negativo
                Retorne ( -1 )   // Retorna un –1 que luego será interpretado
                                 // desde la función llamadora
        Facto = 1                // Inicie la variable Facto en 1
        Cont = 1                 // Inicie la variable Cont en 1
        Mientras Con < = n       // Mientras Cont no haya llegado al
                                 // número recibido como parámetro
                Facto = Facto * Cont  // Multiplique sucesivamente todos los
                                      // enteros
Cont = Cont + 1                  // comprendidos entre 1 y el número
                                 // recibido como parámetro
        Fin_Mientras             // Fin del ciclo Mientras
        Retorne ( Facto )        // Retorne el valor almacenado en la
                                 // variable Facto
Fin                              // Fin de la función
```

Vamos a desarrollar una pequeña prueba de escritorio a esta función para ver si cumple con el objetivo planteado, que es el de calcular el factorial del número que se le envíe como parámetro. Vamos a comenzar suponiendo que a la función se le llama enviándosele el valor entero –4.

Factorial (-4)

La función *Factorial* recibirá ese número *–4* en el parámetro *n* y procederá a su ejecución que, luego de declarar las variables locales *Facto* y *Cont*, procederá a preguntar si *n* (o sea, el valor recibido como parámetro) es menor que *0*. Como esto es *Verdadero*, entonces la función retorna el número –1 y allí termina. No se olvide que, apenas en una función se encuentra una orden *Retorne*, la función llega en su ejecución hasta allí. Ya desde la función llamadora analizaríamos el resultado que nos retorne la función *Factorial* y sabemos que, en caso de que retorne el número –1, esto querrá decir que el valor que se le envió era negativo y podremos decirle al usuario que *No están definidos los factoriales de números negativos*.

Veamos cuál sería el resultado que retornaría la función *Factorial* en caso de que el llamado fuera:

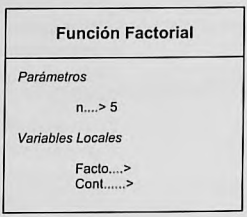
Factorial (5)

En este caso, nuestra prueba iniciaría almacenando este valor 5 en la variable *n* que lo recibiría y declarando las variables locales de la función.

Función Factorial (Entero n)

Variables Locales

Entero : *Facto,*
Cont



Nuestra función comienza, pues, preguntando:

```
Inicio
    Si n < 0
        Retorne (-1)
```

Lo cual es Falso, dado que en la variable n está almacenado el valor 5. Por lo tanto, la prueba continúa con el resto de la función. Se inicializa la variable Facto en 1 y la variable *Cont* en 1.

```
Facto = 1
Cont = 1
```

Función Factorial
Parámetros n....> 5 *Variables Locales* Facto....> 1 Cont.....> 1

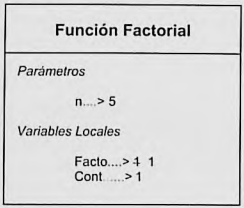
Nuestra función continúa con las órdenes:

```
Mientras Cont < = n
        Facto = Facto * Cont
        Cont = Cont + 1
Fin_Mientras
```

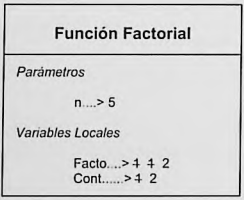
Se pregunta si el contenido de la variable *Cont* (que es 1) es menor que el contenido de la variable n (que es 5). Como es Verdadero, entonces se ejecuta la operación:

```
Facto = Facto * Cont
```

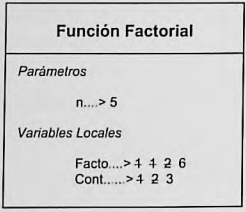
Luego en la variable *Facto* queda almacenado el número 1.



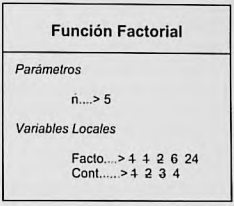
Se incrementa en 1 el contenido de la variable *Cont* y se vuelve a preguntar si dicho valor es menor o igual que el valor almacenado en n (que sigue siendo 5). Como es Verdadero, entonces se vuelve a ejecutar la multiplicación Facto = Facto * Cont.



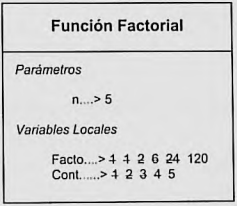
Se vuelve a incrementar en 1 el contenido de la variable *Cont* y volvemos a evaluar la condición del ciclo Mientras. Como el valor almacenado en *Cont* sigue siendo menor que n, entonces volvemos a hacer la multiplicación Facto = Facto * Cont.



Volvemos a incrementar en 1 el contenido de la variable *Cont* y de nuevo se evalúa la condición del ciclo. El valor almacenado en *Cont* sigue siendo menor o igual que el valor almacenado en n, por lo tanto, se ejecuta de nuevo la operación Facto = Facto * Cont.



Se vuelve a incrementar en 1 el contenido de la variable *Cont* (que esta vez ya valdrá 5) y se pregunta de nuevo la condición del ciclo. Como *Cont* sigue siendo menor o igual que *n*, pues ambas almacenarían el valor 5, entonces se ejecuta por última vez la operación que se encuentra en el cuerpo del ciclo, o sea, *Facto = Facto * Cont*.



Se vuelve a incrementar en 1 el contenido de la variable *Cont* y vemos que como almacenaría un número 6 no se cumple la condición del ciclo, o sea, que *Cont* <= *n*, pues n sigue valiendo 5, por lo tanto, nos salimos del ciclo y ejecutamos la orden que está después del *Fin_Mientras*:

Retorne (Facto)

Fin

Que no es más que retornar el valor almacenado en la variable *Facto*. En este caso retornaría el valor 120, que efectivamente corresponde al factorial del

valor recibido que fue el número 5. En el caso de que el llamado a la función *Factorial* sea

Factorial (0)

Sencillamente en la variable *n* se cargará este parámetro, se inicializará la variable *Facto* en 1 y la variable *Cont* en 1. Cuando se llegue al ciclo a la primera vez que se evalúe la condición del mismo (o sea, *Cont* <= *n*) como *Cont* vale 1 y vale 0, entonces la condición será *Falsa* y pasaremos a la instrucción que está después del *Fin_Mientras*. Esta instrucción ordena retornar el valor almacenado en *Facto*, que para este caso sería el número 1. Con ello se cumpliría también con esta función que cuando se le envíe el número 0 ella retorna el número 1 como su factorial.

Podemos pues concluir que la función *Factorial* así como está concebida realmente nos permite que le enviemos un número y ella nos calcula su correspondiente factorial.

Ahora sí vamos a desarrollar el algoritmo completo basado en estas dos funciones, de las cuales ya no tenemos que preocuparnos, pues ya tenemos la absoluta certeza de que la *función Es_Par* permite determinar si un número es par y la *función Factorial* nos permite calcular el factorial de un número.

Algoritmo Determina_Facto_Par

Función Principal

```
Variables Locales
        Entero :    Num,                    // Almacenará el número
                                            // original leído
                    Aux1,                   // Variables Auxiliares
                    Aux2
Inicio
        Escriba "Digite un número entero"   // Solicita un número entero
        Lea Num                             // lo lee y lo almacena en Num
        Aux1 = Es_Par ( Num )               // Aux1 es igual a lo que retorne
                                            // la función Es_Par enviándole
                                            // como parámetro el valor
                                            // almacenado en Num
        Si Aux1 = 0                         // Si ese valor retornado es cero
                Escriba "El número leído es impar"   // entonces el número
                                                     // no es par
```

```
    Sino                                    // Sino (el número si es par)
        Aux2 = Factorial ( Num )            // Aux2 es igual al valor
                                            // que retorne la función
                                            // Factorial enviándole
                                            // como parámetro el
                                            // contenido de la
                                            // variable Num
        Si Aux2 = -1                        // Si ese valor retornado
                                            // por la función Factorial
                                            // es -1 entonces quiere
                                            // decir que el número
                                            // original era negativo
            Escriba "No está definido el factorial para números negativos"
        Sino                                // Sino entonces se
                                            // muestra en pantalla
                                            // que el factorial del
                                            // valor almacenado en
                                            // Num es igual al valor
                                            // almacenado en Aux2
                                            // que fue lo que retornó
                                            // la función Factorial
            Escriba "El factorial de", Num, "es", Aux2
        Fin_Si
    Fin_Si
Fin

Función Es_Par ( Entero n )                 // Función Es_Par que recibe como pará-
metro un                                    // valor entero

Inicio
    Si n / 2 * 2 = n                        // Pregunta si el valor recibido es par
        Retorne ( 1 )                       // entonces que retorne un 1
    Sino                                    // Sino
        Retorne ( 0 )                       // que retorne un 0
    Fin_Si                                  // Fin de la decisión
Fin                                         // Fin de la función
```

```
Función Factorial ( Entero n )          // Función Factorial que recibirá como
                                        // parámetro un entero
Variables Locales                       // Declaración de las variables locales
        Entero : Facto,                 // Almacenará el valor del factorial
                 Cont                   // Permitirá generar los números desde 1
                                        // hasta el parámetro recibido que es a
                                        // quien se le debe calcular el factorial
Inicio
        Si n < 0                        // Si el parámetro recibido es negativo
                Retorne ( -1 )          // Retorna un –1 que luego será
                                        //interpretado desde la función llamadora
        Facto = 1                       // Inicie la variable Facto en 1
        Cont = 1                        // Inicie la variable Cont en 1
        Mientras Con < = n              // Mientras Cont no llegue al número
                                        // recibido como parámetro
                Facto = Facto * Cont    // Multiplique sucesivamente todos
                                        // los enteros
Cont = Cont + 1                         // comprendidos entre 1 y el número
                                        // recibido como parámetro
        Fin_Mientras                    // Fin del ciclo Mientras
        Retorne ( Facto )               // Retorne el valor almacenado en la
                                        // variable Facto
Fin                                     // Fin de la función
```

Ahora usted sí podrá ver lo tan útil que resulta ser para un programador utilizar el concepto de función, pues, como puede ver en este algoritmo, ya no tenemos que realizar una prueba de escritorio detallada a todo el algoritmo, ya que este se basa en las funciones *Es_Par* y *Factorial* y como al hacerle su correspondiente prueba de escritorio a cada una de ellas funcionaron perfectamente, podemos garantizar que el algoritmo completo también está bien. Le sugiero solo como una confirmación que le haga usted la prueba de escritorio a todo el algoritmo. Tenga muy en cuenta los valores que se retornan de cada función.

Adicionalmente, en este algoritmo hemos obtenido una ganancia que yo estoy seguro de que usted todavía no ha reflexionado. Tenemos dos funciones muy confiables que pueden llegar a ser utilizadas sin ningún temor en cualquier otro programa. Cabe anotar que en el programa anterior no era

estrictamente necesario utilizar las variables *Aux1* y *Aux2* pero facilitan la claridad del algoritmo.

11.6. Ejemplo

Leer números hasta que digiten 0 y determinar a cuánto es igual el promedio de los factoriales de los números pares positivos.

Clarificación del objetivo

Vamos a leer varios números, no sabemos cuántos, y en la medida en que los leamos iremos acumulando la suma de los factoriales de los números pares al tiempo que los iremos contando. Cuando nos digiten el valor 0, entonces realizaremos la división entre el valor acumulado de los factoriales de los números pares y la cantidad de números pares que entraron y ese es el resultado que nos solicita este enunciado.

Puede usted notar que, de alguna manera, este ejercicio tiene una leve relación con el ejercicio anterior, pues vamos a necesitar determinar si un número es par y también vamos a necesitar el cálculo del factorial del número. En estas condiciones se hará muy sencillo desarrollar este algoritmo, pues nos vamos a apoyar en las mismas funciones (óigase bien, las mismas funciones) que utilizamos en el ejercicio anterior.

Algoritmo

Algoritmo Prom_Facto_Pares

Función Principal

