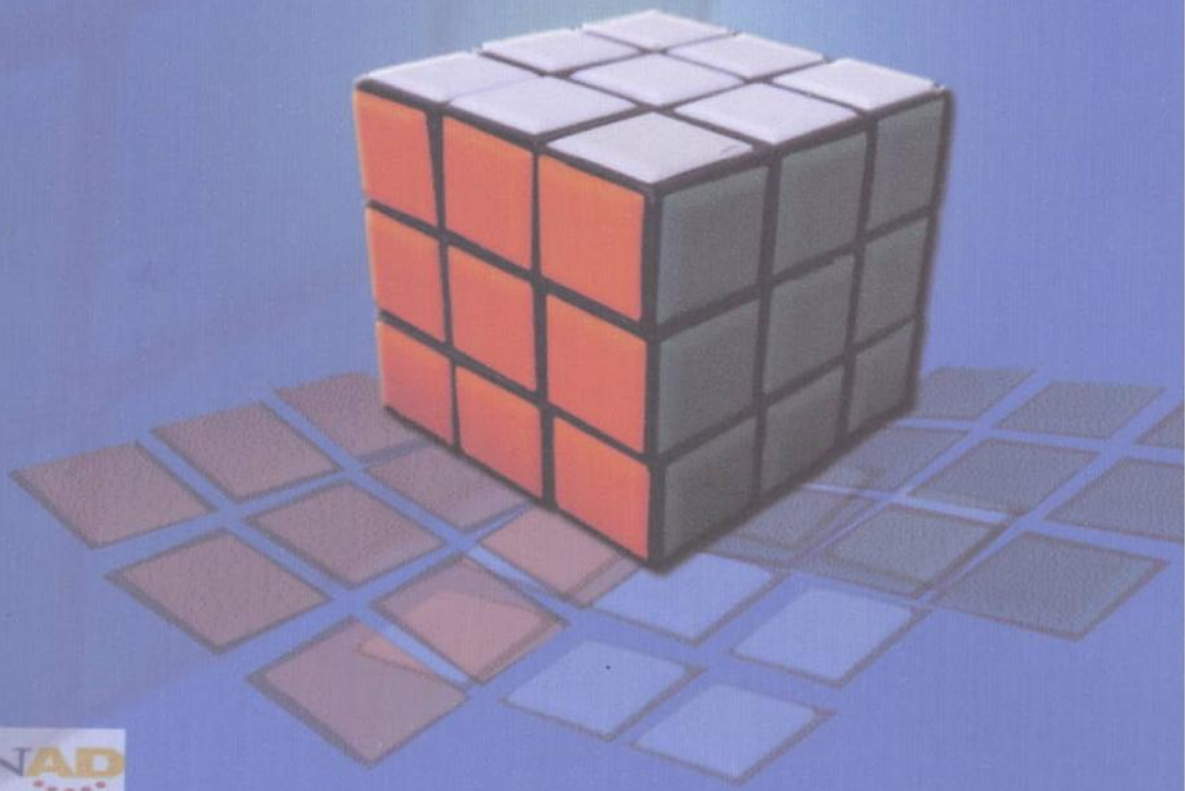
Diseño de Algoritmos

Anívar Chaves Torres



Diseño de Algoritmos

Anívar Chaves Torres
Escuela de Ciencias Básicas,
Tecnología e Ingeniería
Universidad Nacional Abierta y a Distancia
UNAD

Datos de catalogación bibliográfica

CHAVES TORRES, Anívar Néstor. Diseño de algoritmos. Ciudad: editorial, 2012.

ISBN:

Formato: 16,5 x 23 Páginas: 419

Edición y diagramación: Anívar Chaves Torres

Diseño de portada: Anívar Chaves Torres

ISBN:

Impreso por _____

A la sagrada familia,
por su comprensión,
su apoyo y
su confianza.

Agradecimientos

A Jackeline Riascos, Mirian Benavides y Alejandra Zuleta, quienes, pese a sus múltiples ocupaciones, asumieron la tarea de leer el manuscrito y con sus acertadas observaciones hicieron posible corregir muchos errores y mejorar el libro. De igual manera, a William Recalde, quien como estudiante y cómo colega siempre ha estado disponible para revisar mis escritos y ayudarme a mejorarlos.

Al ingeniero Gustavo Velásquez Quintana, decano de la Escuela de Ciencias Básicas, Tecnología e Ingeniería de la Universidad Nacional Abierta y a Distancia - UNAD, por apoyar la publicación de este trabajo.

CONTENIDO

PROLOGO

Este es un libro concebido y elaborado por un profesor de programación, quien a la vez se reconoce como un estudiante permanente del tema; y por tanto, conoce muy bien las dificultades que experimentan los estudiantes para aprender fundamentos de programación y diseño de algoritmos, de igual manera que las necesidades de los profesores de contar material de referencia que incluya conceptos, ejemplos y ejercicios.

Aunque los temas que se desarrollan son comunes en los libros de fundamentos de programación, aquí se presentan con un enfoque didáctico, con un lenguaje sencillo y con un nivel de detalle que cualquier persona lo pueda comprender, pues éste no pretende ser únicamente un documento de consulta, sino un material didáctico para el aprendizaje autónomo.

Como estrategia para facilitar el aprendizaje del diseño de algoritmos se propone proceder de forma inductiva, pues un algoritmo es una solución general para problemas de un mismo tipo y sus pasos se identifican en la medida que se soluciona varios casos particulares del problema. En los ejemplos que se presenta en los capítulos se aplicará esta metodología, primero se propone valores hipotéticos para los datos del problema y se realizan los cálculos para llegar a una solución, luego se pasa a definir variables y establecer expresiones que solucionen el problema para cualquier conjunto de valores.

El libro está organizado en nueve capítulos. El primero es una introducción al tema en el que se estudia las generalidades del computador, tanto en lo físico como en lo lógico, y también el proceso

de construcción de software. Éste tiene como propósito proporcionar al estudiante una perspectiva para el estudio de los siguientes siete capítulos.

En el capítulo dos se desarrollan algunos conceptos fundamentales en programación, como son: tipos de datos, variables y constantes, operadores y expresiones. Es necesario que el estudiante tenga claridad sobre éstos conceptos antes de adentrarse en el diseño de algoritmos, pues serán aplicados en cada uno de los temas siguientes.

El capítulo tres aborda directamente el tema de los algoritmos: conceptos, características, notaciones, estrategia para el diseño y verificación. Éste tiene como propósito ofrecer soporte conceptual, pues el estudiante y el docente deben estar de acuerdo sobre lo que es y lo que no es un algoritmo, sobre cómo debe representarse para evitar ambigüedad en la solución de un problema y cómo verificar la eficacia de la misma.

En el cuarto se abordan los tres tipos de estructuras de programación: secuenciales, selectivas e iterativas. En este capítulo comienza realmente el diseño de algoritmo, pues los anteriores desarrollan el marco de referencia para la comprensión y aplicación de las estructuras. Se muestra cómo llevar a cabo la lectura de datos, el procesamiento y la presentación de los resultados, así como la toma de decisiones y la ejecución repetida de ciertas operaciones o procesos.

En el quinto se estudia uno de los temas más relevantes de la programación: el manejo y la organización de datos en memoria principal a través de arreglos. Estas estructuras de almacenamiento son fáciles de utilizar y muy útiles para el diseño de algoritmos que se aplican sobre conjuntos de datos. En este capítulo se explican detalladamente todas las operaciones que se llevan a cabo en el trabajo con vectores, excepto la búsqueda y ordenación que desarrollan en un capítulo aparte.

En el sexto se explica la estrategia *dividir para vencer* aplicada al diseño de algoritmos. Ésta estrategia consiste en descomponer un problema en subproblemas más fáciles de resolver, para los que se diseña funciones o procedimientos que se invocan desde un algoritmo principal.

En el séptimo se estudia la búsqueda y el ordenamiento, temas de vital importancia cuando se trabaja con grandes volúmenes de datos; se explican detalladamente algoritmos como: búsqueda lineal y binaria, ordenamiento por intercambio, inserción, selección, burbujeo y partición.

En el octavo se trata la recursividad o recursión. Este tema se incluye considerando que muchos problemas de programación son más fáciles de resolver utilizando recursividad y muchos de los algoritmos que los estudiantes deben aprender a utilizar son recursivos, como es el caso del algoritmo de ordenamiento rápido y los algoritmos para manejo estructuras de datos dinámicas. Desde esta perspectiva es importante que los estudiantes conozcan la recursividad desde los primeros cursos.

En el último capítulo, el noveno, se presenta en forma algorítmica la solución al cubo de Rubik o cubo mágico. Este juego se incluye con base en dos hipótesis: primera, es muy difícil armar el cubo si no se conoce las secuencias de movimientos, pues cada vez que se intenta poner en orden una pieza se desordena alguna otra; segunda, es más fácil comprender y asimilar un tema o un concepto cuando se encuentra algo divertido en que aplicarlo. En el algoritmo para armar el cubo se ponen en práctica los temas presentados entre el segundo y octavo capítulo. Adicionalmente, mientras se intenta armar el cubo se ejercitan dos cualidades indispensables para aprender cualquier materia, en especial algoritmos, como son: voluntad y perseverancia.

1. INTRODUCCIÓN

"Sal de la infancia,
amigo y despierta".
Rousseau

Escribir un programa de computador es una actividad compleja que exige conocimientos, habilidades, creatividad y dominio de las técnicas y herramientas de programación. Antes de escribir un programa es necesario pensar detenidamente sobre la naturaleza y las características del problema que se pretende resolver, encontrar la forma de realizar cada operación requerida para solucionar el problema y organizar las actividades a realizar para obtener los resultados esperados. Sólo cuando se cuenta con el diseño de la solución se hace uso de un computador y un lenguaje de programación para escribir el código del programa.

Este libro está orientado a favorecer el aprendizaje de los conceptos y la técnica para analizar los problemas y diseñar soluciones como paso previo a la escritura de programas. Aunque no se aborda la programación bajo ningún lenguaje, se presenta una descripción general de la arquitectura del computador y del proceso de construcción de programas con el fin de proporcionarle al lector una perspectiva que facilite la comprensión de los temas desarrollados en los capítulos siguientes.

1.1 EL COMPUTADOR

El computador es una máquina electrónica diseñada para la administración y procesamiento de datos, capaz de desarrollar complejas operaciones a gran velocidad, siguiendo un conjunto de instrucciones previamente establecido.

Deitel y Deitel (2004: 4) lo definen como "dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades hasta miles de millones de veces más altas que las alcanzables por los seres humanos" el cual procesa datos bajo el control de series de instrucciones llamadas programas de computador.

Lopezcano (1998: 83) lo define como una "máquina capaz de seguir instrucciones para alterar información de una manera deseable" y para realizar algunas tareas sin la intervención del usuario.

Con base en Martínez y Olvera (2000) se considera que un computador se caracteriza por cumplir las siguientes funciones:

- Realizar operaciones aritméticas y lógicas a gran velocidad y de manera confiable.
- Efectuar comparaciones de datos y ejecutar diferentes acciones dependiendo del resultado de la comparación, con gran rapidez.
- Almacenar en memoria tanto datos como instrucciones y procesar los datos siguiendo las instrucciones
- Ejecutar secuencialmente un conjunto de instrucciones almacenadas en memoria (programa) para llevar a cabo el procesamiento de los datos.
- Transferir los resultados de las operaciones a los dispositivos de salida como la pantalla y la impresora o hacia dispositivos de almacenamiento secundario, para que el usuario pueda utilizarlos.
- Recibir programas y datos desde dispositivos periféricos como el teclado, el ratón o algunos sensores.

Es una herramienta de propósito general que puede ser utilizada en diferentes áreas, como: sistemas administrativos, control de procesos, control de dispositivos específicos, diseño asistido por computador, simulación, cálculos científicos, comunicaciones y sistemas de seguridad (Martínez y Olvera, 2000). Es capaz de manejar datos de diferentes formatos, a saber: números, texto, imágenes, sonido y video.

Inicialmente los computadores se utilizaban en actividades que requerían cálculos complejos o manejo de grandes volúmenes de información, como oficinas gubernamentales y universidades. Con el correr de los años se generalizó su utilización en la medida en que aumentó el número de computadores fabricados y bajó su costo. Cada vez es mayor el uso de esta herramienta, por cuanto nadie desea renunciar a sus beneficios en términos de comodidad, rapidez y precisión en la realización de cualquier tipo de tareas; además, cada vez hay en el mercado mayor número de accesorios tanto físicos como lógicos que se aplican a una gama más amplia de actividades.

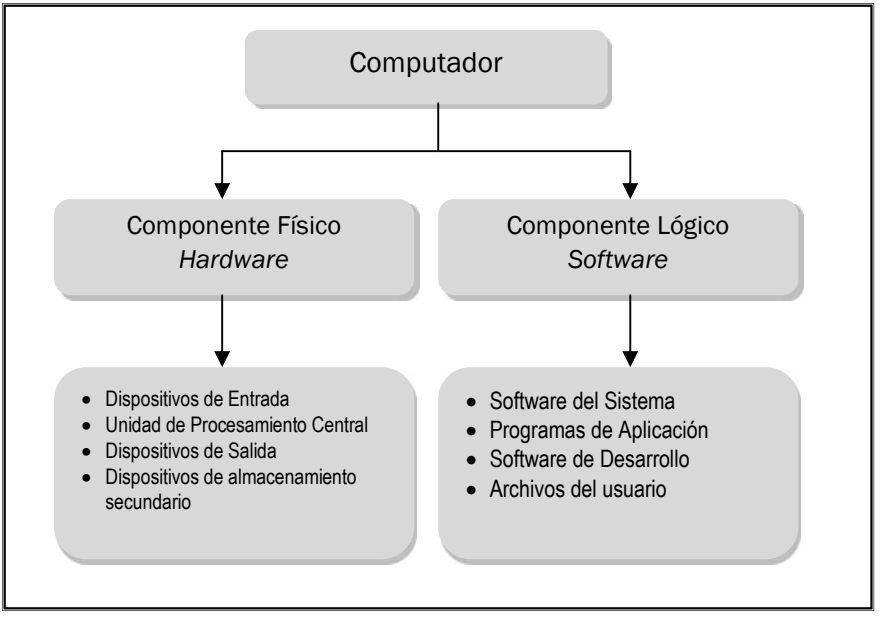
Desde el enfoque de sistemas, el computador es más que un conjunto de dispositivos, pues, lo que hace de éste una herramienta de gran utilidad es la relación y la interacción que se presenta entre sus componentes, regulada por un conjunto de instrucciones y datos que constituyen sus directrices.

Visto de forma superficial, un sistema requiere una entrada o materia prima, sobre la cual ejecutar un proceso o transformación, para generar un producto diferente que constituye la salida. El computador puede ser considerado sistema en cuanto que los datos ingresados sufren algún tipo de transformación (procesamiento) para generar nuevos datos que ofrecen mejor utilidad al usuario, donde dicha utilidad constituye el objetivo del sistema.

A diferencia de cualquier otro tipo de sistema, el computador cuenta con capacidad de almacenamiento, lo que le permite realizar múltiples procesos sobre un mismo conjunto de datos y, también, ofrecer los resultados del proceso en una diversidad de formatos y tantas veces como sea necesario.

El computador cuenta con dos tipos de componentes: unos de tipo físico como las tarjetas, los discos, los circuitos integrados; y otros de tipo lógico: los programas y los datos (Figura 1). La parte física se denomina *hardware*, mientras que la parte lógica se denomina *software*.

Figura 1. Composición del computador

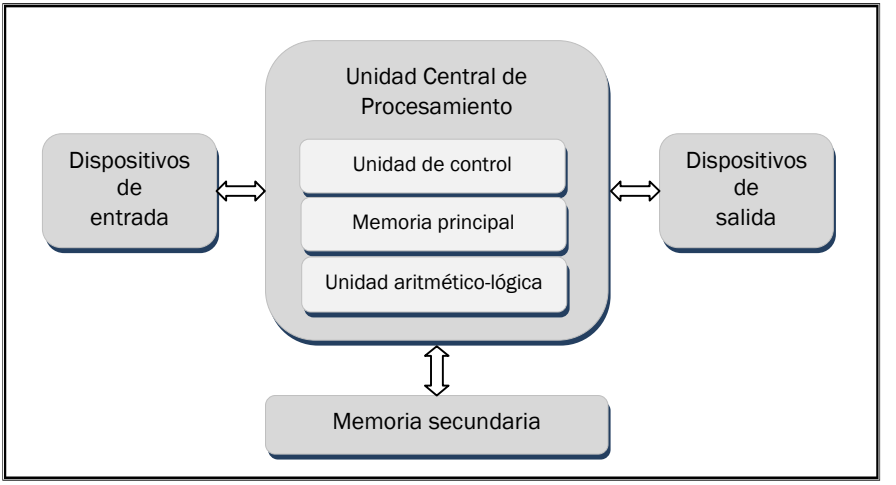


1.1.1 Componente Físico

Este componente, comúnmente conocido por su nombre en inglés: *hardware*, está conformado por todos los dispositivos y accesorios tangibles que pueden ser apreciados a simple vista, como: tarjetas, circuitos integrados, unidad de disco, unidad de CD-ROM, pantalla, impresora, teclado, ratón y parlantes (Plasencia, 2008).

Los elementos del hardware se clasifican según la función que cumplen, esta puede ser: entrada de datos, procesamiento, salida o almacenamiento (Figura 2).

Figura 2. Organización física del computador



Dispositivos de entrada. Los dispositivos de entrada reciben las instrucciones y datos ingresados por el usuario y los transmiten a la memoria principal desde donde serán tomados por el procesador.

Los dispositivos de entrada más utilizados son: el teclado y el ratón, seguidos por otros menos comunes como el escáner, micrófono, cámara, entre otros.

La información ingresada a través de estos medios es transformada en señales eléctricas que se almacén en la memoria principal, donde permanece disponible para ser procesada o almacenada en medios de almacenamiento permanente.

Unidad de Procesamiento Central. Comúnmente se la conoce como CPU, sigla de su nombre en inglés (*Central Processing Unit* - unidad de procesamiento central). Esta es la parte más importante

del computador, por cuanto es la encargada de realizar los cálculos y comparaciones. En palabras de Becerra (1998: 1), "el procesador es el que realiza la secuencia de operaciones especificadas por el programa".

Este componente efectúa la búsqueda de las instrucciones de control almacenadas en la memoria, las decodifica, interpreta y ejecuta; manipula el almacenamiento provisional y la recuperación de los datos, al tiempo que regula el intercambio de información con el exterior a través de los puertos de entrada y salida. (Lopezcano, 1998).

La unidad de procesamiento central cuenta con tres partes importantes que son: unidad de control, unidad aritmético-lógica y registros de memoria. Además de los registros (pequeños espacios de memoria) el procesador requiere interactuar constantemente con la memoria principal.

La unidad de control se encarga de administrar todo el trabajo realizado por el procesador; en otras palabras, podría decirse que controla todo el funcionamiento del computador. Entre las funciones de esta unidad se encuentran:

- Leer de la memoria las instrucciones del programa
- Interpretar cada instrucción leída
- Leer desde la memoria los datos referenciados por cada instrucción
- Ejecutar cada instrucción
- Almacenar el resultado de cada instrucción

La unidad aritmético-lógica realiza una serie de operaciones aritméticas y lógicas sobre uno o dos operandos. Los datos, sobre los que opera esta unidad, están almacenados en un conjunto de registros o bien provienen directamente de la memoria principal. Los resultados de cada operación también se almacenan en registros o en memoria principal (Carretero *et al*, 2001).

Los registros son pequeños espacios de memoria para almacenar los datos que están siendo procesados. La característica principal de los registros es la velocidad con que puede acceder el procesador.

Las operaciones realizadas por la ALU y los datos sobre los cuales actúan son supervisados por la unidad de control.

Memoria principal. Físicamente, es un conjunto de circuitos integrados pegados a una pequeña tarjeta que se coloca en la ranura de la tarjeta principal del computador. Funcionalmente, la memoria es el espacio donde el procesador guarda las instrucciones del programa a ejecutar y los datos que serán procesados.

La memoria cumple un papel muy importante junto al procesador, ya que permite minimizar el tiempo requerido para cualquier tarea; por lo tanto, entre más memoria tenga el equipo mejor será su rendimiento.

Cuando se utiliza el término memoria se hace referencia a la memoria RAM o memoria de acceso aleatorio; sin embargo, existen otros tipos de memoria, como la memoria caché y la memoria virtual, entre otras clasificaciones.

RAM significa memoria de acceso aleatorio, del inglés *Random Access Memory.* Esta memoria tiene como ventaja la velocidad de acceso y como desventaja que los datos se guardan sólo temporalmente; en el momento en que, por alguna razón, se corte la alimentación eléctrica, los datos de la memoria desaparecerán. Por la velocidad de acceso, que es mucho más alta que la velocidad de acceso a disco, entre más datos y programas se puedan cargar en la memoria mucho más rápido será el funcionamiento del computador.

Los diferentes componentes de la unidad computacional se comunican mediante *bits*. Un *bit* es un dígito del sistema binario que puede tomar uno de dos valores: 0 o 1. Los *bits* constituyen la mínima unidad de información para el computador; sin embargo, el término más común para referirse a la capacidad de memoria de un equipo es el *Byte* que es la reunión de 8 *bits* y permite representar hasta 256

caracteres, ya sean letras, dígitos, símbolos o los códigos conocidos como caracteres no imprimibles, cualquier caracter del el sistema ASCII (*American Standard Code for Information Interchange* - código estándar americano para intercambio de información).

Dado que un *byte* (un caracter) es una unidad muy pequeña para expresar la cantidad de información que se maneja en un computador o en cualquiera de sus dispositivos de almacenamiento, se utilizan múltiplos del mismo, como se aprecia en el cuadro 1.

Cuadro 1. Magnitudes de almacenamiento

Unidad	Nombre corto	Capacidad almacenamiento
Bit	B	0 o 1
Byte	B	8 bits
Kilobyte	Kb	1024 Bytes
Megabyte	Mb	1024 Kb
Gigabyte	Gb	1024 Mb
Terabyte	Tb	1024 Gb

Dispositivos de almacenamiento o memoria secundaria. También se la conoce como memoria auxiliar. Ésta es la encargada de brindar seguridad a la información almacenada, por cuanto guarda los datos de manera permanente e independiente de que el computador esté en funcionamiento, a diferencia de la memoria interna que sólo mantiene la información mientras el equipo esté encendido. Los dispositivos de almacenamiento secundario son discos, discos compactos (CD), discos de video digital (DVD) y chips de memoria.

Dispositivos de salida. Permiten presentar los resultados del procesamiento de datos. Son el medio por el cual el computador presenta información al usuario. Los más comunes son la pantalla, la impresora y los parlantes.

Pantalla o monitor: exhibe las imágenes que elabora de acuerdo con el programa o proceso que se esté ejecutando. Pueden ser videos, gráficos, fotografías o texto. Es la salida por defecto, donde se presentan los mensajes generados por el computador, como: solicitud de datos, resultados de procesos y mensajes de error.

Impresora: este dispositivo fija sobre el papel la información que se le envía desde un programa. La impresión puede ser en negro o en colores según el tipo de impresora que se tenga.

Parlantes: estos dispositivos se encargan de amplificar la señal de audio generada por el computador. Actualmente son de uso común debido a la capacidad de procesamiento multimedia de los equipos de cómputo.

1.1.2 Componente Lógico

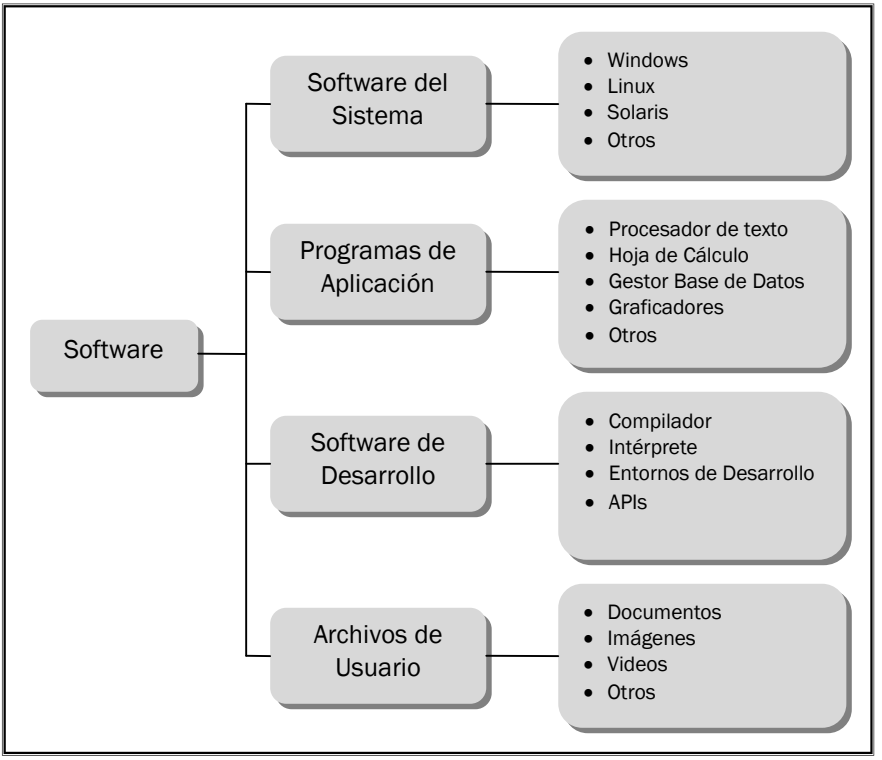
Este componente, también conocido comúnmente como *software*, está conformado por toda la información, ya sean instrucciones o datos, que hace que el computador funcione. Sin el concurso del *software* el *hardware* no realiza ninguna función.

El software está clasificado en cuatro grupos, según la tarea que realiza: software del sistema, programas de aplicación, software de desarrollo y archivos de usuario, como se muestra en la figura 3.

Software del Sistema. También conocido como sistema operativo, es un conjunto de programas indispensables para que el computador funcione. Estos se encargan de administrar todos los recursos de la unidad computacional y facilitan la comunicación con el usuario.

Carretero et al (2001) sugieren que se imagine el computador desnudo e incapaz de llevar a cabo tarea alguna. Aunque el computador esté equipado con el mejor *hardware* disponible, sino cuenta con las instrucciones de un programa en la memoria, que le indiquen qué hacer, no hace nada.

Figura 3. Clasificación del software



El objetivo del sistema operativo es simplificar el uso del computador y la administración de sus recursos, tanto físicos como lógicos, de forma segura y eficiente. Entre sus funciones se destacan tres:

- Gestión de los recursos de la unidad computacional
- Ejecutar servicios solicitados por los programas
- Ejecutar los comandos invocados por el usuario

Para cumplir estas funciones, el sistema operativo cuenta con programas especializados para diversas tareas, como son: la puesta

en marcha del equipo, la interpretación de comandos, el manejo de entrada y salida de información a través de los periféricos, acceso a discos, procesamiento de interrupciones, administración de memoria y procesador.

Algunos sistemas operativos conocidos son: Windows, con sus diferentes versiones; Linux, y sus muchas distribuciones; Netware; Unix, Solaris, entre otros.

Programas de aplicación. Conjunto de programas diferentes al software del sistema. Estos se encargan de manipular la información que el usuario necesita procesar; desarrollan una tarea específica y su finalidad es permitirle al interesado realizar su trabajo con facilidad, rapidez, agilidad y precisión. En esta categoría se tiene varios grupos, como son: procesadores de texto, hoja de cálculo, graficadores, bases de datos, agendas, programas de contabilidad, aplicaciones matemáticas, editores y reproductores de audio, editores y reproductores de video, entre otros. Algunos ejemplos son: Word, Excel, Access, Corel Draw, programas para estadística como SPSS y contabilidad como GC 1.

Software de desarrollo. Esta categoría está conformada por el extenso conjunto de herramientas para la construcción de software, entre ellos se destacan los compiladores, los intérpretes, los entornos integrados de desarrollo, los marcos de trabajo y las Interfaces para programación de aplicaciones (API).

La mayoría de estas herramientas están diseñadas para trabajar con un determinado lenguaje de programación; no obstante, existen algunas aplicaciones que permiten utilizar código de más de un lenguaje, ejemplo de ellos son: la plataforma Java, con su paquete JNI y la plataforma .Net.

Archivos del usuario. Esta categoría está conformada por todos los archivos que el usuario crea utilizando los programas de aplicación; por lo general, no son programas que se puedan ejecutar, sino archivos que contienen los datos introducidos o los resultados del procesamiento de éstos. Ejemplos de archivos de usuario son las

imágenes como fotografías o dibujos, textos como cartas o trabajos, archivos de base de datos, archivos de hoja de cálculo como una nómina o un inventario.

1.2 CONSTRUCCION DE PROGRAMAS

Hace algunas décadas se consideraba que el proceso de preparar programas para computadores era especialmente atractivo porque era gratificante no solo desde lo económico y lo científico, sino también, una experiencia estética como la composición de poesía o música (Knuth, 1969). La programación era considerada un arte, llevarla a cabo era cuestión de talento más que de conocimiento. Hoy en día se cuenta con metodologías y técnicas para hacerlo, de manera que cualquiera persona puede aprender a programar, si se lo propone.

Aunque el tema central de este libro es el diseño de algoritmos y no la programación, se considera importante que el lector tenga un conocimiento general sobre todo el proceso de construcción de un programa, de manera que pueda comprender el papel que juega el algoritmo en la metodología de la programación. En especial, porque los programas se construyen para solucionar un problema y los algoritmos especifican la solución.

En este sentido, cabe anotar que, construir un programa es una tarea compleja que exige no solo conocimiento, sino también creatividad y el ejercicio de ciertas habilidades por parte del desarrollador. Los programas pueden ser tan grandes y complejos como el problema que pretenden solucionar y aunque puede haber cierta similitud entre ellos, así como cada problema tienen sus particularidades, los programas tienen las suyas.

A propósito de la solución de problemas Galve *et al* (1993) mencionan que no existe un método universal que permita resolver cualquier problema, ya que se trata de un proceso creativo donde el conocimiento, la habilidad y la experiencia tienen un papel

importante, cuando se trata de problemas complejos lo importante es proceder de manera sistemática.

La disciplina que se ocupa de estudiar todos los aspectos relacionados con la producción de software, desde la identificación de requisitos hasta su mantenimiento después de la utilización, es la Ingeniería del Software (Sommerville, 2005). Desde esta disciplina se han propuesto diversas metodologías para la construcción de software, algunas de ellas son: proceso unificado, Programación extrema, Desarrollo rápido de aplicaciones, Desarrollo en espiral.

Cada una propone diferentes momentos y actividades para la construcción del software, no obstante hay algunas fases que son comunes a todas, aunque se les dé diferentes nombres. Por mencionar solo tres ejemplos, Joyanes (2000) propone ocho fases: análisis del problema, diseño del algoritmo, codificación, compilación y ejecución, prueba, depuración, mantenimiento y documentación; Bruegge y Dutoit (2002), por su parte, plantean: identificación de requisitos, elaboración de un modelo de análisis, diseño general, diseño detallado, implementación y prueba; y Sommerville (2005) las resume en cuatro: especificación, desarrollo, validación y evolución del software.

Considerando las propuestas anteriores, se organiza el proceso de construcción de programas en seis actividades, como se detallan a continuación.

1.2.1 Análisis del problema

Dado que los problemas no siempre tienen una especificación simple y sencilla, la mitad del trabajo es saber qué problema se va a resolver (Aho et al, 1988).

El primer paso es asegurarse de que se entiende el problema que se pretende solucionar y la magnitud del mismo. Muchas veces, el enunciado del ejercicio o la descripción de necesidades expuesta por un cliente no son lo suficientemente claros como para determinar los

requerimientos que debe cumplir el programa; por ello, el programador debe detenerse y preguntarse si comprende bien el problema que debe solucionar.

En el mundo real los problemas no están aislados, sino interrelacionados; por ello, para intentar solucionar uno en particular es necesario plantearlo con claridad y precisión, estableciendo los alcances y las restricciones. Es importante que se conozca lo que se desea que realice el programa de computador; es decir, identificar el problema y el propósito de la solución.

Al respecto, Joyanes (1996) menciona que la finalidad del análisis es que el programador comprenda el problema que intenta solucionar y pueda especificar con detalle las entradas y las salidas que tendrá el programa, para ello recomienda formularse dos preguntas:

¿Qué información debe proporcionar el programa?

¿Qué información se necesita para solucionar el problema?

Establecer cada una de las funciones que el programa debe realizar y las características que debe tener se conoce como identificación de requisitos o requerimientos. Estos son como las cláusulas de un contrato, definen con cierta precisión el qué hacer, aunque no el cómo. En programas de alta complejidad el listado de requisitos puede ser muy extenso, en pequeños programas, como los que se realizan en los primeros cursos de programación suelen limitarse a unos cuántos; no obstante, establecer los requisitos es una tarea fundamental del análisis.

Una vez identificados los requisitos, ya sea a través de un proceso investigativo o mediante la interpretación del enunciado del problema, Bruegge y Dutoit (2002) recomiendan elaborar un modelo de análisis; esto es, elaborar un documento en el que se consigne toda la información obtenida y generada hasta el momento, preferiblemente utilizando modelos y lenguaje propios del ámbito de la programación. En pequeños ejercicios de programación bastará con especificar la información de entrada, las fórmulas de los cálculos a realizar y la información de salida.

En resumen, el análisis del problema, como mínimo, debe identificar la información relevante relacionada con el problema, las operaciones que deben realizar sobre dicha información y la nueva información que se debe producir; es decir, saber sobre qué datos opera el programa, qué operaciones, cálculos o transformaciones ha de realizar y qué resultados se espera obtener.

1.2.2 Diseño de la solución

Una vez que se ha analizado y comprendido el problema ya se puede pensar en la solución. El diseño consiste en aplicar el conocimiento disponible para responder a la pregunta: ¿cómo solucionar el problema?

Diseñar una solución equivale a generar, organizar y representar las ideas sobre la solución del problema. Para ello se cuenta con técnicas y herramientas de diseño propias de cada modelo de programación, que permiten registrar y comunicar las operaciones y procesos a desarrollar. Bajo el modelo de programación procedural se utiliza, como diseño de solución, los algoritmos; en el modelo funcional, las funciones y en el modelo orientado a objetos, los modelos conceptuales o diagramas.

Las ideas acerca de la solución de un problema, es decir el diseño de la solución, deben representarse utilizando un lenguaje conocido por la comunidad de programadores, de manera que el diseño sea comunicable. En los grandes proyectos de desarrollo de software, las personas que diseñan no son las mismas que escriben el código del programa; por ello, es necesario que la solución en la fase de diseño se documente de tal manera que otras personas puedan comprenderla completamente para llevar a cabo las fases siguientes.

En términos generales, en la fase de diseño se debe pensar en cuatro aspectos de la solución:

La interacción con el usuario: o diseño de la interfaz de usuario; esto es, pensar en cómo se hará el intercambio de datos entre el programa y la persona que lo utiliza, cómo se capturará los datos que

el programa requiere, cómo se presentará los resultados al usuario, cómo el usuario le indicará al programa lo que desea hacer.

El procesamiento de los datos: esto es conocer las operaciones que se realizan sobre los datos según el dominio del problema. Por ejemplo, si se trata de un programa para calcular la nómina de una empresa, se debe conocer los procesos que se llevan a cabo para obtener dicha nómina. Conviene tener presente que no se puede programar un proceso que se desconoce, primero es necesario que el programador conozca muy bien los cálculos que se desarrollan para obtener los resultados deseados; el computador automatizará el proceso, pero el programador debe especificar cómo realizarlo.

La arquitectura del sistema: para solucionar un problema con más facilidad es necesario descomponerlo en subproblemas de manera que cada uno sea más sencillo de resolver (este tema se desarrolla en el capítulo 6), pero todos las partes deben integrarse para ofrecer la solución al problema completo. La forma como se organizan e interactúan las partes de la solución se conoce como arquitectura del sistema. En la fase de diseño se lleva a cabo la descomposición del sistema en diferentes módulos o subsistemas y al mismo tiempo el diseño de la arquitectura que muestra cómo se relacionan los mismos.

La persistencia de datos: todos los programas actúan sobre datos y éstos deben guardarse de forma organizada de manera que el programa pueda almacenarlos y recuperarlos de forma segura. En la fase de diseño se debe decidir cómo se almacenarán los datos en los medios de almacenamiento secundario.

1.2.3 Codificación del programa

Es la traducción de la solución del problema expresada en modelos de diseño, a una serie de instrucciones detalladas en un lenguaje reconocido por el computador, al que se conoce como lenguaje de programación. La serie de instrucciones detalladas se denomina código fuente y debe ser compilada para convertirse en un programa ejecutable.

A diferencia del diseño, la codificación puede ser un trabajo sencillo, siempre que se cuente con el diseño detallado y se conozca las palabras reservadas del lenguaje, con su sintaxis y su semántica.

Es preciso anotar que las etapas en las que se debe concentrar el mayor esfuerzo e interés por desarrollar la destrezas necesarias son: el análisis y el diseño, pues, sin pretender restar importancia al conocimiento del lenguaje, en la etapa de codificación puede ayudar un buen libro de referencia, mientras que no existe un libro que le ayude a analizar el problema y diseñar la solución.

La etapa de codificación incluye la revisión de los resultados generados por el código fuente, línea a línea, y esto no debe confundirse con la prueba del programa que se trata en la sección siguiente.

1.2.4 Prueba y depuración

En el momento de codificar un programa se desarrollan pruebas parciales que permiten verificar que las instrucciones están escritas correctamente y que realizan la función que se espera de ellas; sin embargo, el hecho de que un programa funcione o se ejecute no significa que satisfaga plenamente la necesidad para la cual se desarrolló.

Los errores humanos dentro de la programación de computadores son muchos y aumentan considerablemente con la complejidad del problema. El proceso de identificar y eliminar errores, para dar paso a una solución eficaz se llama *depuración*.

La *depuración o prueba* resulta una tarea tan importante como el mismo desarrollo de la solución, por ello se debe considerar con el mismo interés y entusiasmo. La prueba debe realizarse considerando todas las posibilidades de error que se pueden presentar en la ejecución y utilización del programa.

La prueba tiene como fin identificar las debilidades del programa antes de que sea puesto en marcha, para corregir los errores, sin que

esto implique pérdida de información, tiempo y dinero. Una prueba exitosa es aquella que detecta errores, pues todos los programas los tienen; de manera que las pruebas se diseñan pensando en depurar el programa, no en confirmar que está bien construido.

1.2.5 Documentación

A menudo un programa escrito por una persona, es usado por otra. Por ello la documentación sirve para ayudar a comprender o usar el programa o para facilitar futuras modificaciones (mantenimiento).

La documentación de un programa es la guía o comunicación escrita en sus variadas formas, ya sea en enunciados, dibujos, diagramas o procedimientos.

Existen tres clases de documentación:

Documentación Interna
Documentación técnica
Manual del Usuario

Documentación Interna: son los comentarios o mensajes que se añaden al código fuente para hacerlo más claro y entendible.

En un programa extenso, el mismo programador necesitará de los comentarios en el código para ubicarse y hacer las correcciones o cambios que sean necesarios después de un tiempo. Mucha más importancia tiene esta documentación, si los cambios serán efectuados por personal diferente a quien escribió el código fuente original.

Documentación técnica: es un documento en el que se consigna información relevante respecto al problema planteado y la forma cómo se implementó la solución, algunos datos que se deben incluir son:

Descripción del Problema
Modelos de Análisis

Diseño de la solución
Diccionario de Datos
Código fuente (programa)
Pruebas realizadas

Esta documentación está dirigida a personal con conocimientos de programación; por ende, contiene información técnica sobre el programa y no debe confundirse con el manual del usuario.

La construcción de programas grandes y complejos requiere que se elabore la documentación en la medida en que se avanza en el análisis, diseño y codificación del software, esto debido a dos razones: en primer lugar, a que la construcción de grandes programas requiere el trabajo de varias personas y por ello se debe mantener todas las especificaciones por escrito; en segundo, porque no es posible mantener toda la información del proyecto en la memoria de los responsables, es necesario darle soporte físico.

Manual del Usuario: es un documento destinado al usuario final en el que se describe paso a paso el funcionamiento del programa. Debe incluir información detallada sobre el proceso de instalación, introducción de datos, procesamiento y obtención de resultados, recomendaciones e información sobre posibles errores.

1.2.6 Mantenimiento

Son los cambios que se le deben hacer al programa después de haberse puesto en funcionamiento. Estos cambios pueden tener como fin incluir nuevos procesos o adaptarlo a circunstancias que han cambiado después de que fue desarrollado.

El mantenimiento será más fácil de llevar a cabo si se cuenta con la documentación adecuada del programa.

Las fases que aquí se presentan no necesariamente se desarrollan en el orden en que se describen, pues muchos modelos de desarrollo son iterativos; no obstante, siempre estarán presentes.

2. ELEMENTOS DE PROGRAMACIÓN

La Programación
es el arte y la técnica
de construir y formular algoritmos
de una forma sistemática.
Wirth

Los algoritmos y los programas están conformados por series de operaciones que se desarrollan con datos. A su vez los datos se agrupan en categorías según los valores que pueden contener y las operaciones que se puede realizar sobre ellos. Las operaciones se especifican mediante expresiones conformadas por operadores y variables, éstas últimas hacen referencia a los datos con los que se realiza la operación.

Este capítulo está dedicado al estudio de aquellos elementos que siendo básicos son insustituibles en el diseño de algoritmos y la implementación de programas.