```
Variables Locales                    // Variables locales de
                                     // la función principal
    Entero :    Num,                 // Almacenará cada uno de los números
                                     // leídos
                Acum_Facto,          // Almacenará la suma de todos los
                                     // factoriales de los números pares
                Cont_Pares,          // Almacenará la cantidad de números
                                     // pares a los cuales se les calculó su factorial
                Promedio             // Almacenará el promedio solicitado
Inicio
                                     // Solicita números e indica que finalicen
                                     // con 0
```

son absolutamente correctas y todo lo que tenemos que hacer al momento de desarrollar la prueba de escritorio es tener en cuenta que el llamado a la función se reemplaza internamente por el valor que ella retorne. Quedará pendiente para usted desarrollar la prueba de escritorio de este algoritmo.

11.8. Menús

Concepto general

Un menú sencillamente es un conjunto de opciones que se le presentan a un usuario para que él, de manera voluntaria y libre, escoja cuál ejecutar. Al igual que cuando vamos a un restaurante el mesero nos presenta una lista de opciones (platos) para que nosotros escojamos, en nuestro ejemplo presentaremos un menú de opción leída dado que es la estructura que el seudocódigo nos permite. Los lenguajes de programación nos facilitan herramientas para construir menús de botones, de barras deslizantes y menús gráficos, pero por ahora nos concentraremos en la parte lógica del diseño de un menú. Para ello, vamos a ejemplificar el concepto a través de un ejemplo muy sencillo.

Ejemplo

Brindar las siguientes opciones a través de un menú:

1. Leer un número entero.
2. Determinar si dicho número es primo.
3. Determinar si dicho número es par.
4. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número par.
5. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número primo.
6. Determinar el factorial de dicho número entero.

Es importante que se tenga en cuenta que siempre que nosotros brindemos un menú de opciones deberá existir una que permita terminar con el programa, o sea, salir del menú, que se interpretaría sencillamente como salir del programa. Como usted puede ver, nuestro enunciado ya no es tan sencillo como lo fue en otras oportunidades. Para su solución, vamos a analizarlo opción por opción.

1ª. Opción. Leer un número entero

Para cumplir con este objetivo, vamos a construir una función que nos permita leer un número entero y retornarlo a la función llamadora. Es evidente que esta función no necesitará ningún tipo de parámetros, de tal manera que una posible solución sería la siguiente:

```
Función Lectura_Num ( )                    // Nombre de la función
Variables Locales                          // Declaración de las variables locales
        Entero : N                         // Variable que almacenará el número a leer
Inicio
        Escriba "Digite un número entero"  // Solicita un número entero
        Lea N                              // y lo lee almacenándolo en la
                                           // variable N
        Retorne ( N )                      // Retorna el valor leído
Fin                                        // Fin de la función
```

Por lo simplificada de la función, podemos ver que efectivamente logrará el objetivo de leer un número entero y retornarlo.

2ª. Opción. Determinar si dicho número es primo

Para cumplir efectivamente con este objetivo se hace necesario que construyamos una función que nos permita recibir como parámetro un número entero y que retorne el valor 1 si dicho parámetro es un número primo y 0 si dicho parámetro no es un número primo. Recordemos entonces, primero que nada, *¿qué es un número primo?* Un número primo es un número que solo es divisible exactamente entre 1 y sí mismo. Por ejemplo: el número 19 es primo porque solo es divisible exactamente entre 1 y 19. El número 18 no es primo porque lo dividen exactamente los números 1, 2, 3, 6, 9 y 18. Esta es la definición que por mucho tiempo hemos manejado y que ahora pretendemos programar a través de una función.

Vamos a realizarle un pequeño cambio a la definición sin apartarnos de la esencia de ella, pero lo que necesitamos es facilitar el algoritmo y con él, facilitar su respectiva función. Siendo N un número cualquiera, los posibles divisores exactos de N están en el rango de 1 a N (eso es evidente). Sin embargo, sabiendo que todos, absolutamente todos los números son divisibles exactamente entre 1 y N (siendo N cualquier número), entonces los divisores exactos de un número N cualquiera que nos interesan estarán en el rango de 2 a N-1.

Podríamos decir que si un número N no tiene divisores exactos en el rango de 2 a N-1, entonces con toda seguridad es un número primo y lo contrario

también es cierto, es decir, si un número tiene al menos un divisor exacto en el rango de 2 a N-1, entonces el número no es primo.

Verifiquemos lo anterior con un ejemplo: el número 19 es primo porque no tiene ningún divisor exacto en el rango de 2 a 18, o sea, no hay ningún número que divida exactamente a 19 que sea mayor o igual que 2 y menor o igual que 18. Por eso podemos decir con toda certeza que el 19 es primo y es verdad. El número 24, en cambio, no es primo porque en el rango de 2 a 23 el número 24 tiene los siguientes divisores: 2, 3, 4, 6, 8 y 12. Por esta razón, podemos asegurar que el número 24 no es primo y también es cierto.

Por lo tanto, nuestra función se reduce a determinar si el número que se reciba como parámetro tiene divisores exactos en ese rango (o sea, 2 a N-1, siendo N el número a verificar). De ser así, entonces se retornará 0, pues si tiene divisores exactos en ese rango significa que el número recibido como parámetro no es un número primo, y de no ser así, se retornará 1, pues si no tiene divisores exactos en ese rango es porque los únicos números que lo dividen exactamente son el 1 y el mismo número N, sea cual fuere.

Para optimizar un poco esta función, vamos a involucrarle dos pequeños cambios con el propósito de lograr el objetivo de una manera mucho más eficiente:

a. El rango de 2 a N-1 lo vamos a reducir de 2 a N/2, dado que ningún número tiene un divisor exacto de su mitad entera en adelante. El número 1000, por ejemplo, no tiene ningún divisor exacto entre 501 y 999.

b. No vamos a esperar llegar hasta N/2 en el caso de que se encuentre al menos un divisor, pues con ese simple hecho el número N a evaluar ya no sería primo y sería suficiente para retornar la respuesta solicitada.

Con estos dos pequeños cambios tendremos de una manera altamente eficiente la respuesta a nuestro pequeño objetivo. Para que el ciclo que inicialmente va a ir desde 1 hasta N/2 se interrumpa en cualquier momento, vamos a utilizar una variable que actuará a manera de interruptor y será la que nos va a permitir que nuestro objetivo se logre eficientemente. Sin más preámbulos, una solución a esta función podrá ser la siguiente:

```
Función Es_Primo ( Entero N )
Variables Locales
    Entero : Ind,                  // Servirá para que se generen los
                                   // números enteros de 2 a N/2
                S                  // Variable que actuará como interruptor
                                   // y que servirá de retorno
```

```
Inicio
        Ind = 2                          // Inicia la variable Ind en 2
        S = 1                            // Inicia el interruptor en 1
        Mientras Ind < = N / 2 Y S = 1   // Mientras no se haya llegado a la mitad
                                         // del número y mientras no se haya
                                         // encontrado ningún divisor exacto
                Si N / Ind * Ind = N     // Si el valor almacenado en Ind es un
                                         // divisor exacto de N
                        S = 0            // Cambie el valor del interruptor
                Fin_Si                   // Fin de la decisión
                Ind = Ind + 1            // Pase al siguiente número dentro del rango
        Fin_Mientras                     // Fin del ciclo
        Retorne ( S )                    // Retorne el valor almacenado en S
Fin                                      // Fin de la función
```

Vamos a efectuarle una pequeña prueba de escritorio a esta función para verificar que efectivamente funcione bien. Nuestra función comienza almacenando el parámetro que se le envía en la variable N. Supongamos que se le envía el número 12. No se olvide que esta función retornará 1 si el número (parámetro) es primo o 0 si no lo es. Con la última aclaración quiero recordar que esta función no va a mostrar nada en pantalla, solo va a retornar un 1 o un 0 dependiendo del valor que se reciba como parámetro.

Función Es _Primo
Parámetros: N.....> 12 *Variables Locales:* Ind...> 2 S.....> 1

```
Función Es_Primo ( Entero N )
Variables Locales
        Entero : Ind,
                 S
```

Inicio

Ind = 2

S = 1

Seguidamente asignamos a la variable *Ind* el valor 2 y a la variable *S* el valor 1 y procedemos a entrar en el ciclo mientras que se plantea.

Mientras Ind < = N / 2 **Y** *S = 1*

*Si N / Ind * Ind = N*

S = 0

Fin_Si

Ind = Ind + 1

Fin_Mientras

Conociendo los valores almacenados en las variables respectivas, se evalúa la condición *Mientras Ind < = N / 2* **Y** *S = 1* y vemos que el valor almacenado en *Ind* efectivamente es menor o igual que N/2, dado que *Ind* es igual a 2 y N/2 es igual a 6, y además se ve que el valor almacenado en *S* es igual a 1, por lo tanto, toda la condición es Verdadera. Entonces efectuamos la pregunta:

*Si N / Ind * Ind = N*

Reemplazando por los valores respectivos la decisión se convierte en:

*Si 12 / 2 * 2 = 12*

Lo cual es efectivamente Verdadero porque se está preguntando en el fondo si el número 12 es divisible exactamente entre 2 y así es, por lo tanto, ejecutamos la orden

S = 0

Seguidamente, luego del *Fin_Si* correspondiente, ejecutamos la orden de incrementar en 1 el valor almacenado en la variable *Ind*.

Función Es _Primo
Parámetros: N.....> 12 *Variables Locales:* Ind...> ~~2~~ 3 S.....> ~~1~~ 0

Como encontramos seguidamente el *Fin_Mientras* correspondiente, entonces volvemos a evaluar la condición del ciclo:

Mientras Ind < = N / 2 Y S = 1

Y vemos que el contenido de *Ind* sigue siendo menor o igual que el valor N/2 (o sea, que 3 es menor o igual que 6), pero vemos que el valor de *S* ya no es 1 y, por lo tanto, como las dos condiciones están unidas por un operador **Y**, entonces toda la condición es *Falsa*. Pasamos entonces a la instrucción que se encuentra después del *Fin_Mientras* correspondiente, o sea:

Retorne (S)

Fin

Con lo cual retornaríamos el valor almacenado en *S* (que esta vez sería el número 0) y que coincide con la definición, pues habíamos dicho que en caso de que el parámetro no fuera un número primo entonces debía retornarse 0. De manera que para el caso de que el número no sea primo la función "funciona" (y valga esa redundancia).

Veamos ahora una prueba de escritorio para el caso en que el parámetro recibido sea primo. La función deberá retornar un número 1 indicando que efectivamente el número es primo. Supongamos que el valor recibido como parámetro es el número 7 (que es primo), entonces nuestra pequeña prueba de escritorio comenzaría asignando dicho número a la variable *N* y declarando las variables *Ind* y *S* e iniciándolas con los valores 2 y 1 respectivamente.

Función Es _Primo
Parámetros: N.....> 7 *Variables Locales:* Ind...> 2 S.....> 1

Nuestra función continúa con el planteamiento de un ciclo que depende de que el contenido de *Ind* sea menor o igual que *N/2* y que el contenido de *S* sea 1. Como por esta vez la condición del ciclo es Verdadera, entonces ejecutamos la decisión que sigue:

```
Mientras Ind <= N/2 Y S = 1
        Si N/Ind * Ind = N
                S = 0
        Fin_Si
        Ind = Ind + 1
Fin_Mientras
```

La pregunta *Si N/Ind * Ind = N* se convierte en *Si 7/2 * 2 = 7*, o sea, si 7 es divisible exactamente entre 2, lo cual es *Falso*, por lo tanto, pasamos a la instrucción que se encuentra después del *Fin_Si* correspondiente e incrementamos en 1 el contenido de la variable *Ind*.

Función Es _Primo
Parámetros: N.....> 7 *Variables Locales:* Ind...> ~~2~~ 3 S.....> 1

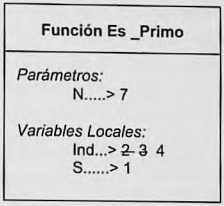
Como lo que sigue es el *Fin_Mientras* correspondiente, entonces volvemos a evaluar la condición del ciclo.

```
Mientras Ind <= N/2 Y S = 1
```

Vemos pues que sigue siendo *Verdadera*, pues el contenido de *Ind* es menor o igual que *N/2*, dado que ambos valores valen 3 y el contenido de la variable *S* sigue siendo 1. Por lo tanto, volvemos a entrar al ciclo y hacemos la decisión:

```
Si N/Ind * Ind = N
```

O sea, que preguntamos si el número 7 (que corresponde al contenido de *N*) es divisible exactamente entre 3 (que corresponde al contenido de *Ind*). Como la respuesta a esta decisión es *Falso*, entonces pasamos a la instrucción que se encuentra después del *Fin_Si* correspondiente, o sea, que volvemos a incrementar en 1 el contenido de *Ind*, quedando esta variable con el valor 4.



Como seguidamente encontramos el fin del ciclo, volvemos a evaluar la condición el ciclo, o sea:

Mientras Ind < = N / 2 **Y** *S = 1*

Vemos entonces que el contenido de *Ind* (que es igual a 4) ya no es menor ni igual que el resultado N/2 (pues este es igual a 3), por lo tanto, esta parte de la decisión es Falsa; aunque el contenido de *S* sigue siendo 1, como están unidas por un operador **Y**, toda la decisión es Falsa y, por lo tanto, nos salimos del ciclo, pasando a ejecutar la instrucción que se encuentra después del respectivo *Fin_Mientras*.

Retorne (S)

Fin

Con lo cual retornamos el valor almacenado en *S*, que es igual a 1, con lo cual se confirma el propósito inicial, pues habíamos dicho que se retornaría 1 si el valor recibido como parámetro es primo. Con esta prueba de escritorio podemos garantizar que la función *Es_Primo* sí nos permite determinar si el valor recibido como parámetro es un número primo o no. Solo existe una condición para su utilización exitosa y es que el valor recibido sea un número positivo.

3ª opción. Determinar si dicho número es par

Para determinar si un dato es par o no, ya no tenemos que preocuparnos, pues ya desarrollamos una función que se encarga de esto y la vamos a utilizar en este programa. Me refiero a la función:

Función Es_Par (Entero n) *// Función Es_Par que recibe como*
// parámetro un valor entero

Inicio

*Si n / 2 * 2 = n* *// Pregunta si el valor recibido es par*

```
            Retorne ( 1 )          // entonces que retorne un 1
        Sino                       // Sino
            Retorne ( 0 )          // que retorne un 0
        Fin_Si                     // Fin de la decisión
Fin                                // Fin de la función
```

Como a esta función ya se le hizo la prueba de escritorio y tenemos la absoluta certeza de que funciona, entonces no tenemos que hacer mayor cosa. Solo utilizarla bien dentro de nuestro programa.

4ª. Opción. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número par

Para cumplir este objetivo, todo lo que tenemos que hacer es sumar todos los números enteros comprendidos entre 1 y el número leído y ese resultado, que deberá quedar en una variable almacenado, enviarlo como parámetro a la función *Es_Par* y ella se encargará de decirnos si este número es par o no.

5ª. Opción. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número primo

Al igual que en el objetivo anterior, debemos sumar todos los números enteros comprendidos entre 1 y el número leído y enviar ese resultado como parámetro a la función *Es_Primo* que acabamos de construir. Esa función se encargará de retornarnos la respuesta acerca de si el número es primo o no.

6ª. Opción. Determinar el factorial de dicho número entero

Para calcular el factorial, ya tenemos una función probada que nos permite obtener ese resultado. Es la función *Factorial* y todo lo que tenemos que hacer es utilizarla bien.

```
Función Factorial ( Entero n )     // Función Factorial que recibirá como
                                   // parámetro un entero
Variables Locales                  // Declaración de las variables locales
    Entero : Facto,          // Almacenará el valor del factorial
                 Cont          // Permitirá generar los números desde 1
                               // hasta el parámetro recibido que es a
                               // quien se le debe calcular el factorial
Inicio
    Si n < 0                   // Si el parámetro recibido es negativo
        Retorne ( -1 )         // Retorna un –1 que luego será
```

```
                                        // interpretado desde la función llamadora
    Facto = 1                           // Inicie la variable Facto en 1
    Cont = 1                            // Inicie la variable Cont en 1
    Mientras Con < = n                  // Mientras Cont no haya llegado al
                                        // número recibido como parámetro
        Facto = Facto * Cont            // Multiplique sucesivamente todos
                                        // los enteros
        Cont = Cont + 1                 // comprendidos entre 1 y el número
                                        // recibido como parámetro
    Fin_Mientras                        // Fin del ciclo Mientras
    Retorne ( Facto )           // Retorne valor almacenado en la variable Facto
Fin                                     // Fin de la función
```

Como a esta función ya se le había hecho su correspondiente prueba de escritorio y ya tenemos absoluta certeza de sus resultados, no tenemos que preocuparnos de ella.

Como puede ver, "armar" un algoritmo a partir del concepto de funciones definitivamente se simplifica pues, cada vez que usted desarrolla una función, va a ser muy posible que en algoritmos futuros la pueda utilizar y estará ahorrando trabajo en el logro de los objetivos de ese nuevo algoritmo. Encontrar los errores, como ya vimos, es muy sencillo cuando se utiliza esa filosofía de trabajo y, sobre todo, comprender la lógica del algoritmo se hace todavía más sencillo.

El enunciado inicial es:

Brindar las siguientes opciones a través de un menú:

1. Leer un número entero.
2. Determinar si dicho número es primo.
3. Determinar si dicho número es par.
4. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número par.
5. Determinar si la suma de todos los enteros comprendidos entre 1 y dicho número es un número primo.
6. Determinar el factorial de dicho número entero.

El algoritmo completo solución para este enunciado podría ser el siguiente:

```
Algoritmo Menu_de_Opciones
Función Principal
Variables Locales
        Entero : Num,                   // Almacenará el número leído
                 Opcion,                // Almacenará la opción escogida por el
                                        // usuario
                 Acum                   // Acumulará la suma de 1 hasta el
                                        // número leído
Inicio
        Opcion = 0                      // Inicializamos la variable Opcion con
                                        // cualquier valor para que la condición
                                        // del ciclo inicialmente sea Verdadera
        Mientras Opcion < > 7           // Mientras no presionen la opción de Salir
                Escriba "1. Leer un número entero"      //Muestre las opciones
                Escriba "2. Determinar si dicho número es primo"
                Escriba "3. Determinar si dicho número es par"
                Escriba "4. Determinar si la suma de 1 a N es par"
                Escriba "5. Determinar si la suma de 1 a N es primo"
                Escriba "6. Calcular el factorial del número leído"
                Escriba "7. Salir"
                Lea Opcion              // Recibe la opción que quiere
// el usuario
                Evalúe ( Opcion )       // Evalúe lo que el usuario digitó
                        Si vale 1 :             // Si escogió la opción 1
                                Num = Lectura_Num ( )  // Entonces llame a la
                                                       // función que lee un
                                                       // entero y lo que
                                                       // retorne almacénelo
                                                       // en la variable Num
                        Si vale 2 :                    // Si el usuario escogió
                                                       // la opción 2
                                Si Es_Primo ( Num ) = 1 // Si lo que retorne la
                                                        // función Es_Primo
```

```
                                    // enviándole como
                                    // parámetro el valor
                                    // almacenado en la
                                    // variable Num es igual
                                    // a 1 quiere decir que el
                                    // número leído es primo
            Escriba "El número es primo"
        Sino
            Escriba "El número no es primo"
        Fin_Si
Si vale 3:                          // Si el usuario escogió
                                    // la opción 3
        Si Es_Par ( Num ) = 1       // Si lo que retorne la
                                    // función Es_Par
                                    // enviándole
                                    // como parámetro el
                                    // valor almacenado en
                                    // la variable Num es
                                    // igual a 1 quiere decir
                                    // que el número leído
                                    // es Par
            Escriba "El número es par"
        Sino
            Escriba "El número no es par"
        Fin_Si
Si vale 4 :                         // Si el usuario escogió
                                    // la opción 4
        Acum = 0                    // Inicialice esta
                                    // variable en 0
        Para Ind = 1 hasta Num      // Acumule en ella
                                    // el valor
                                    // resultante de sumar
                                    // todos los
                                    // enteros comprendidos
                                    // entre 1 y el valor leído
```

```
                    Acum = Acum + Ind
            Fin_Para
            Si Es_Par ( Acum ) = 1      // Si el valor retornado
                                        // por la función Es_Par
                                        // al enviársele como
                                        //parámetro el contenido
                                        // de Acum es 1 entonces
                                        // eso quiere decir que
                                        // el valor de esa suma
                                        // es par
                    Escriba "La suma de 1 a", Num, "es par"
            Sino
                    Escriba "La suma de 1 a", Num, "no es par"
            Fin_Si
    Si vale 5:                          // Si el usuario escoge la
                                        // opción 5
            Acum = 0                    // Inicialice esta variable
                                        // con 0
            Para Ind = 1 hasta Num      // Acumule en ella el
                                        // valor resultante de
                                        // sumar todos los
                                        //enteros comprendidos
                                        // entre 1 y el valor leído
                    Acum = Acum + Ind
            Fin_Para
            Si Es_Primo ( Acum ) = 1 // Si el valor retornado
                                        //por la función Es_primo
                                        // al enviársele como
                                        // parámetro el
                                        // contenido de Acum es
                                        // 1 entonces eso quiere
                                        // decir que el valor de
                                        // esa suma es un
                                        //número primo
            Escriba "La suma de 1 a", Num, "es un número primo"
```

```
                    Sino
                    Escriba "La suma de 1 a", Num, "no es un primo"
                    Fin_Si
            Si vale 6 :                     // Si el usuario escogió
                                            // la opción 6
                    Si Factorial ( Num ) = -1   // Evalúe si lo que retorna
                                            // la función Factorial
                                            // es igual a -1
                                            // entonces quiere decir
                                            // que el número original
                                            // era un número negativo
                    Escriba "No están definidos factoriales de núm negat"
                    Sino                    // De no ser así entonces
                                            // muestre en pantalla
                                            // el valor del factorial
                                            // del número leído
                    Escriba "El factorial de", Num, "es", Factorial ( Num )
                    Fin_Si
            Si vale 7 :                     // Si el usuario escogió
                                            // la opción 7
                    Escriba "Adiós"         // Despedirse
            Sino :                          // Si el usuario no escogió
                                            // ni la opción 1, ni la 2,
                                            // ni la 3, ni la 4, ni la 5,
                                            // ni la 6, ni la 7 quiere
                                            // decir que se equivocó y
                                            // entonces nos toca
                                            // avisarle
                    Escriba "Opción Errada"
        Fin_Evalúe                          // Fin de la estructura
                                            // Evalúe
    Fin_Mientras                            // Fin del ciclo
Fin                                         // Fin de la Función Principal
                                            // (que significa en otras
                                            // palabras Fin del algoritmo)
```

```
Función Lectura_Num ( )                    // Nombre de la función

Variables Locales                          //Declaración de las variables locales
        Entero : N                         // Variable que almacenará el núm a leer
Inicio
        Escriba "Digite un número entero"  // Solicita un número entero
        Lea N                              // y lo lee almacenándolo en la
                                           // variable N
        Retorne ( N )                      // Retorna el valor leído
Fin                                        // Fin de la función

Función Es_Par ( Entero n )                // Función Es_Par que recibe como
                                           // parámetro un valor entero
Inicio
        Si n / 2 * 2 = n                   // Pregunta si el valor recibido es par
                Retorne ( 1 )              // entonces que retorne un 1
        Sino                               // Sino
                Retorne ( 0 )              // que retorne un 0
        Fin_Si                             // Fin de la decisión
Fin                                        // Fin de la función

Función Es_Primo ( Entero N )
Variables Locales
        Entero : Ind,                      // Servirá para que se generen los
                                           // números enteros de 2 a N/2
                        S                  // Variable que actuará como interruptor
                                           // y que servirá de retorno
Inicio
        Ind = 2                            // Inicia la variable Ind en 2
        S = 1                              // Inicia el interruptor en 1
        Mientras Ind < = N / 2 Y S = 1     // Mientras no se haya llegado a la mitad
                                           // del número y mientras no se haya
                                           // encontrado ningún divisor exacto
                Si N / Ind * Ind = N       // Si el valor almacenado en Ind es un
                                           // divisor exacto de N
                        S = 0              // Cambie el valor del interruptor
```

```
            Fin_Si                  // Fin de la decisión
            Ind = Ind + 1           // Pase al siguiente número dentro del rango
        Fin_Mientras                // Fin del ciclo
        Retorne ( S )               // Retorne el valor almacenado en S
Fin                                 // Fin de la función