2.1 TIPOS DE DATOS

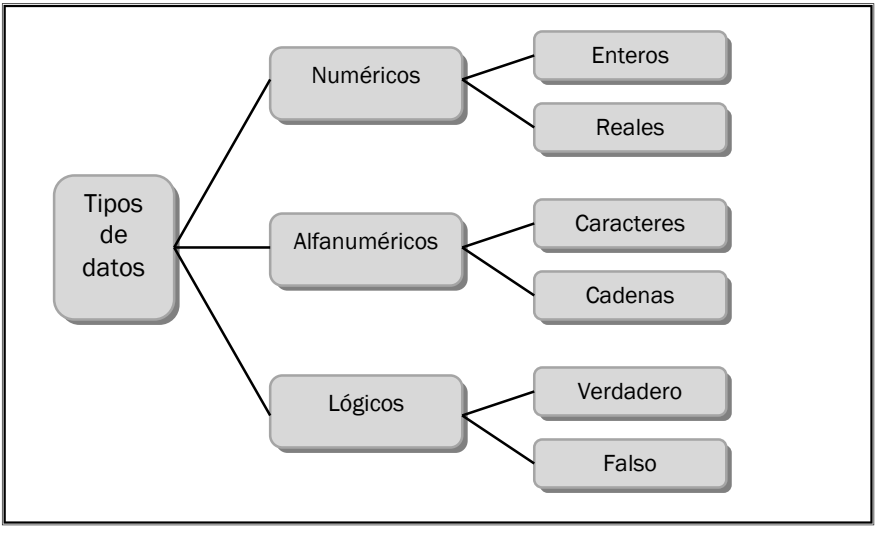
Los tipos de datos son clasificaciones para organizar la información que se almacena en la memoria del computador. Estas abstracciones permiten definir valores mínimos y máximos para cada tipo, de modo que se puede establecer el espacio que cada uno de ellos requiere y de esta manera facilitar la administración de la memoria.

Los tipos de datos se utilizan en la declaración de las variables y en la validación de las operaciones permitidas sobre cada tipo. Por ejemplo, los datos numéricos admiten operaciones aritméticas, mientras que las cadenas pueden ser concatenadas.

Los lenguajes de programación que manejan tipos de datos cuentan con la definición de tres elementos: los tipos primitivos o simples (números, caracteres y booleanos), el conjunto de valores que pueden manejar y las operaciones permitidas. A partir de los datos primitivos se pueden implementar tipos compuestos, como cadenas, vectores y listas (Appleby y Vandekppple, 1998).

Entre los datos primitivos se cuenta con tres tipos: numéricos, alfanuméricos y lógicos o booleanos. A su vez, los datos numéricos pueden ser enteros o reales, mientras que los alfanuméricos pueden ser caracteres o cadenas, como se muestra en la figura 4.

Figura 4. Tipos de datos



2.1.1 Datos numéricos

Los datos de tipo numérico son aquellos que representan cantidades o información cuantificable, como: el número de estudiantes de un curso, el sueldo de un empleado, la edad de una persona, el valor de un electrodoméstico, la extensión en kilómetros cuadrados que tiene un país o la nota de un estudiante.

Datos de tipo número entero. Son datos que se expresan mediante un número exacto; es decir, sin componente fraccionario y pueden ser positivos o negativos. Este tipo se utiliza para representar elementos que no pueden encontrarse de forma fraccionada en el dominio del problema, por ejemplo:

Dato	Valor
La cantidad de empleados de una empresa	150
Las asignaturas que cursa un estudiante	5
El número de goles anotados en un partido de fútbol	3
La cantidad de votos que obtiene un candidato	5678

Es importante anotar que el rango de valores enteros comprendido entre el infinito negativo y el infinito positivo no puede ser manejado en los lenguajes de programación, por razones de almacenamiento, los valores mínimos y máximos a manejar varían dependiendo del lenguaje.

Algunos lenguajes de programación reservan dos bytes para los datos de tipo entero, por lo tanto, el rango de valores admitidos está entre -32.768 y 32.767.

Datos de tipo número real. Son datos que se expresan mediante un número que incluye una fracción y pueden ser positivos o negativos. Este tipo de datos se utiliza para representar elementos que en el dominio del problema tienen valores formados por una parte entera y una parte decimal, por ejemplo:

Dato	Valor
La estatura de una persona (expresada en metros)	1.7
La temperatura ambiente (en grados centígrados)	18.5
La nota de un estudiante (con base 5.0)	3.5
La tasa de interés mensual	2.6

Los lenguajes de programación reservan un número fijo de bytes para los datos de tipo real, por ello el tamaño de estos números también es limitado.

2.1.2 Datos alfanuméricos

Se entiende como datos alfanuméricos aquellos que no representan una cantidad o valor numérico y por ende no se utilizan para cuantificar, sino para describir o cualificar un elemento al que hacen referencia. Por ejemplo, el color de una fruta, la dirección de una casa, el nombre de una persona, el cargo de un empleado, el género.

Los datos alfanuméricos pueden estar formados por caracteres del alfabeto, por números y por otros símbolos; sin embargo, aunque incluyan dígitos no pueden ser operados matemáticamente.

Caracteres. Los caracteres son cada uno de los símbolos incluidos en un sistema de codificación, pueden ser dígitos, letras del alfabeto y símbolos. El sistema más conocido actualmente es ASCII (*American Standard Code for Information Interchange*). Este requiere un byte de memoria para almacenar cada carácter e incluye un total de 256 caracteres. Son ejemplos de caracteres: 1, a, %, +, B, 3.

Cadenas. Son conjuntos de caracteres encerrados entre comillas (“ “) que son manipulados como un solo dato, por ejemplo:

El nombre de una persona: “José”
La ubicación de una universidad: “Calle 19 con 22”
El título de un libro: “Diseño de algoritmos”

2.1.3 Datos lógicos o booleanos

Los datos de este tipo solo pueden tomar dos valores: verdadero o falso. En programación se utilizan sobremanera para hacer referencia al cumplimiento de determinadas condiciones, por ejemplo: la existencia de un archivo, la validez de un dato, la relación existente entre dos datos.

2.2 VARIABLES Y CONSTANTES

Todos los programas de computador, indiferente de la función que cumplan, operan sobre un conjunto de datos. Durante la ejecución de un programa los datos que éste utiliza o genera reposan en la memoria. La información sobre el lugar de almacenamiento de un dato, el tipo y el valor se maneja mediante el concepto de variable. Si el valor se mantiene inalterable durante toda la ejecución del programa se le da el nombre de constante.

2.2.1 Variables

Para manipular un dato en un programa de computador es necesario identificarlo con un nombre y conocer: la posición de memoria donde se encuentra, el tipo al que pertenece y el valor. Para hacer más sencillo el manejo de esta información se cuenta con una abstracción a la que se denomina variable.

En términos de Appleby y Vandekppple (1998) una variable está asociada a una tupla conformada por los atributos:

Identificador (nombre de la variable)
Dirección (posición de memoria donde se almacena el dato)
Tipo de dato (especifica: conjunto de valores y operaciones)
Valor (dato almacenado en memoria)

Identificador. Un identificador es una palabra o secuencia de caracteres que hace referencia a una posición de memoria en la que se almacena un dato.

La longitud de un identificador y la forma de construirse puede variar de un lenguaje de programación a otro; no obstante, hay algunas recomendaciones que se deben tener presente:

- Debe comenzar por una letra comprendida entre A y Z (mayúscula o minúscula)
- No debe contener espacios en blanco
- Debe tener relación con el dato que se almacenará en la posición de memoria (nemotécnico)
- No debe contener caracteres especiales y operadores
- Después de la primera letra se puede utilizar dígitos y el caracter de subrayado (_).

Ejemplo, para almacenar el nombre de una persona, el identificador puede ser:

Nombre
Nbre
Nom

Es frecuente que el identificador represente más de una palabra, ya que se pueden tener datos similares pero que corresponden a diferentes elementos de información, por ejemplo:

Nombre de estudiante
Nombre de profesor
Código estudiante
Código profesor
Teléfono estudiante
Teléfono profesor

En estos casos es recomendable utilizar un número fijo de caracteres de la primera palabra combinado con algunos de la

segunda, teniendo en cuenta de aplicar la misma técnica para la formación de todos los identificadores, de manera que sea fácil recordar el identificador para cada dato.

Los identificadores de los datos anteriores podrían formarse de la siguiente forma:

Dato	**Identificador**	**O también**
Nombre de estudiante	Nom_est	Nbre_e
Nombre de profesor	Nom_pro	Nbre_p
Código estudiante	Cod_est	Cdgo_e
Código profesor	Cod_pro	Cdgo_p
Teléfono estudiante	Tel_est	Tfno_e
Teléfono profesor	Tel_pro	Tfno_p

En la segunda columna, los identificadores se han formado utilizando los tres primeros caracteres de cada palabra, mientras que en la tercera, tomando el primero y los últimos caracteres de la primera palabra y solo el primero de la segunda.

Cada programador tiene su propia estrategia para formar identificadores para: variables, constantes, funciones y tipos de datos definidos por el usuario. Lo importante es tener en cuenta las recomendaciones presentadas anteriormente y utilizar siempre la misma estrategia para facilitarle el trabajo a la memoria (del programador naturalmente).

2.2.2 Tipos de variables

Las variables pueden ser clasificadas con base a tres criterios: el tipo de dato que guardan, la función que cumplen y el ámbito.

De acuerdo al tipo de dato que almacenan, las variables pueden ser de tipo entero, real, caracter, cadena, lógico y de cualquier otro tipo que el lenguaje de programación defina.

En cuanto a la funcionalidad, las variables pueden ser: variables de trabajo, contadores, acumuladores y conmutadores.

El ámbito determina el espacio en el que las variables existen, pueden ser globales o locales.

Variables de trabajo. Son las variables que se declaran con el fin de guardar los valores leídos o calculados durante la ejecución del programa.

Ejemplo:

Real: área
Entero: base, altura
Altura = 10
Base = 20
Área = base * altura / 2

Contadores. Son variables que se utilizan para registrar el número de veces que se ejecuta una operación o un grupo de ellas. El uso más frecuente es como variable de control de un ciclo finito, en cuyo caso guardan el número de iteraciones, pero también pueden registrar el número de registros de un archivo o la cantidad de datos que cumplen una condición.

Los contadores se incrementan o decrementan con un valor constante, por lo regular es de uno en uno.

Ejemplo: se desea ingresar las notas definitivas de los estudiantes de un grupo para calcular el promedio del mismo. Dado que el tamaño del grupo puede variar, se requiere una variable (contador) para registrar el número de notas ingresadas, para luego poder calcular el promedio.

Acumuladores. También se llaman totalizadores. Son variables que se utilizan para almacenar valores que se leen o se calculan repetidas veces. Por ejemplo, si se quisiera calcular el promedio de notas de un grupo de estudiantes, lo primero que se hace es leer las notas y cada una se suma en una variable (acumulador) de modo que después de leer todas las notas se divide la sumatoria de

las mismas sobre el número de estudiantes para obtener el promedio.

En este ejemplo se ha hecho referencia a un contador para conocer el número de estudiantes y a un acumulador para sumar las notas.

El acumulador se diferencia del contador en que éste no tiene incrementos regulares, sino que se incrementa de acuerdo al valor leído o al resultado de una operación.

Conmutadores. También se les llama interruptores, *switchs*, banderas, centinelas o *flags*. Son variables que pueden tomar diferentes valores en la ejecución de un programa y dependiendo de dichos valores el programa puede variar la secuencia de instrucciones a ejecutar, es decir, tomar decisiones.

Ejemplo 1: un conmutador puede utilizarse para informarle a cualquier módulo del programa si un determinado archivo ha sido abierto. Para ello se declara la variable de tipo lógico y se le asigna el valor falso y en el momento en que se abre el archivo se cambia a verdadero. Así en cualquier momento que se desee averiguar si el archivo ha sido abierto basta con verificar el estado del conmutador.

Ejemplo 2: si se busca un registro en un archivo se declara una variable de tipo lógico y se inicializa en falso, para indicar que el registro no ha sido encontrado. Se procede a hacer la búsqueda, en el momento en que el registro es localizado se cambia el valor de la variable a verdadero. Al finalizar la búsqueda, la variable (conmutador) informará si el registro fue encontrado o no, dependiendo si su valor es verdadero o falso, y según éste el programa sabrá que instrucciones ejecutar.

Ejemplo 3: es común que un programa esté protegido con un sistema de seguridad. En algunos casos se tienen diferentes niveles de usuarios, donde dependiendo del nivel tiene acceso a determinados módulos. El programa habilitará o deshabilitará

ciertas opciones dependiendo del nivel de acceso, el cual estará almacenado en una variable (conmutador).

Locales. Una variable es de tipo local cuando solo existe dentro del módulo o función en el que fue declarada. Este tipo de variable es útil cuando se aplica el concepto de programación modular, ya que permite que en cada procedimiento o función se tengan sus propias variables sin que entren en conflicto con las declaradas en otros módulos.

Por defecto todas las variables son locales al ámbito donde se declaran, si se desea declararlas como globales es necesario especificarlas como tales.

Globales. Son variables que se declaran para ser utilizadas en cualquier módulo* del programa. Existen desde que se declaran hasta que se termina la ejecución de la aplicación.

El concepto de variables locales y globales es poco útil en la solución de problemas mediante un solo algoritmo, pero cuando el algoritmo se divide en subrutinas es importante diferenciar entre las variables que desaparecen al terminar un procedimiento o una función y aquellas que trascienden los módulos, y su actualización en un lugar puede afectar otras subrutinas.

Por lo general, los lenguajes de programación definen las variables como locales a menos que se indique lo contrario; por ello, siguiendo esa lógica, en este documento se propone utilizar el modificador *global* antes de la declaración de la variable para especificar que es de ámbito global.

Global entero: x

* Los conceptos de módulo, subrutina, procedimiento y función están relacionados con la estrategia "dividir para vencer" aplicada para enfrentar la complejidad del desarrollo de software. Ésta consiste en partir un problema grande en problemas de menor tamaño, más fáciles de solucionar, y al hacer interactuar las soluciones parciales se resuelve el problema general. Este tema se trata en el capítulo 6.

Se debe tener cuidado de no utilizar el mismo identificador de una variable global para declarar una variable local, pues al actualizarla dentro de la subrutina no se sabría a qué posición de memoria se está accediendo.

2.2.3 Declaración de variables

Esta operación consiste en reservar en memoria el espacio suficiente para almacenar un dato del tipo especificado y a la vez incluir el identificador en la lista de variables del programa.

Para declarar una variable se escribe el tipo de dato seguido del identificador de la variable, de la forma:

TipoDato identificador

Si se desea declarar varias variables de un mismo tipo, se escribe el tipo de dato y la lista de identificadores separados por comas (,) de la forma:

TipoDato identificador1, identificador2, identificador3

Ejemplo:

Entero: edad
Real: estatura, peso, sueldo
Cadena: nombre, apellido, dirección

Aunque hay algunos lenguajes de programación que permiten declarar las variables en el momento en que se las necesita, es aconsejable, en favor de los buenos hábitos de programación, siempre declarar las variables antes de utilizarlas y el sitio más adecuado es el inicio del programa o de la función.

2.2.4 Asignación de valores

Esta operación consiste en almacenar un dato en una posición de memoria haciendo uso de una variable previamente declarada. Es necesario que el dato asignado corresponda al tipo para el cual se declaró la variable.

Una expresión de asignación tiene la forma:

variable = dato

Ejemplo:

Declaración de variables

Cadena: nombre, dirección
Real: sueldo

Asignación de valores

nombre = “José”
dirección = “Carrera 24 15 40”
sueldo = 1000000

También se puede asignar a una variable el resultado de una expresión, de la forma:

variable = expresión

Ejemplo:

Declaración de variables:
Entero: base, altura
Real: área

Asignación:

base = 10

altura = 5
área = base * altura

Una expresión de asignación tiene tres partes, una variable, el signo igual y el valor o la expresión cuyo resultado se asigna a la variable. La variable siempre va a la izquierda del signo igual, mientras que el valor o la expresión siempre estará a la derecha.

2.2.5 Constantes

Una constate hace referencia a una posición de memoria y se forma de igual manera que una variable, con la diferencia de que el dato almacenado no cambia durante la ejecución del programa.

Las constantes se utilizan para no tener que escribir los mismos valores en diversas partes del programa, en vez de ello se utiliza una referencia al mismo. De manera que cuando se requiera cambiar dicho valor basta con hacer el cambio en una sola línea de código, en la que se definió la constante.

Otro motivo para definir una constante es que resulta más fácil recordar y escribir un identificador que un valor. Por ejemplo, es más fácil escribir *pi* que 3.1416, en especial si este valor hay que utilizarlo repetidas veces en el programa.

2.3 OPERADORES Y EXPRESIONES

Los operadores son símbolos que representan operaciones sobre datos. Las expresiones son combinaciones de operadores y operandos (datos) que generan un resultado.

En programación se utilizan tres tipos de operaciones: aritméticas, relacionales y lógicas, y para cada una existe un conjunto de operadores que permite construir las expresiones.

2.3.1 Operadores y expresiones aritméticos

Los operadores aritméticos se aplican sobre datos de tipo numérico y permiten realizar las operaciones aritméticas, éstos se presentan en el cuadro 2.

Cuadro 2. Operadores aritméticos

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
Mod	Módulo

Las expresiones aritméticas combinan los operadores del cuadro 2 con datos numéricos y generan un nuevo número como resultado. Por ejemplo, si se tiene la base y la altura de un triángulo, mediante una expresión aritmética se puede obtener su área.

Ejemplo:

Declaración de variables

Real: base, altura, área

Asignación de valores

base = 10
altura = 15

Expresión aritmética para calcular el área:

base * altura / 2;

Asignación del resultado de una expresión aritmética a una variable:

área = base * altura / 2

En resultado de esta expresión es 75 y se guarda a la variable área.

Los operadores aritméticos se utilizan para operar tanto datos de tipo enteros como reales, excepto el operador ***Mod*** que se aplica únicamente a números enteros.

Algunos autores, como Becerra (1998) y Cairó (2005), hacen diferencia entre la operación división (/) y división entera, y utilizan la palabra reservada ***div*** para la segunda. En este libro se utiliza el símbolo / para toda operación de división. Si los operandos son de tipo entero el resultado será otro entero, en cuyo caso se dirá que se desarrolla una división entera, mientras que si los operandos son reales, o alguno de ellos lo es, el resultado será de tipo real.

El operador *Mod* (módulo) devuelve el residuo de una división entera. Ejemplo:

10 Mod 2 = 0 ya que 10 / 2 = 5 y el residuo es 0
10 Mod 4 = 2 ya que 10 / 4 = 2 y el residuo es 2

La combinación de diferentes operadores aritméticos en una misma expresión puede hacer que ésta resulte ambigua; es decir, que sea posible más de una interpretación y por ende más de un resultado. Para evitar dicha ambigüedad los operadores tienen un orden jerárquico que determina cuál se ejecuta primero cuando hay varios en la misma expresión. Para alterar el orden de ejecución determinado por la jerarquía es necesario utilizar paréntesis.

Como se aprecia en el cuadro 3, en la jerarquía de los operadores, en primer lugar se tiene los paréntesis, que permiten construir subexpresiones, que son las primeras en desarrollarse. Luego se tiene la multiplicación, la división y el módulo, que presentan un nivel de jerarquía mayor que la suma y la resta. Esto significa que si en una expresión se encuentran multiplicaciones y sumas, primero se ejecutan las multiplicaciones y después las

sumas. Cuando se presentan varios operadores del mismo nivel, como suma y resta, la ejecución se hace de izquierda a derecha.

Cuadro 3. Jerarquía de los operadores aritméticos

Nivel de prioridad	Operador	Operación
1	()	Agrupación
2	*, /, Mod	Multiplicación, división y módulo
3	+, -	Suma, resta, incremento y decremento

Ejemplos:

3 * 2 + 5 = 6 + 5 = 11
3 * (2 +5) = 3 * 7 = 21

6 + 4 / 2 = 6 + 2 = 8
(6 + 4) / 2 = 10 / 2 = 5

5 * 3 + 8 / 2 – 1 = 15 + 4 - 1 = 18
5 * (3 + 8) / (2 - 1) = 5 * 11 / 1 = 55

10 + 12 Mod 5 = 10 + 2 = 12
(10 + 12) Mod 5 = 22 % 5 = 2

2.3.2 Operadores y Expresiones Relacionales

Los operadores relacionales permiten hacer comparaciones entre datos del mismo tipo. Las expresiones relacionales dan como resultado un dato de tipo lógico: verdadero o falso. Estas expresiones se utilizan principalmente para tomar decisiones o para controlar ciclos. Los operadores se presentan en el cuadro 4.

Cuadro 4. Operadores relacionales

Operador	Comparación
<	Menor que
<=	Menor o igual que
>	Mayor
>=	Mayor o igual que
=	Igual a
<>	Diferente de

Algunos ejemplos de expresiones relacionales son:

Declaración e inicialización de la variable

Entero x = 5;

Expresiones relacionales:

x < 10 = verdadero
x > 10 = falso
x = 10 = falso

Declaración e inicialización de variables

Real a = 3, b = 1

Expresiones relacionales:

a = b = falso
a > b = *verdadero*;
a < b = falso
a <> b = *verdadero*

2.3.3 Operadores y expresiones lógicos

Estos operadores se utilizan únicamente para operar datos de tipo lógico o booleano (verdadero o falso). Las expresiones lógicas dan como resultado otro dato de tipo lógico. Los operadores se muestran en el cuadro 5.

Cuadro 5. Operadores lógicos

Operador	Operación
Y	Conjunción
O	Disyunción
NO	Negación

En el cuadro 6 se muestra los resultados de operar datos lógicos con los operadores Y, O, NO.

Cuadro 6. Resultado de operaciones lógicas

Operando 1	Operador	Operando 2	=	Resultado
Verdadero	Y	Verdadero	=	Verdadero
Verdadero		Falso		Falso
Falso		Verdadero		Falso
Falso		Falso		Falso
Verdadero	O	Verdadero	=	Verdadero
Verdadero		Falso		Verdadero
Falso		Verdadero		Verdadero
Falso		Falso		Falso
	No	Verdadero	=	Falso
		False		Verdadero

Ahora bien, como los datos lógicos no son comunes en el mundo real, la mayoría de las expresiones lógicas se construyen a partir de expresiones relacionales. Algunos ejemplos son:

Real nota = 3.5;

(nota >= 0) **Y** (nota <= 5.0) = Verdadero

Entero a = 15, b = 8;

(a > 10) **O** (b > 10) = Verdadero
NO (b > 5) = falso

2.3.4 Jerarquía general de los operadores

Los operadores aritméticos, relacionales y lógicos pueden aparecer mezclados en una misma expresión. En estos casos es necesario tener especial atención a la jerarquía de los mismos para asegurarse que cada uno opera sobre tipos válidos y no se generen errores en tiempo de ejecución. Para ello conviene tener en cuenta el orden jerárquico que le corresponde a cada uno, a fin de establecer el orden de ejecución y forzar la prioridad para que las expresiones se desarrollen correctamente. En el cuadro 7 se presenta la jerarquía de todos los operadores estudiados en este capítulo.

Cuadro 7. Jerarquía de los operadores

Prioridad	Operadores
1	()
2	*, /, Mod, NO
3	+, -, , Y
4	>, <, >=, <=, <>, =, O

Tener en cuenta que los operadores del mismo nivel se ejecutan de izquierda a derecha.

3. ALGORITMOS

El tiempo es un maestro de ceremonias
que siempre acaba poniéndonos
en el lugar que nos compete,
vamos avanzando,
parando y retrocediendo
según sus órdenes.
Saramago

La palabra algoritmo proviene de la traducción al latín del término árabe *Al-Khowarizmi*, que significa *originario de Khwarizm,* nombre antiguo del mar de Aral (Galve *et al*, 1993). De otra parte, *Al-Khowarizmi* es el nombre con que se conoció a un matemático y astrónomo persa que escribió un tratado sobre manipulación de números y ecuaciones en el Siglo IX. (Brassard y Bratley, 1997), (Joyanes *et al, 1998*).

3.1 CONCEPTO DE ALGORITMO

Se han propuesto diferentes conceptos de algoritmos, algunos de ellos son: "conjunto finito, y no ambiguo de etapas expresadas en un cierto orden que, para unas condiciones iniciales, permiten resolver el problema en un tiempo finito" (Galve et al, 1993: 3); o también, "conjunto de reglas para efectuar un cálculo, bien sea a mano, o más frecuentemente, en una máquina" (Brassard y Bratley, 1997: 2).

De igual manera se ha considerado que es un procedimiento computacional bien definido que toma un valor o un conjunto de valores como entradas y produce algún valor o conjunto de valores como salida. Así, un algoritmo es una secuencia de pasos computacionales para transformar la entrada en salida. (Comer *et al*, 2001).

Para Becerra (1998), un algoritmo es el camino para solucionar un problema en un computador, es un grupo de instrucciones que se escriben de acuerdo a reglas sintácticas suministradas por un lenguaje de programación.

Mientras que para Hermes (1984:19) "un algoritmo es un procedimiento general con el que se obtiene la respuesta a todo problema apropiado mediante un simple cálculo de acuerdo con un método especificado" y posteriormente menciona que este procedimiento debe estar claramente especificado de manera que no haya lugar a que intervenga la imaginación y la creatividad de quien lo ejecuta.

También puede definirse como "una serie de operaciones detalladas y no ambiguas, a ejecutar paso a paso, y que conducen a la solución de un problema" (Joyanes, 1992: 2).

Cuando se dice que un problema tiene una solución algorítmica significa que es posible escribir un programa de computadora que obtendrá la respuesta correcta para cualquier conjunto de datos de entrada (Baase y Gelder, 2002).

Para que la solución de un problema sea llevada hasta un lenguaje de programación, los pasos expresados en el algoritmo deben ser detallados, de manera que cada uno de ellos implique una operación trivial; es decir, que los pasos no impliquen procesos que requieran de una solución algorítmica. En caso de presentarse esta situación, el algoritmo debe ser refinado, lo que equivale a desarrollar nuevamente el algoritmo para la tarea concreta a la que se hace mención.

Si el problema que se desea solucionar es muy grande o complejo, es recomendable dividirlo en tareas que se puedan abordar independientemente y que resulten más sencillas de solucionar. A esto se le llama diseño modular.

Como ejemplo considérese el algoritmo de Euclides para calcular el Máximo Común Divisor (MCD). El MCD de dos números enteros es el número más grande que los divida a los dos, incluso puede ser uno de ellos, dado que todo número es divisible por sí mismo.

Este algoritmo establece unos pasos concretos que al ejecutarse un número determinado de veces, independiente de los números propuestos, siempre encuentra el mayor divisor común. La solución se presenta más adelante en las cuatro notaciones explicadas en este capítulo.

3.2 CARACTERÍSTICAS DE UN ALGORITMO

Un algoritmo debe tener al menos las siguientes características:

- **Ser preciso**: esto significa que las operaciones o pasos del algoritmo deben desarrollarse en un orden estricto, ya que el desarrollo de cada paso debe obedecer a un orden lógico.

- **Ser definido:** el resultado de la ejecución de un algoritmo depende exclusivamente de los datos de entrada; es decir, siempre que se ejecute el algoritmo con el mismo conjunto de datos el resultado será el mismo.

- **Ser finito**: esta característica implica que el número de pasos de un algoritmo, por grande y complicado que sea el problema que soluciona, debe ser limitado. Todo algoritmo, sin importar el número de pasos que incluya, debe llegar a un final. Para hacer evidente esta característica, en la representación de un algoritmo siempre se incluyen los pasos inicio y fin.

- **Notación**: Para que el algoritmo sea entendido por cualquier persona interesada es necesario que se exprese en alguna de las formas comúnmente aceptadas, esto con el propósito de reducir la ambigüedad propia del lenguaje natural. Entre las notaciones más conocidas están: pseudocódigo, diagrama de flujo, diagramas de Nassi/Shneiderman y funcional.

- **Corrección**: el algoritmo debe ser correcto, es decir debe satisfacer la necesidad o solucionar el problema para el cual fue diseñado. Para garantizar que el algoritmo logre el objetivo, es necesario ponerlo a prueba; a esto se le llama verificación, una forma sencilla de verificarlo es mediante la prueba de escritorio.

- **Eficiencia**: hablar de eficiencia o complejidad de un algoritmo es evaluar los recursos de cómputo que requiere para almacenar datos y para ejecutar operaciones frente al beneficio que ofrece. En cuanto menos recursos requiera será más eficiente el algoritmo.

Si la descripción de un procedimiento o el enunciado de una solución para un problema no cuentan con estas características, no puede considerarse técnicamente un algoritmo. Por ejemplo, las recetas de cocina se parecen mucho a un algoritmo, en el sentido de que describen el procedimiento para preparar algún tipo de alimento, incluso algunos autores, como López, Jeder y Vega (2009) han tomado las recetas de cocina como ejemplo de algoritmos; no obstante, pocas veces están definidas lo suficiente para ser un algoritmo, pues siempre dejan un espacio para la subjetividad de quien ejecuta las acciones.

La diferencia fundamental entre una receta y un algoritmo está en que la primera no está definida, como se muestra en el ejemplo del cuadro 8, tomado de López, Jeder y Vega(2009).

Cuadro 8. Primer algoritmo para preparar una taza de café

1.	Encender una hornilla
2.	Colocar una jarra con leche sobre la hornilla
3.	Esperar a que la leche hierva
4.	Colocar café en una taza
5.	Verter un poco de leche en la taza y batir
6.	Verter más leche en la taza hasta colmarla
7.	Agregar azúcar al gusto

Este tipo de acciones no pueden ser desarrolladas por el computador, de forma automática, porque requieren la intervención del criterio humano. Para que sea un algoritmo, en el sentido que se sustenta en este libro, sería necesario escribirlo como se muestra en el cuadro 9.

Cuadro 9. Segundo algoritmo para preparar una taza de café

1.	Inicio
2.	Encender una hornilla
3.	Colocar una jarra con 200 ml. De leche sobre la hornilla
4.	Colocar 10 gr. de café instantáneo en una taza
5.	Colocar 10 gr. de azúcar en la taza
6.	Mezclar el azúcar y el café
7.	Esperar hasta que la leche hierva
8.	Apagar la hornilla
9.	Verter la leche en la taza
10.	Mezclar hasta que el café y el azúcar se disuelvan
11.	Fin

La primera versión corresponde a una receta, la segunda a un algoritmo. En la segunda versión se indica el principio y el fin, los pasos tienen un orden lógico, las actividades están definidas (cantidades explícitas) de manera que si se aplica varias veces con los mismos insumos se obtendrá el mismo resultado. Es decir que

el producto final depende del algoritmo y no de la persona que lo ejecuta.

3.3 NOTACIONES PARA ALGORITMOS

Se ha definido un algoritmo como el conjunto de pasos para encontrar la solución a un problema o para resolver un cálculo. Estos pasos pueden expresarse de diferentes formas, como: descripción, pseudocódigo, diagramas y funciones matemáticas. La más sencilla, sin duda, es la descripción en forma de prosa; sin embargo, el lenguaje natural es muy amplio y ambiguo, lo que hace muy difícil lograr una descripción precisa del algoritmo, y siendo la claridad una característica fundamental del diseño de algoritmos es preferible utilizar el pseudocódigo. La representación gráfica se hace mediante diagramas que son fáciles de comprender y de verificar, los principales son: diagrama de flujo y diagrama N-S. La representación matemática es propia del modelo de programación funcional y consiste en expresar la solución mediante funciones matemáticas, preferiblemente utilizando Calculo Lambda*.

3.3.1 Descripción textual

Esta es, sin duda, la forma más sencilla de escribir un algoritmo, se trata de elaborar una lista de actividades o pasos en forma de prosa, sin aplicar ninguna notación.

Esta forma de presentar un algoritmo tiene la desventaja de que da lugar a cometer imprecisiones y ambigüedades, dada la amplitud del lenguaje. Es apropiada cuando se escribe la versión preliminar de un algoritmo y se desea centrar la atención en la lógica más que en la forma de comunicar las ideas. Una vez terminado el algoritmo, es recomendable pasarlo a alguna de las notaciones que se explican a continuación.

* El Cálculo Lambda es un sistema formal que define una notación para funciones computables, facilita la definición y evaluación de expresiones y por ello es el fundamento teórico del modelo de programación funcional. Fue desarrollado por Alonso Church en 1936.

Ejemplo 1. Máximo Común Divisor

Uno de los algoritmos más antiguos de que se tienen noticia es el diseñado por Euclides* para calcular el Máximo Común Divisor (MCD) entre dos números. El MCD(a,b) – léase: el máximo común divisor de los números a y b - consiste en identificar un número c que sea el número más grande que divide exactamente a los dos anteriores.

Este algoritmo expresado en forma descriptiva sería:

Se leen los dos números, luego se divide el primero sobre el segundo y se toma el residuo. Si el residuo es igual a cero, entonces el MCD es el segundo número. Si el residuo es mayor a cero se vuelve a efectuar la división del segundo número sobre el residuo de la última división y se continúa dividiendo hasta que el residuo obtenido sea cero. La solución será el número que se utilizó como divisor en la última división.

3.3.2 Pseudocódigo

Es la mezcla de un lenguaje de programación y el lenguaje natural (español, inglés o cualquier otro idioma) en la presentación de una solución algorítmica. Esta notación permite diseñar el algoritmo de forma estructurada pero utilizando el lenguaje y el vocabulario conocido por el diseñador.

Un pseudocódigo es un acercamiento a la estructura y sintaxis de un lenguaje de programación determinado, lo cual facilita la comprensión del algoritmo y su codificación, debido a que se presenta una estrecha correspondencia entre la organización del algoritmo y la sintaxis y semántica del lenguaje.

* Euclides (Tiro, 330 – Alejandría, 300 a.C.), matemático griego. Por encargo del faraón Tolomeo I Sóter (el salvador), escribió la obra Elementos, compuesta por 13 tomos en los que sistematiza en conocimiento matemático de su época; También se le atribuyen otras obras, como: Óptica, Datos, Sobre las divisiones, Fenómenos y elementos de la música (Gran Enciclopedia Espasa, 2005: 4648).

Considerando que el lenguaje natural es muy amplio, mientras que los lenguajes de programación son limitados, es necesario utilizar un conjunto de instrucciones que representen las principales estructuras de los lenguajes de programación. A estas instrucciones se las denomina palabras reservadas y no podrán ser utilizadas como identificadores de variables.

Algunas palabras reservadas son:

Cadena	Función	Lógico	Repita
Caracter	Hacer	Mientras	Retornar
Decrementar	Hasta	Para	Según sea
Entero	Incrementar	Procedimiento	Si
Escribir	Inicio	Real	Si no
Fin	Leer		

Identación

Una de las dificultades que presenta el pseudocódigo es que no es fácil apreciar las estructuras de programación como las bifurcaciones y los ciclos, esto debido a que las instrucciones aparecen una después de otra, aunque no sea esa la secuencia en que se ejecutan.

La identación consiste en dejar una sangría o tabulación para las instrucciones que están dentro de una estructura de programación, para indicar su dependencia y para evidenciar dónde comienza y termina una estructura.

Ventajas de utilizar un Pseudocódigo

Algunas ventajas de esta notación son:

- Ocupa menos espacio
- Permite representar en forma fácil operaciones repetitivas
- Es muy fácil pasar del pseudocódigo al código del programa en un lenguaje de programación.

- Si se aplica identación se puede observar claramente la dependencia de las acciones a las estructuras de programación.

En el cuadro 10, se presenta el algoritmo de Euclides en notación de pseudocódigo. En este ejemplo se observa la aplicación del concepto de identación para resaltar la dependencia de las instrucciones con respecto a la estructura "mientras" y al inicio - fin del algoritmo.

Cuadro 10. Algoritmo para obtener el MCD

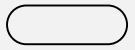
```
1.  Inicio
2.      Entero: a, b, c=1
3.      Leer a, b
4.      Mientras c>0 hacer
5.          c = a Mod b
6.          a = b
7.          b = c
8.      Fin mientras
9.      Escribir "MCD = ", a
10. Fin algoritmo
```

En este algoritmo se lee los dos números a y b, se inicializa en 1 la variable c para asegurarse que el ciclo se ejecute. Se realizan divisiones sucesivas, la primera con los números ingresados y las siguientes tomando el divisor como dividendo y el residuo como divisor, hasta obtener un división entera, cuando esto ocurra el número que haya actuado como divisor será la solución o MCD.

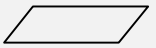
3.3.3 Diagrama de Flujo

Es la representación gráfica de un algoritmo mediante un conjunto de símbolos que representan las operaciones y estructuras básicas de programación. El conjunto de símbolos utilizados para diseñar diagramas de flujo se presentan en la figura 5.

Figura 5. Símbolos utilizados para diseñar diagramas de flujo

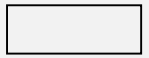


Símbolo utilizado para representar el inicio y el fin de un algoritmo.

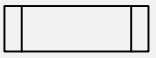


Proceso de entrada y salida de datos. Se debe escribir la operación a realizar: Leer o Escribir y la lista de datos.

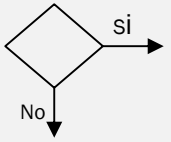
Teclado, se utiliza para representar la instrucción Leer datos por teclado.



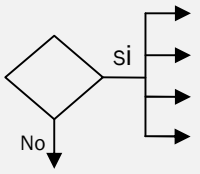
Proceso, representa asignación o cálculo. Se utiliza también para declarar variables.



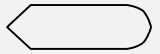
Proceso predefinido, se utiliza para llamar a una subrutina o subprograma.



Decisión, representa operaciones lógicas o de comparación. Tiene dos salidas. El programa seguirá una de ellas dependiendo de si se cumple o no la condición.

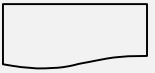


Decisión múltiple, tiene varias salidas.
La variable evaluada puede tomar uno de diferentes valores posibles y dependiendo de éste el programa seguirá un curso de acción. Si el valor es diferente a todos los considerados se seguirá el camino marcado como No.



Pantalla, se utiliza para representar la instrucción Escribir en pantalla o salida de datos por pantalla.

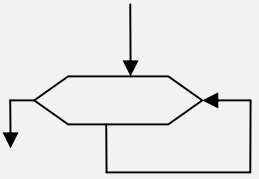
Figura 5. (Continuación)



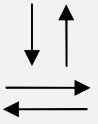
Impresora, se utiliza para representar la instrucción Escribir por impresora o salida de datos por impresora.

Conector en la misma página, se utiliza para unir dos partes del diagrama que no pueden unirse mediante una flecha, pero que están en la misma página.

Conector en diferente página, se utiliza para unir dos partes del diagrama que no pueden unirse mediante una flecha por estar en páginas diferentes.



Bucle, se utiliza para representar estructuras de repetición. Tiene dos salidas y dos entradas, por arriba recibe el flujo proveniente de un proceso anterior, mientras que por la derecha recibe el flujo proveniente del interior del ciclo, por abajo están los procesos que se repiten y por la izquierda se sale del ciclo.



Líneas dirigidas, se utilizan para unir los símbolos del diagrama e indican la dirección del flujo del algoritmo. En lo posible solo se deben utilizar hacia abajo y hacia la derecha.

Comentario, símbolo utilizado para introducir comentarios en el diagrama de flujo.

La representación mediante diagrama de flujo hace que el algoritmo se comprenda fácilmente, por cuanto es visible su estructura y el flujo de operaciones. No obstante, cuando el algoritmo es extenso es difícil de construir el diagrama. Aunque se puede hacer uso de conectores dentro y fuera de la página, el uso de estos afecta la comprensión de la lógica del algoritmo.

Recomendaciones para construir diagramas de flujo

Tomando como referencia a Cairó (2005) se hace algunas recomendaciones para la construcción apropiada de diagramas de flujo:

- El primer símbolo en todo diagrama de flujo debe ser el de *inicio* y al hacer el recorrido, indiferente de los caminos que tenga, siempre debe llegar al símbolo de *fin*.

- Todos los símbolos deben estar unidos con líneas dirigidas, que representan el flujo de entrada y el flujo de salida, a excepción del inicio que no tiene entrada y el fin que no tiene salida.

- Las líneas deben ser rectas, horizontales o verticales y no deben cruzarse. En lo posible hay que procurar utilizar solo líneas hacia abajo y hacia la derecha, con el fin de facilitar la lectura del algoritmo. Excepción a ésta regla son los ciclos.

- Toda línea debe partir de un símbolo y terminar en un símbolo.

- Los conectores deben estar numerados consecutivamente y se deben utilizar solo cuando es necesario.

- Algunos símbolos pueden ser utilizados para representar más de una instrucción y por ello es necesario escribir una etiqueta utilizando expresiones del lenguaje natural. No se debe utilizar sentencias propias de un lenguaje de programación.

- Si se utilizan variables cuyos indicadores son abreviaturas es necesario utilizar el símbolo de comentario para describirlas detalladamente.

- Procurar utilizar solo una página para el diagrama, si esto no es posible es conveniente utilizar conectores numerados y numerar también las páginas.

Ventajas de utilizar diagrama de flujo

Joyanes(1992) señala algunas ventajas de utilizar diagramas de flujo para el diseño de algoritmos:

- Facilita la comprensión del algoritmo

- Facilita la documentación

- Por utilizar símbolos estándares es más fácil encontrar sentencias equivalentes en lenguaje de programación

- Facilita la revisión, prueba y depuración del algoritmo

En la figura 6 se presenta el algoritmo de Euclides en notación de diagrama de flujo.

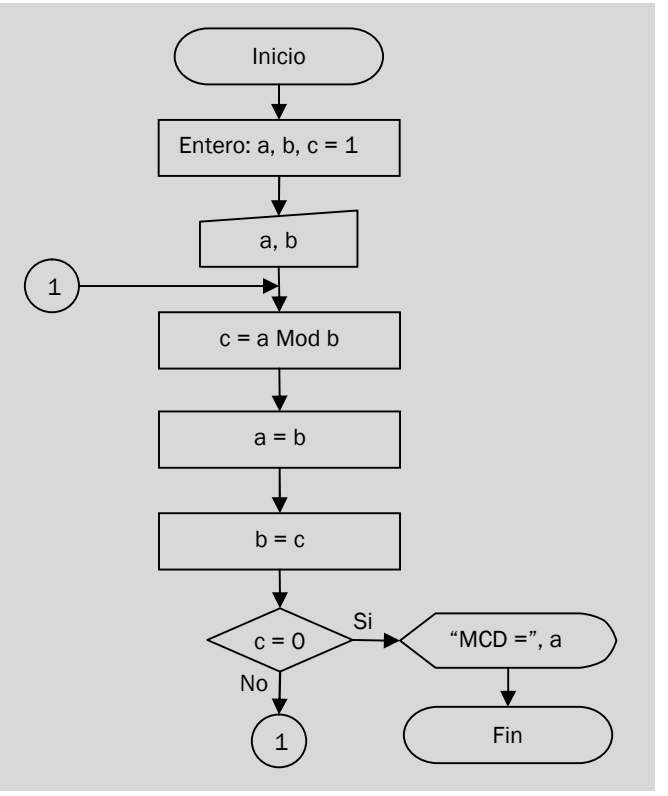
3.3.4 Diagrama de Nassi-Shneiderman

Es una notación propuesta por Isaac Nassi y Ben Shneiderman*, en 1973, como una alternativa a los diagramas de flujo para la

* Isaac Nassi es un experto informático estadounidense. Ha sido director de desarrollo sistemas de Cisco y director de investigaciones de SAP América. También ha trabajado para Apple Computer.

aplicación de la programación estructurada. También se conoce como diagrama de Chapín o simplemente diagrama N-S.

Figura 6. Diagrama de flujo para el algoritmo de Euclides



Ben Shneiderman es matemático/físico e informático estadounidense, catedrático de informática en el Human-Computer interaction Laboratory en la Universidad de Maryland. Su investigación se desarrolla sobre la interacción hombre-máquina. Definió la Usabilidad Universal pensando en las diferentes características de los usuarios y de la tecnología que utilizan.

Un diagrama N-S se construye utilizando instrucciones de pseudocódigo y un conjunto reducido de figuras básicas que corresponden a las estructuras de programación: secuenciales, selectivas e iterativas. No utiliza flechas para establecer el flujo de las operaciones, sino que coloca las cajas que corresponden a las instrucciones unas después o dentro de otras, de manera que es evidente la secuencia, la bifurcación o la repetición en la ejecución del algoritmo.

Nassi y Shneiderman (1973) sostienen que los diagramas por ellos diseñados ofrecen las siguientes ventajas:

- Las estructuras de decisión son fácilmente visibles y comprensibles
- Las iteraciones son visibles y bien definidas
- El alcance de las variables locales y globales es evidente en el diagrama
- Es imposible que el control se transfiera arbitrariamente
- La recursividad se representa de forma trivial
- Los diagramas se pueden adaptar a las peculiaridades del lenguaje de programación a utilizar

A diferencia del pseudocódigo y el diagrama de flujo, el diagrama N-S permite tener una visión más estructurada de los pasos del algoritmo y por ende facilita no solo el paso siguiente que es la codificación, sino también la comprensión y el aprendizaje.

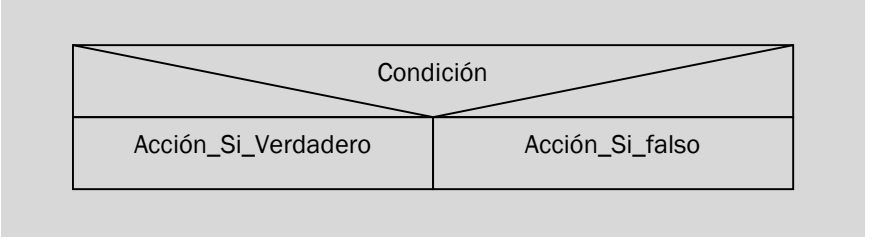
Un programa se representa por un solo diagrama en el que se incluye todas las operaciones a realizar para la solución del problema, comienza con un rectángulo con la etiqueta inicio y termina con otra con la etiqueta fin. Para conectar un diagrama con otro se utiliza la palabra proceso y el nombre o número del subprograma a conectar. Fuera del diagrama se coloca el nombre del programa o algoritmo. En estos diagramas se representa detalladamente los datos de entrada y salida y los procesos que forman parte del algoritmo (Alcalde y García, 1992).

En N-S, las estructuras secuencias, como: declaración de variables, asignación, lectura y escritura de datos e invocación de subrutinas se escriben en un rectángulo, como se muestra en la figura 7.

La estructura selectiva ***si – entonces – si no – fin si***, se representa colocando la condición en un triángulo invertido y éste a su vez en un rectángulo. El flujo de ejecución correspondiente a las alternativas de la decisión se coloca por izquierda y derecha respectivamente. La estructura selectiva se muestra en la figura 8.

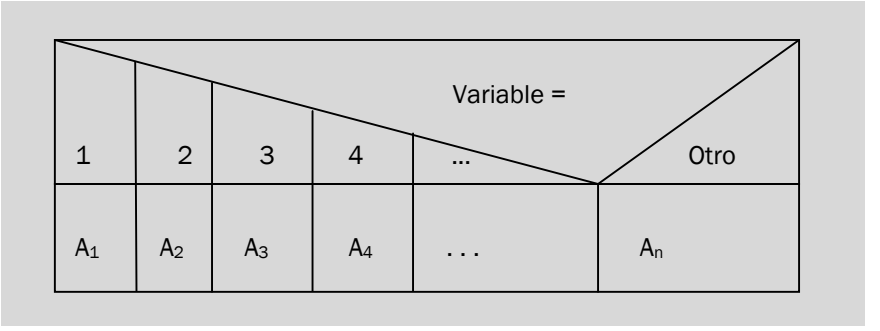
Figura 7. Estructura secuencia en notación N-S

Figura 8. Estructura selectiva en notación N-S



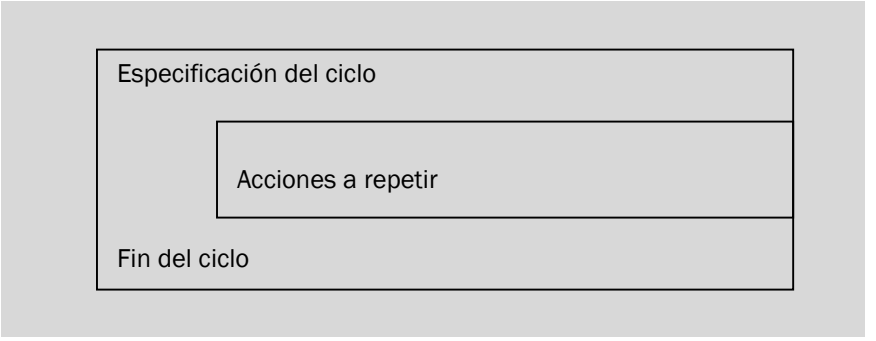
La estructura de decisión múltiple ***según sea*** se representa mediante un triángulo invertido dentro de un rectángulo y por cada posible valor de la variable se coloca un rectángulo hacia abajo. En la figura 9 se muestra la relación entre el valor de la variable y la acción que se ejecuta.

Figura 9. Estructura de selección múltiple en notación N-S



Las estructuras iterativas se representan escribiendo la especificación del ciclo en un rectángulo y las acciones que se repiten en un segundo rectángulo que se coloca dentro del primero. En el documento original los autores proponen tres formas de representar las estructuras iterativas: en el *ciclo mientras* el rectángulo interno se ubica en la parte inferior, el *ciclo para* se ubica al centro y para el ciclo *repita hasta*, en la parte superior, dejando espacio abajo para escribir la especificación del ciclo. En este libro, con la intensión de facilitar el diseño y la interpretación de los algoritmos mediante esta notación, se propone una sola representación, con el rectángulo interno al centro dejando espacio arriba y abajo para la especificación del ciclo, como se muestra en la figura 10.

Figura 10. Estructura iterativa en notación N-S



Como ejemplo de la notación Nassi - Shneiderman, en la figura 11 se presenta el algoritmo de Euclides, para obtener el MCD.

Figura 11. Algoritmo de Euclides en notación N-S

Inicio
Entero a, b, c = 1
Leer a, b
Mientras c > 0 hacer
c = a mod b
a = b
b = c
Fin mientras
Escribir “MCD =”, a
Fin algoritmo

3.3.5 Notación Funcional

En este caso, el término función indica una regla de correspondencia que asociación un elemento del conjunto dominio con otro del conjunto destino, como se entiende en matemáticas; y

no hace referencia a un subprograma como se entiende desde el enfoque de la programación imperativa.

Una función se especifica mediante un nombre, su dominio, rango y regla de correspondencia. Sea *f* una función y *x* un valor de su dominio, la notación *f(x)* representa el elemento *y* del rango que *f* asocia a *x*. Es decir, *f(x)* = *y*. La expresión *f(x)* se conoce como aplicación de la función (Galve *et al*, 1993).

Siendo *f(x,y)* la función Máximo Común Divisor de dos números enteros. En el cuadro 11 se muestra la solución al MCD de dos números aplicando el algoritmo de Euclides y representado en notación funcional.

Cuadro 11. Algoritmo de Euclides en notación funcional

$$f(x,y) = \begin{cases} y & \leftarrow \text{si } (x \; Mod \; y) = 0 \\ f(y, x \; Mod \; y) & \leftarrow si \; (x \; Mod \; y) > 0 \end{cases}$$

Las notaciones presentadas: pseudocódigo, diagrama de flujo, diagrama de Nassi – Shneiderman y funcional son las más importantes, pero hay muchas más. Sobre este tema Santos, Patiño y Carrasco(2006: 15) presentan, bajo el concepto de "Técnicas para el diseño de algoritmos", otras como: organigramas, ordinogramas, tablas de decisión, diagramas de transición de estados y diagramas de Jackson.

3.4 ESTRATEGIA PARA DISEÑAR ALGORITMOS

Para diseñar un algoritmo es primordial conocer el dominio del problema y tener claridad sobre lo que se espera que el algoritmo

haga. La fase de análisis debe proporcionar una descripción precisa de los datos con los que trabaja el algoritmo, las operaciones que debe desarrollar y los resultados esperados.

Un programa de computador desarrolla de forma automática una serie de operaciones que toman un conjunto de datos de entrada, realiza los procesos previamente definidos y genera uno o más resultados. Un algoritmo es la especificación de dicha serie de operaciones; es decir, la definición de las operaciones que posteriormente se realizarán de forma automática. En consecuencia, para diseñar un algoritmo es necesario conocer las operaciones que se requieren para obtener el resultado esperado.

En este orden de ideas, antes de intentar expresar la solución del problema en forma de algoritmo es conveniente tomar un conjunto de datos de entrada, realizar las operaciones y obtener los resultados de forma manual. Este ejercicio permitirá la identificación de todos los datos, separar los de entrada y los de salida y establecer las fórmulas para el desarrollo de los cálculos que permitirán obtener los resultados esperados.

Ejemplo 2. Número par o impar

Este ejemplo trata sobre aplicar la estrategia para diseñar un algoritmo para determinar si un número es par o impar.

Cómo se ha explicado, antes de intentar escribir un algoritmo es necesario pensar en cómo se logra solucionar éste problema para un caso particular, sin pensar en algoritmos ni programas.

Lo primero es establecer el concepto de número par: un número par es aquel que al dividirse para dos el resultado es un número entero y el residuo es cero.

Ahora, supóngase que el número en cuestión es el 10. ¿Cómo confirmar que es un número par?

Si se piensa un poco se encuentra que hay dos formas de hacerlo, las dos implican realizar una división entera sobre dos.

```
10 | 2
0  | 5
```

La primera consiste en tomar el cociente y multiplicarlo por dos, si el resultado es igual al número, al 10 en este caso, se comprueba que el número es par, en caso contrario es impar.

10 / 2 = 5

5 * 2 = 10 => confirmado, 10 es número par

La segunda alternativa es tomar el residuo, si éste es igual a cero, significa que la división entera es exacta, lo que comprueba que el número es par; por el contrario, si el residuo es igual a uno, el número es impar. Para obtener el residuo se utiliza el operador Mod presentado en el apartado 2.3.1.

10 Mod 2 = 0 => confirmado, 10 es número par

De igual manera se prueba con otro número, por ejemplo el 15.

```
15 | 2
1  | 7
```

Aplicando la primera alternativa se tiene que:

15 / 2 = 7

7 * 2 = 14 => 15 es número impar
Aplicando la segunda opción se tiene que:

15 Mod 2 = 1 => 15 es número impar

De estos dos ejemplos se deduce que para solucionar este problema se utiliza un dato de entrada, correspondiente al número que se desea evaluar, el proceso consiste en una división entera de la cual se toma el cociente o el residuo según preferencia del programador, y el resultado esperado es un mensaje: "número par" o "número impar".

Ahora, después de haber solucionado dos casos particulares de este problema, ya se puede proceder a expresar la solución en forma algorítmica, utilizando cualquiera de las notaciones explicadas en la sección 3.3. En el cuadro 12 se presenta la solución en pseudocódigo.

Cuadro 12. Pseudocódigo algoritmo número par/impar

```
1.  Inicio
2.      Entero: n, r
3.      Leer n
4.      r = n Mod 2
5.      Si r = 0 entonces
6.          Escribir n, "es número par"
7.      Si no
8.          Escribir n, "es número impar"
9.      Fin si
10. Fin algoritmo
```

En este algoritmo se declara dos variables, la primera para almacenar el número que se desea evaluar, la segunda para almacenar el residuo de la división entera (línea 4). Para evaluar el contenido de la variable r se utiliza la estructura selectiva **si** que se explica en la sección 4.2.2.

En las figuras 12 y 13 se presenta este mismo algoritmo mediante diagramas de flujo y N-S respectivamente.

Figura 12. Diagrama de flujo del algoritmo número par/impar

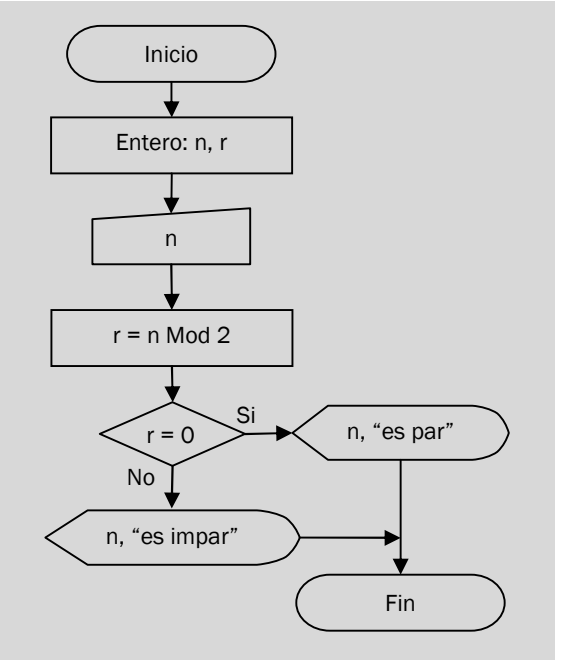
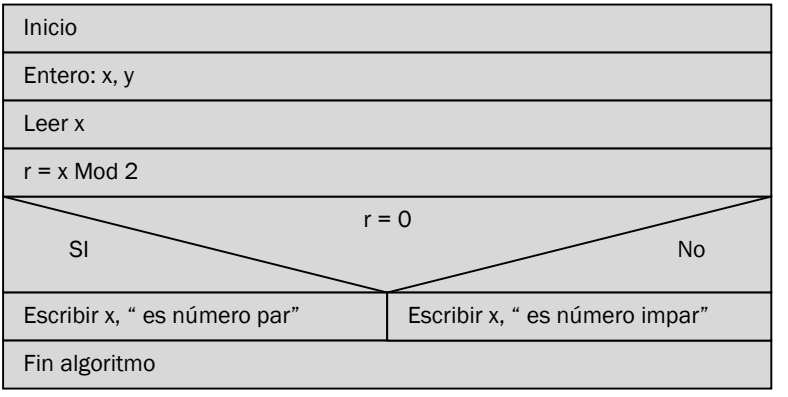


Figura 13. Diagrama de Nassi-Shneiderman del algoritmo número par/impar.



3.5 VERIFICACIÓN DE ALGORITMOS

Después de haber diseñado la solución de un problema mediante cualquiera de las formas vistas anteriormente, es necesario verificar que el algoritmo funciona adecuadamente; es decir, que tenga las características indispensables para ser considerado un buen algoritmo, como son: ser finito, definido, preciso, correcto y eficiente. La verificación del algoritmo también se conoce como prueba de escritorio.

Para verificar la corrección de un algoritmo se crea una tabla con una columna para cada variable o constante utilizada y algunas otras para registrar las ejecuciones, las iteraciones y las salidas, luego se ejecuta paso a paso y se registran en la tabla los valores que toman las variables. Al llegar al final del algoritmo se observará como se transforman los datos entrados o los valores iniciales para generar los datos de salida. Si el proceso es correcto y las salidas también, el algoritmo es correcto. Si alguno de los datos no es el esperado se revisa el diseño y se hacen los cambios necesarios hasta que funcione correctamente.

Aplicando lo explicado, se toma el algoritmo del ejemplo 2 y se verifica tres veces con diferentes números. En la primera ejecución se ingresa el número 6, la división sobre 2 da como residuo 0 y se obtiene como resultado el mensaje: “6 es número par”, lo cual es correcto; en la segunda, se ingresa el número 3, al hacer la división se obtiene como residuo 1 y como resultado el mensaje: “3 es número impar”; en la tercera ejecución se ingresa el número 10, el residuo de la división entera es 0 y el resultado: “10 es número par”.

En el cuadro 13 se presenta la prueba realizada al algoritmo del ejemplo 2, el cual lee un número y muestra un mensaje indicando si es par o impar.

Cuadro 13. Verificación del algoritmo número par/impar

Ejecución	n	R	Salida
1	6	0	6 es número par
2	3	1	3 es número impar
3	10	0	10 es número par

4. ESTRUCTURAS DE PROGRAMACIÓN

El saber que en teoría
un problema se puede resolver
con una computadora
no basta para decirnos
si resulta práctico hacerlo o no.
Baase y Van Gerder

En programación estructurada se utilizan tres tipos de estructuras: secuenciales, aquellas que se ejecutan una después de otra siguiendo el orden en que se han escrito; de decisión, que permiten omitir parte del código o seleccionar el flujo de ejecución de entre dos o más alternativas; y las iterativas, que se utilizan para repetir la ejecución de cierta parte del programa.

4.1 ESTRUCTURAS SECUENCIALES

Estas estructuras se caracterizan por ejecutarse una después de otra en el orden en que aparecen en el algoritmo o el programa, como la asignación de un dato a una variable, la lectura o la impresión de un dato.

4.1.1 Asignación

Esta operación consiste en guardar un dato en una posición determinada de la memoria reservada mediante la declaración de una variable.

La asignación es una operación relevante en el paradigma de programación imperativo, donde todos los datos, los leídos y los obtenidos como resultado de un cálculo, se guardan en memoria en espera de posteriores instrucciones.

Ejemplo:

entero a, b, c
cadena: operación
a = 10
b = 15
c = a + b
operación = "suma"

La asignación de variables se trata con mayor detalle en la sección 2.2.4.

4.1.2 Entrada de datos

Un programa actúa sobre un conjunto de datos suministrados por el usuario o tomados desde un dispositivo de almacenamiento; de igual manera, el algoritmo requiere que se le proporcione los datos a partir de los cuales obtendrá los resultados esperados.

La lectura de datos consiste en tomar datos desde un medio externo o desde un archivo y llevarlos a la memoria donde pueden ser procesados por el programa (Joyanes, 1988). El dispositivo predeterminado es el teclado.

Para la lectura de datos se cuenta con la instrucción *leer* en pseudocódigo y sus equivalentes en las diferentes notaciones.

En pseudocódigo, la instrucción *leer* tiene la siguiente sintaxis

Leer <lista de identificadores de variables>

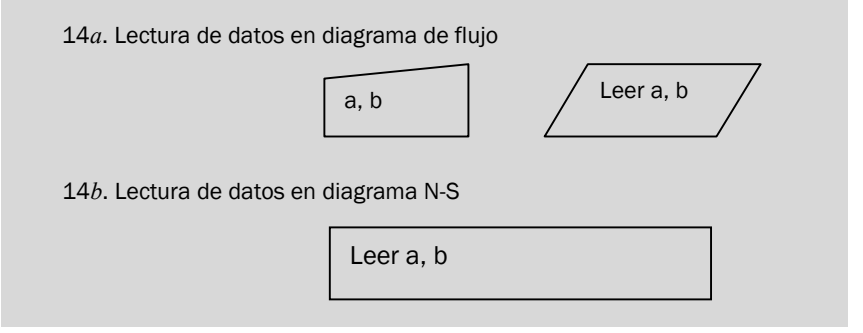
Ejemplo:

Leer a, b

Donde *a* y *b* son las variables que recibirán los valores y, por tanto, deben estar declaradas previamente.

En diagrama de flujo, la instrucción *leer* puede representarse de dos maneras: utilizando el símbolo de lectura por teclado o mediante un proceso de entrada y salida. En diagrama N-S, todas las estructuras secuenciales se escriben dentro de una caja, de manera que la lectura de datos se representa mediante la palabra *leer* dentro de una caja, como se muestran en la figura 14.

Figura 14. Símbolos de entrada de datos



Cualquiera de los dos símbolos de la figura 14*a* es válido para designar una lectura de datos en un diagrama de flujo. La figura 14b es exclusiva para diagramas N-S.

4.1.3 Salida de datos

Todo programa desarrolla una o más tareas y genera uno o más datos como resultado. Para presentar los resultados al usuario se hace uso de las instrucciones y los dispositivos de salida de datos, como la pantalla o la impresora.

Para enviar la información desde la memoria del computador hasta un dispositivo de salida se utiliza la instrucción definida en pseudocódigo como: *escribir* y sus equivalentes en las demás notaciones.

En pseudocódigo la lectura de datos se escribe de la forma:

Escribir <lista de constantes y variables>

Ejemplo:

Escribir a, b

Donde *a y b* son variables previamente definidas

Escribir "Este es un mensaje"

Cuando se escriben más de una variable es necesario separarlas con comas (,) y los mensajes se escriben entre comillas dobles " ". Si una variable es escrita entre comillas se mostrará el identificador y no el contenido.

Ejemplo:

Cadena: nombre
Entero: edad
Nombre = "Juan"
Edad = 25

La forma correcta de mostrar los datos es:

Escribir "Nombre: ", nombre
Escribir "Edad: ", edad

Para que la salida será:

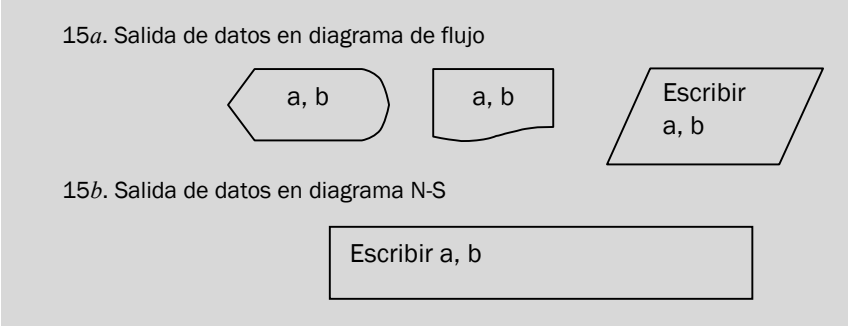
Nombre: Juan
Edad: 25

Escribir de la forma:
Escribir "nombre"
Escribir "edad"

Solo mostrará las etiquetas: "nombre" y "edad"

En diagrama de flujo la salida de datos puede representarse mediante tres símbolos: salida por pantalla, salida por impresora y proceso de entrada y salida (figura 15*a*), mientras que en diagrama N-S la instrucción *escribir* se coloca dentro de una caja, como se muestra en la figura 15*b*.

Figura 15. Símbolos de salida de datos



4.1.4 Ejemplos con estructuras secuenciales

Ejemplo 3. Sumar dos números

Este algoritmo lee dos números, los almacena en variables, luego los suma y muestra el resultado. Se trata de representar algorítmicamente el desarrollo de la operación sumar utilizando dos valores. Por ejemplo:

3 + 2 = 5

Si esta operación se la convierte en una expresión aritmética con variables se tiene:

$r = x + y$

Si se asigna o lee valores para las variables x e y se puede calcular el resultado. Ejemplo:

$x = 6$
$y = 7$

$r = x + y$
$r = 6 + 7$
$r = 13$

De este ejemplo se deduce que el algoritmo requiere que se le suministren dos datos, estos son los datos de entrada, que se almacenarán en las variables x e y. Con éstos se desarrolla la expresión (proceso) y el resultado es el dato que interesa al usuario, por tanto éste es el dato de salida.

Entrada: x, y
Proceso: r = x + y
Salida: r

Ahora que se comprende la solución general del problema, ésta se puede expresar de forma algorítmica. En el cuadro 14 se presenta el algoritmo en notación de pseudocódigo, en las figuras 16 y 17 los diagramas de flujo y N-S, respectivamente.

Cuadro 14. Pseudocódigo del algoritmo sumar dos números

```
1. Inicio
2.     Entero: x, y, r
3.     Leer x, y
4.     r = x + y
5. Fin algoritmo
```

Figura 16. Diagrama de flujo para sumar dos números

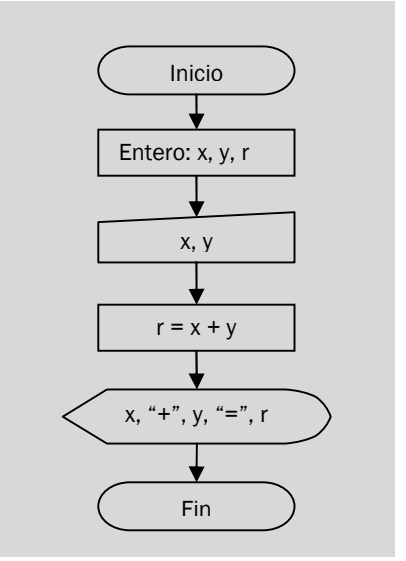


Figura 17. Diagrama N-S para sumar dos números

Inicio
Entero: x, y, r
Leer x, y
r = x + y
Escribir x, “ + “, y, “ = “, r
Fin algoritmo

En este algoritmo se declaran tres variables: las dos primeras (x, y) corresponden a los dos datos de entrada, la tercera (r) guarda el resultado del proceso. La salida para este ejercicio es, sin duda, el resultado de la operación; no obstante, en muchos programas es necesario mostrar también algunos de los datos entrados para que la salida sea más entendible, en este ejemplo se muestra los operandos y el resultado de la operación.

Para verificar si este algoritmo es correcto se crea una tabla como la que se presenta en el cuadro 15, se ejecuta el algoritmo línea por línea, proporcionando los datos de entrada y realizando las operaciones. Los datos se escriben en las columnas de la tabla etiquetadas con las variables y luego se verifica si la salida es correcta. (La verificación de algoritmos se explica en la sección 3.5).

Cuadro 15. Verificación del algoritmo para sumar dos números

Ejecución	X	y	r	Salida
1	3	4	7	3 + 4 = 7
2	5	8	13	5 + 8 = 13
3	2	3	5	2 + 3 = 5

Es importante tener presente que al verificar el algoritmo se debe ejecutar estrictamente los pasos de éste y no proceder de memoria, ya que muchas veces lo que se tiene en mente y lo que se escribe no es lo mismo y lo que se quiere verificar es lo que está escrito.

Ejemplo 4. El cuadrado de un número.

El cuadrado de un número es igual a multiplicar el número por sí mismo, de la forma:

$$3^2 = 3 * 3 = 9$$
$$5^2 = 5 * 5 = 25$$

Ahora, para generalizar la operación se utiliza una variable para el número que se desea elevar al cuadrado y se tiene una expresión de la forma:

$$num^2 = num * num = ?$$

Si llevamos este proceso a un algoritmo y luego a un programa, el computador podrá proporcionar el resultado para cualquier número. Ahora bien, antes de proceder a diseñar el algoritmo se

organiza la información que se tiene en forma de: entrada, salida y proceso.

Entrada: número
Salida: el cuadrado del número
*Proceso: cuadrado = número * número*

Con esta información se procede a diseñar la solución del problema. En el cuadro 16 se presenta el pseudocódigo, y en las figuras 18 y 19 los diagramas de flujo y N-S.

Cuadro 16. Pseudocódigo para calcular el cuadrado de un número

```
1.  Inicio
2.      Entero: num, cuad
3.      Leer num
4.      cuad = num * num
5.      Escribir cuad
6.  Fin algoritmo
```

En este algoritmo se declaran dos variables: *num* para almacenar el número ingresado por el usuario y *cuad* para almacenar el resultado del proceso, es decir, el cuadrado del número. En el cuadro 17 se presenta tres ejecuciones de prueba para verificar la corrección del algoritmo: en la primera se ingresa el número 3 y se obtiene como resultado el 9, en la segunda se ingresa el número 2 y se obtiene el cuatro y en la tercera se ingresa el 7 y el resultado es 49, estos resultados confirman que el algoritmo es correcto.

Cuadro 17. Verificación del algoritmo el cuadrado de un número

Ejecución	num	cuad	Salida
1	3	9	9
2	2	4	4
3	7	49	49

Figura 18. Diagrama de flujo para calcular el cuadrado de un número

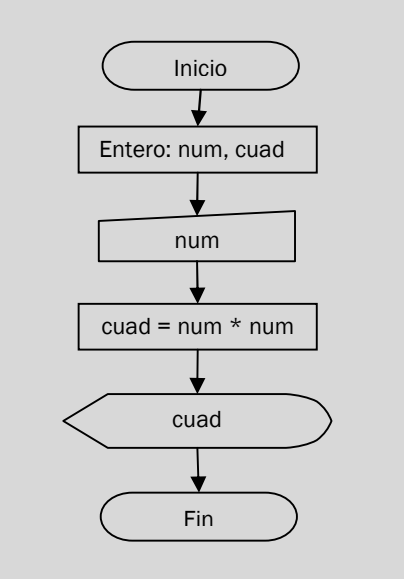


Figura 19. Diagrama N-S para calcular el cuadrado de un número

Inicio
Entero: num, cuad
Leer num
cuad = num * num
Escribir cuad
Fin algoritmo

Ejemplo 5. Precio de venta de un producto

Considérese el siguiente caso: una tienda de licores adquiere sus productos por cajas y los vende por unidades. Dado que una caja de cualquier producto tiene un costo c y contiene n unidades,

se desea calcular el precio p para cada unidad, de manera que se obtenga el 30% de utilidad.

Lo primero que hay que hacer es comprender bien el dominio del problema, esto es, poder obtener los resultados y solucionar el problema de forma manual.

Ejemplo 1: se compra una caja de ron en $ 240.000.oo (c), ésta contiene 24 unidades (n) y se desea obtener el 30% de utilidad. ¿A qué precio (p) se debe vender cada unidad?

Para solucionar este problema es necesario realizar tres cálculos, primero, aplicar el porcentaje de utilidad sobre el costo de la caja; segundo, sumar el costo más la utilidad para obtener el precio total de la caja; y tercero, dividir el valor total sobre el número de unidades (n) para obtener el precio unitario (p).

Primero:

*utilidad = 240.000.oo * 30/100*
utilidad = $ 72.000.oo

Segundo:

total = $ 240.000.oo + $ 72.000.oo
total = $ 312.000.oo

Tercero:

p = $ 312.000.oo / 24
p = $ 13.000.oo

Finalmente tenemos el precio de venta de cada unidad: $ 13.000.oo

Ejemplo 2: supóngase que se compra otro producto, cuyo costo por caja es de $ 600.000.oo y contiene 12 unidades. Entonces se tiene que:

Primero:

*utilidad = $ 600.000.oo * 30/100*
utilidad = $ 180.000.oo

Segundo:

total = $ 600.000.oo + $ 180.000.oo
total = $ 780.000.oo

Tercero:

p = $ 780.000.oo / 12
p = $ 65.000.oo

El precio de venta de cada unidad es de $ 65.000.oo

Con estos dos ejemplos se logra comprender cómo se soluciona el problema, ahora es necesario generalizar el proceso de solución de manera que se pueda especificar un algoritmo; para ello hay que representar los valores mediante variables y los cálculos con fórmulas que se aplican sobre dichas variables, así:

Sea:

c = costo de la caja
n = número de unidades
p = precio de venta por unidad

De estos datos: los dos primeros son entradas, el último es la salida. Los cálculos a realizar son:

*utilidad = c * 30/100*
total = c + utilidad
p = total / n

Habiendo llegado a este punto ya se puede proceder a diseñar un algoritmo para solucionar cualquier caso de este problema. En el

cuadro 18 se muestra el pseudocódigo, en la figura 20 el diagrama de flujo y en la 21 el diagrama N-S.

Cuadro 18. Pseudocódigo para calcular el precio unitario de un producto

```
1.  Inicio
2.      Entero: n
        Real: c, utilidad, total, p
3.      Leer c, n
4.      utilidad = c * 30/100
5.      total = c + utilidad
6.      p = total / n
7.      Escribir "Precio = ", p
8.  Fin algoritmo
```

Figura 20. Diagrama de flujo para calcular el precio unitario de un producto

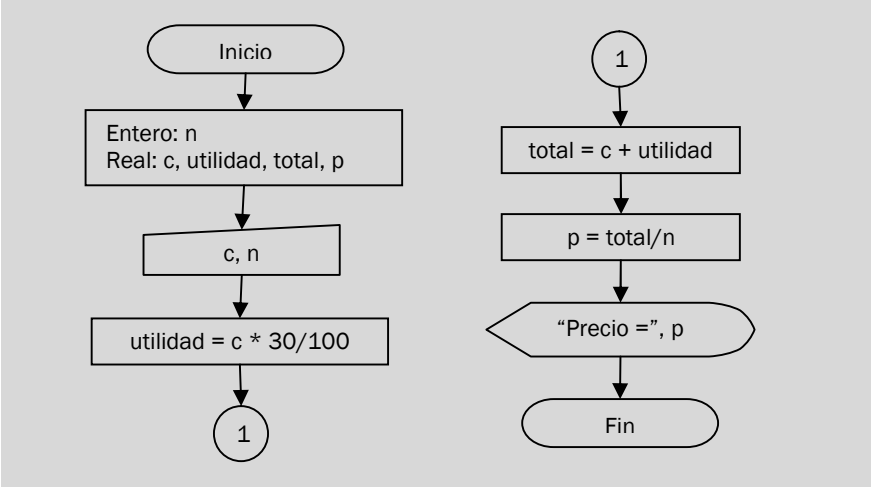


Figura 21. Diagrama N-S para calcular el precio unitario de un producto

Inicio
Entero: n Real: c, utilidad, total, p
Leer c, n
utilidad = c * 30/100
total = c + utilidad
p = total / n
Escribir “Precio = ”, p
Fin algoritmo

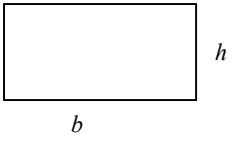
En el cuadro 19 se presentan los datos de tres pruebas efectuadas para verificar la corrección de este algoritmo, las dos primeras columnas corresponden a los datos de entrada y la última a la salida que genera el algoritmo.

Cuadro 19. Verificación del algoritmo para calcular el precio unitario de un producto

C	n	Utilidad	total	P	Salida
240.000	24	72.000	312.000	13.000	Precio = 13.000
600.000	12	180.000	780.000	65.000	Precio = 65.000
300.000	10	90.000	390.000	39.000	Precio = 39.000

Ejemplo 6. Área y perímetro de un rectángulo

Se requiere un algoritmo para calcular el área y el perímetro de un rectángulo de cualquier dimensión.



Para solucionar este problema es necesario conocer las fórmulas para obtener tanto el área como el perímetro de un rectángulo.

Siendo:

b = base
h = altura

Las fórmulas a utilizar son:

*Área = b * h*
*Perímetro = 2 * (b + h)*

Si el rectángulo tiene una base de 10 cm y una altura de 5 cm, se obtiene:

*Área = 10 * 5*
Área = 50

*Perímetro = 2 * (10 + 5)*
Perímetro = 30

Cómo se aprecia en el ejemplo, para hacer los cálculos es necesario contar con la base y la altura, de manera que éstos corresponden a los datos que el usuario debe suministrar y que el algoritmo ha de leer, mientras que el área y el perímetro son los datos que el algoritmo debe calcular y presentar al usuario; por lo tanto:

Datos de entrada: base (b) y altura (h)
Datos de salidas: área (a) y perímetro (p)
Procesos:
*a = b * h*
*p = 2 * (b + h)*

El diseño de la solución en notación de pseudocódigo se presenta en el cuadro 20.

Cuadro 20. Pseudocódigo para calcular el área y perímetro de un rectángulo

```
1.  Inicio
2.      Entero: b, h, a, p
3.      Leer b, h
4.      a = b * h
5.      p = 2 (b + h)
6.      Escribir "área:", a
7.      Escribir "perímetro:", p
8.  Fin algoritmo
```

Para verificar que el algoritmo es correcto se realiza la prueba de escritorio, paso a paso, así:

Paso 2. Se declaran variables: b, h, a, p correspondientes a: base, altura, área y perímetro respectivamente.

Paso 3. Se lee desde teclado base y altura y se almacena en las variables b y h, supóngase los valores 5 y 8

Paso 4. Se calcula el área, 5 * 8 = 40 (a = b * h) y se almacena en la variable a

Paso 5. Se calcula el perímetro, 2 * (5 + 8) = 26 y se guarda en la variable p

Pasos 6 y 7. Se muestra el contenido de las variables a y p con su respectivo mensaje.

En el cuadro 21 se presentan los resultados de la verificación con tres conjuntos de datos.

Cuadro 21. Verificación del algoritmo área y perímetro de un rectángulo

Ejecución	b	h	a	p	Salida
1	5	8	40	26	área: 40 perímetro: 26
2	3	4	12	14	área: 12 perímetro: 14
3	8	4	32	24	área: 32 perímetro: 24

Ejemplo 7. Tiempo dedicado a una asignatura

Se sabe que un profesor universitario contratado como hora cátedra dedica una hora a preparar una clase de dos horas y por cada cuatro horas de clase desarrolladas hace una evaluación, la cual le toma dos horas calificar. Si al docente se le asigna un curso de *n* horas al semestre, ¿cuántas horas trabajará en total?

La mejor forma de comprender un problema es tomar un caso particular y desarrollarlo. Supóngase que el profesor ha sido contratado para el curso de matemáticas cuyo número de horas (nh) es de 64 al semestre.

Si para cada par de horas de clase dedica una hora a su preparación se tiene que el tiempo de preparación (tp) es:

tp = 64 / 2
tp = 32 horas

En general: tp = nh/2

Se sabe también que hace una evaluación por cada cuatro horas de clase, de manera que se puede calcular el número de evaluaciones (ne) que realizará durante el semestre:

ne = 64 / 4

ne = 16

Para cualquier caso: ne = nh / 4

Ahora, si para calificar cada evaluación requiere dos horas, el tiempo dedicado en el semestre a calificar (tc) es:

tc = 16 * 2
tc = 32

Utilizando variables: tc = ne * 2

Finalmente, para obtener el tiempo total (tt) que el docente dedica al curso de matemáticas sólo hay que sumar el tiempo de preparación, el tiempo de desarrollo de las clases y el tiempo de calificación de exámenes, así:

tt = 32 + 64 + 32
tt = 128

Esto es: tt = tp + nh + tc

De esta manera se obtiene que para un curso de 64 horas de clase, el docente dedica 128 horas de trabajo. Como se quiere calcular este valor para cualquier curso es necesario diseñar un algoritmo, al cual se le proporciona el número de horas de clase semestral y el devolverá el total de horas dedicadas.

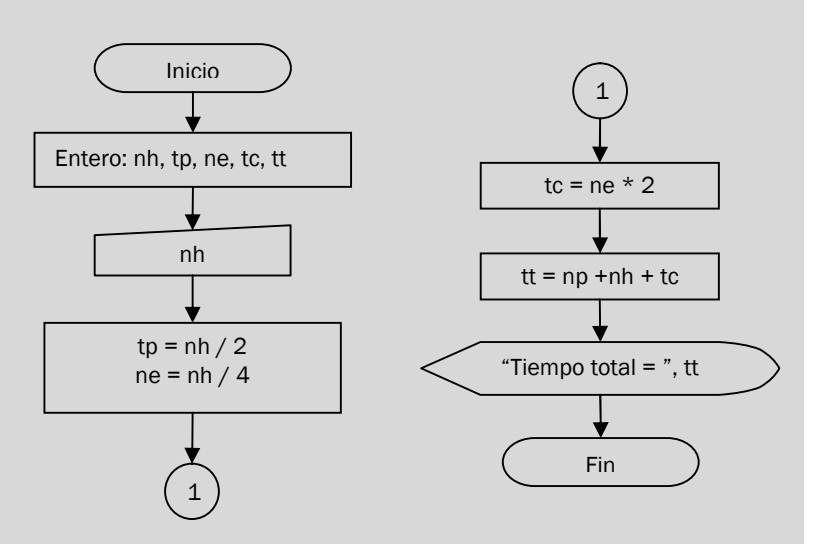
De lo anterior se deduce que:

Entrada: número de horas de clase (nh)
Salida: tiempo total dedicado (tt)
Cálculos a realizar:
tp = nh/2
ne = nh / 4
tc = ne * 2
tt = tp + nh + tc

El algoritmo, en notación de diagrama de flujo se presenta en la figura 22.

Para tener la seguridad de que el algoritmo realiza los cálculos correctamente se realiza la prueba con tres casos diferentes, los resultados se presentan en el cuadro 22.

Figura 22. Diagrama de flujo para calcular el tiempo dedicada a una asignatura



Cuadro 22. Verificación del algoritmo tiempo dedicado a una asignatura

Ejec.	nh	tp	ne	tc	tt	Salida
1	64	32	16	32	128	Tiempo total = 128
2	80	40	20	40	160	Tiempo total = 160
3	96	48	24	48	192	Tiempo total = 192

4.1.5 Ejercicios propuestos

Diseñar los algoritmos para solucionar los siguientes problemas y representarlos mediante pseudocódigo, diagrama de flujo o diagrama N-S.

1. Leer tres notas y calcular el promedio

2. Calcular el área de un triángulo

3. Calcular el área y el perímetro de un círculo

4. Calcular el volumen y la superficie de un cilindro

5. Dado el ancho, largo y alto de una caja, calcular el volumen y la cantidad de papel (en cm^2) necesario para cubrirla.