Función Factorial ( Entero n )      // Función Factorial que recibirá como
                                    // parámetro un entero
Variables Locales                   // Declaración de las variables locales
            Entero : Facto,         // Almacenará el valor del factorial
                     Cont           // Permitirá generar los números desde 1
                                    // hasta el parámetro recibido que
                                    // es a quien se le debe calcular
                                    // el factorial
Inicio
        Si n < 0            // Si el parámetro recibido es negativo
            Retorne ( -1 )          // Retorna un –1 que luego será
                                    // interpretado desde la función llamadora
        Facto = 1                   // Inicie la variable Facto en 1
        Cont = 1                    // Inicie la variable Cont en 1
        Mientras Con < = n          // Mientras Cont no haya llegado al
                                    // número recibido como parámetro
            Facto = Facto * Cont    // Multiplique sucesivamente todos
                                    // los enteros
            Cont = Cont + 1         // comprendidos entre 1 y el número
                                    // recibido como parámetro
        Fin_Mientras                // Fin del ciclo Mientras
        Retorne ( Facto )           // Retorne el valor almacenado en la
                                    // variable Facto
Fin                                 // Fin de la función
```

Usted deberá desarrollarle la prueba de escritorio a la *Función Principal* dado que a las demás funciones ya se le hizo su correspondiente prueba. Cuando necesite el resultado de alguna de las demás funciones, todo lo que tiene que hacer es colocar el resultado que usted sabe que debe retornar la función, pues esta ya ha sido probada.

Es importante que tenga en cuenta que cuando se brinda un menú no existe un orden en las opciones que escoja el usuario. Él podrá escoger cualquier opción y el algoritmo deberá funcionar. Igualmente, el usuario podrá escoger una y otra opción tantas veces como quiera y solo se saldrá del algoritmo cuando escoja la opción de *Salir* (para este caso la opción 7). De la misma manera, tenga en cuenta que todo menú deberá tener siempre una opción para salir naturalmente.

Usted deberá realizarle una prueba de escritorio a la *Función Principal* y ajustar todo el algoritmo para cuando el número original sea negativo, pues este caso no está considerado en él intencionalmente. Procure hacer todos los cambios en la función principal y deje las demás funciones intactas.

11.9. Ejercicios

Nota aclaratoria: en los siguientes enunciados se deberá construir la función solicitada y una *Función Principal* que haga uso de la función solicitada.

1. Construir una función que reciba como parámetro un entero y retorne su último dígito.

2. Construir una función que reciba como parámetro un entero y retorne sus dos últimos dígitos.

3. Construir una función que reciba como parámetro un entero y retorne la cantidad de dígitos.

4. Construir una función que reciba como parámetro un entero y retorne la cantidad de dígitos pares.

5. Construir una función que reciba como parámetro un entero y retorne la cantidad de dígitos primos.

6. Construir una función que reciba como parámetro un entero y retorne el carácter al cual pertenece ese entero como código ASCII.

7. Construir una función que reciba como parámetro un carácter y retorne el código ASCII asociado a él.

8. Construir una función que reciba como parámetro un entero y retorne 1 si dicho entero está entre los 30 primeros elementos de la serie de Fibonacci. Deberá retornar 0 si no es así.

9. Construir una función que reciba un entero y le calcule su factorial sabiendo que el factorial de un número es el resultado de multiplicar sucesivamente todos los enteros comprendidos entre 1 y el número dado. El factorial de 0 es 1. No están definidos los factoriales de números negativos.

10. Construir una función que reciba como parámetro un entero y retorne el primer dígito de este entero.

11. Construir una función que reciba como parámetro un entero y un dígito y retorne 1 si dicho entero es múltiplo de dicho dígito y 0 si no es así.

12. Construir una función que reciba como parámetro un entero y un dígito y retorne 1 si dicho dígito está en dicho entero y 0 si no es así.

13. Construir una función que reciba como parámetro un entero y un dígito y retorne la cantidad de veces que se encuentra dicho dígito en dicho entero.

14. Construir una función que reciba como parámetros dos números enteros y retorne el valor del mayor.

15. Construir una función que reciba como parámetros dos números enteros y retorne 1 si el primer número es múltiplo del segundo y 0 si no.

16. Construir una función que reciba como parámetro un entero y retorne 1 si corresponde al código ASCII de una letra minúscula (los códigos ASCII de las letras minúsculas van desde 97, que es el código de la letra a, hasta 122, que es el código de la letra z). Deberá retornar 0 si no es así.

17. Construir una función que reciba como parámetro un entero y retorne 1 si corresponde al código ASCII de un dígito (los códigos ASCII de las letras minúsculas van desde 48, que es el código del dígito 0, hasta 57, que es el código del dígito 9). Deberá retornar 0 si no es así.

18. Construir una función que reciba como parámetro un valor entero y retornar 1 si dicho valor es el factorial de alguno de los dígitos del número. Deberá retornar 0 si no es así.

19. Construir una función que reciba como parámetro un entero y retorne 1 si dicho valor es un número de mínimo 3 dígitos. Deberá retornar 0 si no es así.

20. Construir una función que reciba como parámetro un entero y retorne 1 si en dicho valor todos los dígitos son iguales. Deberá retornar 0 si no es así.

21. Construir una función que reciba como parámetro un entero y retorne 1 si en dicho valor el primer dígito es igual al último. Deberá retornar 0 si no es así.

22. Construir una función que reciba como parámetro un entero y retorne 1 si dicho valor es múltiplo de 5. Deberá retornar 0 si no es así.

23. Construir una función que reciba como parámetro dos enteros y retorne 1 si la diferencia entre los dos valores es un número primo. Deberá retornar 0 si no es así.

24. Construir una función que reciba como parámetro dos enteros de dos dígitos cada uno y retorne 1 si son inversos. Ejemplo: 83 es inverso de 38. Deberá retornar 0 si no es así.

25. Construir una función que reciba como parámetro un entero y un dígito menor o igual a 5 y retorne el dígito del número que se encuentre en la posición especificada por el dígito que llegó como parámetro.

26. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne el mayor de los datos del vector.

27. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la posición en la cual se encuentra el mayor de los datos del vector.

28. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la cantidad de números primos almacenados en el vector.

29. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la cantidad de números que pertenecen a los 30 primeros elementos de la serie de Fibonacci.

30. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la posición del mayor número primo almacenado en el vector.

31. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne el promedio entero del vector.

32. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne el promedio real del vector.

33. Construir una función que reciba como parámetros un vector de 10 posiciones enteras y un valor entero y retorne 1 si dicho valor entero se encuentra en el vector. Deberá retornar 0 si no es así.

34. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la posición del número entero que tenga mayor cantidad de dígitos.

35. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la posición en la que se encuentre el mayor número primo que termine en 3 almacenado en el vector.

36. Construir una función que reciba como parámetro un entero y retorne ese elemento de la serie de Fibonacci.

37. Construir una función que reciba como parámetros dos enteros, el primero actuará como base y el segundo como exponente, y retorne el resultado de elevar dicha base a dicho exponente.

38. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y retorne la cantidad de números terminados en 3 que contiene el vector.

39. Construir una función que reciba como parámetros un vector de 10 posiciones enteras y un dígito y que retorne la cantidad de veces que dicho dígito se encuentra en el vector. No se olvide que un mismo dígito puede estar varias veces en un solo número.

40. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y un dígito y que retorne la cantidad de números del vector que terminan en dicho dígito.

41. Construir una función que reciba como parámetro un vector de 10 posiciones enteras y un dígito y que retorne la cantidad de números del vector en donde dicho dígito está de penúltimo.

42. Construir una función que reciba como parámetro una matriz de 3x4 entera y retorne la cantidad de veces que se repite el mayor dato de la matriz.

43. Construir una función que reciba como parámetro una matriz 3x4 entera y retorne la cantidad de números primos almacenados en la matriz.

44. Construir una función que reciba como parámetro una matriz 3x4 entera y retorne la cantidad de veces que se repite el mayor número primo de la matriz.

45. Construir una función que reciba como parámetros una matriz 4x4 entera y un valor entero y retorne la cantidad de veces que se repite dicho valor en la matriz.

46. Construir una función que reciba como parámetro una matriz 4x4 entera y retorne el número de la fila en donde se encuentre por primera vez el número mayor de la matriz.

47. Construir una función que reciba como parámetro una matriz 4x4 entera y retorne el número de la columna en donde se encuentre por primera vez el número mayor de la matriz.

48. Construir una función que reciba como parámetro una matriz 4x4 entera y retorne la posición exacta en donde se encuentre almacenado el mayor número primo.

49. Construir una función que reciba una matriz 5x5 y retorne el valor de su moda. La moda de un conjunto de datos es el dato que más se repite.

50. Construir una función que reciba una matriz 5x5 y retorne la cantidad de veces que se repite su moda.

Capítulo 12

Consejos y reflexiones sobre programación

En los siguientes párrafos pretendo condensar en una serie de consejos y reflexiones toda mi experiencia como programador y como profesor de Lógica de Programación. Algunos de estos comentarios obedecen a criterios puramente técnicos y otros sencillamente son recomendaciones que me han hecho mucho más fácil el trabajo de la programación y su correspondiente convivencia con diferentes lenguajes de computador. Por esta razón, espero que este capítulo sea para usted un aporte significativo en su tarea de formación como programador.

12.1. Acerca de la lógica

Siempre que usted vaya a resolver un problema, sea muy lógico. Esto quiere decir que, sencillamente, guíese por sus mínimos razonamientos y busque siempre el camino más obvio y sencillo. No existe un problema que se resuelva con lógica cuya solución no sea sencilla. Antes de comenzar a pensar en la lógica de programación, piense en su propia lógica. Diseñe las soluciones pensando en sus propias reglas y, luego sí, ajústese a las reglas que la lógica de programación impone para facilitar la posterior codificación.

Es muy importante que usted poco a poco destine parte de su tiempo a resolver problemas, así no sean estrictamente de programación, dado que en esos momentos en donde usted se sienta a pensar detenidamente en la búsqueda de solución de un determinado problema, en esos momentos, es donde usted realmente está utilizando su cerebro. Normalmente, nuestro cerebro se va acostumbrando, y así es como lo orienta la educación formal, a buscar entre sus conocimientos las soluciones que se estén necesitando, pero muchas veces tenemos que crear soluciones y es allí en donde nos encontramos que son muy pocas las veces en las que ponemos a funcionar nuestro cerebro.

Yo espero que usted, amigo lector, no se me vaya a ofender, ya que no es el propósito de este párrafo incomodarlo. Lo que sí quiero es que usted piense cuántas veces realmente se ha sentado a crear una solución de un problema y encontrará que son muy pocas las veces, ya que en aquellas oportunidades en donde ha tratado de hacerlo y, por ventura, ha encontrado la solución es porque su cerebro ha buscado en su "biblioteca de conocimientos" alguna solución análoga a problemas parecidos y la ha "ajustado" al problema en mención.

Por tal motivo, es muy importante que de vez en cuando resuelva acertijos matemáticos, problemas con palitos, dados, cartas e incluso hasta resolver adivinanzas. Este tipo de problemas le van permitiendo a usted buscar soluciones espontáneas, originales, creadas por usted mismo y que además son solución a un determinado problema planteado. Todos esos juegos de lógica que más de una noche nos han puesto a pensar son los que van haciendo que el cerebro cree soluciones y no las busque en las que ya conoce. Es muy importante que tenga en cuenta todo esto, dado que en programación usted va a necesitar crear soluciones a problemas determinados basadas en sus conceptos y en el conocimiento que tenga de las herramientas y de los conceptos aquí planteados.

Siempre que usted se enfrente a un problema, no lo olvide, busque el camino más lógico para resolverlo. ¿Cómo saber cuál es el camino más lógico? Pues sencillamente la solución más obvia es la que demarca cuál es el camino más lógico. Siempre busque la solución más obvia antes de comenzar a aplicar teorías y conceptos como los planteados en este libro. La aplicación de dichas teorías y conceptos debe ser un paso posterior. Inicialmente, lo que usted debe tener aproximadamente claro es un camino de solución y si se detiene a pensar en el problema no será raro que la mayoría de las veces tenga el camino más obvio de solución a dicho problema.