6. Leer un número entero y separar sus dígitos en: miles, centenas, decenas y unidades.

7. Calcular el interés y el valor futuro de una inversión con interés simple.

8. Calcular el interés y valor futuro de una inversión con interés compuesto.

9. Si la universidad financia las matrículas de los estudiantes a cuatro cuotas mensuales iguales con un interés del 2% sobre el saldo. Dado el valor de matrícula y el número de cuotas, un estudiante desea saber ¿Cuál será el valor de cada cuota? ¿Cuánto pagará en total?

10. Un vendedor recibe un sueldo base más el 10% de comisión sobre sus ventas. Si en un mes cualquiera hace tres ventas por valores: v1, v2 y v3, ¿cuánto recibirá por comisión? y ¿cuánto en total?

11. Un cliente de un supermercado adquiere n productos a precio unitario p. Si por temporada el producto tiene un descuento del 10% que se hace efectivo en caja ¿cuál es el valor del descuento? ¿cuánto deberá pagar?

12. Un estudiante desea saber cuál será su calificación final en Programación. Dicha calificación se compone del promedio de tres notas parciales. Cada nota parcial se obtiene a partir de un taller, una evaluación teórica y una evaluación práctica. Los talleres equivalen al 25% de la nota del parcial, las evaluaciones teóricas al 35% y las evaluaciones prácticas al 40%.

13. Un viajero desea conocer cuántos dólares obtendrá por su capital en pesos.

14. Facturar el servicio de electricidad. El consumo mensual se determina por diferencia de lecturas.

15. Las utilidades de una empresa se distribuyen entre tres socios así: socio A = 40%, socio B = 25% y socio C = 35%. Dada una cantidad de dinero ¿cuánto corresponderá a cada uno?

16. El dueño de una tienda compra un artículo por x pesos y desea obtener el 30% de utilidad. ¿Cuál es el precio de venta del artículo?

17. Un almacén tiene como política obtener el 30% de utilidad sobre cada artículo y por temporada ofrece un descuento del 15% sobre el precio de venta en todos sus productos. Dado el costo de un producto calcular el precio de venta, de manera que pueda efectuar el descuento sobre el precio de venta y obtener el 30% de utilidad sobre el costo.

18. Tres personas deciden invertir su dinero para fundar una empresa. Cada una de ellas invierte una cantidad distinta.

Obtener el porcentaje que cada cual invierte con respecto a la cantidad total invertida.

19. Calcular el sueldo de un empleado que ha trabajado n horas extras diurnas y m horas extras nocturnas, siendo que cada hora extra diurna tiene un incremento del 25% sobre el valor de una hora corriente y cada hora extra nocturna un incremento del 35%.

20. Un estudiante desea saber la nota mínima que deberá obtener en la evaluación final de cálculo después de conocer las notas de los dos parciales, sabiendo que la materia se aprueba con 3.0 y la nota definitiva se obtiene de la siguiente manera: 30% para cada uno de los parciales y 40% para el final.

21. Dado el valor del capital de un préstamo, el tiempo y el valor pagado por concepto de interés, calcular la tasa de interés que se aplicó.

22. Conociendo el tiempo que un atleta tarda en dar una vuelta al estadio (400 m) se requiere estimar el tiempo que tardará en recorrer los 12 km. establecido para una competencia.

23. Resolver una ecuación cuadrática ($aX^2 + bX + c = 0$) teniendo en cuenta que a, b y c son valores enteros y pueden ser positivos o negativos.

24. Dado el valor que un cliente paga por un producto, calcular qué valor corresponde al costo del producto y cuánto al IVA. Considerando que el porcentaje del IVA puede variar en el tiempo y de un producto a otro, este dato se lee por teclado.

25. Un profesor diseña un cuestionario con n preguntas, estima que para calificar cada pregunta requiere m minutos. Si el cuestionario se aplica a x estudiantes, cuánto tiempo (horas y minutos) necesita para calificar todos los exámenes.

4.2 ESTRUCTURAS DE DECISIÓN

En la programación, como en la vida real, es necesario evaluar las circunstancias y actuar en consecuencia, pues no siempre se puede establecer por anticipado el curso de las acciones. En la ejecución de un programa, muchas expresiones estarán supeditadas al cumplimiento de determinadas condiciones; por ello, el programador ha de identificar estas situaciones y especificar, en el programa, qué hacer en cada caso (Timarán *et al*, 2009).

Todo problema, tanto en programación como en otros ámbitos, incluye un conjunto de variables que pueden tomar diferentes valores. La presencia o ausencia de determinadas variables, así como su comportamiento, requieren acciones diferentes en la solución. Esto hace que los programas no sean secuencias simples de instrucciones, sino estructuras complejas que implementan saltos y bifurcaciones según el valor de las variables y las condiciones definidas sobre éstas

Por ejemplo, para realizar una división se requieren dos variables: dividendo y divisor, que pueden contener cualquier número entero o real. No obstante, la división sobre cero no está determinada y por tanto se tiene como condición que el divisor sea diferente de cero. Al implementar esta operación en un programa se debe verificar el cumplimiento de esta restricción y decidir si se realiza el cálculo o se muestra un mensaje de error. Si no se implementa esta decisión y al ejecutarse el programa la variable toma el valor cero se presentará un error de cálculo y la ejecución se interrumpirá.

Implementar una decisión implica evaluar una o más variables para determinar si cumple una o más condiciones y establecer un flujo de acciones para cada resultado posible. Las acciones pueden consistir en ejecutar una o más expresiones o en omitirlas, o en seleccionar una secuencia de instrucciones de entre dos o más alternativas disponibles.

4.2.1 Condición

En esta sección se menciona repetidas veces el término condición, de manera que parece oportuno hacer mayor claridad sobre el mismo. Una condición es una expresión relacional o lógica que puede o no cumplirse dependiendo de los valores de las variables involucradas en la expresión. Cuando se utiliza una expresión relacional, la condición es una comparación entre dos variables del mismo tipo o una variable y una constante, mientras que cuando se utiliza una expresión lógica se tiene una o más expresiones relacionales combinadas con operadores lógicos: *y, o, no*; cuyo resultado depende de la tabla de verdad del correspondiente operador.

Los siguientes son ejemplos de condiciones que contienen expresiones relacionales:

a. $x = 5$
b. $y <= 10$
c. $x > y$

Al escribir el programa se especifica la condición mediante expresiones como estas, pero al ejecutarlo se evalúan y el resultado puede ser verdadero o falso dependiendo del valor de las variables. Supóngase que:

x = 8
y = 9

En tal caso se tendría que la condición *a* no se cumple por ende genera un resultado *falso*, la *b* si se cumple generando así un resultado *verdadero* y la *c* también devuelve *falso*.

Obsérvese algunos ejemplos de condiciones formadas por expresiones lógicas:

a. $x > 0$ ***Y*** $x < 10$

b. $x == 5$ ***O*** $y == 9$
c. ***NO****(x>y)*

La condición *a* devuelve *verdadero* si las dos expresiones relacionales son verdaderas; la condición *b* devuelve *verdadero* si al menos una de las expresiones relacionales que la conforman es verdadera y la *c* niega el resultado de la expresión relacional que está entre paréntesis. Si las variables tiene los valores indicados anteriormente ($x = 8, y = 9$) la condición *a* es *verdadera*, la *b* es *verdadera* y la *c* es *verdadera*.

4.2.2 Tipos de decisiones

Las decisiones pueden ser de tres clases: simples, dobles y múltiples.

Decisión simple: consiste en decidir si ejecutar u omitir una instrucción o un conjunto de instrucciones. En este caso se determina qué debe hacer el programa si la condición es verdadera, pero si no lo es, simplemente se pasa el control a la instrucción que está después de la estructura de decisión.

Un ejemplo de decisión simple se presenta cuando se calcula el valor absoluto de un número. El valor absoluto de un número es el mismo número, pero si éste es negativo es necesario multiplicarlo por -1 para cambiar de signo. Nótese que esta operación sólo es necesaria en el caso de los números negativos. Para incluir decisiones simples en un algoritmo se utiliza la sentencia *Si*.

Decisión doble: se presenta cuando se tienen dos alternativas de ejecución y dependiendo del resultado de la evaluación de la condición se ejecuta la una o la otra. Si la condición es verdadera se ejecuta la primera instrucción o bloque de instrucciones y si es falsa, la segunda.

Como ejemplo de este tipo de decisión se puede tomar el caso en que se evalúa una nota para decidir si un estudiante aprueba o

reprueba una asignatura, dependiendo si la nota es mayor o igual a 3.0.

Decisión múltiple: consiste en evaluar el contenido de una variable y dependiendo del valor de ésta ejecutar una secuencia de acciones. En este caso, el conjunto de valores para la variable debe ser mayor a dos y solo se evalúa la condición de igualdad. Cada posible valor está asociado a una secuencia de ejecución y éstas son mutuamente excluyentes. Este tipo de decisiones se implementan haciendo uso de la sentencia *Según sea*.

Como ejemplo de decisión múltiple considérese el caso de un algoritmo que permite ejecutar cualquiera de las operaciones aritméticas básicas: suma, resta, multiplicación y división, sobre un conjunto de números. Al ejecutarse, el usuario escoge la operación y el algoritmo realiza la operación correspondiente.

4.2.3 Estructura SI

Esta instrucción evalúa un valor lógico, una expresión relacional o una expresión lógica y retorna un valor lógico, con base en éste se toma una decisión simple o doble.

Decisión simple

Como decisión simple su sintaxis en pseudocódigo es:

SI <condición> ENTONCES
    Instrucciones
FIN SI

En las figuras 23 y 24 se presenta su representación en diagrama de flujo y N-S, respectivamente.

Figura 23. Decisión simple en notación diagrama de flujo

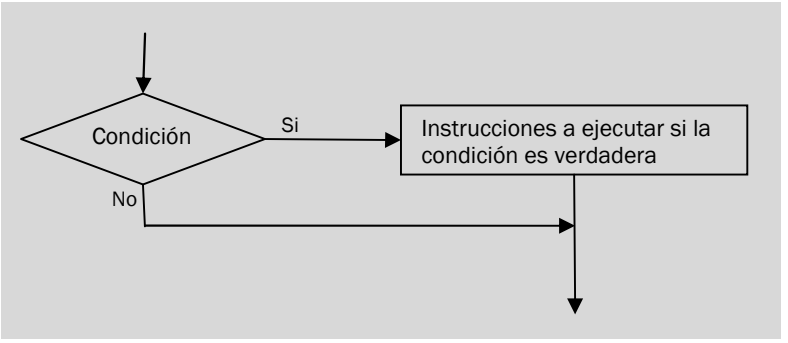
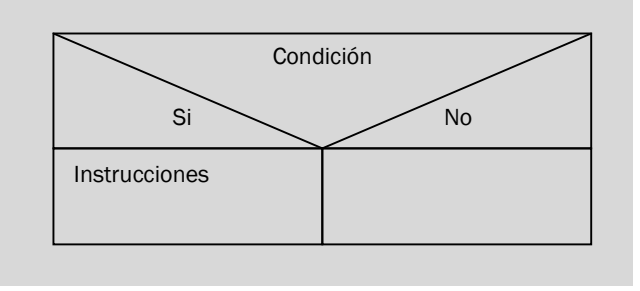


Figura 24. Decisión simple en notación diagrama N-S



Ejemplo 8. Calcular el valor absoluto de un número

El valor absoluto de un número es el mismo número para el caso de los positivos y el número cambiado de signo para los negativos; es decir, el valor absoluto es la distancia que hay desde el 0 al número y como las distancias no pueden ser negativas, éste siempre será positivo.

|5| = 5
|-3| = 3

En el cuadro 23 se presenta el pseudocódigo de este ejercicio.

Cuadro 23. Pseudocódigo para calcular el valor absoluto de un número

```
1.  Inicio
2.      Entero: num
3.      Leer num
4.      Si num < 0 entonces
5.          num = num * -1
6.      Fin si
7.      Escribir "Valor absoluto = ", num
8.  Fin algoritmo
```

La estructura de decisión se aplica en la línea 4 para determinar si el número es negativo, si la evaluación de la condición da un resultado verdadero se ejecuta la línea 5, si da un resultado falso termina la estructura *Si* y se ejecuta la instrucción que está después de *Fin si*, en la línea 7.

Si al ejecutar este algoritmo se introduce el número 5 se obtiene que, al llegar a la decisión, la condición no se cumple y por tanto se procede a escribir el mismo número. Si en una segunda ejecución se introduce el número -3, al evaluar la condición ésta genera como resultado *verdadero*, por tanto se ejecuta la operación que está dentro de la decisión: multiplicar el número por -1 y finalmente se muestra el número 3. Estos resultados se aprecian en el cuadro 24.

Cuadro 24. Verificación del algoritmo para calcular valor absoluto

Ejecución	num	Salida
1	5	Valor absoluto = 5
2	-3	Valor absoluto = 3

En la figura 25 se presenta el diagramas de flujo y en la 26 el diagrama N-S.

Figura 25. Diagrama de flujo de valor absoluto de un número

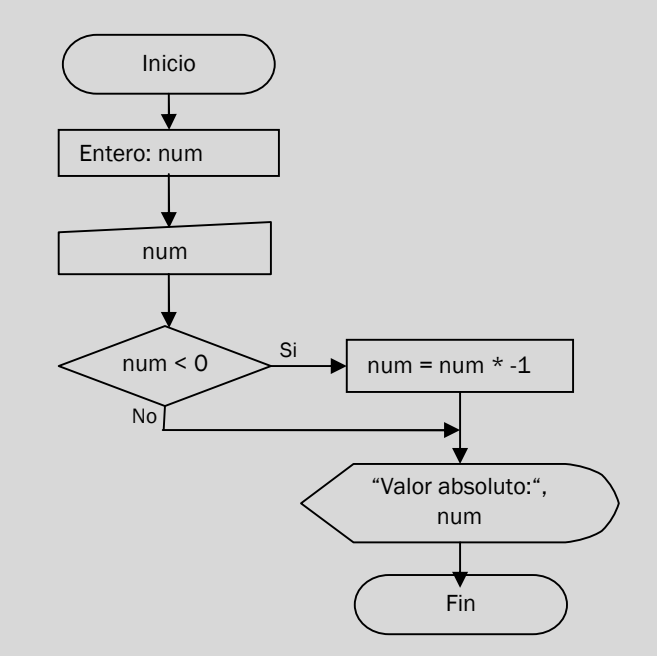
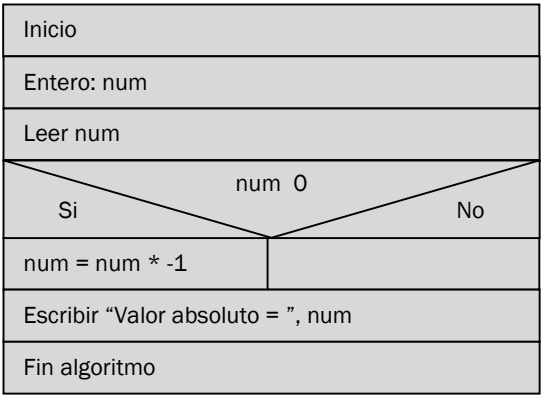


Figura 26. Diagrama N-S para calcular el valor absoluto de un número



Decisión doble

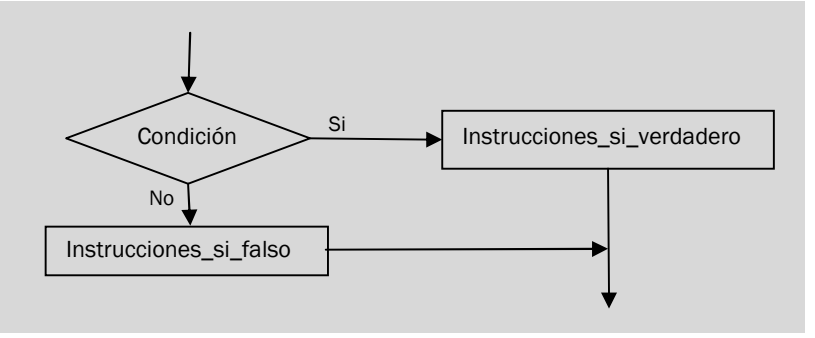
Se implementa una decisión doble cuando se cuenta con dos opciones para continuar la ejecución del algoritmo y éstas dependen de una condición; es decir, que se cuenta con una o más instrucciones para el caso que la condición sea verdadera y con otra u otro conjunto de instrucciones para el caso de que sea falsa.

Las decisiones simples se utilizan para incluir saltos en la ejecución del algoritmo o del programa, las decisiones dobles permiten hacer bifurcaciones. Su sintaxis en pseudocódigo es:

SI <condición> ENTONCES
Instrucciones_si_verdadero
SI NO
Instrucciones_si_falso
FIN SI

En las figuras 27 y 28 se presenta su representación en diagrama de flujo y N-S, respectivamente.

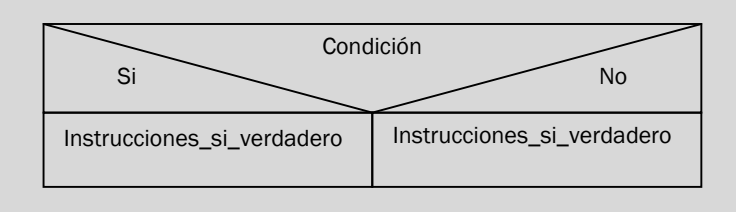
Figura 27. Decisión doble en notación diagrama de flujo



Observe que en pseudocódigo se escribe la instrucción *Fin Si* para indicar hasta donde se extiende la estructura condicional. En diagrama de flujo, el fin del condicional está determinado por la

unión de los dos caminos, marcados como *Si* y *No*. En el diagrama N-S la estructura condicional tiene dos bloques, en el de la izquierda se escriben las instrucciones que deben ejecutarse cuando la condición se cumple y en el de la derecha se colocan las que deben ejecutarse cuando la condición no se cumple, el fin del condicional estará marcado por la terminación de los dos bloques y la continuación en una sola caja.

Figura 28. Decisión simple en notación diagrama N-S



Ejemplo 9. División

Como se mencionó, antes de ejecutar una división es necesario verificar que el divisor sea diferente de cero, ya que en caso contrario se generará un error de cómputo y el programa se cancelará. En consecuencia, es necesario utilizar la sentencia *Si* para decidir si se ejecuta la división o se muestra un mensaje de error.

El algoritmo para realizar una división, representado mediante pseudocódigo, se presenta en el cuadro 25. En la línea 4 se especifica la condición que debe cumplirse para proceder a efectuar la división, en caso de que la condición no se cumpla la ejecución salta a la cláusula *Si no*, en la línea 7 y se ejecuta la línea 8, mostrándose un mensaje de error.

Para probar si el algoritmo funciona correctamente se ejecuta paso a paso, se introduce un par de números, se verifica el

cumplimiento de la condición y la ruta que toma dependiendo de ésta. En el cuadro 26 se presentan tres pruebas.

Cuadro 25. Pseudocódigo para realizar una división

```
1.  Inicio
2.      Entero: dividendo, divisor, cociente
3.      Leer dividendo, divisor
4.      Si divisor != 0 entonces
5.          cociente = dividendo / divisor
6.          Escribir cociente
7.      Si no
8.          Escribir "Error, divisor = 0"
9.      Fin si
10. Fin algoritmo
```

Cuadro 26. Verificación del algoritmo para hacer una división

Ejec.	Dividendo	divisor	cociente	Salida
1	15	3	5	5
2	8	0		Error, divisor = 0
3	12	3	4	4

La solución de este ejercicio en notación de diagrama de flujo se presenta en la figura 29 y en diagrama N-S en la figura 30.

Ejemplo 10. Número mayor y número menor

Este ejercicio consiste en leer dos números enteros diferentes y decidir cuál de ellos es el mayor y cuál es el menor.

Para solucionar este ejercicio es necesario declarar dos variables (x, y), introducir los números y guardarlos en ellas, luego decidir si x es mayor que y o lo contrario.

Figura 29. Diagrama de flujo para hacer una división

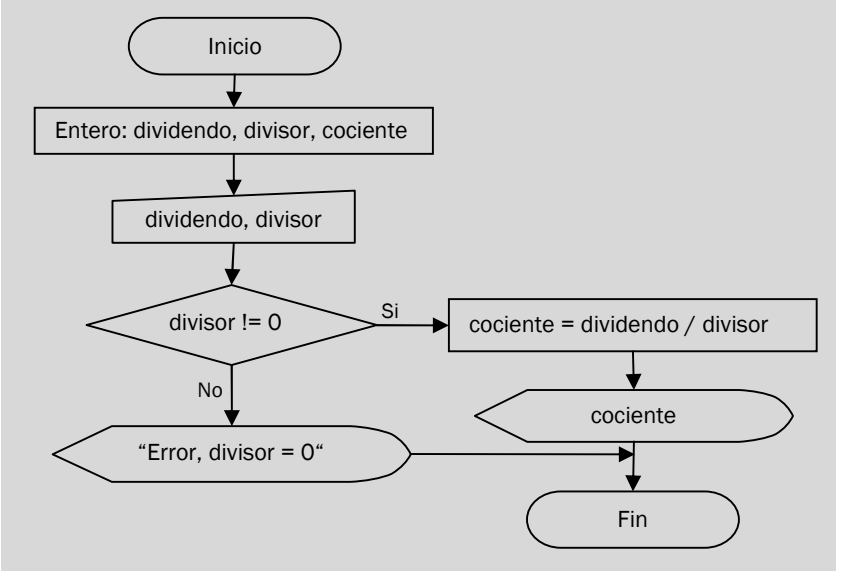
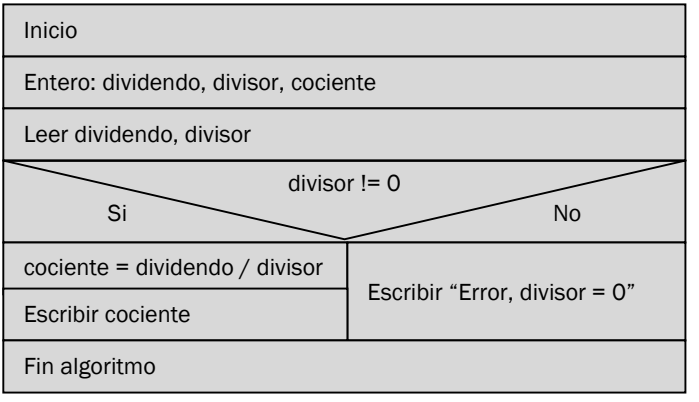


Figura 30. Diagrama N-S para hacer una división



El algoritmo del ejemplo 10, en notación de pseudocódigo, se presenta en el cuadro 27.

Cuadro 27. Pseudocódigo para identificar número mayor y menor

```
1.  Inicio
2.      Entero: x, y
3.      Leer x, y
4.      Si x > y entonces
5.          Escribir "Número mayor: ", x
6.          Escribir "Número menor", y
7.      Si no
8.          Escribir "Número mayor: ", y
9.          Escribir "Número menor", x
10.     Fin si
11. Fin algoritmo
```

En el cuadro 28 se muestra el comportamiento del algoritmo al ser probado con dos conjuntos de datos.

Cuadro 28. Verificación del algoritmo número mayor y menor

x	y	x > y	Salida
23	12	Verdadero	Número mayor: 23 Número menor: 12
5	11	Falso	Número mayor: 11 Número menor: 5

Encontrará más ejemplos desarrollados en la sección 4.2.6.

4.2.4 Estructura SEGÚN SEA

Muchas decisiones deben tomarse no solo entre dos alternativas sino de un conjunto mayor. Estos casos bien pueden solucionarse utilizando decisiones dobles anidadas; sin embargo, en favor de la

claridad del algoritmo y la facilidad para el programador, es mejor utilizar una estructura de decisión múltiple, la cual es fácil de pasar a un lenguaje de programación, ya que éstos incluyen alguna instrucción con este fin.

La instrucción *Según sea* determina el valor de una variable y dependiendo de éste sigue un curso de acción. Es importante tener en cuenta que solo se verifica la condición de igualdad entre la variable y la constante. Con base en Joyanes (1996) se propone la siguiente sintaxis:

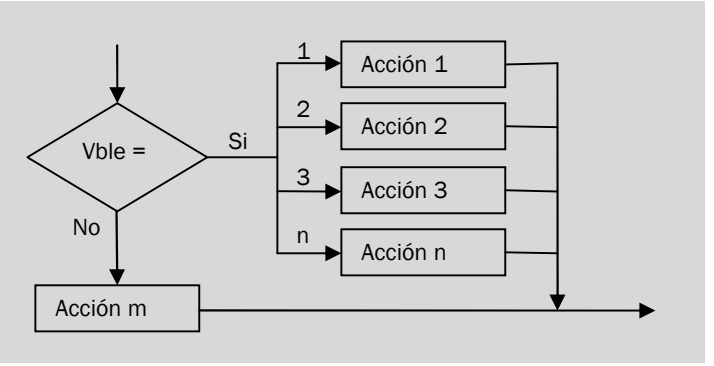
Según sea *<variable >* ***hacer***
    Valor_1:
        Acción 1
    Valor_2:
        Acción 2
    Valor_3:
        Acción 3
    Valor_n
        Acción n
    Si no
        Acción m
Fin según sea

Dado que se evalúa la igualdad entre la variable y la constante, todas las alternativas son mutuamente excluyentes; eso significa que sólo se podrá ejecutar un conjunto de acciones. Si ninguna de las alternativas se cumple, por no presentarse ninguna igualdad, se ejecutará el grupo de acciones asociadas con la cláusula *si no*. Esta última es opcional, en caso de no aparecer y no cumplirse ninguno de los casos, simplemente la ejecución del algoritmo continuará en la línea siguiente a *fin según sea*.

En diagrama de flujo la decisión múltiple se representa mediante un rombo en el que se coloca la variable y dos salidas, una de ellas se marca con la palabra *Si* y se subdivide según los posibles valores de la variable, los mismos que se colocan junto al flujo que les corresponde; en la otra salida se escribe la palabra *No* y las

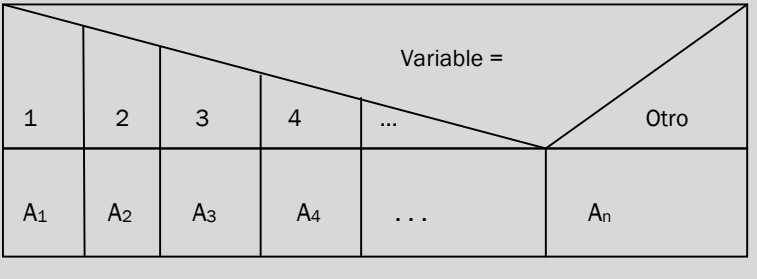
acciones a realizar en caso de que el valor de la variable no coincida con ninguno de los valores establecidos, como se muestra en la figura 31.

Figura 31. Decisión múltiple en notación diagrama de flujo



En diagrama N-S se representa mediante una caja con un triángulo invertido en el que se coloca la variable y el signo igual, y se divide la caja en tantas sub-cajas como alternativas tenga la decisión, incluyendo una caja para el caso en que la variable tome un valor diferente a los contemplados en las opciones, esta se coloca bajo el rótulo *Otro*, como se muestra en la figura 32.

Figura 32. Decisión múltiple en notación N-S



Ejemplo 11. Número romano

Este algoritmo lee un número arábigo y muestra su equivalente en romano, pero sólo contempla los 10 primeros números, de manera que si el número digitado es superior a 10 se mostrará el mensaje: número no válido.

La solución de ejercicios de este tipo se facilita utilizando la estructura de decisión múltiple, ya que para cada valor hay un equivalente.

El pseudocódigo se muestra en el cuadro 29, en el que se observa la estructura de decisión múltiple implementada entre las líneas 4 y 17.

Cuadro 29. Pseudocódigo para números romanos

```
1.  Inicio
2.      Entero: num
3.      Leer num
4.      Según sea num hacer
5.          1: Escribir "I"
6.          2: Escribir "II"
7.          3: Escribir "III"
8.          4: Escribir "IV"
9.          5: Escribir "V"
10.         6: Escribir "VI"
11.         7: Escribir "VII"
12.         8: Escribir "VIII"
13.         9: Escribir "IX"
14.         10: Escribir "X"
15.     Si no
16.         Escribir "Número no valido"
17.     Fin según sea
18. Fin algoritmo
```

Los diagrama de flujo y N-S se presentan en la figuras 33 y 34 respectivamente. En los diagramas se aprecia las diferentes rutas de ejecución que se generan a partir de la estructura de decisión múltiple.

Figura 33. Diagrama de flujo para número romano

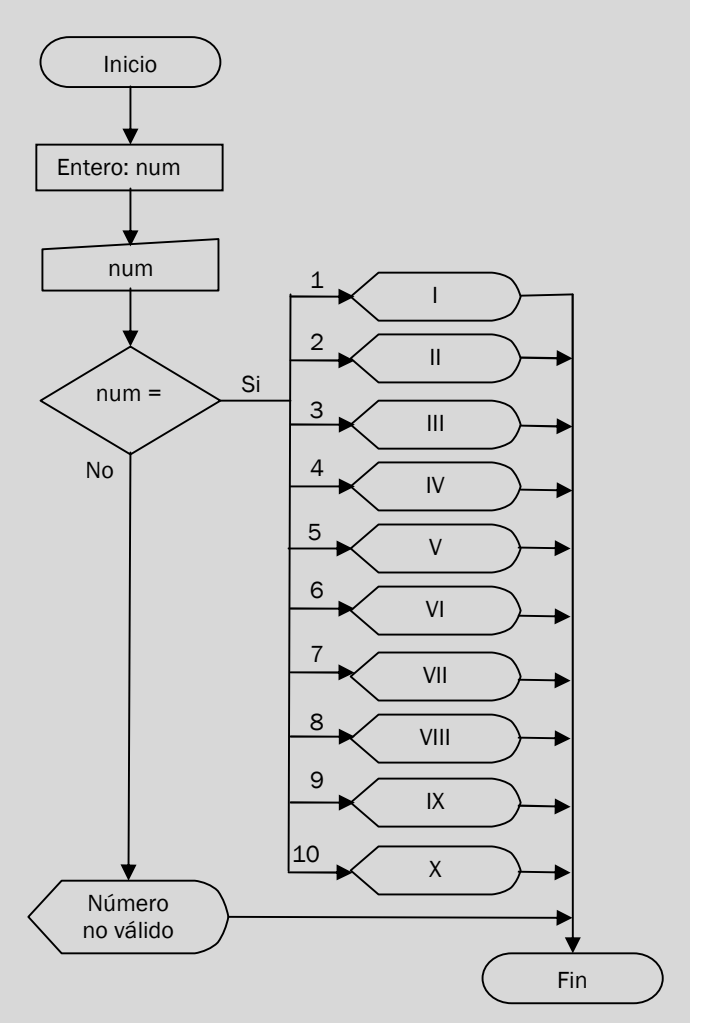
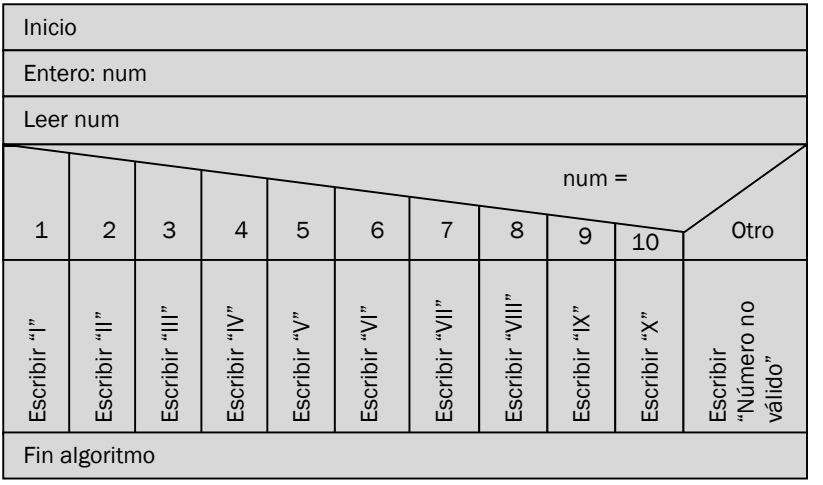


Figura 34. Diagrama N-S para número romano



En el pseudocódigo de este ejercicio se programa 10 posibles valores para la variable *num*, éstos se escriben después de la instrucción *según sea*, seguidos de dos puntos, y para cada uno de ellos se especifica las instrucciones que se deben realizar si el contenido de la variable coincide con dicho valor. Finalmente se tiene la cláusula *si no* para cualquier otro valor mayor a 10 o menor a 1.

En el cuadro 30 se muestra los resultados de tres ejecuciones con diferentes números.

Cuadro 30. Verificación del algoritmo número romano

Ejecución	num	Salida
1	3	III
2	9	IX
3	12	Número no valido

Ejemplo 12. Nombre del día de la semana

Este algoritmo lee un número entre uno y siete correspondiente al día de la semana y muestra el nombre del día, haciendo corresponder el uno (1) con lunes, dos con martes y así sucesivamente, para lo cual utiliza la sentencia *según sea*. Si el número digitado es menor a uno o mayor a siete, muestra un mensaje de error. El pseudocódigo se muestra en el cuadro 31, el diagrama de flujo en la figura 35 y el diagrama N-S en la 36.

Cuadro 31. Pseudocódigo del algoritmo día de la semana

```
1.  Inicio
2.      Entero: día
        Cadena: nomdía
3.      Leer día
4.      Según sea día hacer
5.          1: nomdía = "Lunes"
6.          2: nomdía = "Martes"
7.          3: nomdía = "Miércoles"
8.          4: nomdía = "Jueves"
9.          5: nomdía = "Viernes"
10.         6: nomdía = "Sábado"
11.         7: nomdía = "Domingo
12.     Si no
13.         nomdía = "Número no válido"
14.     Fin según sea
15.     Escribir nomdía
16. Fin algoritmo
```

En este ejercicio no se escribe el nombre del día en cada caso de la decisión, se declara una variable y se le asigna el nombre que corresponda (líneas 5 a 11). Al final, después de haber cerrado la estructura de decisión, se muestra el nombre del día (línea 15).

Figura 35. Diagrama de flujo del algoritmo nombre del día de la semana

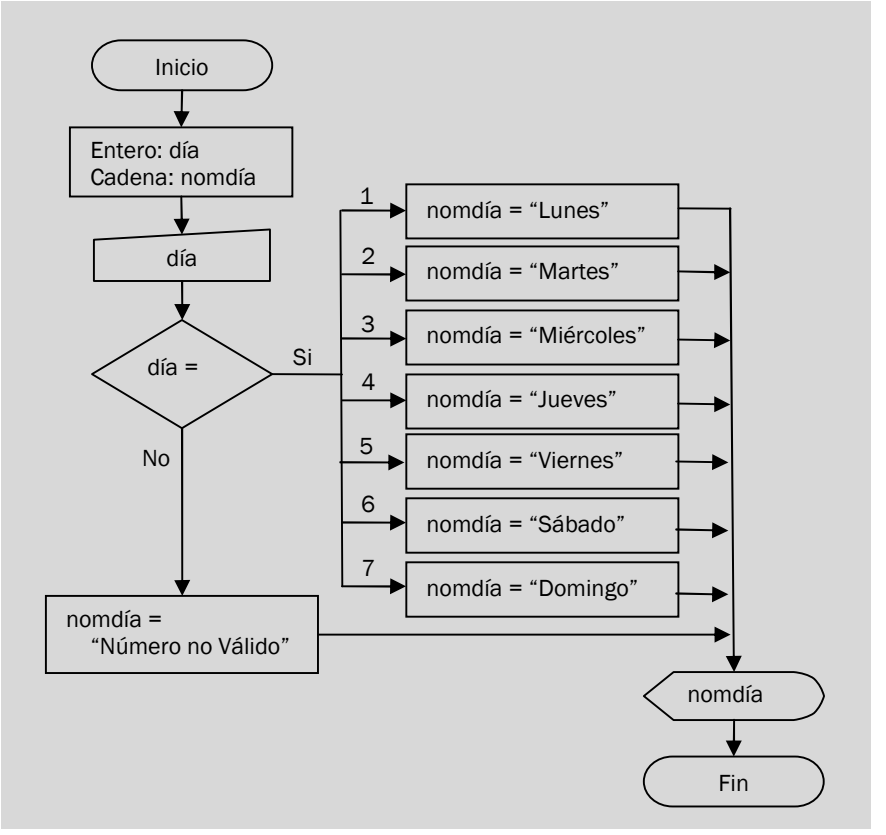
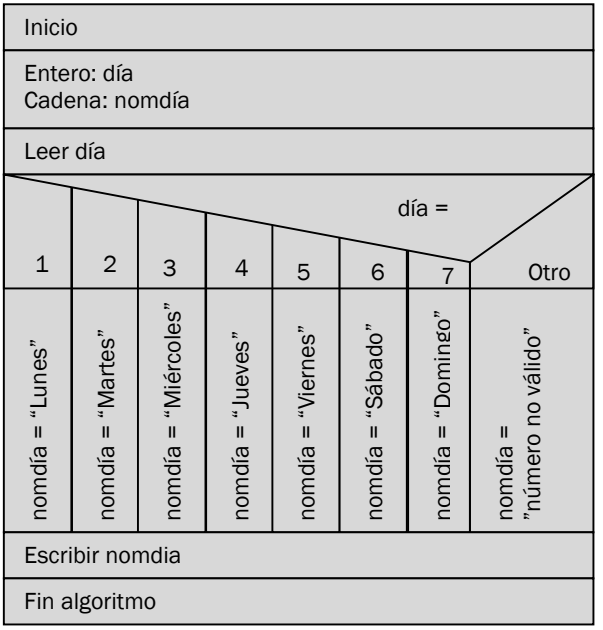


Figura 36. Diagrama N-S del algoritmo día de la semana



En el cuadro 32 se muestran tres casos utilizados para verificar el funcionamiento del algoritmo.

Cuadro 32. Verificación del algoritmo día de la semana

Ejecución	Día	nomdía	Salida
1	6	Sábado	Sábado
2	4	Jueves	Jueves
3	9	Número no válido	Número no válido

4.2.5 Decisiones anidadas

Se denominan *anidadas* a las decisiones que se escriben una dentro de otra; lo que significa que después de haber tomado una decisión es necesario tomar otra. En pseudocódigo el anidamiento tiene la forma de la estructura que se muestra en el cuadro 33.

Cuadro 33. Decisiones anidadas

```
Si <condición 1> entonces
        Si <condición 2> entonces
            Instrucciones 1
         Si no
               Si <condición 3> entonces
                     Instrucciones 2
                Si no
                      Instrucciones 3
               Fin si
        Fin si
Si no
        Si <condición 4> entonces
             Instrucciones 4
        Si no
             Instrucciones 5
        Fin si
Fin si
```

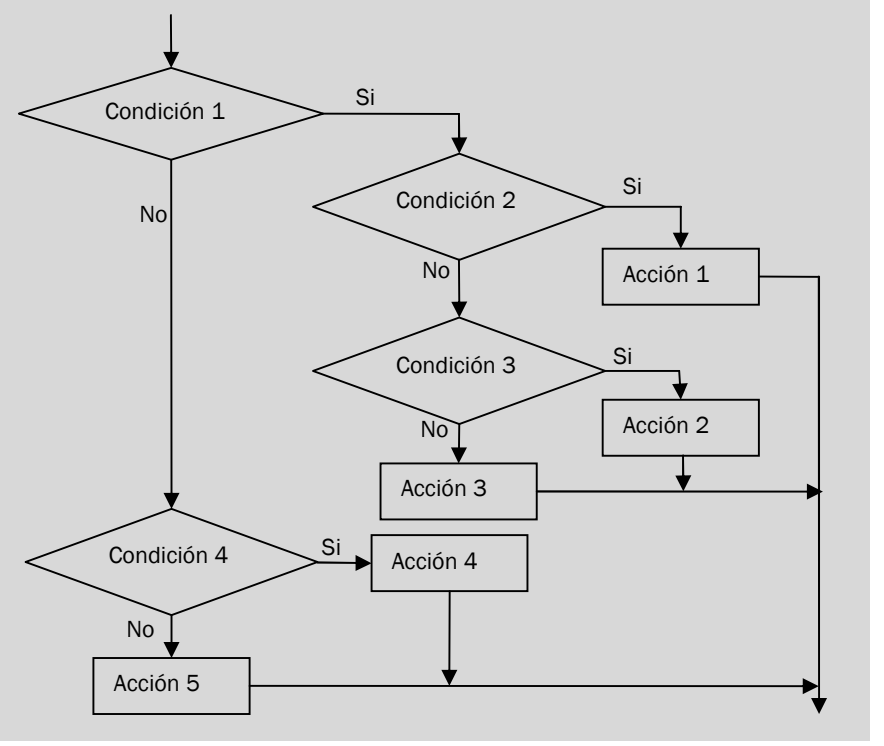
Si condición-1 es verdadera se evalúa condición-2 y si esta también es verdadera se ejecuta instrucciones-1. De forma que instrucciones-1 se ejecuta únicamente si condición-1 y condición-2 son verdaderas; instrucciones-2, si condición-1 es verdadera y condición-2 es falsa.

Cuando la segunda decisión se escribe para el caso de que la primera sea verdadera se dice que se anida por verdadero, como ocurre con la condición-2. Si la segunda decisión se evalúa cuando la primera no se cumple, se dice que se anida por falso. Tanto por verdadero como por falso se pueden anidar todas las decisiones que sean necesarias.

El anidamiento se puede hacer con la estructura *Si* de igual manera con *Según sea* y se pueden combinar las dos: un *Si* dentro de un *Según sea* o lo contrario, tantos niveles como la solución del problema lo requiera.

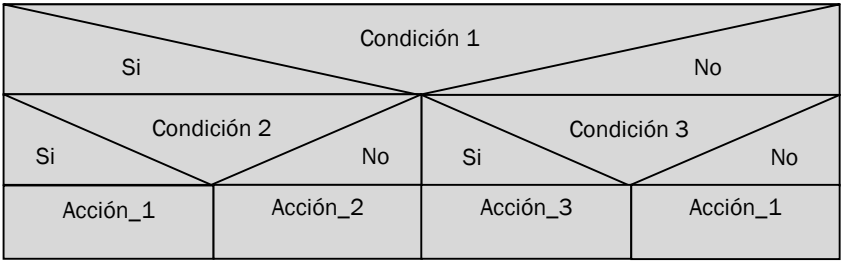
La representación en diagrama de flujo se muestra en la figura 37. En este tipo diagrama es muy fácil de apreciar el anidamiento de las decisiones y de comprender la dependencia de las operaciones con respecto a las decisiones, basta con evaluar la condición y seguir la flecha que corresponda.

Figura 37. Diagrama de flujo de decisiones anidadas



El Diagrama N-S puede parecer complicado, pero no lo es, basta con leer una caja a la vez, de arriba hacia abajo y de izquierda a derecha (ver figura 38). *Condición 1* establece dos caminos, siguiendo por el de la izquierda encuentra la *Condición 2*, ésta también proporciona dos rutas. Si la *Condición 1* es falsa se toma el camino de la derecha y se encuentra con la *Condición 3*, la cual proporciona también dos caminos a seguir, dependiendo del resultado de la condición.

Figura 38. Diagrama N-S de decisiones anidadas



Ejemplo 13. Comparación de dos números

Dados dos números enteros, estos pueden ser iguales o diferentes, si son diferentes ¿cuál de ellos es mayor? y ¿cuál es menor?

En este caso se leen dos números desde el teclado y se almacenan en dos variables: n1 y n2.

El algoritmo requiere dos decisiones, la primera está dada por la condición:

n1 = n2 ?

Si esta condición es verdadera muestra un mensaje, si no lo es debe tomar otra decisión con base en la condición:

n1 > n2 ?

Si esta condición es verdadera mostrará un resultado, si es falsa otro.

En el cuadro 34 se presenta la solución en notación de pseudocódigo, los diagramas de flujo y N-S se presentan en las figuras 39 y 40.

Cuadro 34. Pseudocódigo del algoritmo comparar números

```
1.  Inicio
2.      Entero: n1, n2
3.      Leer n1, n2
4.      Si n1 = n2 entonces
5.          Escribir "números iguales"
6.      Si no
7.          Si n1 > n2 entonces
8.              Escribir n1, "Mayor"
9.              Escribir n2, "Menor"
10.         Si no
11.             Escribir n2, "Mayor"
12.             Escribir n1, "Menor"
13.         Fin si
14.     Fin si
15. Fin algoritmo
```

Para verificar la corrección del algoritmo se hacen tres pruebas. En la primera ejecución se introduce el número 3 para cada una de las variables, en consecuencia al evaluar la primera condición (línea 4) se obtiene el valor verdadero y se muestra el mensaje "números iguales" (línea 5). En la segunda ejecución se ingresan los números 5 y 2, esta vez la primera condición no se cumple y la ejecución continúa en la línea 7 donde se encuentra con otra condición, ésta

sí se cumple y por ende se ejecutan las líneas 8 y 9. En la tercera ejecución se introducen los números 6 y 8, al evaluar la primera condición ésta no se cumple, salta a la línea 7 y evalúa la segunda condición, dado que ésta también es falsa, se ejecutan las líneas 11 y 12.

Figura 39. Diagrama de flujo del algoritmo comparar números

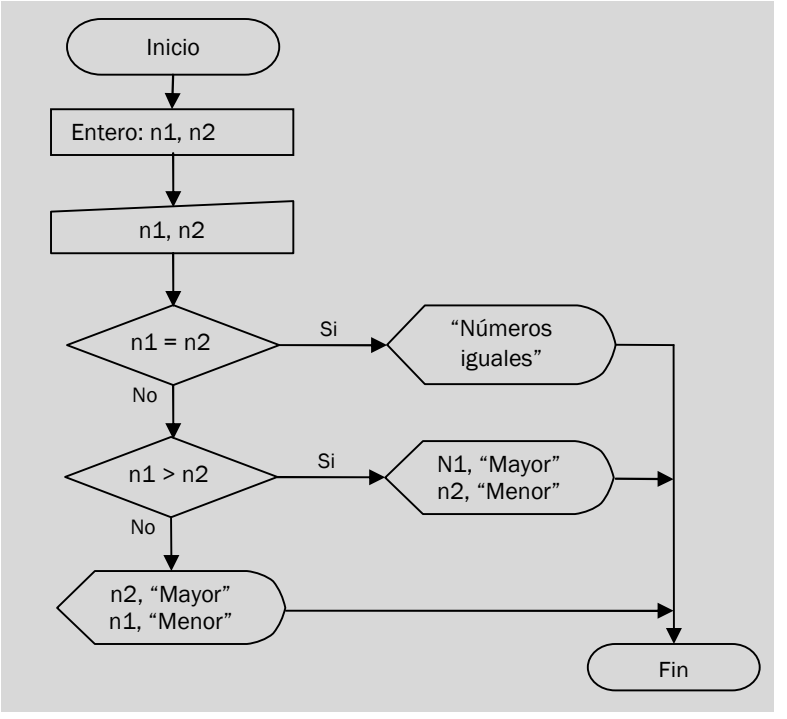
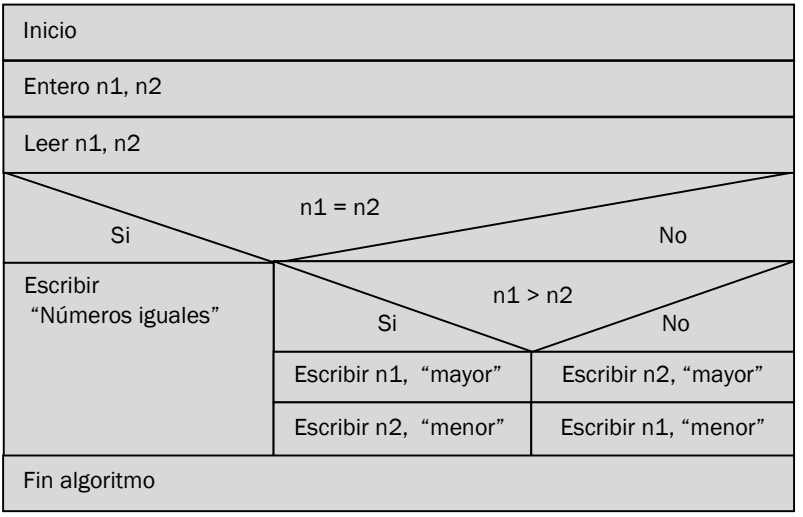


Figura 40. Diagrama N-S del algoritmo comparar números



Los resultados obtenidos en las pruebas del algoritmo se muestran en el cuadro 35.

Cuadro 35. Pseudocódigo del algoritmo comparar números

Ejecución	n1	n2	Salida
1	3	3	Números iguales
2	5	2	5 Mayor 2 Menor
3	6	8	8 Mayor 6 Menor

Ejemplo 14. Calcular aumento de sueldo

La empresa La Generosa S.A desea aumentar el sueldo a sus empleados, para ello ha establecido las siguientes condiciones: quienes ganan hasta $ 800.000 tendrán un incremento del 10%, quienes devengan más de $ 800.000 y hasta 1'200.000 recibirán

un aumento del 8% y los demás del 5%. Se requiere un algoritmo que calcule el valor del aumento y el nuevo salario para cada empleado.

Para comprender el problema considérese los siguientes casos:

Un empleado devenga $ 700.000; dado que este valor es inferior a $ 800.000.oo tendrá un incremento del 10%, por tanto:

Aumento = 700.000 * 10 / 100 = 70.000
Nuevo sueldo = 700.000 + aumento
Nuevo sueldo = 770.000

Otro empleado devenga $ 1'000.000; este valor es superior a 800.000 e inferior a 1'200.000, por tanto tendrá un incremento del 8%, en consecuencia:

Aumento = 1'000.000 * 8 / 100 = 80.000
Nuevo sueldo = 1'000.000 + aumento
Nuevo sueldo = 1'080.000

Un tercer empleado tiene un sueldo de 1'500.000, como este valor es superior a 1'200.000 el porcentaje de aumento es del 5%, se tiene que:

Aumento = 1'500.000 * 5 / 100 = 75.000
Nuevo sueldo = 1'500.000 + aumento
Nuevo sueldo = 1'575.000

De esto se desprende que:

Entradas: sueldo (sue)
Salida: valor aumento (aum) y nuevo sueldo (nsue)
Cálculos: aumento = sueldo * porcentaje (por)
nuevo sueldo = sueldo + aumento

El diseño de la solución en notación pseudocódigo se presenta en el cuadro 36.

Cuadro 36. Pseudocódigo del algoritmo nuevo sueldo

```
1.  Inicio
2.      Real: sue, por, aum, nsue
3.      Leer sue
4.      Si sue <= 800000 entonces
5.          por = 10
6.      Si no
7.          Si sue <= 1200000 entonces
8.              por = 8
9.          Si no
10.             por = 5
11.         Fin si
12.     Fin si
13.     aum = sue * por /100
14.     nsue = sue + aum
15.     Escribir “Aumento:”, aum
16.     Escribir “Nuevo sueldo:”, nsue
17. Fin algoritmo
```

El diagrama de flujo se presenta en la figura 41 y en diagrama N-S en la figura 42.

Figura 41. Diagrama de flujo del algoritmo nuevo sueldo

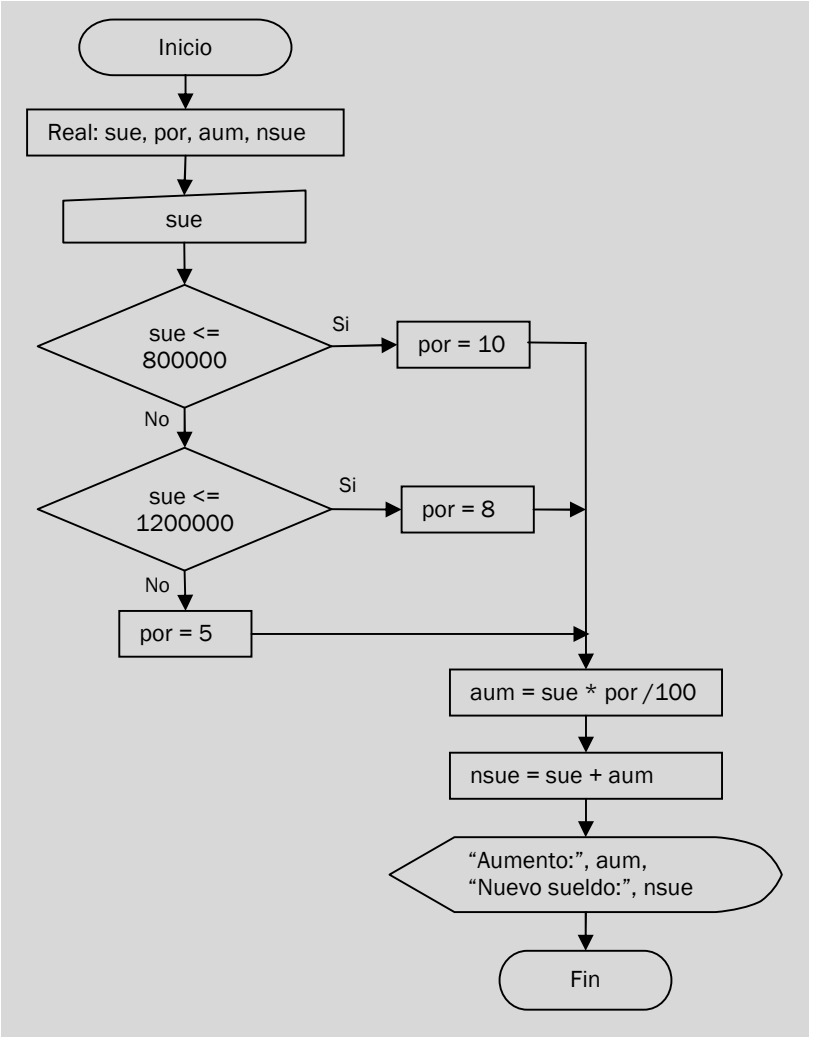
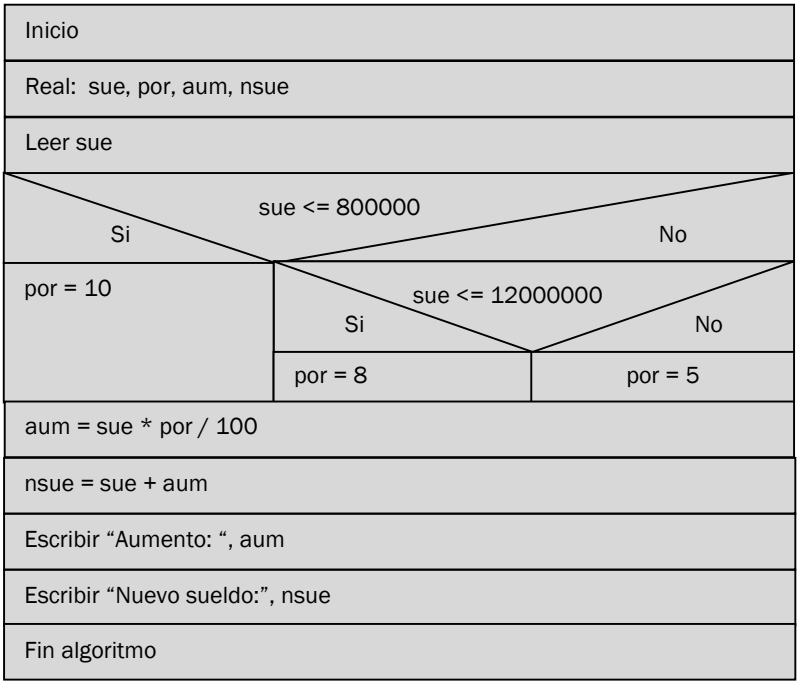


Figura 42. Diagrama N-S del algoritmo nuevo sueldo



Para verificar que el algoritmo funciona correctamente se prueba con los datos que se analizó anteriormente. Los resultados de esta prueba se muestran en el cuadro 37.

Cuadro 37. Verificación del algoritmo nuevo sueldo

Ejec.	sue	por	aum	nsue	Salida
1	700.000	10	70.000	770.000	Aumento: 70.000 Nuevo sueldo: 770.000
2	1'000.000	8	80.000	1'080.000	Aumento: 80.000 Nuevo sueldo: 1'080.000
3	1'500.000	5	75.000	1'575.000	Aumento: 75.000 Nuevo sueldo: 1'575.000

4.2.6 Más ejemplos de decisiones

Ejemplo 15. Selección de personal

Una empresa desea contratar un profesional para cubrir una vacante, los requisitos para ser considerado elegible son: ser profesional y tener entre 25 y 35 años inclusive, o contar con formación académica de especialista o superior en cuyo caso no se tiene en cuenta la edad. Se requiere un algoritmo que evalúe los datos de cada candidato e informe si es apto o no apto.

Siguiendo la estrategia propuesta en este documento, se analizan casos particulares, se identifican los datos y las operaciones, para luego proceder a diseñar una solución genérica.

A la convocatoria se presenta Pedro, quien tiene formación de pregrado en administración de empresas y 28 años de edad. Al confrontar sus datos con las condiciones del problema se encuentra que su edad está en el rango establecido como válido y cuenta con formación de pregrado, por lo tanto es considerado apto.

Luego se presenta Lucía, quien es contador público y tienen una especialización en alta gerencia y 38 años. Al revisar las condiciones que debe cumplir se encuentra que la edad está fuera de rango, pero al contar con especialización es catalogada como apta.

Finalmente, se presenta Carlos, de 45 años y con formación de pregrado en economía. Al examinar el cumplimiento de la primera condición se encuentra que su edad está por encima del límite establecido, esta situación podría ser ignorada si contara con formación especializada, pero se verifica que su nivel de estudios es de pregrado; por tanto es declarado no apto.

Ahora, se trata de diseñar un algoritmo que tome esta decisión de forma automática para cualquier persona que desee saber si cumple los requisitos.

En este caso se requiere tres datos de entrada: el nombre, la edad y la formación académica. Para minimizar la posibilidad de error al ingresar el tercer dato, se presenta al usuario un menú del cual seleccione la opción que corresponda:

Formación académica

1. Tecnológico
2. Profesional
3. Especialista
4. Magister
5. Doctor

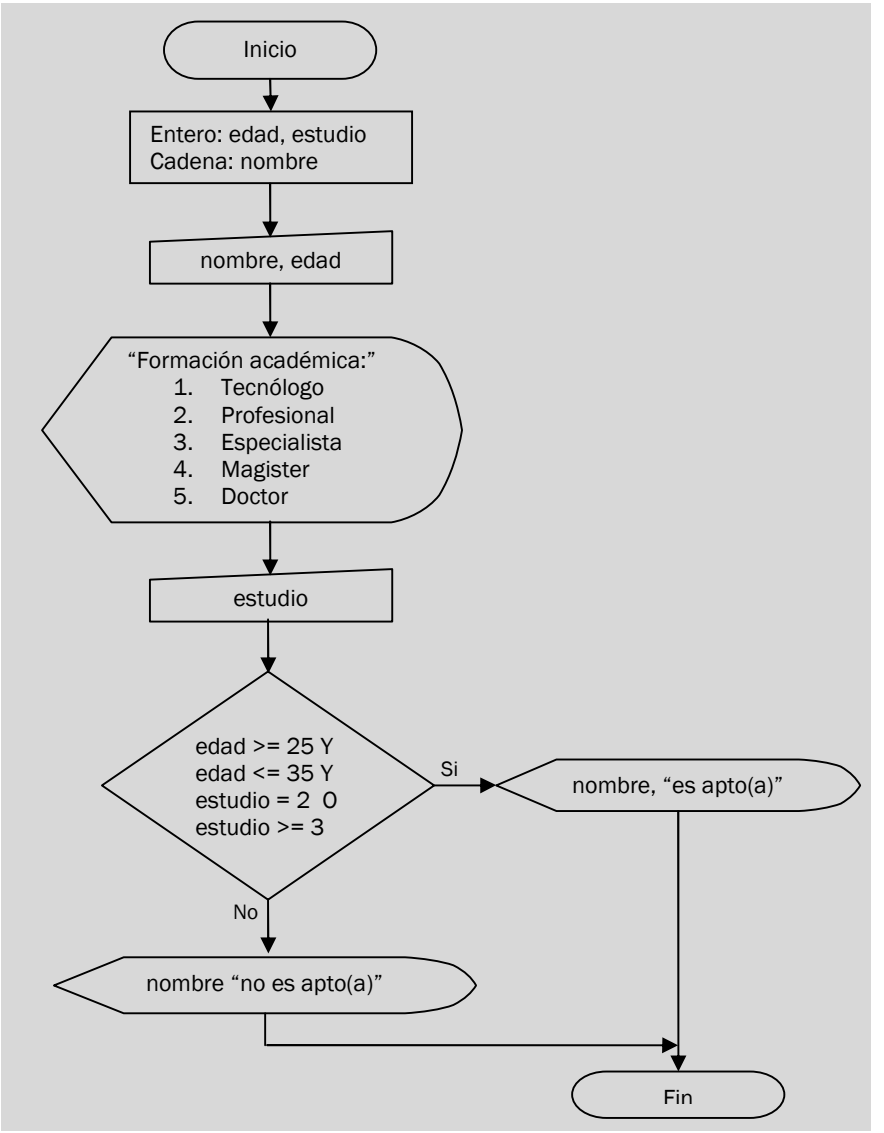
De esta manera la condición se establece sobre datos numéricos. El algoritmo para solucionar este problema se presente como diagrama de flujo en la figura 43.

Los resultados de la verificación del algoritmo se presentan en el cuadro 38.

Cuadro 38. Verificación del algoritmo selección de personal

Ejec.	Nombre	Edad	estudio	Salida
1	Pedro	28	2	Pedro es apto
2	Lucía	38	3	Lucía es apta
3	Carlos	45	2	Carlos no es apto

Figura 43. Diagrama de flujo para seleccionar un empleado



Ejemplo 16. Auxilio de transporte

Se requiere un algoritmo que decida si un empleado tiene derecho al auxilio de transporte. Se conoce que todos los empleados que devengan un salario menor o igual a dos salarios mínimos legales tienen derecho a este rubro.

Para identificar los datos del problema y la forma de solucionarlo se proponen dos ejemplos:

Supóngase un salario mínimo legal de $ 600.000.oo y José devenga un salario mensual de $750.000.oo. El planteamiento del problema estipula que tiene derecho al auxilio si su salario es menor o igual a dos veces el salario mínimo legal; entonces:

750.000.oo <= 2 * 600.000.oo
750.000.oo <= 1'200.000.oo

Esta expresión relacional es verdades; por tanto, José tiene derecho al auxilio de transporte.

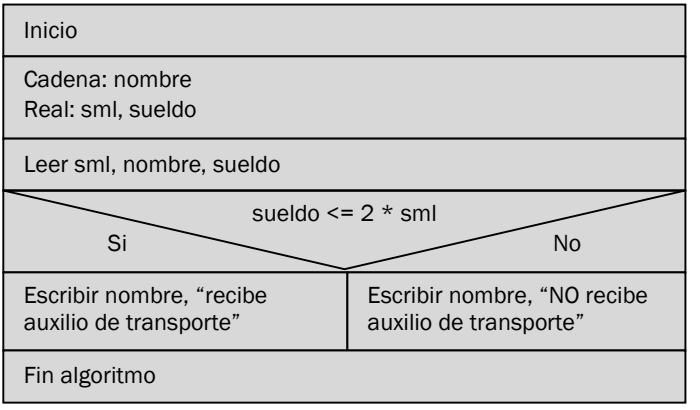
Luis devenga mensualmente $ 1'300.000.oo ¿tiene derecho al auxilio?

1'300.000.oo <= 2 * 600.000.oo
1'300.000.oo <= 1'200.000.oo

La condición no se cumple; en consecuencia, Luis no recibe auxilio de transporte.

El diseño de la solución a este problema se propone mediante el diagrama N-S de la figura 44.

Figura 44. Diagrama N-S del algoritmo auxilio de transporte



Los resultados de la verificación de este algoritmo, con los ejemplos propuestos se muestran en el cuadro 39.

Cuadro 39. Verificación del algoritmo auxilio de transporte

Ejec.	Sml	nombre	sueldo	sueldo <= 2 * sml	Salida
1	515.000	José	750.000	Verdadero	José recibe auxilio de transporte
2	515.000	Luis	1'200.000	Falso	Luis NO recibe auxilio de transporte

Ejemplo 17. Nota definitiva

En la universidad Buena Nota se requiere un algoritmo para calcular la nota definitiva y decidir si el estudiante aprueba o reprueba la asignatura. La nota final se obtiene a partir de dos notas parciales y un examen final, donde el primer parcial equivale al 30%, el segundo parcial al 30% y el examen final al 40%, y la nota mínima aprobatoria es 3.0.

Si el promedio de los dos parciales es menor a 2.0, el estudiante no puede presentar examen final y pierde la materia por bajo promedio, en este caso la nota definitiva es el promedio de los parciales, si el promedio es igual o superior a 2.0 puede presentar el examen final.

Si la nota del examen final es inferior a 2.0, se desconoce las notas parciales y la nota definitiva es la obtenida en el examen final. Si la nota es igual o superior a 2.0 se calcula la nota definitiva aplicando los porcentajes mencionados a los parciales y al final.

Si la nota definitiva es igual o superior a 3.0 el estudiante aprueba la asignatura; si es inferior a 3.0 pierde la materia; sin embargo, puede habilitarla, siempre y cuando en el examen final obtenga nota mayor o igual a 2.0, en este caso la nota definitiva será la que obtenga en la habilitación.

Para comprender mejor esta situación considérese estos casos:

Raúl obtuvo las siguientes notas:

Parcial 1 = 2.2
Parcial 2 = 1.6
Promedio = 1.9
Nota definitiva = 1.9

Siguiendo la información del problema Raúl no puede presentar examen final dado que el promedio de los dos parciales en menor a 2.0; en consecuencia la nota final es el promedio de los parciales y no puede presentar ni examen final ni habilitación. Pierde la asignatura.

Karol obtuvo las siguientes notas:

Parcial 1 = 3.4
Parcial 2 = 2.0
Promedio = 2.7

Puede presentar examen final

Examen final = 1.5

Dado que su nota de examen final es menor a 2.0, ésta pasa a ser la nota definitiva y no puede habilitar.

Las notas de Carlos fueron:

Parcial 1 = 3.5
Parcial 2 = 2.5
Promedio = 3.0

Puede presentar examen final

Examen final = 2.2

Nota definitiva = 3.5 * 30% + 2.5 * 30% + 2.2 * 40% = 2.7

Carlos no aprueba la materia, pero como tiene nota mayor a 2.0 en el examen final puede habilitar

Habilitación = 3.5
Nota definitiva = 3.5. Aprueba la materia

Ana, por su parte, obtuvo las siguientes notas:

Parcial 1 = 3.7
Parcial 2 = 4.3

Promedio = 4.0

Presenta examen final

Examen final = 3.5

Nota definitiva = 3.8

Por lo tanto, Ana aprueba la materia.

Ahora, que se comprende bien el problema ya se puede diseñar un algoritmo para solucionar cualquier caso. Este algoritmo se presenta en el cuadro 40.

Cuadro 40. Pseudocódigo algoritmo Nota Definitiva

```
1.  Inicio
2.      Real: p1, p2, ef, nd, prom, nh
3.      Leer p1, p2
4.      prom = (p1 + p2)/2
5.      Si prom < 2.0 entonces
6.          nd = prom
7.          Escribir “pierde materia por bajo promedio”
8.      si no
9.          Leer ef
10.         Si ef < 2.0 entonces
11.             nd = ef
12.             Escribir “Pierde materia y no puede habilitar”
13.         si no
14.             nd = p1 * 0.3 + p2 * 0.3 + ef * 0.4
15.             Si nd >= 3.0 entonces
16.                 Escribir “Ganó la materia”
17.             si no
18.                 Escribir “Perdió la materia pero puede habilitar”
19.                 Leer nh
20.                 nd = nh
21.                 Si nh >= 3.0 entonces
22.                     Escribir “Ganó la habilitación”
23.                 si no
24.                     Escribir “Perdió la habilitación”
25.                 Fin si
```

Cuadro 39. (Continuación)

```
26.                 Fin si
27.             Fin si
28.         Fin si
29.         Escribir "Nota definitiva:", nd
30.     Fin algoritmo
```

En este algoritmo se declaran variables para almacenar las dos notas parciales, el promedio, el examen final, la nota definitiva y la nota de habilitación. Para comenzar se leen los dos parciales, se calcula el promedio y se evalúa (línea 5), si éste es menor a 2.0, se asigna a la nota definitiva, se muestra un mensaje, la nota y el algoritmo termina.

Si el promedio es superior a 2.0 se lee el examen final y se evalúa (línea 10) si es menor a 2.0 se asigna como definitiva, se muestra un mensaje y salta al final del algoritmo. Si el examen final es superior a 2.0 se hace el cálculo de la nota definitiva y se decide si aprueba o pierde, si pierde puede presentar habilitación, esta es la última oportunidad de aprobar la asignatura.

En el cuadro 41 se muestra tres conjuntos de datos con los que se prueba el algoritmo y los resultados obtenidos.

Cuadro 41. Verificación del algoritmo Nota Definitiva

Ejec	P1	P2	prom	ef	nd	nh	Salida
1	2.2	1.6	1.9	-	1.9		Perdió la materia por bajo promedio Nota definitiva: 1.9
2	3.4	2.0	2.7	1.5	1.5		Pierde materia Nota definitiva: 1.5
3	3.5	2.5	3.0	2.2	2.7	3.5	Perdió la materia pero puede habilitar Ganó la habilitación Nota definitiva: 3.5
4	3.7	4.3	4.0	3.5	3.8		Ganó la materia Nota definitiva: 3.8

Ejemplo 18. El hermano mayor

Este algoritmo lee los nombres y las edades de tres hermanos y decide cuál es el nombre del hermano mayor.

Supóngase los siguientes casos:

José, Luis y María son hermanos, el primero tiene 19 años, el segundo 15 y la tercera 23. ¿Cuál de ellos es el mayor? Evidentemente, María.

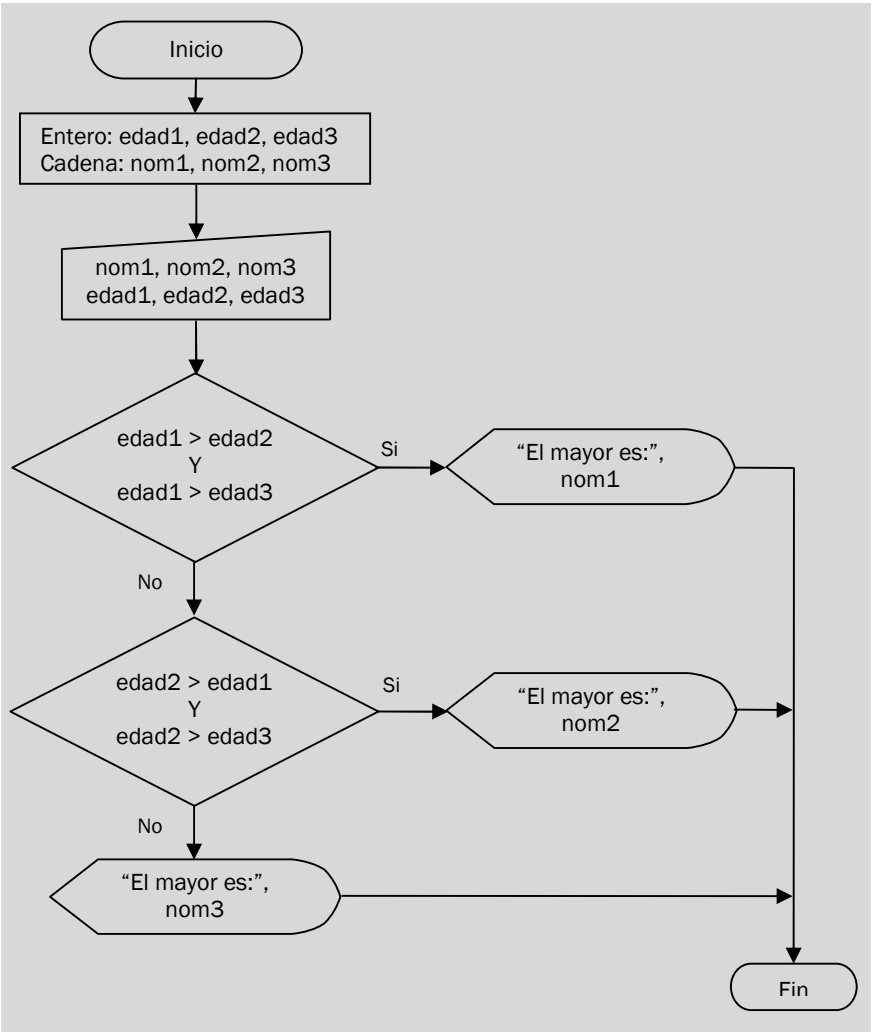
Carlos tiene 32, rocío tiene 28 y Jesús tiene 25. En este caso, el mayor es Carlos.

Martha tiene 8 años, Ana 10 y Camilo 4. De los tres, Ana es mayor.

Ahora se trata de diseñar un algoritmo para que ésta decisión la tome un programa de computador. Es necesario declarar tres variables de tipo cadena y relacionarlas con tres de tipo numérico, así se comparan las edades y se muestra el nombre como resultado. El diagrama de flujo de este algoritmo se presenta en la figura 45.

Para verificar que el algoritmo es correcto se revisa paso a paso registrando los valores correspondientes a las variables y la salida por pantalla. En el cuadro 42 se muestran los datos de prueba y los resultados obtenidos.

Figura 45. Diagrama de flujo del algoritmo hermano mayor



Cuadro 42. Verificación algoritmo del hermano mayor

Ejec.	nom1	edad1	nom2	edad2	nom3	edad3	Salida
1	José	19	Luis	15	María	23	María
2	Carlos	32	Rocío	28	Jesús	25	Carlos
3	Martha	8	Ana	10	Camilo	4	Ana

Ejemplo 19. Descuento a mayoristas

El almacén Gran Distribuidor vende camisas al por mayor y hace descuentos según la cantidad facturada: en cantidades superiores o iguales a 1000 unidades hace el 10% de descuento; entre 500 y 999, el 8%; entre 200 y 499, el 5%; y en menos de 200 no hay descuento. Dada la cantidad facturada y el precio unitario, se requiere calcular el descuento que se le hace a un cliente y el valor a pagar.

Antes de proceder con el diseño del algoritmo es conveniente comprender bien el problema a partir de un caso particular para realizar los cálculos.

Si una camisa tiene un costo de $ 80.000 y un cliente hace un pedido de 350 unidades, entonces tiene derecho a un descuento del 5%:

Unidades: 350
Valor unitario: 80.000
Valor total = 350 * 80.000 = 28.000.000
Porcentaje descuento: 5%
Valor descuento = 28.000.000 * 5 / 100 = 1.400.000
Valor a pagar = 28.000.000 – 1.400.000 = 26.600.000

En este caso el cliente tendrá un descuento por valor de $ 1.400.000 y pagará en total $ 26.600.000.

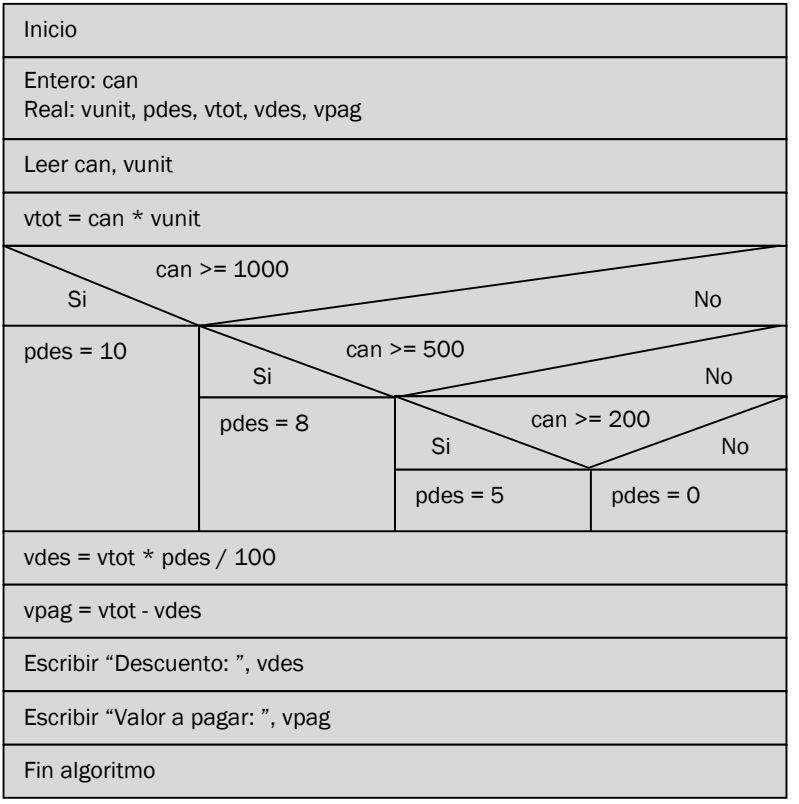
Ahora se puede identificar los datos de entrada, de salida y los procesos a realizar:

Entrada: cantidad, valor unitario
Salida: valor descuento, valor a pagar
Procesos:

Total = cantidad * valor unitario
Valor descuento = total * porcentaje descuento / 100
Valor a pagar = total - descuento

La solución se presenta mediante diagrama N-S en la figura 46.

Figura 46. Diagrama N-S algoritmo descuento a mayoristas



En el cuadro 43 se muestra los datos de prueba de este algoritmo.

Cuadro 43. Verificación algoritmo descuento a mayorista

Ejec.	can	vunit	Vtot	pdes	vdes	Vpag	Salida
1	350	80000	28000000	5	1400000	26600000	Descuento 1400000 Pagar 26600000
2	600	20000	12000000	8	960000	11040000	Descuento 960000 Pagar 11040000
3	1100	30000	33000000	10	3300000	29700000	Descuento 3300000 Pagar 29700000

Ejemplo 20. Depósito a término

El banco Buena Paga ofrece diferentes tasas de interés anual para depósitos a término dependiendo del tiempo por el que se hagan. Si el depósito es por un periodo menor o igual a seis meses la tasa es del 8% anual; entre siete y 12 meses, 10%; entre 13 y 18, 12%; entre 19 y 24, 15%; y para periodos superiores a 24 meses el 18%. Se requiere un algoritmo para determinar cuánto recibirá un cliente por un depósito, tanto por concepto de interés como en total.

Considérese algunos casos particulares:

Caso 1: Pedro Pérez hace un depósito de un millón de pesos ($ 1'000.000.oo) durante cinco meses. Según la información que se da en el planteamiento del problema, el bando le pagará una tasa de interés del 8% anual.

Dado que la tasa de interés está expresada en años y el depósito en meses, lo primero que hay que hacer es pasarla a meses, para ello se divide sobre 12.

Tasa interés mensual = 8 / 12
Tasa interés mensual = 0.667

Ahora, se tienen los siguientes datos:

Capital = 1000000.oo
Tiempo = 5
Tasa interés mensual = 0.667 %

Para conocer cuánto recibirá por concepto de interés se aplica al valor del capital el porcentaje que se acaba de obtener y se multiplica por el número de periodos, en este caso cinco meses, de la forma:

Interés = 1000000.oo * (0.667 / 100) * 5
Interés = 33350

Y para obtener el valor futuro de la inversión, se suma el interés al capital:

Valor futuro = 1000000 + 33350
Valor futuro = 1033350

Caso 2: José López desea depositar cinco millones por un periodo de un año y medio (18 meses). En este caso, el banco le ofrece un interés de 12% anual.

La tasa de interés convertida a meses es de:

Tasa interés mensual = 12 / 12 = 1

Entonces, los datos son:

Capital = 5000000
Tasa interés mensual = 1 %
Tiempo = 18

De dónde se obtiene:

Interés = 5000000 * (1 / 100) * 18
Interés = 900000

Valor futuro = capital + interés
Valor futuro = 5000000 + 900000
Valor futuro = 5900000

Con base en los dos ejemplos se concluye que:

Datos de entrada: capital, tiempo
Datos de salida: interés, valor futuro
Procesos:
    Determinar la tasa de interés anual (es una decisión)
    Interés mensual = tasa interés anual / 12
    Interés = capital * tasa interés mensual / 100 * tiempo
    Valor futuro = capital + interés

El algoritmo de solución a este ejercicio se presenta, en notación de pseudocódigo, en el cuadro 44.

En este algoritmo se utiliza las estructuras de decisión *si anidadas* para determinar el porcentaje correspondiente a la tasa de interés anual, dependiendo del tiempo. Se aplica decisiones anidadas y no la estructura *según sea*, ya que ésta sólo puede utilizarse para evaluar la condición de igualdad entre el contenido de la variable y un valor constante, pero no para relaciones de mayor o menor.

En el cuadro 45 se muestra los resultados de verificar el algoritmo con los datos utilizados en los dos ejemplos desarrollados al comienzo de este ejercicio, más un tercero por valor de 8000000 a 36 meses.

Cuadro 44. Pseudocódigo algoritmo depósito a término

```
1.  Inicio
2.      Real: capital, tasaintanual, tasaintmes, interés, valfuturo
3.      Entero: tiempo
4.      Leer capital, tiempo
5.      Si tiempo < 1 entonces
6.          Escribir "Tiempo no valido"
7.          tasaintanual = 0
8.      Si no
9.          Si tiempo <= 6 entonces
10.             tasaintanual = 8
11.         Si no
12.             Si tiempo <= 12 entonces
13.                 tasaintanual = 10
14.             Si no
15.                 Si tiempo <= 18 entonces
16.                     tasaintanual = 12
17.                 Si no
18.                     Si tiempo <= 24 entonces
19.                         tasaintanual = 15
20.                     Si no
21.                         tasaintanual = 18
22.                     Fin si
23.                 Fin si
24.             Fin si
25.         Fin si
26.     Fin si
27.     tasaintmes = tasaintanual / 12
28.     interés = capital * tasaintmes / 100 * tiempo
29.     valfuturo = capital + interés
30.     Escribir "Interes = ", interés
31.     Escribir "Valor futuro = ", valfuturo
32. Fin algoritmo
```

Cuadro 45. Verificación del algoritmo depósito a término

Capital	Tiempo	tasaintanual	tasaintmes	interés	Valfuturo
1000000	5	8	0.667	33350	1033350
5000000	18	12	1	900000	5900000
8000000	36	18	1.5	4320000	12320000

Ejemplo 21. Empresa editorial

Una empresa editorial tiene tres grupos de empleados: vendedores, diseñadores y administrativos. Se requiere calcular el nuevo sueldo para los empleados teniendo en cuenta que se incrementarán así: administrativos 5%, diagramadores 10% y vendedores 12%.

Como un caso particular se calcula el nuevo sueldo de la secretaria, quien devenga actualmente un sueldo de $ 600.000.oo. Su cargo es de tipo administrativo, por tanto le corresponde un incremento del 5%.