La lógica es ese conjunto de razonamientos que nos permiten solucionar fácilmente determinados problemas o lograr fácilmente determinados objetivos. Cada persona puede tener un enfoque diferente en cuanto a dicha solución y es muy importante, cuando se trabaja en equipo, escuchar cuál es la solución de los otros. Indiscutiblemente que para cada problema ha de existir una solución óptima, obvia y además muy sencilla. ¿Por qué razón cada persona puede llegar a encontrar una solución diferente a un determinado problema...? Son múltiples las explicaciones, pero se debe destacar dentro de ellas el entorno social, la preparación, el conocimiento, la convivencia y la utilización de conceptos nuevos acerca de la misma lógica, su mismo entorno personal y muchas más razones que pueden hacer que una persona vea la solución de un problema con una óptica diferente a como la podemos ver nosotros.

Lo que para una persona es absolutamente ilógico para otra es completamente lógico y es posible que ambas tengan la razón (o al menos crean tenerla) dadas sus propias condiciones personales e intelectuales. Sin embargo, podemos decir que si se mira un problema con una óptica aproximadamente neutral nos podremos aproximar a la solución más sencilla y obvia. ¿Cómo poder llegar a obtener una óptica aproximadamente neutral...? Considero que solo hay una forma de acercarse a este concepto y es estudiando conceptos que nos permitan lograr este objetivo. No es fácil determinar en qué momento hemos alcanzado una lógica aproximadamente normal, pero cuando el problema a solucionar puede ser resuelto con la utilización de la tecnología, entonces esta se convierte en el catalizador y en la regla de medida para saber hasta dónde nuestra solución es realmente óptima o no.

No olvide que la lógica computacional le permitirá buscar soluciones que puedan implementarse con tecnología. Por esta razón es que la buena utilización de la misma nos va a permitir saber hasta dónde nos hemos acercado a la solución óptima. Me arriesgaría a decir que un problema que se solucione con lógica computacional solo tendrá una y solo una solución óptima. Es decir, la solución más sencilla de implementar, la más obvia y la más entendible, a la luz de los conceptos de la misma lógica computacional.

12.2. Acerca de la metodología para solucionar un problema

Es muy posible que a esta altura usted se haya dado cuenta de algo en cuanto a la metodología para solucionar un problema y es que esta metodología no solo se refiere a problemas de la lógica computacional, sino que involucra cualquier problema. Note usted que los tres pasos explicados en este libro son, ni más ni menos, el secreto para que cualquier problema, sin importar de qué orden sea, tenga, mínimamente, una buena solución.

El objetivo es lo principal en todo lo que nos proponemos hacer en nuestras vidas; muchas empresas han fracasado solamente porque no tenían el objetivo suficientemente claro, muchos propósitos se han ido al piso solo porque el objetivo no era tan diáfano como se pensaba. En cuestión de lógica de programación, el objetivo es lo principal. Recuerde: "No dé ningún paso hacia delante si no tiene exageradamente claro el objetivo". Cuando usted tiene claro el objetivo, entonces obtiene dos ganancias: sabe claramente qué es lo que quiere hacer y sabe claramente hasta dónde debe llegar, o sea, en dónde debe parar. Pareciera que estas dos ganancias fueran lo mismo pero en realidad no lo son, aunque no puedo negar que sí están fuertemente conexas.

El objetivo marca el faro hacia el cual debemos enrutar todos nuestros esfuerzos y la clave para lograrlo es, precisamente, no perderlo de vista. Sin un objetivo suficientemente claro, no podemos empezar a desarrollar nada, pues es posible que estemos tomando un camino equivocado y no nos estemos dando cuenta. ¿Cómo saber que el objetivo está realmente claro? Pues con el solo hecho de que usted ha conceptualizado perfectamente lo que quiere lograr, el hecho de que usted pueda explicar con absoluta seguridad qué es lo que pretende obtener, ya con eso usted podrá sentir que el objetivo está completamente claro.

No dé ni un solo paso hasta tanto no tenga una absoluta certeza de lo que usted quiere lograr. No avance porque es posible que llegue a avanzar por el camino equivocado y termine logrando algo muy distinto a lo que había sido su objetivo original. Primero que nada, un objetivo claro y ahí sí podrá usted pensar en todos los demás elementos que conforman esta metodología.

El objetivo es como ese farol que ilumina una calle, inicialmente, oscura. Cuando usted tiene bien claro el objetivo a lograr, inmediatamente el camino para lograrlo automáticamente se aclara. Cuando a usted le dicen que debe ir hasta ese edificio azul que se ve al fondo, inmediatamente comienza a buscar todos los caminos para llegar a él y verá más de uno que le permita lograr llegar hasta el edificio en mención. Eso es lo mismo que pasa cuando usted tiene un objetivo claro. Ese camino para lograr un objetivo es lo que se llama algoritmo. El algoritmo es el conjunto de acciones (en el caso de un seudocódigo, es el conjunto de órdenes) que nos permiten lograr un objetivo. Cuando hemos depurado nuestra lógica frente a la solución de varios problemas, podemos decir que dicho algoritmo se aproxima altamente a la solución óptima.

Cuando se trate de algoritmos computacionales, deberán ir orientados a facilitar la codificación posterior precisamente para tener toda la certeza de que el computador va a ejecutar lo que nosotros hemos concebido como solución. Trate de utilizar, en sus algoritmos, órdenes que sean de fácil conversión a cualquier lenguaje de programación convencional. No olvide que lo más importante en el desarrollo de soluciones computacionales es la lógica que se haya utilizado para solucionar un determinado problema. Siempre tenga en cuenta que de una buena lógica utilizada en la construcción de los algoritmos dependen los buenos programas, es decir, las buenas aplicaciones.

Igualmente, de nada sirve la presunción de que nuestros algoritmos están bien, o sea, de que cumplen los objetivos planteados, si no nos lo demostramos a nosotros mismos haciendo unas buenas "pruebas de escritorio". Siempre debe tener en cuenta que un algoritmo al que no se le ha hecho una rigurosa prueba de escritorio no es más que un conjunto de órdenes posiblemente

incoherente. Lo único que le da verdadero respaldo a un algoritmo es la prueba de escritorio, que es la que nos permite saber si realmente el algoritmo está cumpliendo el objetivo o si no.

La prueba de escritorio nos brinda dos elementos que a mi juicio son lo más importante en la construcción de los algoritmos: en primera instancia, la prueba de escritorio nos va a permitir saber si realmente el algoritmo cumple el objetivo o no, o sea, en términos coloquiales, la prueba de escritorio nos permite saber si el algoritmo está bien. En segundo lugar, la prueba de escritorio nos va a permitir saber en dónde está el error o los errores de un determinado algoritmo para ser corregido, porque no solo se necesita saber si cumple o no con el objetivo un determinado algoritmo, sino además saber en dónde están las fallas y las razones por las cuales no está cumpliendo el objetivo.

Solo hasta cuando usted ha realizado una prueba de escritorio rigurosa y ha confrontado los resultados finales con el objetivo inicial, solo hasta ese momento, podrá realmente saber si su algoritmo estaba bien concebido o no. ¿Cómo hacemos entonces para realizar una prueba de escritorio rigurosa? Pues muy sencillo, escriba su algoritmo en formato de seudocódigo y coloque en el papel un rectángulo para lo que va a pasar en la pantalla y otro rectángulo para lo que va a pasar en la memoria. En el rectángulo de la memoria, coloque las variables que va a utilizar su algoritmo y comience a ejecutar línea a línea cada orden escrita. No suponga nada. No se salte ninguna orden. No presuma ningún resultado. Usted tendrá que ejecutar el algoritmo tal y como el computador lo va a realizar. Él no supone nada, no se salta ninguna línea ni presume ningún resultado.

Solo de esta forma podemos llegar a los mismos resultados que llegaría el computador. Esto quiere decir que, si al realizar una prueba de escritorio vemos que nuestro algoritmo arroja unos resultados que no coinciden con el objetivo, entonces, si el computador ejecutara dicho algoritmo (convertido en programa, por supuesto), también arrojaría resultados errados. Vaya utilizando las variables de memoria en la medida en que las va utilizando el algoritmo y cada vez que le coloque un nuevo valor a una variable, tache el valor que tenía almacenada antes dicha variable. Sin suponer nada, ejecute una a una cada instrucción del algoritmo y, cuando haya terminado, mire lo que queda en la pantalla o mejor dicho en el rectángulo de la pantalla.

Si lo que usted obtiene allí coincide con el objetivo inicial que buscaba lograr, entonces su algoritmo estará bien y podrá comenzar a codificarlo. Si usted nota que los resultados presentados en su "pantalla" no coinciden con los que inicialmente había planteado como objetivo, entonces esto querrá decir que su algoritmo está mal y que, por lo tanto, usted deberá corregir los errores que

la misma prueba de escritorio le indique en donde están. No olvide que en el instante de desarrollar una prueba de escritorio no se presume nada, no se supone nada ni se salta ninguna instrucción.

Podríamos concluir en este conjunto de reflexiones acerca de la metodología para solucionar un problema que de la realización eficiente de las pruebas de escritorio depende el tiempo que usted gane o pierda, de la buena construcción de algoritmos dependen las buenas aplicaciones de programación y de la claridad de los objetivos depende su prestigio como programador.

12.3. Acerca de las variables y los operadores

Cuando usted vaya a desarrollar un algoritmo, no pierda mucho tiempo pensando en las variables que va a necesitar. Comience por declarar las variables que saltan a la vista. Por ejemplo, si su algoritmo comienza diciendo *Leer un número entero y...*, ya con eso es suficiente para que usted sepa que va a necesitar mínimamente una variable de tipo entero, dado que al leer un número este debe quedar almacenado en algún lugar de memoria y ese lugar indiscutiblemente tiene que ser una variable. Igualmente, del mismo enunciado se puede concluir el tipo de dato que necesitamos.

Cuando vaya a iniciar un algoritmo, declare las variables que saltan a la vista, como cuando usted va a preparar un arroz con pollo salta a la vista que va a necesitar arroz y pollo, pero también sabemos que no solo esos dos elementos forman parte de un arroz con pollo, también necesitaremos sal, condimentos y otros productos que no saltan a la vista de manera tan obvia como el arroz y el pollo. Entonces, inicialmente, declare las variables que saltan a la vista en el mismo enunciado, no se detenga a pensar en las demás variables que va a necesitar, pues estas van surgiendo en la medida que las va necesitando en el desarrollo del mismo algoritmo.

En este libro se han explicado y se han utilizado tres tipos de datos: los datos de tipo entero, los datos de tipo carácter y los datos de tipo real. Cada uno tiene unas características técnicas que permiten manipular y manejar variables con ese tipo de datos. Sin embargo, tenga en cuenta que muchos lenguajes de programación cuentan con tipos de datos diferentes adicionales a los aquí explicados y en más de una oportunidad esos tipos de datos nos pueden facilitar de una manera mucho más simplificada el logro de un determinado objetivo. Por eso es muy bueno que usted tenga algún leve conocimiento de los tipos de datos que permite el lenguaje de programación con el cual va a codificar sus algoritmos.

No olvide tampoco que, siempre que usted almacene un valor determinado en una variable, el valor anterior se pierde, pues el nuevo valor reemplaza el

dato anterior. Por eso, cuando usted necesite almacenar el dato anterior de una variable, pues sencillamente tendrá que utilizar otra para que ese dato no se pierda. Igualmente, en algunos lenguajes es posible almacenar en una variable de un tipo de dato otro tipo de dato, esto quiere decir que en algunos lenguajes puede usted almacenar un entero en una variable de tipo carácter. Claro que recalco que esta es una particularidad de determinados lenguajes de programación. Por estas razones, es muy importante que usted conozca algunas características del lenguaje de programación con el cual va a codificar sus algoritmos, ya que podrá utilizar apropiadamente algunos recursos de dicho lenguaje en la construcción de sus algoritmos.

También es muy importante que tenga en cuenta que, por las características técnicas de los tipos de datos en los lenguajes de programación, los datos de tipo entero y real tienen unos topes de almacenamiento. Para ello, es muy bueno que usted conozca esos topes de almacenamiento y, si se da el caso del siguiente enunciado: *Leer un número entero y determinar si tiene más de 40 dígitos*, usted inmediatamente sepa que con los tipos de datos convencionales no se puede desarrollar.

En cuanto a la precedencia de ejecución de los operadores, tenga en cuenta que los que primero se resuelven son los paréntesis más internos, o sea, aquellos paréntesis que no tienen más paréntesis adentro. Esto es aceptado por todos los lenguajes de programación. También es importante que sepa que primero se desarrollan las potencias, luego las multiplicaciones y divisiones y luego las sumas y restas. Esta precedencia, tanto con paréntesis como con operadores, es lo que permite que una misma operación, por compleja que sea, tenga exactamente los mismos resultados en cualquier lenguaje de programación, dado que se rigen por las mismas reglas.

Cuando, al desarrollar una prueba de escritorio, tenga que resolver una expresión que incluya paréntesis y operadores y obviamente variables, no vaya a suponer nada, resuelva la operación tal y como lo haría el computador; de esta manera, y solo así, usted podrá notar si los resultados son los correctos o no, es decir, si la expresión sirve para lograr el objetivo o no.

12.4. Acerca de las estructuras básicas