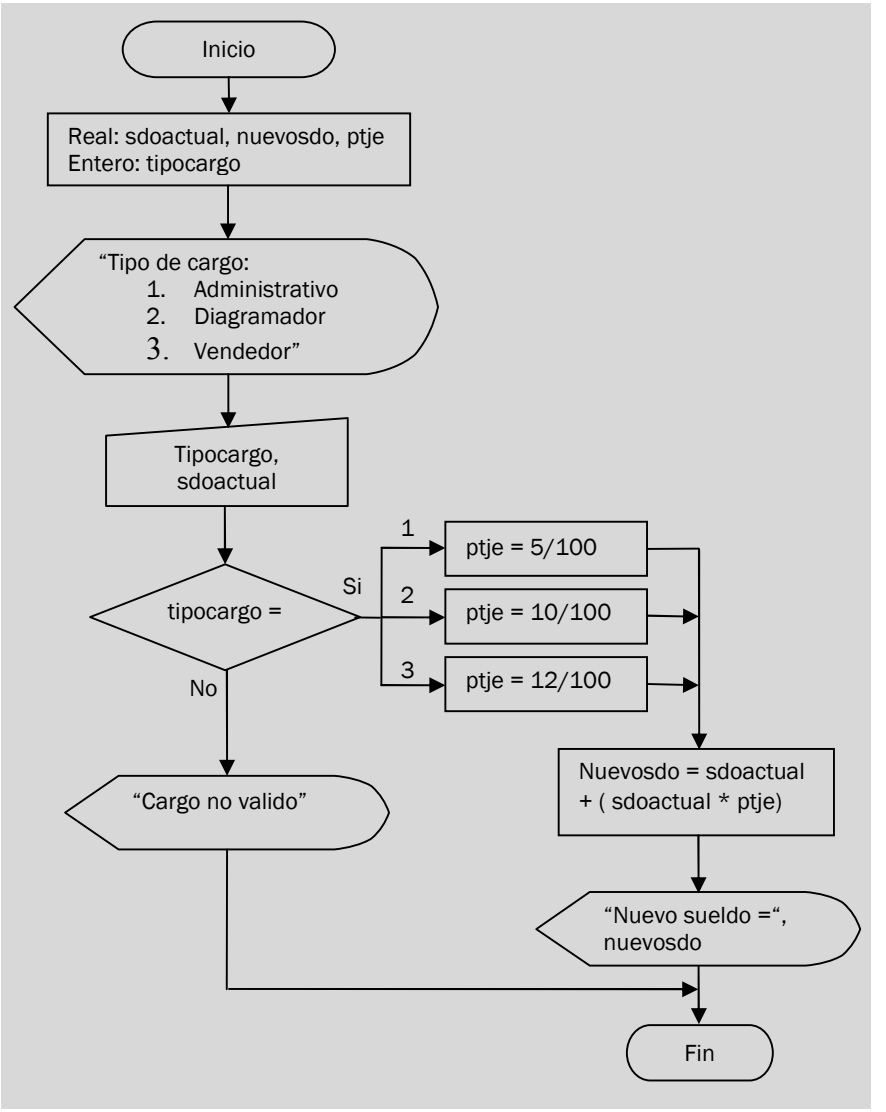
Cargo: secretaria
Tipo de cargo: administrativo
Sueldo actual: 600.000.oo
Porcentaje incremento: 5%

Por lo tanto

Valor incremento: 30.000.oo
Nuevo sueldo: 630.000.oo

En este ejercicio se tiene como dato de entrada el sueldo actual y el tipo de cargo que desempeña (administrativo, diseñador, vendedor), ya que el porcentaje de incremento depende del cargo y se aplica sobre el valor del sueldo vigente. El algoritmo para solucionar este problema se presenta en notación de diagrama de flujo en la figura 47.

Figura 47. Diagrama de flujo del algoritmo Empresa editorial



En este algoritmo se declaran tres variables de tipo real para sueldo actual, nuevo sueldo y porcentaje de incremento, y una variable entera para capturar la opción correspondiente al tipo de cargo que tiene el empleado. Se toma una decisión (múltiple) sobre el tipo de cargo y se establece el porcentaje a aplicar, se realiza el cálculo y se muestra el nuevo sueldo. Si la opción seleccionada como cargo no está entre 1 y 3 se muestra un mensaje de error.

Se proponen tres conjuntos de datos para verificar si el algoritmo es correcto. Los resultados de las ejecuciones se presentan en el cuadro 46.

Cuadro 46. Verificación del algoritmo Empresa editorial

Ejec.	tipocargo	sdoactual	Ptje	nuevosdo	Salida
1	1	600.000	0.05	630.000	Nuevo sueldo = 630.000
2	2	800.000	0.1	880.000	Nuevo sueldo = 880.000
3	3	700.000	0.12	784.000	Nuevo sueldo = 784.000
4	4	500.000			Cargo no válido

Ejemplo 22. Descuento en motocicleta

La distribuidora de motocicletas Rueda Floja ofrece una promoción que consiste en lo siguiente: las motos marca Honda tienen un descuento del 5%, las de marca Yamaha del 8% y las Suzuki el 10%, las de otras marcas el 2%. Se requiere calcular el valor a pagar por una motocicleta.

Considérense algunos ejemplos particulares de este problema.

Pedro desea comprar una motocicleta Honda cuyo precio es de seis millones de pesos. ¿Cuánto debe pagar en realidad por aquella moto?

En la descripción del problema se dice que este tipo de motos tienen un descuento del 5%, entonces hay que calcular el valor del

descuento que se le hará a la moto que desea Pedro; se tienen los siguientes datos:

Valor moto = 6'000.000.oo
Porcentaje descuento = 5/100 (5%)
Valor descuento = 6'000.000.oo * 5 / 100 = 300.000.oo
Valor neto = 6'000.000.oo – 300.000.oo = 5'700.000.oo

Dado que la motocicleta que Pedro desea tiene un descuento del 5% el valor neto a pagar es de $ 5'700.000.oo.

Juan quiere comprar una motocicleta Yamaha que tiene un precio de lista de $ 8'500.000.oo. Dado que esta marca tiene un descuento de 8%, se tiene que:

Valor moto = 8'500.000.oo
Porcentaje descuento = 8/100 (8%)
Valor descuento = 8'500.000.oo * 8 / 100 = 680.000.oo
Valor neto = 8'500.000.oo – 680.000.oo = 7'820.000.oo

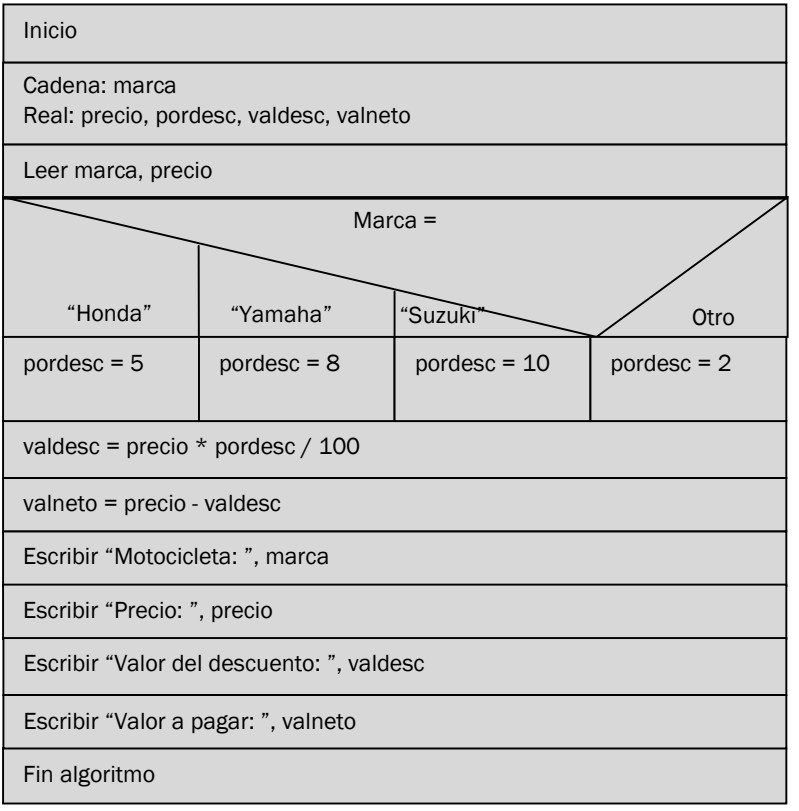
Por lo tanto, Juan debe pagar la suma de $ 7'820.000.oo por la motocicleta Yamaha.

Con base en los ejemplos anteriores se puede generalizar el procedimiento, así:

Porcentaje de descuento = depende de la marca (5%, 8%, 10% o 2%)
Valor descuento = valor moto * porcentaje descuento
Valor neto = valor moto – valor descuento

En consecuencia los datos a introducir corresponden a: precio de lista y la marca de la motocicleta. El algoritmo que soluciona este problema se presenta en notación N-S en la figura 48.

Figura 48. Descuento en motocicleta



Para verificar que el algoritmo es correcto se realiza la prueba de escritorio considerando cada una de las alternativas de decisión, como se muestra en el cuadro 47.

Cuadro 47. Verificación del algoritmo descuento en motocicleta

Ejec.	Marca	precio	pordesc	Valdesc	valneto	Salida
1	Honda	6'000.000	5	300.000	5'700.000	Motocicleta: Honda Precio: 6'000.000 Valor del descuento: 300.000 Valor a pagar: 5'700.000
2	Yamaha	8'500.000	8	680.000	7'820.000	Motocicleta: Yamaha Precio: 8'500.000 Valor del descuento: 680.000 Valor a pagar: 7'820.000
3	Suzuki	3'800.000	10	380.000	3'420.000	Motocicleta: Suzuki Precio: 3'800.000 Valor del descuento: 380.000 Valor a pagar: 3'420.000
4	Kawasaki	5'000.000	2	100.000	4'900.000	Motocicleta: Kawasaki Precio: 5'000.000 Valor del descuento: 100.000 Valor a pagar: 4'900.000

Ejemplo 23. Comisión de ventas

La empresa V&V paga a sus vendedores un salario básico más una comisión sobre las ventas efectuadas en el mes, siempre que éstas sean mayores a $ 1.000.000. Los porcentajes de bonificación son:

Ventas mayores a $ 1.000.000 y hasta $ 2.000.000 = 3%
Ventas mayores a $ 2.000.000 y hasta $ 5.000.000 = 5%
Ventas mayores a $ 5.000.000 = 8%

Se desea conocer cuánto corresponde al vendedor por comisión y cuánto recibirá en total en el mes.

Siguiendo el enfoque que se ha propuesto para este libro, lo primero es considerar algunos ejemplos que permitan comprender el problema y la forma de solucionarlo. En este orden de ideas se plantean cuatro ejemplos:

Jaime fue contratado con un salario básico de $ 600.000.oo y durante el último mes vendió la suma de $ 1'200.000.oo. Dado que vendió más de un millón tiene derecho a comisión; entonces, el siguiente paso es decidir qué porcentaje le corresponde. En la descripción del ejercicio se establece que si las ventas están entre uno y dos millones el vendedor recibirá una comisión del 3% sobre el valor de las ventas. Esto es:

Vendedor(a): Jaime
Salario básico: 600.000.oo
Valor ventas: 1'200.000.oo
Porcentaje comisión: 3%
Valor comisión = 1'200.000 * 3 / 100 = 36.000
Sueldo del mes = 600.000 + 36.000 = 636.000

Jaime recibirá $ 36.000 por concepto de comisión y el sueldo total será de $ 636.000.oo

Susana tiene un salario básico de $ 700.000.oo y en el último mes sus ventas fueron por valor de $ 3'500.000.oo. Este caso, confrontando los requerimientos del problema, ella tiene derecho a una comisión del 5%. Los resultados del ejercicio para Susana son:

Vendedor(a): Susana
Salario básico: 700.000.oo
Valor ventas: 3'500.000.oo
Porcentaje comisión: 5%
Valor comisión = 3'500.000 * 5 / 100 = 175.000
Sueldo del mes = 700.000 + 175.000 = 875.000

Susana ha conseguido una comisión de ventas por valor de $ 175.000 para un sueldo total de $ 875.000.oo.

Carlos tiene un sueldo básico de $ 600.000.oo y en el último mes sus ventas fueron de $ 950.000.oo. Como no alcanzó a vender el millón no recibirá comisión.

Vendedor(a): Carlos
Salario básico: 600.000.oo
Valor ventas: 950.000.oo
Porcentaje comisión: Ninguno
Valor comisión = Ninguna
Sueldo del mes = 600.000

Carlos recibe únicamente el salario básico.

Rocío es conocida como la mejor vendedora de la empresa, tiene asignado un salario básico de $ 800.000.oo. En el último mes logró vender siete millones de pesos.

Vendedor(a): Rocío
Salario básico: 800.000.oo
Valor ventas: 7'000.000.oo
Porcentaje comisión: 8%
Valor comisión = 7'000.000 * 8 / 100 = 560.000
Sueldo del mes = 800.000 + 560.000 = 1'360.000

Rocío obtiene una comisión de $ 560.000.oo y un sueldo total de $ 1'360.000.oo

De los anteriores ejemplos se identifica los datos de entrada y los cálculos a realizar, los datos de salida los especifica el planteamiento del ejercicio: vendedor, comisión y sueldo total. La solución algorítmica se presenta en el cuadro 48 en la notación de pseudocódigo.

Los datos a ingresar son:

Nombre del vendedor
Sueldo básico
Valor de ventas

Los cálculos a realizar son:
Valor comisión = valor ventas * porcentaje comisión (3, 5, 8) / 100
Sueldo mes = sueldo básico + valor comisión

Cuadro 48. Algoritmo comisión de ventas

```
1.  Inicio
2.      Cadena: nom
3.      Real: sbasico, venta, pcomi, vcomi, smes
4.      Leer nom,sbasico,venta
5.      Si venta > 1000000 entonces
6.          Si venta <= 2000000 entonces
7.              pcomi = 3
8.          Si no
9.              Si venta <= 5000000 entonces
10                 pcomi = 5
11             Si no
12                 pcomi = 8
13             Fin si
14         Fin si
15     Si no
16         pcomi = 0
17     Fin si
18     vcomi = venta * pcomi / 100
19     smes = sbasico + vcomi
20     Escribir "Vendedor(a):", nom
21     Escribir "Comisión:", vcomi
22     Escribir "Sueldo total:", smes
23  Fin algoritmo
```

Para verificar si el algoritmo funciona correctamente se ejecuta el pseudocódigo, paso a paso, con los datos de los ejemplos explicados previamente para los cuales ya se hicieron los cálculos y

se conoce los resultados. Los resultados se presentan en el cuadro 49.

Cuadro 49. Verificación del algoritmo comisión de ventas

Nom	sbasico	Venta	pcomi	vcomi	smes	Salida
Jaime	600000	1200000	3	36000	636000	Vendedor(a): Jaime Comisión: 36000 Sueldo total: 636000
Susana	700000	3500000	5	175000	875000	Vendedor(a): Susana Comisión: 175000 Sueldo total: 875000
Carlos	600000	950000	0	0	600000	Vendedor(a): Carlos Comisión: 0 Sueldo total: 600000
Rocío	800000	7000000	8	560000	1360000	Vendedor(a): Rocío Comisión: 560000 Sueldo total: 1360000

Ejemplo 24. Factura del servicio de energía

Se requiere un algoritmo para facturar el servicio de energía eléctrica. El consumo del mes se determina por diferencia de lecturas. El valor del kilovatio (kW*) es el mismo para todos los usuarios, pero se hace un descuento para algunos estratos, así:

Estrato 1: 10%
Estrato 2: 6%
Estrato 3: 5%

Para todos los estratos se aplica un descuento del 2% si el consumo es superior a 200 kW. Se desea que el programa muestre el consumo y el valor a pagar por éste servicio. (El planteamiento de

* kW es el símbolo de kilovatio. Un kilovatio es una unidad de medida de electricidad equivalente a 1000 vatios. Equivale, también, a la energía producida o consumida por una potencia de un kW durante una hora. (Círculo Enciclopedia Universal)

este ejercicio es meramente didáctico y no tiene relación alguna con la facturación de este servicio en el mundo real).

En una empresa real se mantiene los registros de las lecturas y por ende sólo es necesario hacer una lectura y se hace la diferencia con la última que se tenga almacenada, pero en este ejercicio es necesario leer los dos datos para calcular la diferencia. Para comprender mejor este planteamiento obsérvese los siguientes ejemplos:

José es un usuario del servicio de energía que reside en estrato 1, el mes anterior su medidor registró 12345 y en este mes, 12480. Dado que está en estrato 1 recibirá un descuento del 10%, y si el valor del kW es de $ 500 el consumo y el valor a pagar se obtienen de la siguiente forma:

Usuario: José
Lectura 1: 12345
Lectura 2: 12480
Consumo: 12480 – 12345 = 135
Valor kW: 500
Valor consumo: 67500
Estrato: 1
Porcentaje de descuento: 10%
Valor del descuento = 67500 * 10 / 100 = 6750
Valor a pagar = 67500 – 6750 = 60750

De este ejemplo se puede identificar los datos y los procesos necesarios para solucionar el ejercicio. Los datos de entrada son:

Usuario,
Estrato socioeconómico,
Lectura 1,
Lectura 2 y
valor del kW.

El porcentaje a aplicar como descuento es una decisión basada en el estrato socio-económico del usuario. Los valores a calcular son:

Consumo = lectura 2 – lectura 1
Valor consumo = consumo * valor kW
Valor descuento = valor consumo * porcentaje descuento / 100
Valor a pagar = valor consumo – valor descuento

El paso siguiente es organizar las acciones en forma algorítmica. El diagrama de flujo de la solución a este ejercicio se presenta en la figura 49.

Para verificar que la solución es correcta se ejecuta paso a paso, se introduce los datos que se utilizaron para analizar el problema y se confrontan los resultados generados por el algoritmo con los obtenidos manualmente.

Ejemplo 25. Calculadora

Este algoritmo lee dos números y un operador aritmético, aplica la operación correspondiente y muestra el nombre de la operación y el resultado.

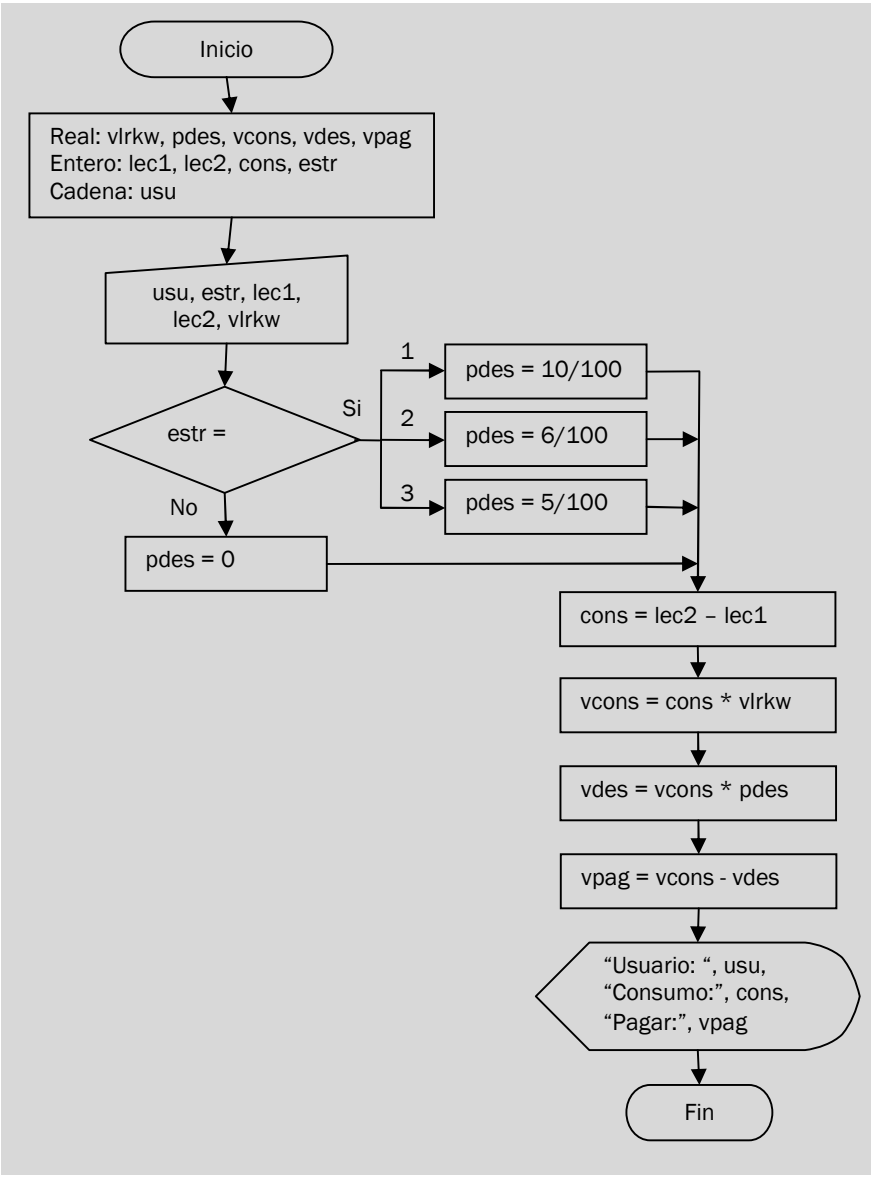
En este caso se trata de identificar el operador que el usuario ha introducido y realizar la operación correspondiente. Los operadores aritméticos a tener en cuenta son:

+ suma
- resta
* multiplicación
/ división

Por ejemplo:

Primer número: 3
Segundo número: 4
Operador: +

Figura 49. Facturación servicio de energía



El algoritmo debe realizar la operación suma y mostrar el resultado, así:

Suma: 3 + 4 = 7

Otro ejemplo:

Primer número: 6
Segundo número: 2
Operador: -

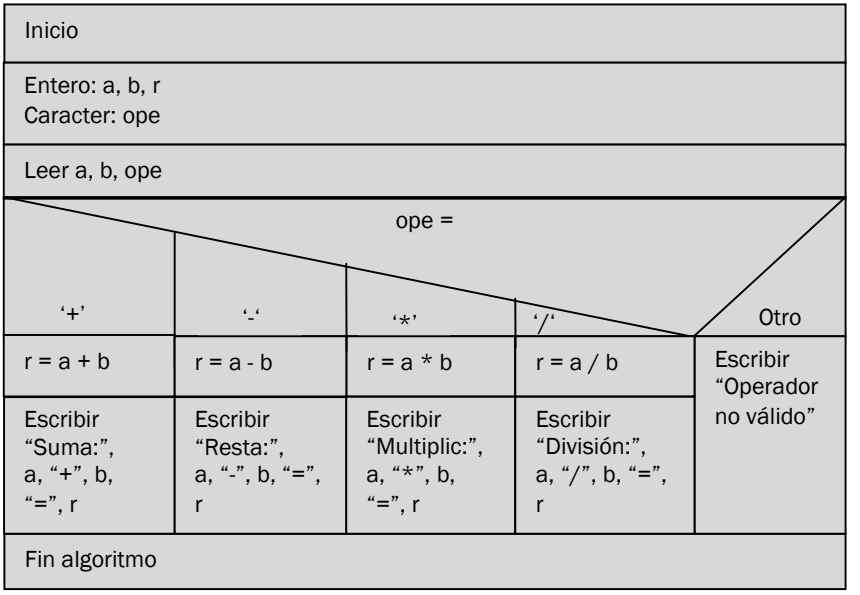
El resultado será:

Resta: 6 – 2 = 4

El algoritmo para solucionar este ejercicio requiere una decisión múltiple aplicada al operador y en cada alternativa de decisión se realiza la operación correspondiente y se muestra el resultado. El diagrama N-S de este algoritmo se presente en la figura 50.

Se declaran las variables: a, b, y r para primer número, segundo número y resultado, y ope para el operador. Al hacer la verificación del algoritmo, probando todos los operadores, se obtiene los resultados que se muestran en el cuadro 50.

Figura 50. Diagrama N-S del algoritmo Calculadora



Cuadro 50. Verificación del algoritmo calculadora

Ejec	a	b	ope	r	Salida
1	3	4	+	7	Suma: 3 + 4 = 7
2	6	2	-	4	Resta: 6 – 2 = 4
3	2	5	*	10	Multiplic: 2 * 5 = 10
4	27	9	/	3	División: 27 / 9 = 3
5	8	7	X		Operador no válido

4.2.7 Ejercicios propuestos

Diseñar algoritmos para solucionar los siguientes problemas y representarlos mediante pseudocódigo, diagramas de flujo y diagramas N-S.

1. Para integrar la selección de baloncesto, además de ser un buen jugador, es necesario una estatura de 1.70 m como mínimo. Dada la estatura de un aspirante decidir si es apto.

2. Dados dos enteros, se desea saber si alguno de ellos es múltiplo del otro.

3. Dado dos números racionales a/b y d/e se desea saber si son equivalentes y si no los son, cuál es el mayor y cuál el menor.

4. Una empresa está pagando una bonificación a sus empleados dependiendo del tiempo de servicio y del estado civil, así: para empleados solteros: si llevan hasta cinco años, el 2% del sueldo; entre 6 y 10 años el 5%; más de 10 años el 10%. Para empleados casados: si llevan hasta 5 años, el 5% del sueldo; entre 6 y 10 años el 10%, más de 10 años el 15%.

5. En un almacén se hace un descuento del 8% a los clientes cuya compra sea superior a un millón y del 5% si es superior $500.000 y menor o igual a un millón ¿Cuánto pagará una persona por su compra?

6. Se sabe que un año es bisiesto cuando es divisible para cuatro y no para 100 o cuando es divisible para cuatrocientos. Se desea determinar si un año n es bisiesto.

7. Se requiere un algoritmo que lea tres números y los ordene de forma ascendente.

8. Se tiene un recipiente cilíndrico de radio r y altura h y una caja de ancho a, largo b y altura c. Se desea saber cuál de ellos tiene mayor capacidad de almacenamiento.

9. Un supermercado hace un descuento del 10% por la compra de 10 unidades o más del mismo artículo ¿Cuánto deberá pagar un cliente por su compra?

10. Una compañía de seguros abrió una nueva agencia y estableció un programa para captar clientes, que consiste en lo siguiente: si el monto por el que se contrata el seguro es menor que $5.000. 000 pagará el 3%, si el monto es mayor o igual a $5.000 000 y menor de 20.000.000 pagará el 2% y si el monto es mayor o igual a 20.000.000 el valor a pagar será el 1.5 % ¿Cuál será el valor de la póliza?

11. La universidad ABC tiene un programa de estímulo a estudiantes con buen rendimiento académico. Si el promedio en las cuatro materias que se cursan en cada semestre es mayor o igual a 4.8, el estudiante no debe pagar matrícula para el siguiente semestre; si el promedio es superior o igual a 4.5 y menor de 4.8, el estudiante tendrá un descuento del 50%; para promedios mayores o iguales a 4.0 y menores a 4.5 se mantiene el valor; mientras que para promedios menores se incrementa en un 10% respecto al semestre anterior. Dadas las notas definitivas de las materias determinar el valor de la matrícula para el siguiente semestre.

12. Una distribuidora de computadores hace un descuento dependiendo de la cantidad comprada. En compras de menos de cinco equipos no hay descuento, en compras de cinco y hasta nueve equipos hará un descuento del 5% y en compras de 10 o más el descuento será del 10%. ¿Cuál será el valor del descuento? ¿Cuánto pagará el cliente?

13. Un restaurante ofrece servicio a domicilio con las siguientes condiciones. Si el pedido es superior a $ 20.000 el servicio a domicilio no tiene ningún costo adicional, si es mayor a $ 10.000 y hasta $20.000 se cobrará un incremento de $2.000, y si es menor a 10.000 tendrá un incremento de $4.000. ¿Qué valor deberá cancelar el cliente?

14. Una distribuidora tiene tres vendedores a los que paga un sueldo básico más una comisión del 5% sobre las ventas, siempre que éstas sean mayor o iguales a $ 1.000.000. Además, el vendedor que haya vendido más durante el mes recibirá un 2% adicional sobre el valor de las ventas, indiferente del monto de éstas. ¿Cuál será el sueldo de cada vendedor?

15. En una entrevista para contratar personal se tienen en cuenta los siguientes criterios: educación formal, edad y estado civil. Los puntajes son: para edades entre 18-24 años, 10 puntos; entre 25 - 30, 20 puntos; 31 - 40 años, 15 puntos; mayores de 40, 8 puntos. Para estudios de bachillerato 5 puntos, tecnológicos 8 puntos, profesionales 10 puntos, postgrado 15 puntos. Estado civil soltero 20 puntos, casado 15 puntos, unión libre 12 puntos, separado 18 puntos. Se requiere calcular el puntaje total para un entrevistado.

16. Dado un año y un mes, se desea saber cuántos días tiene dicho mes, teniendo en cuenta que febrero cambia dependiendo si el año es bisiesto.

17. Dado un número entero entre 1 y 9999 expresarlo en palabras

18. Una joyería establece el valor de sus productos según el peso y el material. Se requiere un algoritmo que lea un valor de referencia x y a partir de éste calcule el valor del gramo de material utilizado y el valor del producto. El gramo de plata procesado tiene un costo de 3x, el platino 5x y el oro 8x.

19. Una empresa otorga una prima especial de maternidad / paternidad a sus empleados, dependiendo del número de hijos: para empleados que no tienen hijos, no hay prima, para un hijo la prima será del 5% del sueldo, para dos hijos será del 8%, para tres hijos el 10%, para cuatro el 12%, para un número mayor el 15%.

20. Un almacén de electrodomésticos tiene la siguiente política de venta: en ventas de contado se hace un descuento del 10%, mientras que en ventas a crédito se carga un porcentaje del valor del producto, dependiendo del número de cuotas mensuales, y se divide para el número de cuotas. Para 2 meses no hay interés de financiación, para 3 meses se incrementa el valor en 5%, para 4 meses el 10%, para 5 meses el 15% y para 6 meses el 20%. Se requiere conocer el valor del producto, el valor a pagar por el cliente, el valor del descuento o incremento y el valor de cada cuota en caso de que las haya.

21. Se requiere un algoritmo para calcular el área de una de las principales figuras geométricas: triangulo, rectángulo, círculo, rombo y trapecio. Se sabe que las áreas se obtienen con las siguientes fórmulas: área triángulo = b*h/2, área rectángulo = b*h, área círculo = Pi * radio^2, área rombo = D*d/2, área trapecio = ½h(a+b)

22. Se requiere un algoritmo para facturar llamadas telefónicas. El valor del minuto depende del tipo de llamada, así: local = $ 50, regional = 100, larga distancia nacional = 500, larga distancia internacional = 700, a celular = 200. Las llamadas de tipo regional y larga distancia nacional tienen un descuento del 5% si su duración es de 5 minutos o más.

23. Una empresa dedicada a alquilar vehículos requiere un programa para calcular el valor de cada uno de los préstamos. El valor se determina teniendo en cuenta el tipo de vehículo y el número de días que el usuario lo utiliza. El valor diario por categoría es: automóvil = $ 20000, campero = $ 30000, microbús = $ 50000 y camioneta = $ 40000.

24. La universidad José Martí otorga una beca por el 100% del valor de la matrícula al estudiante con mayor promedio en cada grupo y por el 50% al segundo mejor promedio. Conociendo los nombres y promedios de los cuatro mejores estudiantes de un grupo se desea conocer los nombres de quienes recibirán la beca.

25. Una empresa patrocinadora de atletas apoya a los deportistas que en las marcas de entrenamiento del último mes cumplen estas condiciones: el tiempo mayor no supera los 40 minutos, el tiempo menor sea inferior a 30 minutos y el tiempo promedio es menor a 35. Se requiere un algoritmo para decidir si un deportista cumple las condiciones para ser patrocinado.

26. Un estudiante de ingeniería necesita tres libros: Cálculo, Física y Programación, cuyos precios son: *v1, v2, v3* respectivamente. El estudiante cuenta con una cantidad *x* de dinero que ha recibido como apoyo económico para este fin. Se requiere un algoritmo que le ayude a decidir si puede comprar los tres, dos o un libros. El estudiante desea dar prioridad a los libros de mayor valor, ya que el dinero sobrante deberá devolverlo y los libros de menor valor serán más fácil de adquirir con recursos propios.

4.3 ESTRUCTURAS DE ITERACIÓN

Este tipo de estructuras también se conocen como ciclos, bucles o estructuras de repetición y son las que se utilizan para programar que una o más acciones se ejecuten repetidas veces. Son muy importantes en programación ya que hay muchas actividades que deben ejecutarse más de una vez.

Considere el caso de una aplicación para elaborar una factura. El cliente puede comprar un producto, pero también puede comprar dos, tres o más. Las mismas operaciones ejecutadas para facturar el primer producto deben ejecutarse para el segundo, el tercero, ..., y el enésimo producto. Las estructuras iterativas permiten que se escriba las instrucciones una vez y se utilicen todas las que sea necesario.

Todas las estructuras de repetición tienen como mínimo dos partes: definición y cuerpo. La definición incluye una condición que indica cuántas veces ha de repetirse el cuerpo del ciclo; y, si no se conoce con anticipación el número de repeticiones, la condición indicará las circunstancias en que el cuerpo del ciclo se repite, por ejemplo utilizando una variable de tipo conmutador. El cuerpo del ciclo está conformado por la instrucción o el conjunto de instrucciones que incluye la estructura de repetición y que serán ejecutadas repetidas veces

4.3.1 Control de iteraciones

Para establecer la condición de iteración y tener control sobre la misma, casi siempre, es necesario utilizar una variable, ya se trate de un contador o un conmutador. A esta variable se la denomina **variable de control de ciclo** [Joyanes y Zahonero, 2002].

Una variable de control de iteraciones tiene tres momentos:

Inicialización. Para que una variable sea utilizada en la definición de una estructura iterativa es necesario conocer su valor inicial. Ya que se trata siempre de contadores o conmutadores, su valor no será leído en la ejecución del programa y es necesario que el programador lo asigne antes de incluir la variable en una condición. Si la variable no se inicializa puede ocurrir que el ciclo no se ejecute ninguna vez o que se ejecute infinitamente.

Evaluación. La evaluación de la variable puede realizarse antes o después de cada iteración, esto depende de la estructura utilizada, para decidir si ejecuta una vez más el cuerpo de ciclo o si sale y continúa con la ejecución de las demás instrucciones.

Actualización. Si la variable se inicializa en un valor que permite la ejecución del ciclo es necesario que, en cada iteración, se actualice de manera que en algún momento de la ejecución se produzca la salida del ciclo. Si el bucle está controlado por una variable y esta no se actualiza se genera un ciclo infinito.

Al especificar iteraciones controladas por una variable es importante formularse las preguntas: ¿Cuál es el valor inicial de la variable? ¿Qué valor debe tener la variable para que el cuerpo del ciclo se repita? ¿Cómo cambia la variable? Las respuestas a estas preguntas obligarán a definir los tres momentos referidos y evitar errores de lógica.

4.3.2 Estructura MIENTRAS

Este ciclo consiste en un conjunto de instrucciones que se repiten mientras se cumpla una condición. De igual manera que en las decisiones, la condición es evaluada y retorna un valor lógico que puede ser verdadero o falso. En el caso del ciclo *mientras* las instrucciones contenidas en la estructura de repetición se ejecutarán solamente si al evaluar la condición se genera un valor *verdadero*; es decir, si la condición se cumple; en caso contrario, se ejecutará la instrucción que aparece después de *Fin mientras*.

A diferencia de otros ciclos, la estructura *mientras* comienza evaluando la expresión condicional, si el resultado es *verdadero* se ejecutan las instrucciones del cuerpo del ciclo; al encontrarse la línea *fin mientras* se vuelve a evaluar la condición, si ésta se cumple se ejecutan nuevamente las instrucciones y así sucesivamente hasta que la condición deje de cumplirse, en cuyo caso, el control del programa pasa a la línea que aparece después de *fin mientras*. Si en la primera pasada por el ciclo la condición no se cumple las instrucciones que están dentro del ciclo no se ejecutarán ni una sola vez.

En pseudocódigo, el ciclo mientras se escribe de la siguiente forma:

Mientras *<condición>* ***hacer***
    Instrucciones que se repiten . . .
Fin mientras

En diagrama de flujo el ciclo mientras puede representarse mediante una decisión y un conector como se aprecia en la figura 51 o mediante un hexágono con dos entradas y dos salidas como se muestra en la figura 52. La primera forma es cómo se ha utilizado tradicionalmente, en este caso el ciclo no es explícito en el diagrama y sólo se identifica cuando se ejecuta paso a paso el algoritmo; en la segunda forma, utilizando un símbolo que denota iteración, el ciclo es evidente. El programa DFD que se utiliza comúnmente para elaborar diagramas de flujo utiliza la segunda forma, quizá por ello cada día es más común la utilización del símbolo de iteración.

En diagrama de Nassi – Shneiderman se representa mediante una caja para la definición del ciclo y cajas internas para las instrucciones que se repiten, como se muestra en la figura 53.

Una vez conocida la representación de esta estructura iterativa es necesario aprender cómo se utiliza en el diseño de algoritmos, para ello se presentan algunos ejemplos.

Figura 51. Representación ciclo mientras en diagrama de flujo (versión 1)

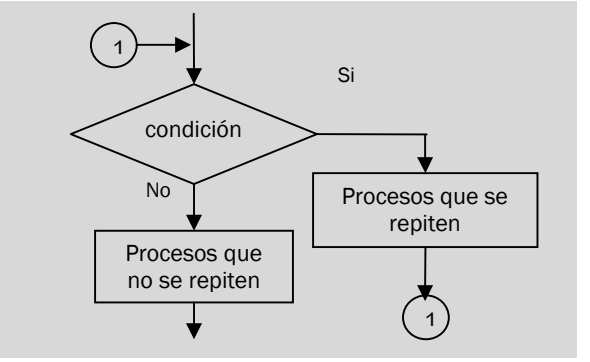


Figura 52. Representación ciclo mientras en diagrama de flujo (versión 2)

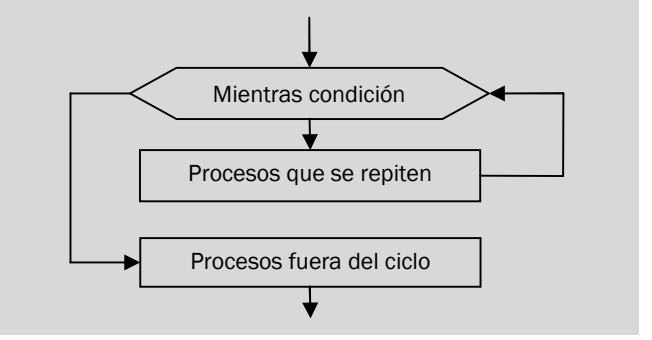
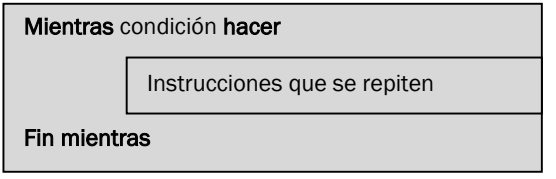


Figura 53. Representación ciclo mientras en diagrama N-S



Ejemplo 26. Generar números

Este algoritmo utiliza iteraciones para generar y mostrar los números del 1 al 10.

Lo primero por hacer es declarar una variable de tipo entero para el control del ciclo y la misma se utilizará para mostrar los números. La variable se inicializa en 1, luego se define el ciclo, dentro de éste se muestra el número y se incrementa la variables (contador), de esta manera se llevan a cabo las tres operaciones fundamentales para controlar un ciclo mediante una variable de tipo contador: se inicializa, se evalúa en la definición del ciclo y se actualiza aproximándose progresivamente a 10, para dar fin al ciclo. El pseudocódigo se presenta en el cuadro 51, el diagrama de flujo en la figura 54 y el diagrama N-S en la figura 55.

Cuadro 51. Pseudocódigo del algoritmo para generar números

```
1.  Inicio
2.      Entero:  num = 1
3.      Mientras num <= 10 hacer
4.          Escribir num
5.          num = num + 1
6.      Fin mientras
7.  Fin algoritmo
```

En el pseudocódigo, el ciclo se encuentra entre las líneas 3 y 6, en la 3 se define el ciclo y en la 6 termina, las líneas 4 y 5 corresponden al cuerpo del ciclo; es decir, las instrucciones que se repiten. Como el ciclo *mientras* necesita que la variable esté inicializada, esta operación se lleva a cabo en la línea 2. En este caso particular, cada vez que la ejecución llegue a la línea 6 se producirá un retorno a la línea 3 para evaluar la condición del ciclo, si la condición es verdadera se ejecutan las líneas 4 y 5, pero si es falsa la ejecución salta a la línea 7.

Figura 54. Diagrama de flujo del algoritmo para generar números

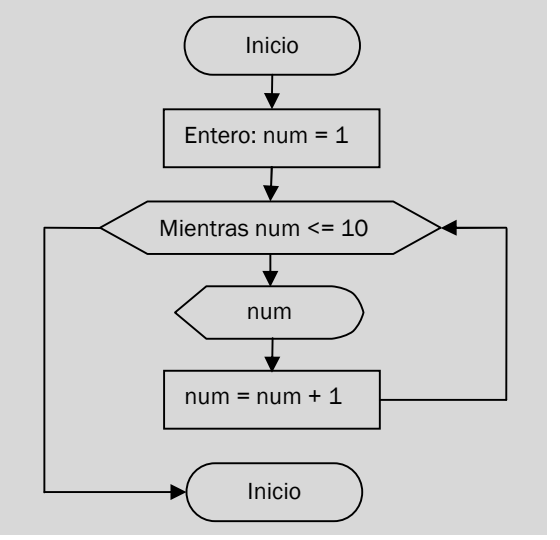


Figura 55. Diagrama N-S del algoritmo para generar números

Inicio	
Entero: num = 1	
Mientras num <= 0 **hacer**	
	Escribir num
	Num = num + 1
Fin mientras	
Fin algoritmo	

En el diagrama de flujo se aprecia con más claridad el comportamiento de la estructura iterativa. Al declarar la variable se le asigna un valor inicial (1), en la definición del ciclo se establece la condición (num <= 10), si ésta se cumple la ejecución continúa por el flujo que sale hacia abajo, se muestra la variable y se incrementa en uno, luego vuelve a la definición del ciclo para evaluar la

condición, si es verdades se ejecutan nuevamente las instrucciones, si es falsa sale por la izquierda y va al fin del algoritmo. El ciclo termina cuando num =11.

En el diagrama N-S también es fácilmente identificable el ciclo ya que toda la estructura se encuentra en una misma caja y el cuerpo del ciclo aparece internamente. Sólo cuando el ciclo ha terminado se pasa a la siguiente caja.

Para verificar el funcionamiento de este algoritmo se realiza la prueba de escritorio cuyos resultados se muestran en el cuadro 52.

Cuadro 52. Verificación algoritmo para generar números

Iteración	Num	Salida
	1	
1	2	1
2	3	2
3	4	3
4	5	4
5	6	5
6	7	6
7	8	7
8	9	8
9	10	9
10	11	10

Ejemplo 27. Divisores de un número

Se requiere un algoritmo para mostrar los divisores de un número en orden descendente.

Antes de comenzar a solucionar el ejercicio es necesario tener claridad sobre lo que se entiende por divisor de un número.

Dados dos números enteros a y b, se dice que b es divisor de a si se cumple que al efectuar una división entera a/b el residuo es 0. Por ejemplo, si se toman los números 27 y 3, se puede comprobar que 3 es divisor de 27 dado que al hacer la división 27/3 = 9 y el residuo es 0. Ahora bien, para mayor facilidad, el residuo de una división entera se puede obtener directamente utilizando el operador Mod.(Los operadores aritméticos se tratan en la sección 2.3.1)

Volviendo al ejercicio, en éste se pide que se muestren todos los divisores de un número comenzando por el más alto; por tanto, la solución consiste en tomar los números comprendidos entre el número ingresado y el número uno, y para cada valor examinar si cumple con la condición de ser divisor del número. Para ello es necesario utilizar una variable de control que comience en el número y disminuya de uno en uno hasta 1.

Si se toma, por ejemplo, el número 6, se deben examinar todos los números entre 6 y 1 y seleccionar aquellos que sean divisores de seis, así:

Número	contador	Número Mod contador	Divisor
6	6	0	Si
	5	1	No
	4	2	No
	3	0	Si
	2	0	Si
	1	0	Si

En la última columna se puede constatar que los divisores de seis, en orden descendente son: 6, 3, 2, 1.

Un segundo ejemplo: si se toma el número 8, se genera la lista de números desde ocho hasta uno y se verifica para cada uno si es o no divisor de ocho.

Número	Contador	Número Mod contador	Divisor
8	8	0	Si
	7	1	No
	6	2	No
	5	3	No
	4	0	Si
	3	2	No
	2	0	Si
	1	0	Si

De esto se obtiene que los divisores de 8, en orden descendente, son: 8, 4, 2, 1.

Como se observa en los ejemplos anteriores la clave para encontrar los divisores está en generar la lista de números, y para ello es apropiado utilizar una estructura iterativa y una variable de tipo contador, como se aprecia en el pseudocódigo del cuadro 53.

Cuadro 53. Pseudocódigo algoritmo divisores de un número

```
1.  Inicio
2.      Entero: num, cont
3.      Leer num
4.      cont = num
5.      Mientras cont >= 1 Hacer
6.          Si num Mod cont = 0 entonces
7.              Escribir cont
8.          Fin si
9.          cont = cont – 1
10.     Fin mientras
11. Fin algoritmo
```

Ya se mencionó la importancia de utilizar un ciclo para generar una lista de números, pero no se muestran todos los números ya que sólo se requieren los que cumplen la condición de ser divisores del número leído. La variable de control del ciclo se inicializa en el número ingresado, el ciclo se ejecuta mientras esta variable sea mayor a 1 (línea 5) y la variable se decrementa de uno en uno (línea 9). En el cuadro 54 se muestra los resultados de la verificación con los números 6 y 8.

Cuadro 54. Verificación algoritmo divisores de un número

Ejecución	num	cont	Salida
1	6	6	6
		5	
		4	
		3	3
		2	2
		1	1
		0	
2	8	8	8
		7	
		6	
		5	
		4	4
		3	
		2	2
		1	1
		0	

Las representaciones mediante diagrama de flujo y diagrama N-S se presentan en las figuras 56 y 57.

Figura 56. Diagrama de flujo algoritmo divisores de un número

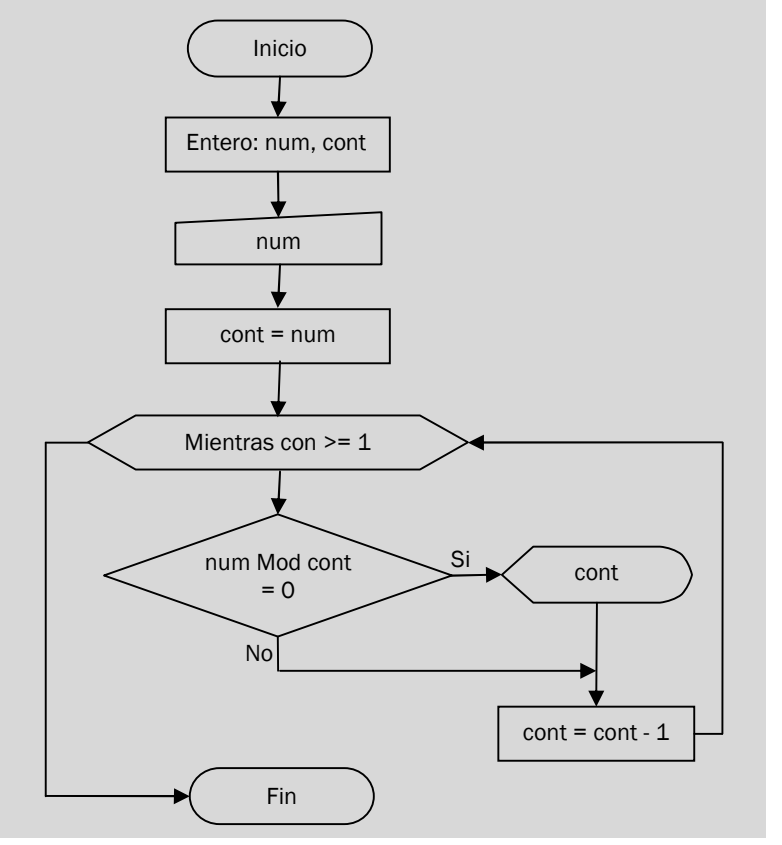
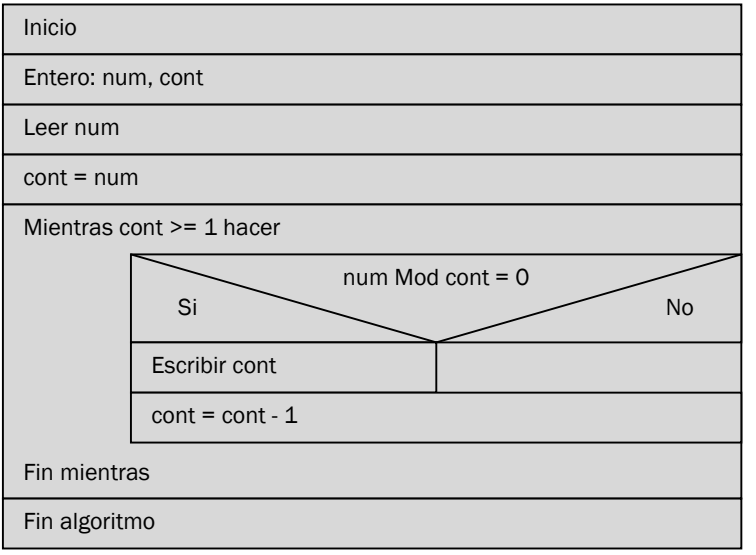


Figura 57. Diagrama N-S del algoritmo divisores de un número



4.3.3 Estructura HACER MIENTRAS

Esta estructura se propone como alternativa al ciclo *Repita – Hasta que*, por cuanto éste ya no figura en los nuevos lenguajes de programación. El ciclo *Hacer - Mientras*, permite ejecutar repetidas veces una instrucción o un grupo de instrucciones, con la particularidad de que evalúa la condición que controla el ciclo después de cada iteración; es decir que, la primera evaluación de la condición se hace después de haber ejecutado las instrucciones del ciclo, lo que garantiza que el cuerpo del bucle se ejecutará al menos una vez.

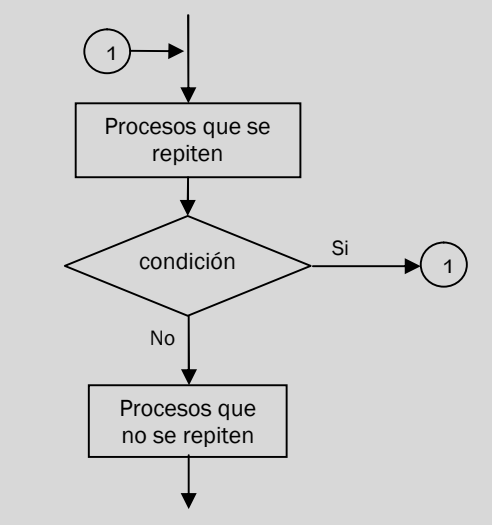
Cuando en un programa se encuentra la palabra *Hacer*, se siguen ejecutando las líneas que están a continuación, en el momento en que se encuentra la instrucción *Mientras*, se evalúa la condición asociada, la cual debe retornar un valor lógico (verdadero o falso). Si el valor es verdadero el control regresa a la instrucción

Hacer y ejecuta nuevamente las instrucciones que le siguen. Si la condición es falsa la ejecución continúa en la instrucción que está después de *Mientras*. En pseudocódigo se escribe de así:

Hacer
Instrucción 1
Instrucción 2
. . .
Instrucción n
Mientras <condición>

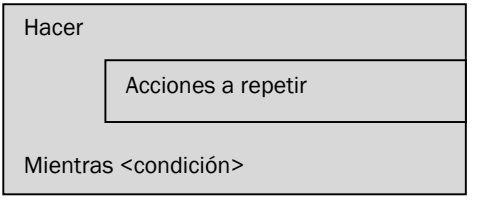
En diagrama de flujo no se conoce un símbolo para representar este tipo de estructura; por ello, se lo implementa mediante un símbolo de decisión y un conector para regresar a un proceso anterior y repetir la secuencia mientras la condición sea verdadera, como se muestra en la figura 58.

Figura 58. Ciclo Hacer - mientras en Diagrama de Flujo



En diagrama N-S se utiliza una caja para representar el ciclo y cajas internas para las acciones que conforman el cuerpo del ciclo, como se presenta en la figura 59.

Figura 59. Ciclo Hacer – Mientras en Diagrama N-S



Ejemplo 28. Sumar enteros positivos

Este algoritmo lee números enteros y los suma mientras sean positivos. Cuando se introduce un número negativo o el cero el ciclo termina, se muestra la sumatoria y termina el algoritmo. El pseudocódigo se muestra en el cuadro 55.

Cuadro 55. Pseudocódigo del algoritmo sumar enteros

```
1.  Inicio
2.      Entero: num = 0, suma = 0
3.      Hacer
4.          suma = suma + num
5.          Leer num
6.      Mientras num > 0
7.      Escribir suma
8.  Fin algoritmo
```

En este algoritmo se declara dos variables y se inicializan, esto debido a que la primera instrucción a ejecutar es la suma y es necesario tener control sobre el contenido de las variables. En la línea 5 se lee un número, pero para efectuar la suma es preciso volver atrás, lo cual está sujeto a la condición de ser entero positivo. Si se cumple la condición la ejecución regresa a la línea 3, en la 4

efectúa la suma y vuelve a leer un número, y así sucesivamente hasta que se ingrese el cero o un número negativo. Sólo al terminar el ciclo muestra el resultado de la sumatoria y termina la ejecución. En la Figura 60 se presenta el diagrama de flujo y en la 61 el diagrama N-S.

Figura 60. Diagrama de Flujo del algoritmo sumar enteros

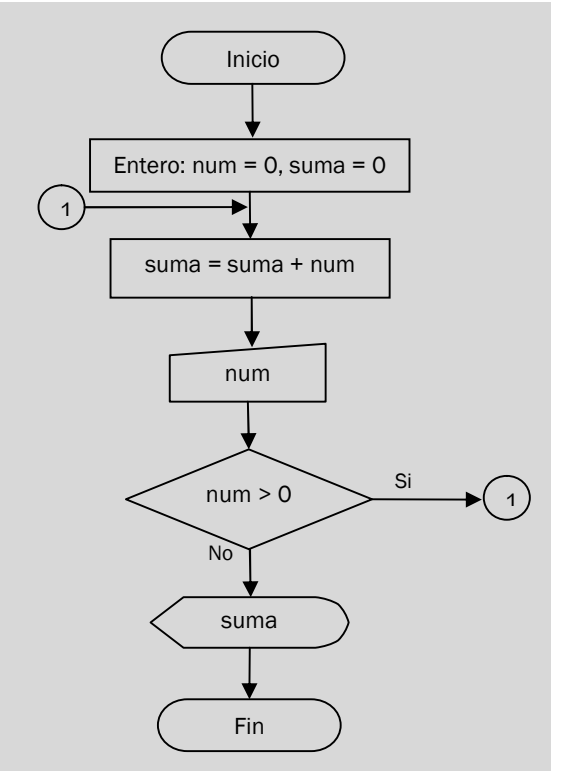
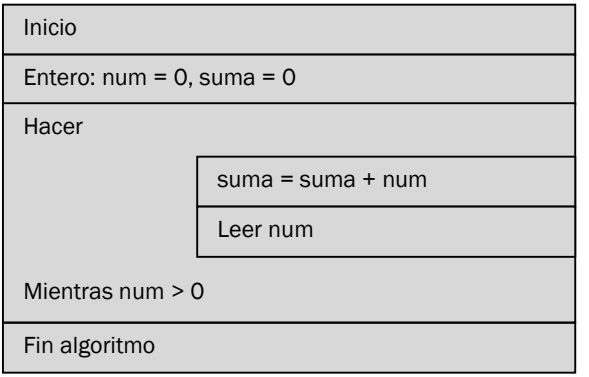


Figura 61. Diagrama N-S del algoritmo sumar enteros



En el cuadro 56 se muestra los resultados de la verificación del algoritmo.

Cuadro 56. Verificación del algoritmo sumar enteros

Iteración	Num	Suma	num > 0	Salida
	0	0		
1	4	4	V	
2	8	12	V	
3	1	13	V	
4	5	18	V	
5	0		F	
				18

Ejemplo 29. Número binario

Dado un número en base 10, este algoritmo calcula su equivalente en base 2; es decir, convierte un decimal a binario.

Una forma de pasar un valor en base 10 (decimal) a base 2 (binario) consiste en dividir el valor sobre 2 y tomar el residuo, que puede ser 0 o 1, retomar el cociente y dividir nuevamente para 2.

Esta operación se efectúa repetidas veces hasta que el resultado sea 0. No obstante, esta solución tiene una breve dificultad que hay que resolver y es que los residuos forman el número en binario, pero deben leerse en sentido contrario; es decir, el primer número que se obtuvo ocupa la posición menos significativa mientras que el último debe ocupar la primera posición en el número binario.

Considérese el número 20

20 / 2 = 10	Residuo = 0
10 / 2 = 5	Residuo = 0
5 / 2 = 2	Residuo = 1
2 / 2 = 1	Residuo = 0
1 / 2 = 0	Residuo = 1

Si los números se toman en el orden que se generan se obtiene: 00101, este número equivale a 5 en decimal, el cual está muy lejos del valor original 20.

Los números que se generan se toman en orden inverso, el último residuo es el primer dígito del binario; por lo tanto, el número binario generado en esta conversión es: 10100. Este sí es equivalente a 20.

Ahora, la pregunta es ¿cómo hacer que cada nuevo dígito que se obtiene se ubique a la izquierda de los anteriores?

Para solucionar este asunto, se utiliza una variable para llevar control de la posición que debe ocupar el dígito dentro del nuevo número. La variable inicia en 1 y en cada iteración se la multiplica por 10 para que indique una posición cada vez más significativa. Entonces se tiene que:

```
Entrada: número en base 10 (decimal)
Salida: número en base 2
Proceso:
        dígito = decimal Mod 2
        binario = binario + digito * posición
```

decimal = decimal / 2
posición = posición * 10

La solución a este problema se presenta en el cuadro 57.

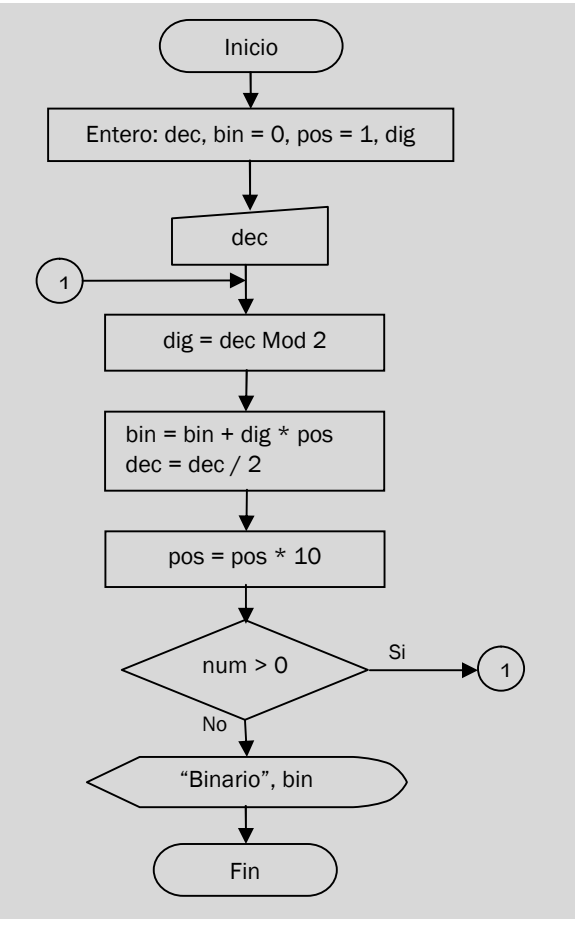
Cuadro 57. Pseudocódigo del algoritmo número binario

```
1.  Inicio
2.      Entero:  dec, bin = 0, pos = 1, dig
3.      Leer dec
4.      Hacer
5.          dig = dec Mod 2
6.          bin = bin + dig * pos
7.          dec = dec / 2
8.          pos = pos * 10
9.      Mientras dec > 0
10.     Escribir "Binario:", bin
11. Fin algoritmo
```

En el algoritmo se aprecia que las operaciones de las líneas 5 a 8 se repiten mientras haya un número para dividir ($dec > 0$). En la línea 5 se obtiene un digito (0 o 1) que corresponde al residuo de dividir el número decimal sobre dos, ese dígito se multiplica por la potencia de 10 para convertirlo en 10, 100, 1000 o cualquiera otra potencia si se trata de un uno, para luego sumarlo con los dígitos que se hayan obtenido anteriormente (línea 6). El número decimal se reduce cada vez a la mitad, de esta manera tiende a cero (línea 7), al tiempo que la potencia que indica la posición del siguiente uno se multiplica por 10 en cada iteración (línea 8).

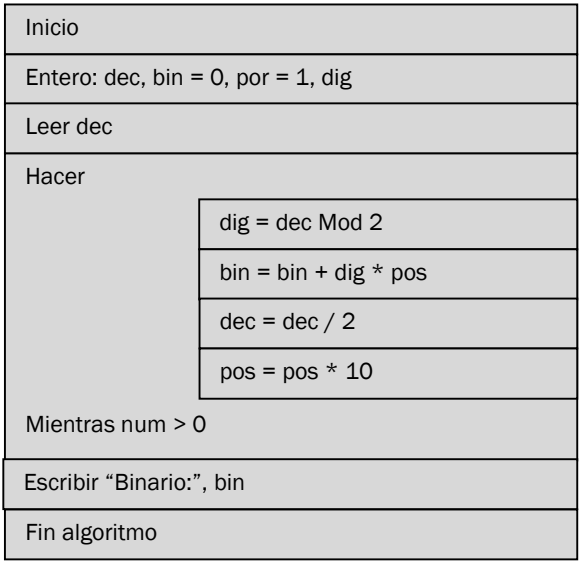
Los diagramas de flujo y N-S de este algoritmo se presentan en las figuras 62 y 63 respectivamente.

Figura 62. Diagrama de flujo del algoritmo número binario



Para probar este algoritmo se calcula el equivalente binario para los números 20 y 66 obteniendo como resultados 10100 y 1000010 respectivamente, los cuales son correctos. Los datos generados en las pruebas se presentan en el cuadro 58.

Figura 63. Diagrama N-S del algoritmo número binario



Cuadro 58. Verificación del algoritmo número binario

Iteración	dig	dec	Bin	pos	Salida
			0	1	
		20	0		
1	0	10	0	10	
2	0	5	0	100	
3	1	2	100	1000	
4	0	1	100	10000	
5	1	0	10100	100000	Binario: 10100
		66	0	1	
1	0	33	0	10	
2	1	16	10	100	
3	0	8	10	1000	
4	0	4	10	10000	
5	0	2	10	100000	
6	0	1	10	1000000	
7	1	0	1000010		Binario: 1000010

4.3.4 Estructura PARA

Esta estructura, al igual que las anteriores, permite ejecutar repetidas veces una instrucción o un grupo de ellas, pero a diferencia de otras instrucciones de repetición, ésta maneja el valor inicial, el valor de incremento o decremento y el valor final de la variable de control como parte de la definición del ciclo.

Cuando al ejecutarse un algoritmo se encuentra una instrucción *para* la variable de control (contador) toma el valor inicial, se verifica que el valor inicial no sobrepase el valor final y luego se ejecutan las instrucciones del ciclo. Al encontrar la instrucción *fin para*, se produce el incremento y se vuelve a verificar que la variable de control no haya superado el límite admitido, y se vuelven a ejecutar las instrucciones que están dentro del ciclo, y así sucesivamente tantas veces como sea necesario hasta que se supere el valor final establecido.

El ciclo *para* termina en el momento en que la variable de control (contador) sobrepasa el valor final; es decir, que la igualdad está permitida y las instrucciones se ejecutan cuando el contador es igual al valor final.

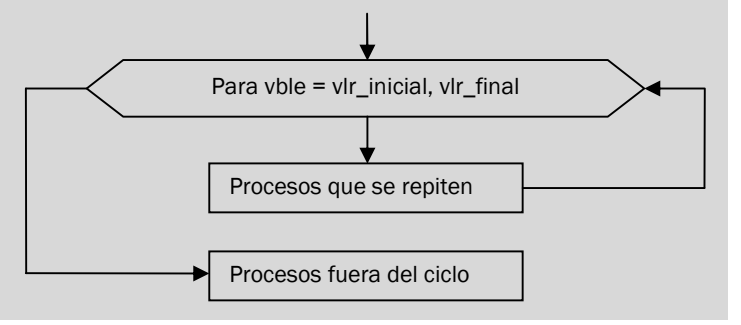
Algunos lenguajes de programación definen la sintaxis de este ciclo incluyendo una condición que se debe cumplir, de forma similar al ciclo mientras, y no un valor final para la variable, como se propone en pseudocódigo.

Este ciclo puede presentarse de tres maneras: la primera es la más común, cuando se produce un incremento de 1 en cada iteración, en cuyo caso no es necesario escribir explícitamente este valor. En pseudocódigo se expresa de la siguiente forma:

Para *variable = valor_inicial* ***hasta*** *valor_final* ***hacer***
    Instrucciones
Fin para

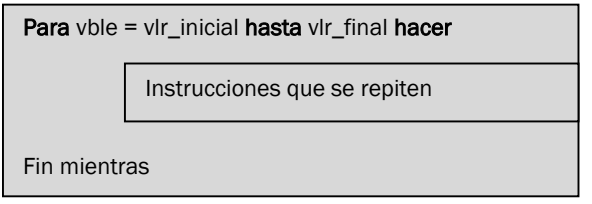
En diagrama de flujo se representa como se muestra en la figura 64 y en diagrama N-S, como la figura 65.

Figura 64. Ciclo para en Diagrama de Flujo (Versión 1)



En el diagrama de flujo no es necesario escribir todas las palabras, éstas se reemplazan por comas (,) excepto la primera.

Figura 65. Ciclo para en DiagramaN-S (Versión 1)



El segundo caso de utilización del ciclo *para* se presenta cuando el incremento es diferente de 1, en cuyo caso se escribirá la palabra *incrementar* seguida del valor a sumar en cada iteración.

En pseudocódigo se escribe:

Para *vble = vlr_inicial* ***hasta*** *vlr_final* ***incrementar*** *valor* ***hacer***
    Instrucciones que se repiten
Fin para

Diagrama de flujo y en diagrama N-S se representa como en la figuras 66 y 67.

Figura 66. Ciclo Para en Diagrama de Flujo (Versión 2)

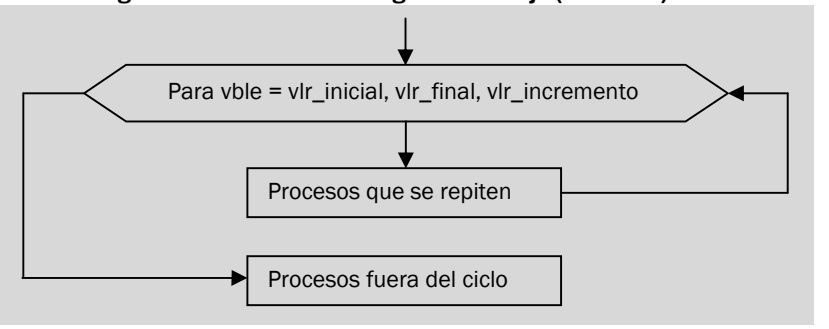
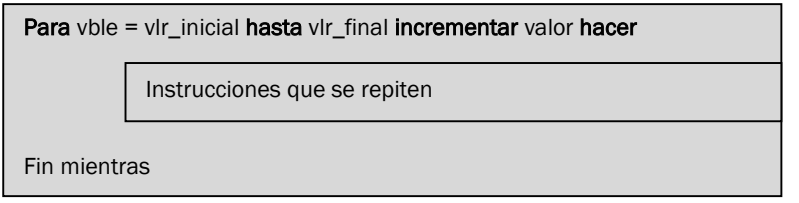


Figura 67. Ciclo Para en N-S (Versión 2)



El tercer caso se presenta cuando el ciclo *para* no se incrementa desde un valor inicial hasta un valor mayor, sino que disminuye desde un valor inicial alto hasta un valor menor. Para ello es suficiente con escribir *decrementar* en vez de *incrementar*.

En pseudocódigo se tiene:

Para contador = valor_inicial hasta valor_final decrementar valor
hacer
Instrucciones
Fin para

En este caso para que se ejecute la primera iteración será necesario que el valor inicial sea mayor que el valor final, en caso

contrario, simplemente pasará a ejecutarse la instrucción que sigue al *fin para*.

Es importante tener en cuenta que cuando la variable de control debe disminuir en cada iteración, siempre hay que escribir *decrementar* y el valor, aunque sea -1.

En diagrama de flujo será suficiente con anteponer el signo menos (-) al valor a decrementar.

Ejemplo 30. Sumatoria iterativa

Dado un número n, entero positivo, se calcula y se muestra la sumatoria de los números desde 1 hasta n.

En este caso se requiere una estructura iterativa para generar los números desde el uno hasta n y cada valor comprendido en este intervalo se almacena en una variable de tipo acumulador.

Ejemplo: si se introduce el número 3, la sumatoria será:

1 + 2 + 3 = 6

Si se introduce el número 6, la sumatoria será:

1 + 2 + 3 + 4 + 5 + 6 = 21

El ejercicio se puede sistematizar de la siguiente forma:

Entrada: número
Salida: sumatoria
Proceso:

$$\text{sumatoria} = \sum_{i=1}^{i=n} i$$

La solución de este ejercicio se presenta en el cuadro 59 en notación de pseudocódigo y en las figuras 68 y 69 los diagramas de flujo y N-S.

Cuadro 59. Pseudocódigo algoritmo sumatoria

```
1.  Inicio
2.      Entero: num, i, suma = 0
3.      Leer num
4.      Para i = 1 hasta num hacer
5.          suma = suma + i
6.      Fin para
7.      Escribir "Sumatoria:", suma
8.  Fin algoritmo
```

Figura 68. Diagrama de flujo de algoritmo sumatoria

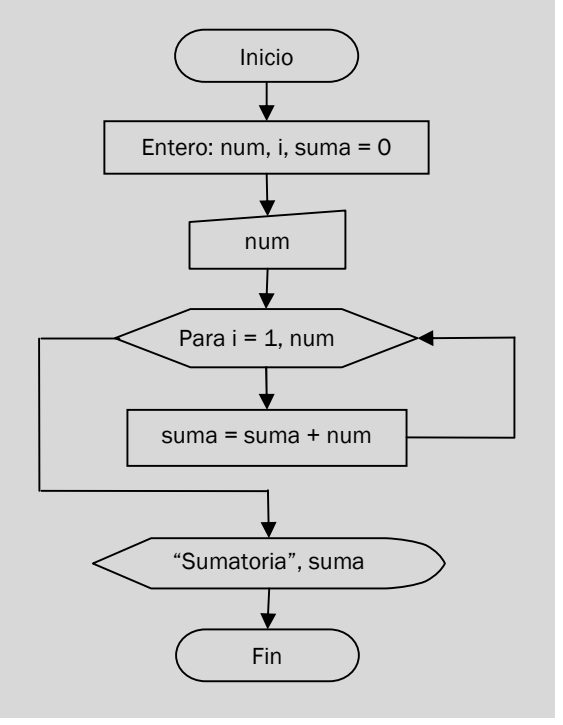
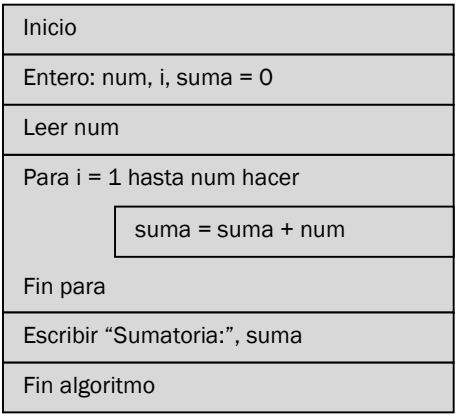


Figura 69. Diagrama N-S del algoritmo sumatoria



Dado que el ciclo *para* incrementa automáticamente la variable, dentro del ciclo solo aparece la instrucción para sumar los números generados.

En el cuadro 60 se muestra el comportamiento de las variables al ejecutar el algoritmo para calcular la sumatoria de los números del 1 al 6.

Cuadro 60. Verificación de algoritmo sumatoria

Iteración	Num	i	suma	Salida
			0	
	6		0	
1		1	1	
2		2	3	
3		3	6	
4		4	10	
5		5	15	
6		6	21	Sumatoria: 21

Ejemplo 31. Tabla de multiplicar

Comúnmente se conoce como tabla de multiplicar de un número a la lista de los primeros 10 múltiplos. Por ejemplo, la tabla del 2:

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

Para mostrarla de esta forma se puede utilizar la estructura iterativa *para*, la que hace variar el multiplicador desde 1 hasta 10 y en cada iteración se calcula el producto y muestra los datos. El pseudocódigo se muestra en el cuadro 61, los diagramas de flujo y N-S en las figuras 70 y 71.

Cuadro 61. Pseudocódigo del algoritmo tabla de multiplicar

```
1.  Inicio
2.      Entero:  m, n, r
3.      Leer m
4.      Para n = 1 hasta 10 hacer
5.          r = m * n
6.          Escribir m, '*', n, '=', r
7.      Fin para
8.  Fin algoritmo
```

Figura 70. Diagrama de flujo del algoritmo tabla de multiplicar

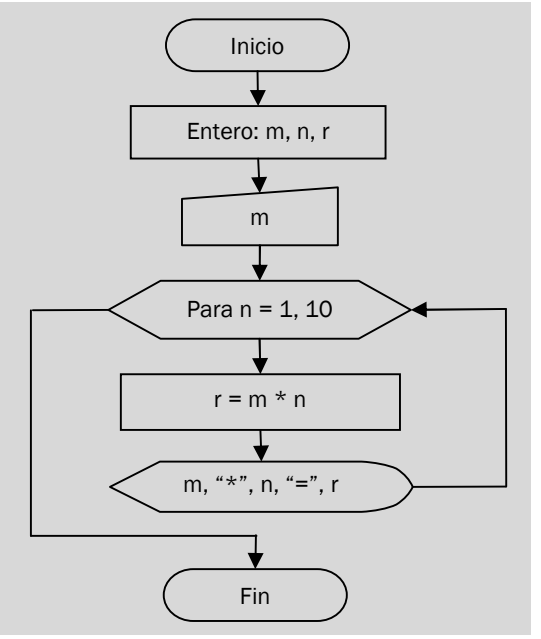


Figura 71. Diagrama N-S del algoritmo tabla de multiplicar

Inicio
Entero: m, n, r
Leer m
Para n = 1 hasta 10 hacer
r = m * n
Escribir m, “*”, n “=”, r
Fin para
Fin algoritmo

Para verificar el funcionamiento del algoritmo se genera la tabla del 4, los resultados se presentan en el cuadro 62.

Cuadro 62. Verificación del algoritmo tabla de multiplicar

Iteración	m	n	r	Salida
	4			
1		1	4	4 * 1 = 4
2		2	8	4 * 2 = 8
3		3	12	4 * 3 = 12
4		4	16	4 * 4 = 16
5		5	20	4 * 5 = 20
6		6	26	4 * 6 = 24
7		7	28	4 * 7 = 28
8		8	32	4 * 8 = 32
9		9	36	4 * 9 = 36
10		10	40	4 * 10 = 40

4.3.5 Estructuras Iterativas anidadas

Las estructuras iterativas, al igual que las selectivas, se pueden anidar; es decir, se pueden colocar una dentro de otra.

El anidamiento de ciclos se conforma por un ciclo externo y uno o más ciclos internos, donde cada vez que se repite el ciclo externo, los internos se reinician y ejecutan todas las iteraciones definidas (Joyanes, 2000).

Un ejemplo que permite ilustrar fácilmente el concepto de ciclos anidados es el reloj digital. El tiempo se mide en horas, minutos y segundos, las horas están conformadas por minutos y los minutos por segundos, de manera que si se escribe un ciclo para las horas, otro para los minutos y otro para los segundos, el ciclo de los minutos en cada iteración deberá esperar a que se ejecuten las 60

iteraciones del ciclo de los segundos (0..59) y el de las horas deberá esperar que se ejecute el de los minutos (0..59).

Ejemplo 32. Reloj digital

Para implementar un reloj digital es necesario declarar tres variables, para controlar: horas, minutos y segundos. Cada una de estas variables controla un ciclo, así: los segundos se incrementan desde 0 a 59, los minutos también desde 0 a 59 y las horas desde 1 a 12 o de 1 a 24.

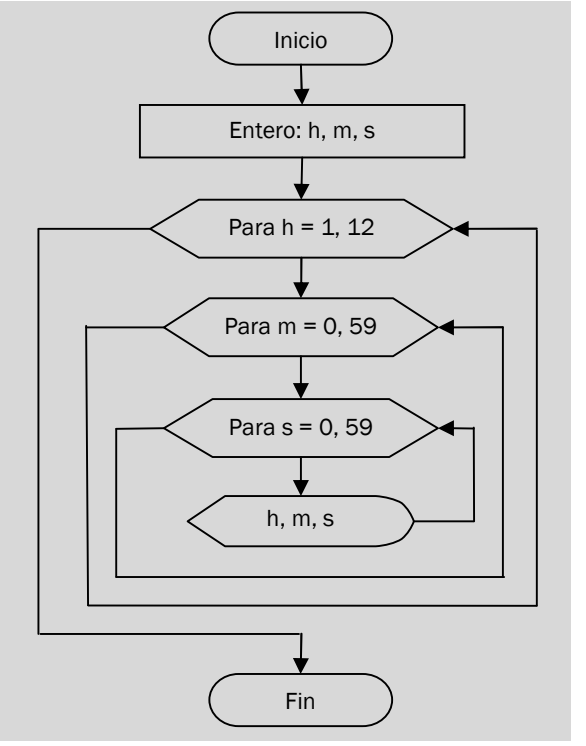
La variable que controla las horas se incrementa en uno cada vez que la variable que controla los minutos ha hecho el recorrido completo desde 0 a 59, a su vez, la variable de los minutos se incrementa en uno cada vez que la variable de los segundos ha completado su recorrido desde 0 a 59. Para que esto ocurra es necesario que el ciclo de las horas contenga al de los minutos y éste, al de los segundos, como se aprecia en el cuadro 63.

Cuadro 63. Pseudocódigo algoritmo Reloj digital

```
1.  Inicio
2.      Entero: h, m, s
3.      Para h = 1 hasta 12 hacer
4.          Para m = 0 hasta 59 hacer
5.              Para s = 0 hasta 59 hacer
6.                  Escribir h,":", m, ":", s
7.              Fin para
8.          Fin para
9.      Fin para
10. Fin algoritmo
```

Este mismo algoritmo expresado en diagrama de flujo se presenta en la figura 72 y en diagrama N-S en la 73.

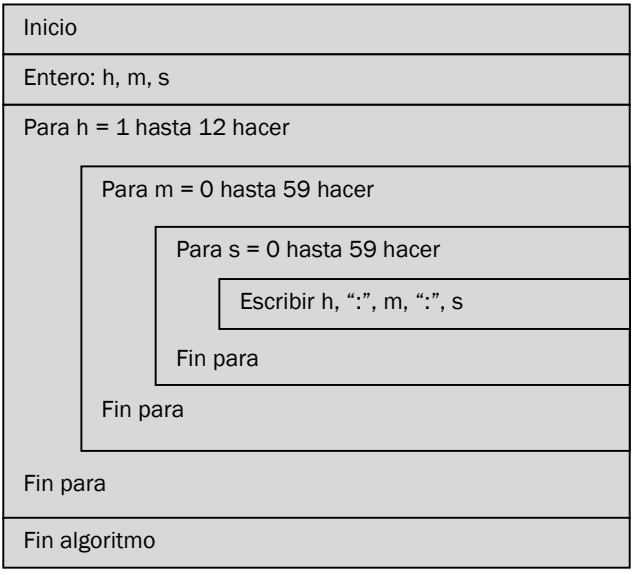
Figura 72. Diagrama de flujo del algoritmo Reloj digital



Al ejecutar este algoritmo se tendría una salida de la forma:

1:0:0
1:0:1
1:0:2
...
1:0:58
1:0:59
1:1:0
1:1:1
1:1:2
...

Figura 73. Diagrama N-S del algoritmo Reloj digital



1:59:58
1:59:59
2:0:0
2:0:1
2:0:2
...

Este algoritmo está diseñado para terminar la ejecución al llegar a 12:59:59. Para hacer que el reloj se ejecute indefinidamente sería necesario colocar los tres ciclos dentro de otro, para que al llegar a 12:59:59 se reinicie la variable h y vuelva a comenzar el conteo. El ciclo externo tendría que ser un ciclo infinito de la forma: *mientras verdadero hacer*.

Ejemplo 33. Nueve tablas de multiplicar

En un ejemplo anterior se utilizó un ciclo para generar la tabla de multiplicar de un número, en éste se utiliza ciclos anidados para

generar las tablas desde dos hasta 10. El primer ciclo está asociado con el multiplicando y hace referencia a cada una de las tablas a generar, por tanto toma valores entre 2 y 10; el ciclo interno está relacionado con el multiplicador, se encarga de generar los múltiplos en cada una de las tablas, en consecuencia toma valores entre 1 y 10. El pseudocódigo se presenta en el cuadro 64.

Cuadro 64. Pseudocódigo para Nueve tablas de multiplicar

```
1.  Inicio
2.      Entero: m, n, r
3.      Para m = 2 hasta 10 hacer
4.          Escribir "Tabla de multiplicar del ", m
5.          Para n = 1 hasta 10 hacer
6.              r = m * n
7.              Escribir m, "*", n, "=", r
8.          Fin para
9.      Fin para
10. Fin algoritmo
```

Al ejecutar este algoritmo se tendrá una salida de la forma:

Tabla de multiplicar del 2
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
...
Tabla de multiplicar del 3
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
...
Y así sucesivamente hasta

10 * 10 = 100

Los diagramas de flujo y N-S para este algoritmo se presentan en las figuras 74 y 75.

Figura 74. Diagrama de flujo para 9 tablas de multiplicar

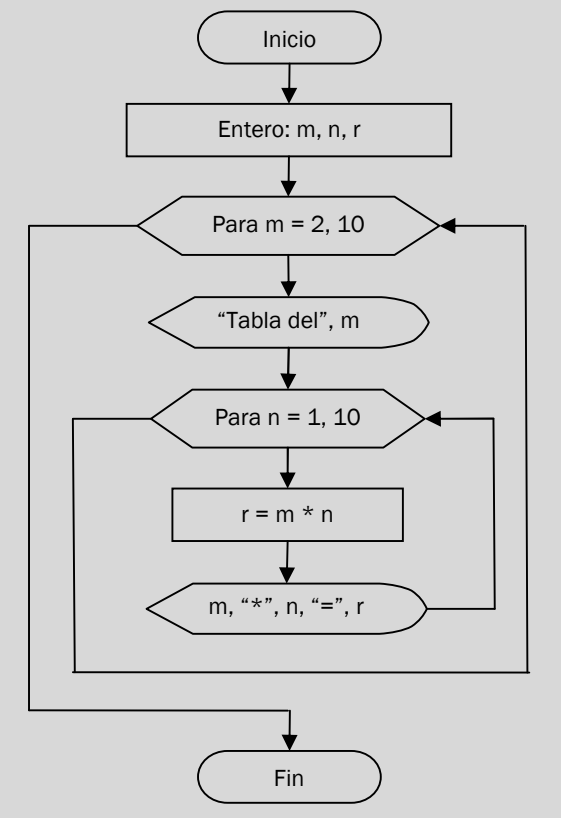
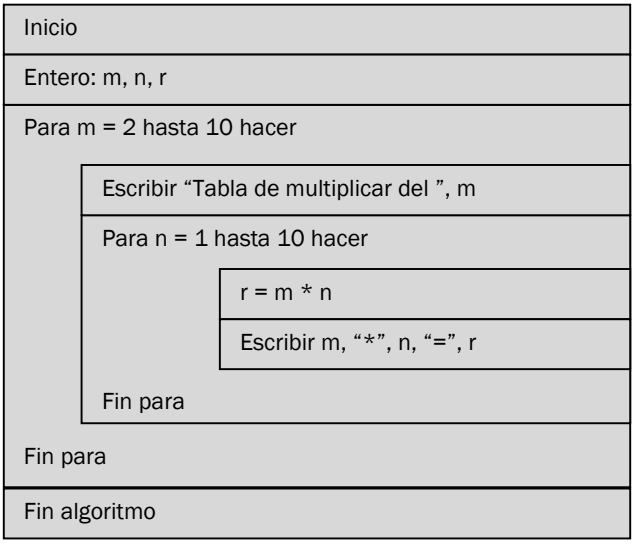


Figura 75. Diagrama N-S para Nueve tablas de multiplicar



4.3.6 Más ejemplos de iteraciones

Ejemplo 34. Mayor, menor y promedio de n números

Este ejercicio consiste en leer n números y luego reportar el número menor y el número mayor de la lista y calcular el valor promedio.