En el tiempo que me he desempeñado como programador y como profesor de Lógica de Programación he notado que realmente la solución de ningún problema se sale de estas tres estructuras. Sé que es un poco arriesgado asegurarlo, pero también sé que este libro busca exponer, como su nombre lo

indica, la esencia de la lógica de programación y puedo garantizarle que el acertado manejo de estas tres estructuras es realmente la esencia en mención.

La estructura de secuencias no es más que la aceptación del principio mínimo de trabajo de los algoritmos, es decir, el hecho de que cada orden se ejecuta después de la anterior y antes de la posterior. Además, todas las órdenes se van a ejecutar secuencialmente, es decir, desde el principio hasta el fin. Podríamos decir que esta estructura es la que gobierna todo el concepto general de la programación y, a la vez, permite mantener el hilo lógico conceptual del diseño de algoritmos.

La segunda estructura está formada por las decisiones. Ya hemos definido que no es más que la escogencia de uno de entre dos ramales lógicos que dependen de una condición. Una decisión está formada por una formulación de una condición. Si esta es verdadera, entonces se ejecutarán las órdenes correspondientes; si no lo es, entonces se ejecutarán las otras órdenes. Es muy importante que usted sepa que las órdenes a ejecutar en caso de que la condición sea verdadera estarán escritas entre la formulación de la condición y la palabra clave *Sino* (en caso de que se considere la parte falsa de la condición) o la palabra *Fin_Si* (que finaliza la decisión).

Las órdenes a ejecutar en el caso de que la condición sea falsa estarán comprendidas entre la palabra clave *Sino* y el fin de la condición, o sea, el *Fin_Si* correspondiente. Para ello, tenga en cuenta que el planteamiento de una condición no necesariamente involucra un *Sino* correspondiente, ya que existen decisiones en donde, si su condición no se cumple, no nos interesa ejecutar alguna orden determinada.

Utilice apropiadamente los operadores relacionales y los operadores booleanos, para ello recuerde que un operador relacional permite obtener una respuesta de verdadero o falso y un operador booleano permite conectar expresiones relacionales. La notación tanto para operadores relacionales como para operadores booleanos no es estándar, razón por la cual usted deberá saber cómo se expresa cada una en el lenguaje de programación con el cual va a codificar sus algoritmos. Incluso puede utilizar esa notación en el desarrollo de sus algoritmos, ya que de una vez se va acostumbrando a la notación propia del lenguaje de programación que va a utilizar.

Algunos lenguajes también permiten realizar operaciones de asignación en medio de una expresión lógica y, por lo tanto, más de una vez van a facilitar el logro de un determinado objetivo. Como usted puede ver, aunque no se desmiente que el algoritmo es el soporte lógico para lograr un objetivo, tampoco se puede negar la inmensa importancia del conocimiento del lenguaje

de programación con el cual se van a convertir en programas lo que en el momento solo sean algoritmos expresados en seudocódigo. Normalmente, cada lenguaje cuenta con recursos y elementos que van a facilitar la construcción de unos algoritmos y, por lo tanto, el logro de unos objetivos, aunque dificulten un poco el logro de otros. Eso es lo que justifica precisamente la existencia de tantos lenguajes de programación en el medio informático.

Como se pudo ver, para la toma de decisiones también se cuenta con una estructura que nos permite escoger uno de entre varios ramales lógicos (me refiero a la estructura casos). Cada lenguaje de programación tiene una implementación propia de esta estructura de decisión, pero, en esencia, la lógica es la misma, o sea, que vamos a encontrar una equivalencia entre lo que nosotros programemos y la correspondiente codificación. Siempre tenga en cuenta que, para que el computador tome una decisión, esta tendrá que expresarse en términos de variables, constantes y operadores (tanto relacionales como booleanos).

Procure revisar bien sus algoritmos para que verifique si todas las decisiones son absolutamente necesarias, dado que una de las órdenes que más le toma tiempo al computador es tomar una decisión. Obviamente que hablo en términos del mismo computador, pues él trabaja en millonésimas de segundo y, por lo tanto, lo que para nosotros pudiera parecer muy veloz para él pudiera parecer muy lento. Si un computador demora un segundo en resolver una decisión, estará demorando demasiado, ya que en ese mismo lapso de tiempo podrá realizar millones de operaciones. Por este motivo es muy importante que usted tenga en cuenta que uno de los factores que permiten medir la eficiencia de un programa es la cantidad de decisiones que él mismo involucre.

Muchas veces colocamos decisiones en nuestros algoritmos, a diestra y siniestra, desconociendo que este factor va en detrimento de la misma solución lógica del algoritmo. Por lo tanto, siempre que usted haya terminado de desarrollar un algoritmo y lo haya probado y vea que cumple plenamente con el objetivo, tómese un tiempito para revisar cuáles decisiones pueden estar convirtiendo su solución en algoritmo ineficiente, lo cual no contradice el logro del objetivo, pues usted puede llegar hasta la plaza de su pueblo recorriendo todas las calles o recorriendo solo aquellas que lo llevan allá en línea recta.

Los ciclos son la tercera estructura y con ellos podemos, normalmente, lograr desarrollar algoritmos mucho más genéricos, es decir, sin las restricciones que sí se necesitan cuando no se tiene esta estructura a disposición. Los ciclos no son más que esquemas que nos permiten, en cualquier lenguaje de programación, repetir conjuntos de instrucciones una cantidad finita de veces de tal forma que, acorde con un manejo correcto de variables, se pueda lograr más

de un objetivo de manera absolutamente genérica y sin tener que estar atado a condiciones especiales.

Con los ciclos, todo lo que tenemos que hacer es tener mucho cuidado de que las condiciones sean coherentes con el cuerpo de los mismos de manera que se cumpla que el conjunto de instrucciones a repetir se itere una cantidad definida de pasos. Es muy útil recordar que la buena utilización de operadores relacionales y operadores booleanos es lo que nos permite lograr unos ciclos eficientes y ante todo muy funcionales.

12.5. Acerca de las técnicas de representación de algoritmos

Hemos revisado los conceptos acerca de las diferentes técnicas de representación de algoritmos y hemos mostrado cómo quedarían algunos de ellos representados con dichas técnicas. Las técnicas lo que van a permitir, cada una en su estilo, es colocar a su disposición herramientas para que la representación se haga mucho más sencilla porque si usted se detiene a pensar un momento no es tan fácil representar una idea por su misma naturaleza etérea e intangible. Por este motivo, la explicación de estas técnicas busca que usted pueda tener unos elementos conceptuales que le permitan de una manera sencilla y simplificada representar sus ideas.

La diagramación libre permitió durante mucho tiempo que la idea general de solución de un problema fuera expresada de manera gráfica. Así se pudo, poco a poco, ir estandarizando soluciones para determinados problemas comunes. Como se pudo ver, se explicó un cuadro comparativo de las diferentes técnicas y allí se expusieron las ventajas y desventajas de la diagramación libre. Considero que la diagramación como tal tiene la ventaja de mostrar lo que podríamos decir que es un "dibujo" que sirve como solución a un problema determinado, o sea, que sirve para lograr un objetivo planteado.

Aun a pesar de que realizar correcciones a un diagrama de flujo ya realizado es una tarea en más de una ocasión dispendiosa e incómoda, aun así, podría asegurarles que uno entiende cualquier cosa de una manera mucho más fácil si se la explican a través de un gráfico. Por esta razón, es mucho más fácil entender un mapa que entender una dirección. Este es el factor realmente rescatable de la técnica de diagramas de flujo; sin embargo, el gran problema que se presentaba al momento de realizarle una o más correcciones es lo que ha desplazado poco a poco la utilidad de esta técnica. Además que resultaba ser muy desventajoso, hablando en términos técnicos, el hecho de que el programador tenga

toda la libertad de hacer lo que le venga en gana en el papel para representar sus ideas, dado que si bien algunos programadores son muy ordenados otros no lo son, y si después le corresponde a alguien codificar lo que en su momento un programador altamente desordenado planteó a través de una maraña de líneas y símbolos que él erróneamente llamó diagrama de flujo, se verá en grandes aprietos para entenderlo.

Por este motivo, poco a poco se fue viendo la necesidad de buscar otra forma de expresión de los algoritmos. Fue allí en donde nació la idea de la diagramación rectangular estructurada que permitía que el programador expresara cualquier idea de una manera aproximadamente gráfica pero no con tanta libertad como se hacía en los diagramas de flujo. También presentaba la pequeña dificultad de realizar correcciones, pero se hacía de todas formas mucho más fácil la tarea de corregir que en los diagramas de flujo. No se puede negar que la representación gráfica como tal de un algoritmo a través de la diagramación rectangular estructurada no es tan entendible a primera vista como sí lo es con los diagramas de flujo, pero sí se debe aceptar que es mucho más técnica y por ello facilita mucho la posterior codificación, otro gran problema que colocaban los diagramas de flujo.

Bajo mi óptica diría que a alguien se le ocurrió quitarle las líneas a la diagramación rectangular estructurada y adicionarle unos nuevos estándares y prácticamente fue así como se llegó al concepto del seudocódigo, que pasó a ser la solución óptima dentro del objetivo de lograr una forma de representación de ideas que permitiera fácilmente una posterior codificación, que tuviera unas reglas establecidas por la misma lógica y que, además, fuera, desde todo punto de vista, entendible.

El seudocódigo abre un camino para una posterior codificación simplificada y además es muy entendible, teniendo en cuenta que se circunscribe a unas reglas lógicas que estandarizan el desarrollo del algoritmo como tal. Sin desconocer las inmensas bondades de las otras dos técnicas, considero que el seudocódigo permite lograr de una manera muy sencilla la representación de una determinada idea solución orientada al logro de un determinado objetivo.

Finalmente, es muy bueno que usted conozca las tres técnicas y de vez en cuando las practique, dado que existen muchos autores de manuales y de libros que utilizan indistintamente una u otra y entonces allí se hará muy importante que las conozcamos para que podamos entender lo que nos quieren representar técnicamente.

12.6. Acerca de la tecnología

Usted ha podido notar que en el mundo actual existe un despliegue de tecnología que no solo facilita sino que también condiciona la vida del ser humano. Los lenguajes de programación no son la excepción dentro de esa carrera comercial en la cual se encuentran inmersos todos los fabricantes de tecnología. Con gran frecuencia se encuentra uno con la decepción de que el *software* de desarrollo, o sea, los lenguajes de programación que existen en el mercado, son modificados y, teóricamente, mejorados con tanta vertiginosidad que uno escasamente alcanza a conocer algunas de sus bondades cuando ya las están cambiando por otras.

Por eso, no hay que preocuparse de que la tecnología vaya avanzando mucho más rápido de como avanzan nuestros conocimientos, pues lo que nos corresponde es tratar de estar a la vanguardia en dichos avances. ¿Cómo se hace para estar al día con todos los lenguajes de programación? Pues si usted en algún momento ha tenido este propósito es verdaderamente imposible. Permanentemente trate de mantenerse actualizado en cuanto a los avances del lenguaje de programación con el que usted usualmente trabaja. Si está vinculado a una empresa en donde se trabaja con un lenguaje determinado, pues entonces ha de procurar aprovechar las políticas de actualización de la empresa para tratar de mantenerse al día en las mejoras que dicho lenguaje vaya teniendo.

Sin embargo, tenga en cuenta que por más que se estudie es muy posible que usted siempre esté un poco rezagado de las actualizaciones de los lenguajes de programación, dado que esta tecnología nos llega luego de que ha pasado algún tiempo de prueba y otro de comercialización en los países industrializados. La ventaja de trabajar con un solo lenguaje de programación es que de una u otra forma usted estará, por la simple necesidad de uso del mismo, mucho más actualizado que lo normal (obviamente si la empresa tiene unas políticas de actualización tecnológica apropiadas). La desventaja es que usted se va a ir "encasillando" poco a poco en un solo lenguaje y, probablemente, ni siquiera llegue a conocer las ventajas y los avances de otros lenguajes de programación. Sin embargo, ha de entender y aceptar usted que no es fácil para una empresa estar cambiando su plataforma de desarrollo cada seis meses solo porque salió un lenguaje de programación aparentemente mejor.

Por esta razón, se hace muy importante ir poco a poco conociendo las actualizaciones en determinado lenguaje y, sin "casarse" de manera exclusiva con él, sí tener alguna preferencia por un lenguaje que por los demás. Considero que eso le permite a uno tener un poco más de destreza en la programación, así sea orientando sus algoritmos a la implementación en un lenguaje determinado.

Tenga en cuenta que, cada vez que usted se sienta a desarrollar un algoritmo, ha de buscar que dicho algoritmo sea lo más genérico posible no solo en su concepción lógica, sino en su posterior implementación, es decir, el algoritmo debe estar escrito en términos que se facilite la codificación en cualquier lenguaje. No puedo negarle que, en la medida en que uno se acostumbra a un determinado lenguaje de programación, en esa misma medida se va acostumbrando a la utilización de elementos y conceptos propios de dicho lenguaje en la construcción de los algoritmos. ¿Bueno o malo...? Eso no sabría respondérselo, pues encuentro tantas cosas buenas como malas, pero sé que es así como medianamente podemos llegar a funcionar teniendo en cuenta que definitivamente los lenguajes de programación avanzan mucho más rápido que nuestro aprendizaje o al menos así sucede en el mercado informático colombiano.

Considero que la mejor fuente para tener una percepción de los avances de los lenguajes de programación, así dicha percepción solo sean ideas de mejoramiento mas no instrucciones puntuales en sí, la constituyen las revistas técnicas en donde explican bondades y beneficios de las mejoras de los lenguajes de programación. Estas revistas lo que permiten es que tengamos una idea del camino que están siguiendo los avances en el desarrollo de las herramientas de programación. Ya si nos interesa conocer en detalle dichas mejoras, entonces tendríamos que trabajar con el lenguaje en mención y de esta forma podríamos conocer, de manera aplicativa, cuáles han sido dichos avances.

Utilice preferiblemente uno o dos lenguajes de programación si es independiente. Si la utilización de dicho lenguaje de programación está determinada por la empresa, entonces estudie todas sus bondades. Estoy seguro que más de una vez se encontrará con muchas instrucciones del mismo lenguaje desconocidas para usted y que le sacarán de más de un lío programático.

12.7. Acerca de las decisiones

En cuanto a las decisiones, realmente es poco lo que hay que acotar, dado que en la sección "Acerca de las estructuras básicas" se ha dicho la mayoría. Tal vez podría agregar que no se ate solo a la estructura *Si-Entonces-Sino*, pues también tenga en cuenta que la estructura *casos* le va a permitir simplificar más de un conjunto de decisiones y hacer que su algoritmo sea mucho más entendible y fácil de codificar.

Siempre evalúe bien las condiciones de las decisiones y realícele la prueba de escritorio a las decisiones tal y como se ha recomendado en todo este libro, es decir, sin suponer absolutamente nada y sin saltarse ningún paso, debido a que se puede encontrar, más de una vez, con que saltarse una instrucción

que involucre una decisión va a significar un cambio realmente profundo en el desarrollo del algoritmo y en su posterior ejecución. Revise que las decisiones que aparezcan en su algoritmo realmente sean necesarias y omita todas aquellas que, por razones estrictamente lógicas, estén sobrando.

Las decisiones son el único camino que tenemos en la programación para escoger uno de entre dos ramales lógicos, por eso es una de las herramientas que debemos utilizar de manera eficiente, pues su excesivo uso, tal como ya se expuso, representa ineficiencia en la construcción del algoritmo. Cuantas menos decisiones tenga un algoritmo más eficiente será la ejecución de su correspondiente programa.

12.8. Acerca de los ciclos

Como vimos, existen diferentes tipos de ciclos y la mayoría de ellos están contemplados en los lenguajes de programación. Normalmente me preguntan: ¿cuál de las estructuras cíclicas es la mejor? A lo cual yo respondo que realmente esta pregunta no se puede resolver en cuanto a una comparación y escogencia del mejor ciclo. Comencemos por decir que en la inmensa mayoría de las veces todos los ciclos son equivalentes. Como vimos, existían unos casos que no permitían una implementación de algunos ciclos, pero, en general, la mayoría de las veces son equivalentes.

No puedo negarle que uno poco a poco se va acostumbrando a la permanente utilización de uno o dos esquemas de ciclos, pero ello no es porque sean mejores o no, sencillamente es porque uno se acostumbra a ellos y no más. Cuando necesite un ciclo en un algoritmo, simplemente utilice aquel que usted vea más a la mano, es decir, aquel que vea más lógico, aquel con el cual usted vea que le resulta más fácil implementar el conjunto de instrucciones a repetir. No se ate obligatoriamente a un esquema de ciclos, pero no se extrañe si con el tiempo nota que utiliza uno o dos esquemas de ciclos muchos más que los otros.

La forma de escritura de algunos esquemas de ciclos en determinados lenguajes de programación puede llegar a ser un poco confusa y, en muchos casos, casi caprichosa. Tenemos que acostumbrarnos a ello si estamos trabajando con un lenguaje de estos o si nos toca trabajar con ellos. La sintaxis (que no es más que la forma como se escriben las órdenes en un lenguaje de programación) está diseñada por los creadores del lenguaje y, por lo tanto, no es cambiable a nuestro gusto, aunque puedo garantizarle que no se preocupe si usted se acostumbra a (o le toca) trabajar con un determinado lenguaje de programación; por más difícil que sea, usted en pocos días verá que su sintaxis es entendible y hasta llegará a defenderla, al fin y al cabo somos unos "animales de costumbre".

12.9. Acerca de los vectores

Los vectores establecen una herramienta de programación tan fuerte que más de una vez vamos a recurrir a ellos para que nuestros algoritmos sean realmente eficientes. No sobredimensione los vectores, esto quiere decir que no utilice vectores con una dimensión exagerada para no correr el riesgo de quedarse corto de memoria. Trate de ser lo más exacto en la dimensión de los vectores, pues tenga en cuenta que sobredimensionarlos implica estar reservando espacios de memoria que probablemente nunca se van a utilizar en la práctica.

Sé que usted estará pensando que no existe una técnica que permita darle a los vectores una dimensión dependiendo de las necesidades instantáneas de ejecución del programa. La verdad es que sí existe una técnica y no es exactamente con vectores, pero no son tema de este libro; sin embargo, en la medida en que usted utilice bien los vectores, empezará a notar como sus programas no solo pueden llegar a aprovechar eficientemente los recursos del computador, sino que logrará objetivos cada vez más genéricos y con menos esfuerzo tecnológico.

Siempre que utilice un vector tenga en cuenta la buena utilización de una variable que servirá como subíndice para referenciar cada uno de los elementos almacenados en el vector. No olvide que cuando se hace referencia a *V [i]* se está citando el valor que se encuentra almacenado en el vector *V* en la posición *i* y cuando se hace referencia a *i* sencillamente se está citando la posición de un determinado dato del vector.

Este ejemplo es válido si asumimos que estamos dentro de un algoritmo en donde estamos utilizando un vector que hemos llamado *V* y que utiliza un subíndice que hemos llamado *i*. De la buena utilización de subíndices en los vectores depende la buena aplicación de los mismos en los algoritmos.

12.10. Acerca de las matrices

Las matrices, al igual que los vectores, tienen una precaución en su utilización y es que no se vayan a sobredimensionar por el temor de quedarnos cortos de memoria. Siempre trate al máximo de que la dimensión de las matrices que utilice sea lo más justa posible, precisamente para que usted no esté reservando todo un espacio que posteriormente dentro del algoritmo no va a utilizar. Trate, al igual que con las variables y los vectores, de que los nombres de las matrices sean lo más mnemónicos posibles, pues de esto también dependerá que en cualquier momento usted vea un algoritmo y se ubique en cuanto a su concepción lógica. Tenga también en cuenta que la utilización de matrices

normalmente exigirá la utilización de dos subíndices, uno para las filas y otro para las columnas, y que no necesariamente estos dos subíndices tienen que llamarse *i* y *j*. Sencillamente necesitan ser dos variables enteras y nada más. Al igual que con los vectores, de la buena utilización de los subíndices de una matriz obtendremos la buena aplicación del concepto de matriz dentro de nuestros algoritmos. No utilice matrices en donde no las vaya a necesitar, siempre verifique si realmente el enunciado exige la presencia de matrices o no para que no tenga que estar reservando espacios de memoria que no va a usar.

Si los datos que usted lee deben ser almacenados, por alguna razón, en forma de filas y columnas, entonces usted necesitará una matriz. Si necesita que, además de la consideración anterior, sus datos permanezcan para una posterior utilización, entonces usted también necesitará en ese momento la utilización de una matriz. Siempre encontrará razones lógicas que van a validar o a rechazar la utilización de matrices dentro de la estructura lógica de un algoritmo.

12.11. Acerca de las funciones

Las funciones son, como yo les llamo en todos los cursos de programación que he dictado, la gran "*vedette*" de la programación, pues con las funciones, como pudimos ver en el contexto del libro, se solucionan los que considero que son los problemas más grandes que tiene la programación y que precisamente no son técnicos, sino de concepción humana. Cuando usted vaya a construir una función, tenga en cuenta que cuanto más genérica la construya más le va a servir para más adelante. ¿Cómo lograr funciones verdaderamente genéricas? Verificando qué parámetros necesita. Solo allí usted podrá encontrar el verdadero sentido de las funciones y su gran utilidad. En mi concepto, la programación conceptualmente no ha entregado una estructura más práctica y más simplificada que el concepto de función. Yo diría que hasta de allí en adelante la programación no ha evolucionado, pues toda la teoría actual de la programación se basa en esa unidad de trabajo, en esa "célula" funcional. Por tal motivo, yo le diría que, si al leer usted este libro y llegar a este punto encuentra que no solo ha entendido claramente qué es una variable, qué es una secuencia, qué es una decisión, qué es un ciclo, qué es un vector y qué es una matriz y, además, ha entendido claramente qué es una función y cómo se aplican en ella todos los conceptos anteriores, entonces este libro logró su objetivo inicial, que era colocar a su disposición todos los elementos básicos que constituyen la esencia de la lógica de programación.

Bibliografia

Articulos

A, Azad and D. Smith. Teaching an introductory programming language in a general education course. 2014, Journal of Information Technology Education: Innovations in Practice, Vol. 13, págs. 57-67. 2014

A.M. Vega and A. Espinel. Aspectos fundamentales para la enseñanza de la programación básica en ingeniería. Revista Avances en Sistemas e Informática, Vol. 7, pp. 7-13. Marzo 2010,

C. Romero and M. Rosero. Modelo de Enseñanza y su relación con los procesos metacognitivos en programación de sistemas. Asoc. Colombiana de Facultades de Ingeniería ACOFI. Bogotá. Revista Educ. en Ingeniería, pág. 3. Junio 2014

H. Paz-Penagos. ¿How to develop metacognition through problem solving in higher education? Revista de Ingeniería e Investigación, Vol. 31, págs. 75-80. 2009.

H. Paz-Penagos. Aprendizaje autónomo y estilo cognitivo: diseño didáctico, metodología y evaluación. Revista Educación en Ingenieía, Vol. 9, págs. 53-65. 2014

S. Fincher. ¿What are we doing when we teach programming? 29th ASEE/IEEE Frontiers in Education Conference. San Juan, Puerto Rico. 10-13 de Noviembre de 1999, Sesión 12a4

Libros

Attard et al. Student Centered Learning. An insight into theory and practice. Bucarest : Lifelong learning programme - European Community, 2010.

C. Boyer. Historia de la Matemática. Madrid (España) : Alianza Editorial, 2010.

D. Ausubel. Sicología Educativa: Un punto de vista cognoscitivo. Ciudad de México: Trillas, 1986

Deitel and Deitel. C++ Programming. New York : Prentice Hall, 2013.

G. Small. El cerebro digital. Madrid. Editorial Urano, 2011.

H. Schildt. C Programming. México : McGraw Hill, 2010.

H. Schildt. C++ Programming. Vancouver : McGraw Hill, 2010.

J. Bruner. Actos de significado: Mas allá de la revolución cognitiva. Madrid. Alianza Editorial, 2009

J.S. Bruner. Hacia un teoría de la instrucción. Ciudad de México : Hispanoamericana, 1969.

L. Vigotsky, Lev. El desarrollo de los procesos sicológicos superiores. México : Eitorial Grijalbo, 1981.

M. Costa and F. Costa. Metodología da pesquisa: preguntas e resposas. 1a Ed. Sao Paulo. DoAutor, 2013.

M. Felleisen et al. How to design programs. 2a Ed. Boston : MIT Press, 2013.

O.I. Trejos Buriticá. Algoritmos Problemas Básicos. Pereira (Colombia) : Papiro, 2008.

O.I. Trejos. Aprendizaje en Ingeniería: un problema de comunicación. Pereira (Colombia) : Tesis Doctoral - Universidad Tecnológica de Pereira, 2012.

O.I. Trejos. Fundamentos de Programación. Pereira : Papiro, 2006.

O.I. Trejos. Significado y Competencias. Pereira. Papiro, 2013

O'Farrel. Enhancing student learning through assesament. New York : Electronic Press, 2012.

P. Van Roy. Concepts, Techniques and Models of Computer Programming. Estocolmo. Université Catholique de Louvain, 2008.

P. Van Roy. Techniques and methods in programming computer. Louvaine : University Press, 2008.

Stewart. Historia de las Matemáticas en los últimos 10000 años. Barcelona (España) : Editorial Crítica, 2012.

Muchas personas confunden la Programación con la Lógica de Programación. La primera involucra el conocimiento de técnicas e instrucciones de un determinado lenguaje a través de los cuales se hace sencillo lograr que el computador obtenga unos resultados mucho más rápido que nosotros. La segunda involucra, de una manera técnica y organizada, los conceptos que nos permiten diseñar, en términos generales, la solución a problemas que pueden llegar a ser implementados a través de un computador.

Luego de muchos años de estudio de estos factores, pude condensar en este libro los que, considero, son los conceptos fundamentales para aprender realmente a programar, o sea, lo que he llamado la esencia de la lógica de programación, pues busco que usted conozca estos elementos conceptuales y, luego de dominarlos, se enfrente sin ningún problema no solo a cualquier objetivo que pueda ser alcanzable a través de computadores, sino además a cualquier lenguaje de programación.

- Conozca los conceptos realmente básicos para aprender a programar
- Resuelva problemas computables

Omar Iván **Trejos** Buriticá

Ingeniero de Sistemas y Computación, Doctorado en Ciencias de la Educación, Maestría en Comunicación Educativa y Especialización en Instrumentación Física. Docente de la Universidad Tecnológica de Pereira

ediciones de la U

Contenidos libres en:
www.edicionesdelau.com

ISBN: 978-958-762-720-6