Supóngase que el usuario ingresa la siguiente lista de datos:
2
14
55
6
12
34
17

Se tiene que:
Cantidad de números ingresados: 7
Número mayor: 55
Número menor: 2
Sumatoria: 140
Promedio = 140 / 7 = 20

Para solucionar este problema, lo primero que hay que plantear es cómo se sabrá cuando terminar la lectura de datos, ya que no se especifica la cantidad de números que serán ingresados y en el planteamiento del problema no se da ninguna condición que permita saber cuándo terminar el ciclo. Este tipo de problemas es muy común, no se conoce con antelación la cantidad de datos a procesar.

Para implementar ciclos cuando no se conoce el número de iteraciones a ejecutar hay al menos tres formas muy sencillas de hacerlo: la primera consiste en definir una variable para almacenar este dato (*n*) y se pide al usuario que la ingrese, de la forma "*cuántos números desea ingresar?*" y el dato ingresado se utiliza como valor final para implementar una variable de control; la segunda, utilizar un conmutador o bandera, se trata de definir una variable con dos posibles valores: si y no o 0 y 1, por ejemplo, preguntar al usuario "*Otro número (S/N) ?*", si ingresa "*S*" se continúa leyendo datos, en caso contrario el ciclo termina; y la tercera, hacer uso de un centinela que termine el ciclo cuando se cumpla una condición, por ejemplo, el ciclo termina cuando se ingresa el número 0, y se presenta al usuario un mensaje de la forma: "*Ingrese un número (0 para terminar): "*.

Para solucionar este ejercicio se opta por la primera propuesta. Se cuenta con la siguiente información:

Datos de entrada: cantidad de números, números
Datos de salida: promedio, mayor y menor

Procesos:
suma = suma + número

promedio = suma / cantidad de números

Para identificar los números menor y mayor se puede optar por una de estas dos alternativas: primera: inicializar la variable del número menor en un número muy grande y la variable del número mayor en 0, de manera que en la primera iteración se actualicen; segunda: no inicializar las variables y asignarles el primera dato que se ingrese tanto a la variable del mayor como del menor. En este caso se optó por la segunda, el algoritmo se presenta en el cuadro 65.

Cuadro 65. Pseudocódigo para número menor, mayor y promedio

```
1.  Inicio
2.    Entero: can, num, suma=0, menor, mayor, cont = 0
3.    Real: prom
4.    Leer can
5.    Mientras cont < can hacer
6.        Leer num
7.        suma = suma + num
8.        Si cont = 0 entonces
9.            menor = num
10            mayor = num
11        Si no
12            Si num < menor  entonces
13                menor = num
14            Fin si
15            Si num > mayor entonces
16                mayor = num
17            Fin si
18        Fin si
19        cont = cont + 1
20    Fin mientras
21    prom = suma / can
```

Cuadro 65. (Continuación)

```
22    Escribir "Número menor:", menor
23    Escribir "Promedio:", prom
24    Escribir "Número mayor:", mayor
25  Fin algoritmo
```

En este algoritmo se utilizan las variables: *can, num, mayor, menor* y *prom* que son variables de trabajo; *cont* es el contador que permite controlar el ciclo, *suma* es el acumulador que totaliza los números ingresados para luego poder obtener el promedio.

Es importante comprender qué instrucciones deben ir dentro del ciclo y cuales fuera, ya sea antes o después. No hay una regla de oro que se pueda aplicar a todos los casos, pero puede ayudar el recordar que todo lo que está dentro del ciclo se repite; es decir, hay que preguntarse cuántas veces tendrá que ejecutarse la instrucción, si la respuesta es solo una vez, no hay razón para que esté dentro del bucle. En este ejemplo, leer la cantidad de números a trabajar sólo debe ejecutarse una vez ya que esta cantidad permitirá establecer el límite de las repeticiones y calcular el promedio también debe hacerse solo una vez, por lo tanto se ubican fuera del ciclo, pero sumar el número entrado debe realizarse tantas veces como números se ingresen, por ende esta instrucción aparece dentro del bucle.

El ciclo se ejecutará siempre que *cont* sea menor que el contenido de la variable *can*, como *cont* inicia en 0, el ciclo se ejecuta para cualquier valor de *can* que sea mayor o igual a 1, pero si el valor ingresado para *can* fuera 0, el ciclo no se ejecutaría y se generaría un error de división sobre 0 al calcular el promedio. La última instrucción del ciclo es *cont* = *cont* + *1*, ésta es indispensable, ya que el contador debe incrementarse en cada iteración para que alcance el valor de la variable *can*. No se debe olvidar que todo ciclo debe tener un número limitado de iteraciones,

por ello al diseñar el algoritmo se debe prever la forma en que el ciclo terminará.

En el cuadro 66 se presenta el comportamiento de las variables y los resultados de una ejecución.

Cuadro 66. Verificación del algoritmo Número menor, mayor y promedio

Iteración	Can	num	menor	Mayor	Cont	suma	Prom	Salida
	7				0	0		
1		2		2	1	2		
2		14	2	14	2	16		
3		55	2	55	3	71		
4		6	2	55	4	77		
5		12	2	55	5	89		
6		34	2	55	6	123		
7		17	2	55	7	140		
							20	Número menor: 2 Promedio = 20 Número mayor: 55

Ejemplo 35. Procesamiento de notas

En un grupo de n estudiantes se realizaron tres evaluaciones parciales. Se requiere calcular la nota definitiva de cada estudiante y conocer la cantidad de estudiantes que aprobaron, la cantidad de estudiantes que reprobaron y la nota promedio del grupo, considerando que el valor porcentual de cada parcial lo acordaron entre el docente y el grupo, y la nota mínima aprobatoria es 3,0.

Aplicando la estrategia que se ha propuesto para comprender el problema supóngase un caso particular, como éste:

Grupo de segundo semestre: 20 estudiantes
Materia: programación

Porcentaje para primera evaluación: 30%
Porcentaje para segunda evaluación: 30%
Porcentaje para tercera evaluación: 40%

Ahora la solución para un estudiante:

El estudiante Pedro Pérez obtiene las siguientes notas:
Evaluación 1: 3,5
Evaluación 2: 4,0
Evaluación 3: 2,8
Cuál es la nota definitiva de Pedro?

Aplicando los porcentajes previamente mencionados se tiene que:

Nota definitiva = 3,5 * 30% + 4,0 * 30% + 2,8 * 40%
Nota definitiva = 1,05 + 1,2 + 1,12
Nota definitiva = 3,4

El siguiente paso es decidir si el estudiante aprueba o reprueba la materia:

Nota definitiva >= 3,0? (3,4 >= 3.0?) Si

Pedro Pérez aprobó el curso; en consecuencia se lo cuenta como uno de los que aprobaron, y para poder obtener el promedio del curso es necesario sumar su nota a una variable de tipo acumulador que al final se dividirá sobre el contador de estudiantes (20).

Este mismo procedimiento se tiene que llevar a cabo para cada uno de los estudiantes del grupo.

A continuación se identifican y clasifican los datos del problema:

Datos de entrada: nombre del estudiante, nota1, nota2, nota3,
porcentaje1, porcentaje2, porcentaje3.

Datos de salida: nota definitiva, promedio, número pierden,
número ganan

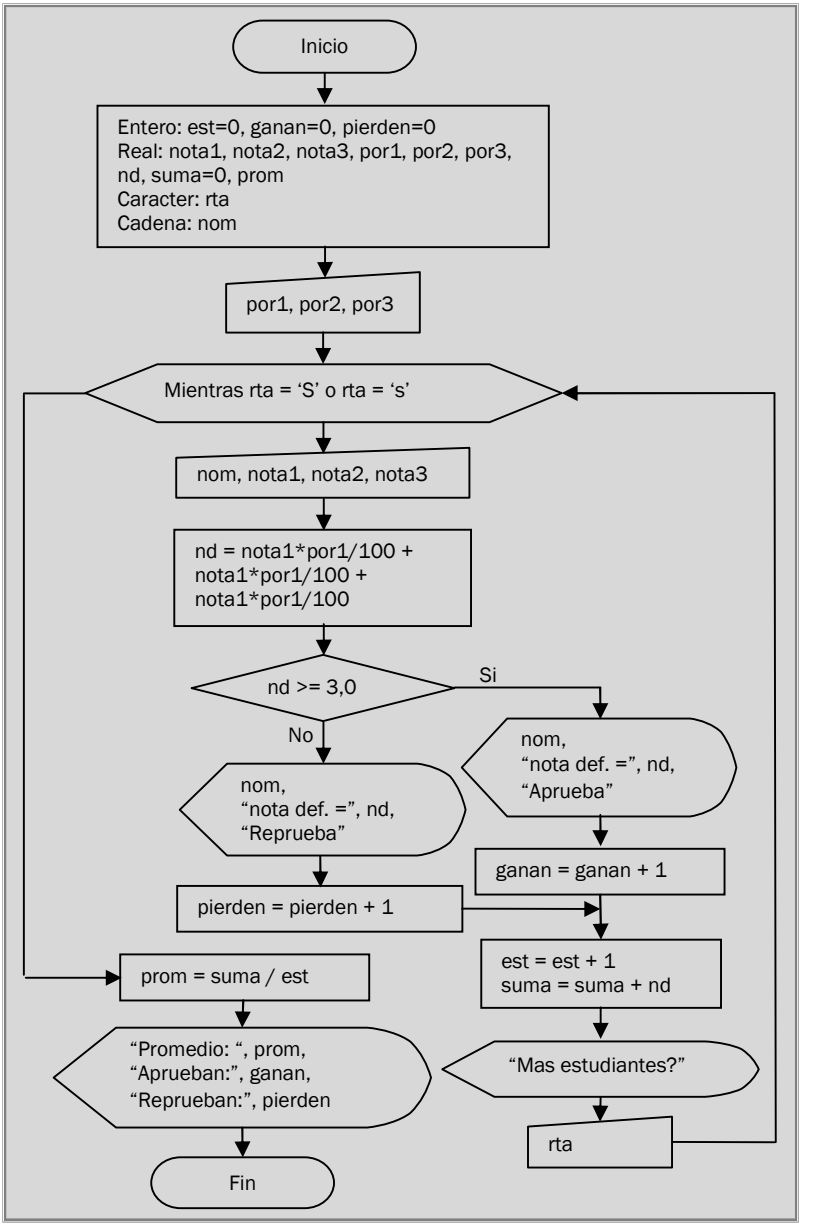
Proceso:
nota definitiva = nota1 * porcentaje1 + nota2 * porcentaje2
+ nota3 * porcentaje3
suma = suma + nota definitiva
promedio = suma / numero estudiantes

En el análisis del ejercicio anterior se mencionó que hay tres formas de programar un ciclo cuando no se conoce por anticipado el número de iteraciones y se implementó la primera; en este ejemplo, se utilizará la segunda, que consiste en preguntar al usuario si desea continuar entrando datos. En el momento en que responda negativamente el ciclo termina.

En este algoritmo se utiliza tres contadores y un acumulador además de las variables de trabajo. Las variables utilizadas para registrar el número de estudiantes, el número de estudiantes que ganan y el número que pierden la materia son contadores, mientras que la variable utilizada para totalizar las notas definitivas es un acumulador. El ciclo no está controlado por la cantidad predefinida de estudiantes, sino por el contenido de la variable *rta* de tipo caracter que debe tomar como valores '*S*' o '*N*'.

El diseño de la solución se presenta en la figura 76 en notación de diagrama de flujo y los resultados de una ejecución de prueba con cuatro estudiantes se muestran en el cuadro 67.

Figura 76. Diagrama de flujo para procesamiento de notas



Cuadro 67. Verificación del algoritmo procesamiento de notas

por1	por2	Por3	nom	Nota 1	Nota 2	Nota 3	Nd	ganan	pierden	est	suma	rta	prom
								0	0	0	0	S	
30	30	40											
			A	3.0	3.5	4.0	3.5	1		1	3.5	S	
			B	2.0	2.5	2.0	2.5		1	2	6	S	
			C	4.0	3.2	2.0	3.0	2		3	9	S	
			D	2.2	3.5	2.5	2.7		2	4	11.7	n	2.9

La salida en pantalla es la siguiente:

Nombre: A
Nota def: 3,5
Aprueba

Nombre: B
Nota def: 2,5
Reprueba

Nombre: C
Nota def: 3,0
Aprueba

Nombre: D
Nota def: 2,7
Reprueba

Promedio: 2,9
Aprueban: 2
Reprueban: 2

Ejemplo 36. Serie Fibonacci

Esta conocida serie propuesta por el matemático italiano del mismo nombre corresponde a un modelo matemático que explica la

reproducción de los conejos y fue publicada por primera vez en 1202 en una obra titulada *Liberabaci.*

Fibonacci planteó el problema de la siguiente manera: supóngase que una pareja de conejos da lugar a dos descendientes cada mes y cada nuevo ejemplar comienza su reproducción después de dos meses. De manera que al comprar una pareja de conejos, en los meses uno y dos se tendrá una pareja, pero al tercer mes se habrán reproducido y se contará con dos parejas, en el mes cuatro solo se reproduce la primera pareja, así que el número aumentará a tres parejas; en el mes cinco, comienza la reproducción de la segunda pareja, con lo cual se obtendrán cinco parejas, y así sucesivamente. Si ningún ejemplar muere, el número de conejos que se tendrán cada mes está dado por la sucesión: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...,

Si se asume que los dos primeros números son constantes; es decir, si se establece que los dos primeros números de la sucesión son 0 y 1, los siguientes se pueden obtener sumando los dos anteriores, así:

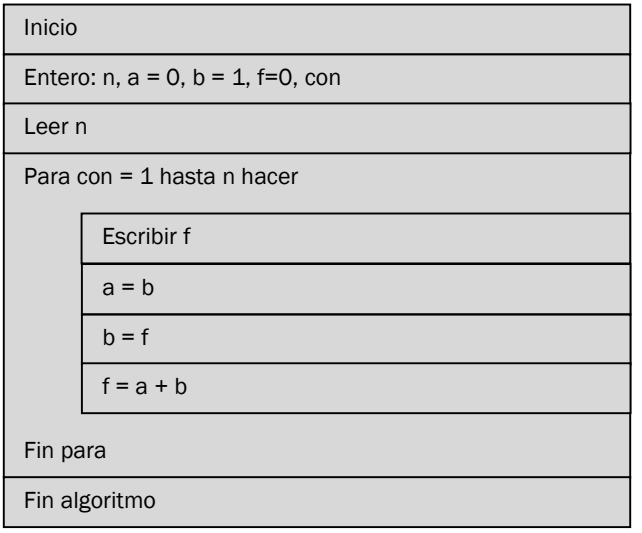
Término	Valor	Obtenido de
1	0	Constante
2	1	Constante
3	1	0 + 1
4	2	1 + 1
5	3	2 + 1
6	5	3 + 2
7	8	5 + 3
8	13	8 + 5
9	21	13 + 8
10	34	21 + 13

Este tema se desarrolla con mayor detalle en la sección 8.7 donde se propone una solución recursiva.

En este ejemplo se desea diseñar un algoritmo que genere n primeros términos de esta serie.

Para solucionar este ejercicio haciendo uso de una estructura iterativa es necesario, en primera instancia, determinar la cantidad de números a generar o sea conocer el valor de n; en segunda, y dado que cada número se obtiene por la suma de los dos términos anteriores, se utiliza tres variables: una para el valor generado y dos más para mantener en memoria los dos últimos datos, adicionalmente se define una variable para controlar el ciclo. La solución se presenta mediante un diagrama N-S en la figura 77.

Figura 77. Diagrama N-S para Serie Fibonacci



Para mirar el comportamiento de las variables en el cuadro 68 se presenta los resultados de la verificación del algoritmo, obsérvese como cambian los valores de una variable a otra.

Cuadro 68. Verificación del algoritmo serie Fibonacci

Iteración	n	Con	A	B	f	Salida
	10	0	0	1	0	
1		1	1	0	1	0
2		2	0	1	1	1
3		3	1	1	2	1
4		4	1	2	3	2
5		5	2	3	5	3
6		6	3	5	8	5
7		7	5	8	13	8
8		8	8	13	21	13
9		9	13	21	34	21
10		10	21	34	55	34

Ejemplo 37. Máximo común divisor

Dados dos números enteros, se requiere encontrar el máximo común divisor de los mismos.

El máximo común divisor (MCD) entre dos números es el número más grande que los divide a ambos, incluso puede ser uno de ellos dado que todo número es divisor de sí mismo. El MCD se puede obtener de forma iterativa o recursiva, en esta sección se soluciona aplicando la primera y en la sección 8.7 se presenta la solución recursiva aplicando el algoritmo de Euclides.

Al abordar este problema, seguramente, la primera idea que se le ocurre al lector es descomponer los números en sus divisores y tomar los comunes, tal como se lo enseñaron en el colegio. Esa es una solución válida; sin embargo, es difícil de implementar de forma algorítmica.

Aplicando el algoritmo de Euclides, el MCD se obtiene de la siguiente manera: se divide el primer número sobre el segundo, si la división es exacta (residuo = 0) el MCD es el segundo número, si la

división no es exacta se divide el divisor sobre el residuo y si el módulo es cero el MCD es el número que se colocó como divisor; en caso contrario, se repite esta operación hasta obtener una división entera.

Por ejemplo, se busca el MCD de los números 12 y 8

```
12 | 8
---|---
4  | 1
```

Como la división es inexacta se efectúa una segunda división tomando como dividendo el anterior divisor y como divisor el residuo.

```
 8 | 4
---|---
0  | 2
```

Esta segunda operación es una división exacta (residuo = 0) por tanto el divisor es la solución del ejercicio; es decir, el MCD entre 12 y 8 es 4.

Con base en el ejemplo anterior se puede organizar los datos del ejercicio así:

Datos de entrada: a, b (dos números enteros)
Datos de salida: MCD
Proceso: c = a Mod b

La solución se presenta mediante pseudocódigo en el cuadro 69

Cuadro 69. Pseudocódigo del algoritmo Máximo Común Divisor

```
1.  Inicio
2.     Entero:  num1, num2, a, b, c
3.     Leer num1, num2
4.     a = num1
5.     b = num2
6.     Hacer
7.          c = a Mod b
8.          a = b
9.          b = c
10     Mientras c != 0
11     Escribir "MCD:", a
12  Fin algoritmo
```

La verificación de este algoritmo con los números 12 y 8, 6 y 20 genera los datos que se presentan en el cuadro 70.

Cuadro 70. Verificación del algoritmo Máximo Común Divisor

Ejecución	Iteración	a	b	c	Salida
1		12	8		
1	1	12	8	4	
1	2	8	4	0	
1		4	0		MCD: 4
2		6	20		
2	1	6	20	6	
2	2	20	6	2	
2	3	6	2	0	
		2	0		MCD: 2

Ejemplo 38. Invertir dígitos

Dado un número entero, se desea invertir el orden de sus dígitos. Supóngase que se toma el número 123 al invertirlo se obtiene el número 321.

Para cambiar el orden de los dígitos es necesario separarlos comenzando por el último, para ello se divide sobre 10 y se toma el residuo utilizando el operador módulo.

```
123 | 10
  3   12
```

Esta operación es equivalente a la expresión: 123 Mod 10 = 3

De esta manera se obtiene el dígito de la última posición del número original, el 3, que será el primero en el número invertido; luego se obtiene el siguiente dígito, el 2, dividiendo el cociente sobre 10, y así sucesivamente.

```
12 | 10
 2   1
```

En la medida en que se extraen los dígitos del número original se van organizando de izquierda a derecha para conformar el nuevo número, para ello es necesario multiplicar por 10 el número que va generando y sumarle el último dígito obtenido, así:

3 * 10 + 2 = 32

Luego se obtiene el tercero y último número, el 1, y se realiza la operación:

```
 1 | 10
 1   0
```

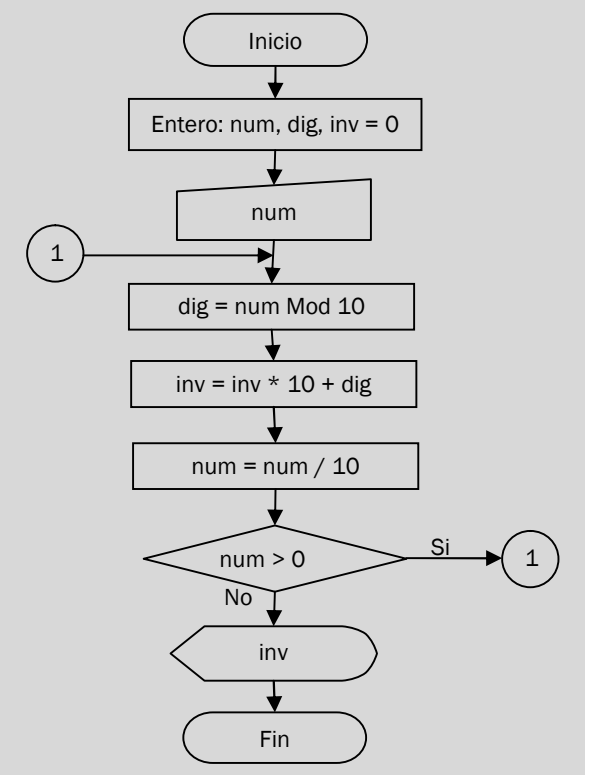
32 * 10 + 1 = 321

Las operaciones se repiten hasta que el cociente de la división sea 0, en cuyo caso se habrán invertido todos los dígitos. En este orden de ideas, se tienen los siguientes datos

Datos de entrada: número
Datos de salida: número invertido
Procesos:
Dígito = número Mod 10
Número = número / 10
Número invertido = número invertido * 10 + digito

El diagrama de flujo se presenta en la figura 78.

Figura 78. Diagrama de flujo para invertir los dígitos de un número



En este algoritmo se utiliza la estructura iterativa *hacer – mientras*, como en la notación de diagrama de flujo no se cuenta con un símbolo para esta estructura se modela mediante una decisión y un conector que lleva el control de la ejecución hasta el primer proceso que se repite.

Los datos obtenidos en la verificación paso a paso del algoritmo se muestran en el cuadro 71.

Cuadro 71. Verificación del algoritmo invertir dígitos de un número

Iteración	num	dig	inv	Salida
	123			
1	123	3	3	
2	12	2	32	
3	1	1	321	
	0			321

Ejemplo 39. Número perfecto

Un número es perfecto si se cumple que la sumatoria de los divisores menores al número da como resultado el mismo número. Se desea un algoritmo para determinar si un número n es perfecto.

Para comprender mejor el concepto de número perfecto considérese dos casos: 6 y 8. Los divisores de 6 son: 1, 2, 3 y 6; y los divisores de 8 son: 1, 2, 4 y 8 (todo número es divisor de sí mismo). Si se suman únicamente los divisores menores a cada número se tiene:

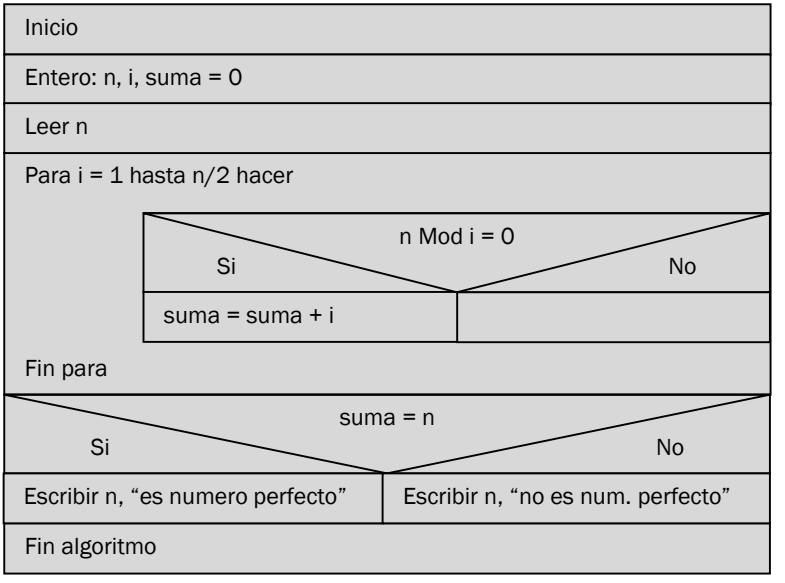
$$\sum_{D_6 = 1}^{D_6 < 6} D_6 = 1 + 2 + 3 = 6 \qquad \sum_{D_8 = 1}^{D_8 < 8} D_8 = 1 + 2 + 4 = 7$$

Léase D_n como divisor de n. Por tanto, se tiene que: la sumatoria de los divisores de 6, menores a 6, es igual a 6, de lo que se

concluye que 6 es número perfecto; con respecto al segundo número, se lee: la sumatoria de los divisores de 8, menores a 8, es igual a 7, en consecuencia 8 no es número perfecto.

Ahora, para saber si un número es perfecto se requieren tres procesos básicos: identificar los divisores, sumarlos y verificar si la sumatoria es igual al número. Para identificar los divisores menores a n es preciso hacer un recorrido entre 1 y $n/2$ y verificar para cada número si es o no es divisor de n, para esto se utiliza una estructura iterativa; para sumarlos se utiliza una variable de tipo acumulador que se actualiza dentro del ciclo; y finalmente, una estructura de decisión ubicada fuera del ciclo determinará si el número es o no es perfecto. La solución a este ejercicio se presenta en la figura 79.

Figura 79. Diagrama N-S para número perfecto



En el cuadro 72 se presenta los resultados de la verificación de este algoritmo con los números 6 y 8.

Cuadro 72. Verificación del algoritmo Número perfecto

Ejecución	n	i	suma	Salida
1	6		0	
1		1	1	
1		2	3	
1		3	6	6 es número perfecto
2	8		0	
2		1	1	
2		2	3	
2		3	3	
		4	7	8 no es número perfecto

Ejemplo 40. Potenciación iterativa

Dados dos números enteros: b que es la base y e que es el exponente, se requiere calcular el resultado de la potenciación.

La potenciación es una operación matemática cuyo resultado es el producto de multiplicar la base por sí misma tantas veces como indique el exponente. Entre sus propiedades se tienen: si el exponente es 0, el resultado es 1 y si el exponente es 1, el resultado es el mismo valor de la base. En este ejercicio, para facilitar la solución, se limita el exponente a los números enteros positivos. Por ejemplo:

$2^3 = 2 * 2 * 2 = 8$
$3^5 = 3 * 3 * 3 * 3 * 3 = 243$

De manera que para desarrollar una potenciación se debe implementar una estructura iterativa y dentro de ella realizar multiplicaciones sucesivas de la base. El exponente indica el número de iteraciones a realizar. Los datos son:

Datos de entrada: base, exponente
Datos de salida: resultado

Proceso: producto = producto * base

En el cuadro 73 se presenta el pseudocódigo de este ejercicio.

Cuadro 73. Pseudocódigo Potencia iterativa

```
1.  Inicio
2.      Entero: b, e, p = 1, con
3.      Leer b, e
4.      Para con = 1 hasta e hacer
5.          p = p * b
6.      Fin para
7.      Escribir p
8.  Fin algoritmo
```

Cualquier número multiplicado por 0 da como resultado 0; por esto, la variable p (producto), que es un acumulador de resultados de multiplicación, se inicializa en 1, pues si se inicializara en 0 al final se tendría como resultado el 0. Los resultados de la verificación de este algoritmo se presentan en el cuadro 74.

Cuadro 74. Verificación del algoritmo potencia iterativa

Ejecución	Iteración	b	e	Con	P	Salida
1					1	
1	1	2	4	1	2	
1	2			2	4	
1	3			3	8	
1	4			4	16	16
2		3	5		1	
2	1			1	3	
2	2			2	9	
2	3			3	27	
2	4			4	81	
	5			5	243	243

Ejemplo 41. Número primo

Se denomina primo al número entero que solo cuenta con dos divisores: el uno y el mismo número. Se solicita un algoritmo que dado un número n decida si es o no es primo.

Para determinar si un número cumple con esta propiedad se procede a verificar si tiene algún divisor diferente de uno y el mismo número, en cuyo caso se demuestra que no es primo, en caso contrario se dice que si es primo; pero no es necesario examinar todos los números, para saber si es o no primo un número basta con buscar divisores entre dos y la raíz cuadrada del número*.

Como ejemplos se examinan los números 9 y 19. La raíz cuadrada de 9 es 3, por tanto se busca divisores entre 2 y 3 y se encuentra que el 3 es divisor, por tanto 9 no es número primo. Para 19 se tiene que la raíz cuadrada entera es 4, por tanto se busca divisores entre 2 y 4, obteniéndose que los números 2, 3 y 4 ninguno es divisor de 19, por tanto se concluye que 19 es número primo. Podría verificarse para los números que siguen hasta 18, pero el resultado será el mismo.

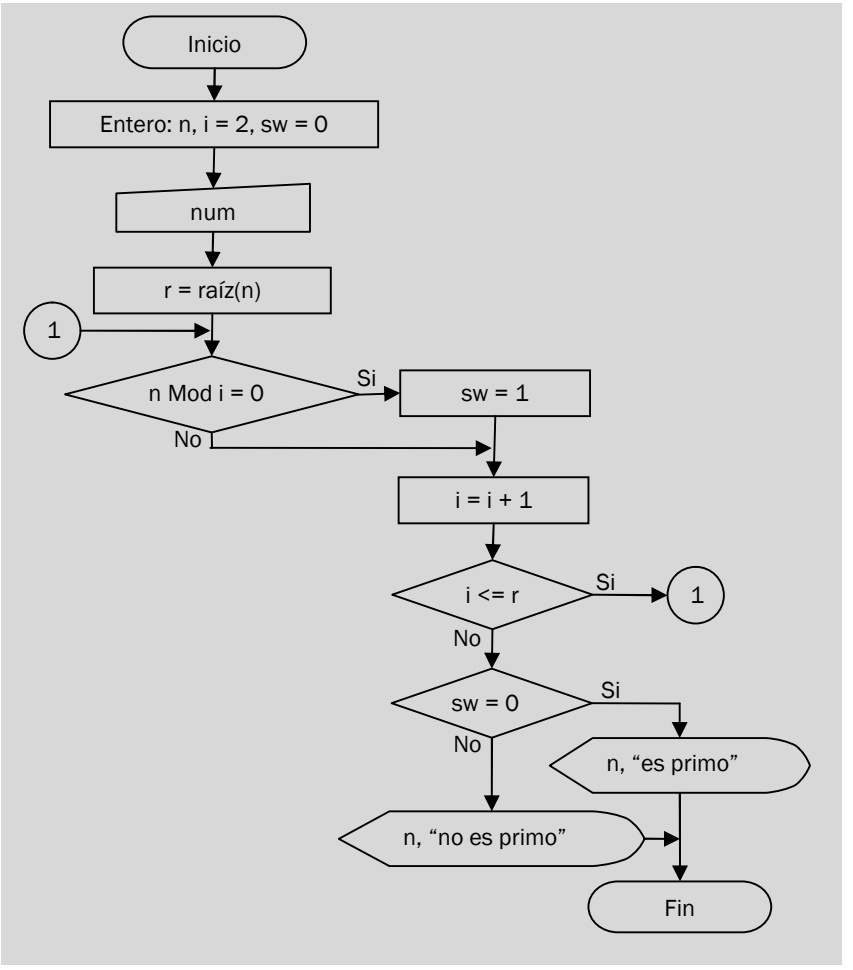
Para determinar si un número es primo se tienen los siguientes datos y operaciones:

Datos de entrada: número
Datos de salida: mensaje “número primo” ó “número no primo”
Proceso: número Mod i donde i: 2 … raíz(número)

La solución algorítmica se presenta en la figura 80 en notación de diagrama de flujo.

* El propósito de este ejercicio es aplicar una estructura iterativa en la solución, por ello se procede a buscar divisores entre todos los números comprendidos entre 2 y la raíz entera del número. Existe otro método más rápido para comprobar si un número es primo, se trata de la solución propuesta por Eratóstenes (matemático griego del siglo III a. C) quien propone que para verificar si un número es primo solo hay que dividir para: 2, 3, 5 y 7. (Gran Enciclopedia Espasa, 2005).

Figura 80. Diagrama de flujo para número primo



En este algoritmo se han declarado cuatro variables: *n, r, i, sw*; *n* para el número, *r* para calcular la raíz cuadrada de *n*, *i* para hacer el recorrido entre 2 y *r* y *sw*. Esta última es una variable tipo conmutador o bandera, su propósito es informar si al hacer las iteraciones para la variable *i* se encontró algún divisor de *n*, la variable se

inicializa en 0, si al finalizar el ciclo se mantiene en 0 significa que no se encontró ningún divisor y por tanto el número es primo, pero si la variable ha tomado el valor 1 significa que hay al menos un divisor y por tanto el número no es primo.

En el cuadro 75 se muestra los resultados de la ejecución paso a paso para los números 9 y 19.

Cuadro 75. Verificación del algoritmo Número primo

Ejecución	Iteración	n	r	i	sw	Salida
1		9	3		0	
	1			2		
	2			3	1	9 no es primo
2		19	4		0	
	1			2		
	2			3		
	3			4		19 es primo

Ejemplo 42. Puntos de una línea

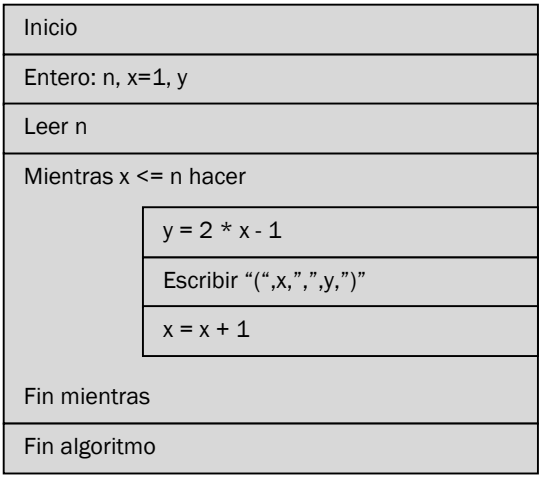
Dada la ecuación lineal $y = 2x - 1$ se desea un algoritmo para calcular n puntos por los que pasa la línea a partir de $x = 1$.

Si se toma n = 4, los puntos serían:

X = 1 → y = 2(1) – 1 = 1 → (1,1)
X = 2 → y = 2(2) – 1 = 3 → (2,3)
X = 3 → y = 2(3) – 1 = 5 → (3,5)
X = 4 → y = 2(4) – 1 = 7 → (4,7)

En este caso los datos de entrada corresponde a n, los datos de salida a los puntos (x,y), tantos como indique n y el proceso se limita al desarrollo de la ecuación reemplazando x por el valor que corresponda. Para ello se utiliza una estructura iterativa, como se aprecia en el diagrama N-S de la figura 81.

Figura 81. Diagrama N-S para Puntos de una línea



Los datos generados al verificar la corrección de este algoritmo se presentan en el cuadro 76.

Cuadro 76. Verificación del algoritmo Número primo

Ejecución	N	x	y	Salida
1	4	1	1	(1,1)
1		2	3	(2,3)
1		3	5	(3,5)
		4	7	(4,7)

Ejemplo 43. Raíz cuadrada

Calcular la raíz cuadrada entera de un número.

Se sabe que la raíz cuadrada de un número es otro número que al elevarse al cuadrado es igual al primero.

$\sqrt{4} = 2 \quad \rightarrow 4 = 2^2$

$\sqrt{25} = 5 \quad \rightarrow 25 = 5^2$

Algunos números, como los anteriores, tienen una raíz cuadrada exacta, entera; mientras que para otros es un número real, como la raíz de 10 y para los números negativos la raíz es un número complejo o imaginario.

$\sqrt{10} = 3{,}16227766$

La raíz entera de un número, en el caso de los números que no tienen una raíz exacta, se obtiene truncando la parte decimal, por ejemplo, la raíz entera de 10 es 3.

En este ejercicio se tiene: dato de entrada es el número, dato de salida es la raíz y el proceso es calcular el cuadrado de los números, desde 1 hasta la raíz cuadrada del número. Se sabe que se ha encontrado la raíz entera cuando el siguiente número al elevarse al cuadrado da un valor superior al número ingresado. La solución a este ejercicio se presenta en notación de pseudocódigo en el cuadro 77.

Cuadro 77. Pseudocódigo del algoritmo raíz cuadrada

```
1   Inicio
2       Entero: n, i = 0, c
3       Leer n
4       Hacer
5           i = i + 1
6           c = (i+1) * (i+1)
7       Mientras (c <= n)
8       Escribir “La raíz de”, n, “es”, i
9   Fin algoritmo
```

En el cuadro 78 se presenta los resultados de la ejecución paso a paso de este algoritmo.

Cuadro 78. Verificación del algoritmo raíz cuadrada

Ejecución	N	I	C	Salida
1	25	0		
1		1	4	
1		2	9	
1		3	16	
1		4	25	
1		5	36	La raíz de 25 es 5
2	10	0		
2		1	4	
2		2	6	
2		3	16	La raíz de 10 es 3

4.3.7 Ejercicios propuestos

Diseñar los algoritmos para solucionar los problemas que se plantean a continuación. Se sugiere intercalar notación de pseudocódigo, diagrama de flujo y diagrama N-S; y de igual manera, las tres estructuras iterativas estudiadas en este capítulo.

1. Un estudiante cuenta con siete notas parciales en el curso de Introducción a la programación, se requiere un algoritmo para calcular el promedio de dichas notas.

2. Diseñar un algoritmo para leer números enteros hasta que se introduzca el 0. Calcular el cuadrado de los números negativos y el cubo de los positivos.

3. Diseñar un algoritmo para ingresar números, tantos como el usuario desee. Al finalizar el ciclo reportar cuántos números pares y cuántos impares se registraron, cuánto suman los pares y cuánto los impares.

4. El director de la escuela Buena Nota desea conocer el promedio de edad de sus estudiantes en cada grado. La escuela ofrece educación desde transición a quinto de primaria y cuenta con un grupo de estudiantes en cada grado. Diseñar un algoritmo que lea la edad y el grado de cada estudiante de la escuela y genere el correspondiente informe.

5. Un entrenador le ha propuesto a un atleta recorrer una ruta de cinco kilómetros durante 10 días, para determinar si es apto para la prueba de 5 kilómetros. Para considerarlo apto debe cumplir las siguientes condiciones:

   - Que en ninguna de las pruebas haga un tiempo mayor a 20 minutos.
   - Que al menos en una de las pruebas realice un tiempo menor de 15 minutos.
   - Que su promedio sea menor o igual a 18 minutos.

   Diseñar un algoritmo para registrar los datos y decidir si es apto para la competencia.

6. Una compañía de seguros tiene contratados a n vendedores. Cada vendedor recibe un sueldo base y un 10% extra por comisiones de sus ventas. Se requiere un algoritmo para calcular el valor a pagar a cada empleado y los totales a pagar por concepto de sueldos y comisiones.

7. Elaborar un algoritmo para procesar las notas definitivas de Biología para un grupo de n estudiantes. Se desea conocer el promedio del grupo, clasificar a los estudiantes en: excelentes, buenos, regulares y descuidados, según la nota obtenida y contar cuántos pertenecen a cada categoría. La escala es:

   Nota >= 4.8: excelente
   4.0 <= nota <= 4.7: bueno
   3.0 <= nota <= 3.9: regular
   nota <= 2.9: descuidado

8. Diseñar un algoritmo que genere el n-ésimo número de la serie Fibonacci.

9. Se aplicó una encuesta a n personas solicitando su opinión sobre el tema del servicio militar obligatorio para las mujeres. Las opciones de respuesta fueron: a favor, en contra y no responde. Se solicita un algoritmo que calcule qué porcentaje de los encuestados marcó cada una de las respuestas.

10. Se requiere un algoritmo que mediante un menú cumpla las funciones de una calculadora: suma, resta, multiplicación, división, potenciación y porcentaje. El menú contará con la opción apagar para terminar la ejecución del algoritmo.

11. Se requiere un algoritmo para facturar una venta con *n* artículos. En cantidades mayores a 10 unidades de un mismo artículo se aplicará un descuento del 5%.

12. Diseñar un algoritmo que lea un número entero y sume los dígitos que lo componen.

13. En el programa Ingeniería de Sistemas se necesita un algoritmo para conocer los porcentajes de estudiantes que trabajan y de los que se dedican únicamente a sus estudios, discriminados por género.

14. El almacén Buena Ropa cuenta con los registros mensuales de ventas y desea un algoritmo para determinar: en qué més se tuvo las ventas más altas, en cuál las más bajas y el promedio mensual de ventas.

15. Diseñar un algoritmo para calcular el factorial de un número *n*.

16. Se requiere un algoritmo para encontrar los números primos existentes entre 1 y n

17. Dado un grupo de 20 estudiantes que cursaron la materia Algoritmos, se desea saber cuál es el promedio del grupo, cuál

fue la nota más alta y cuál la más baja, cuántos aprobaron el curso y cuántos reprobaron.

18. Dada la ecuación $y = 2x^2+3x - 4$ calcular los puntos por los que pasa la parábola en el intervalo -5, 5.

19. Diseñar un algoritmo para encontrar los primeros n números perfectos.

20. Diseñar un algoritmo para validar una nota. La nota debe ser un valor real entre 0 y 5.0. Si se ingresa un valor fuera de este intervalo se vuelve a leer, hasta que se ingrese una nota válida.

21. Diseñar un algoritmo para validar una fecha en el formato dd/mm/aaaa. Se considera que una fecha es válida si está entre 01/01/1900 y 31/12/2100, el mes entre 1 y 12 y el día entre 1 y 31, teniendo en cuenta que algunos meses tienen 28 (o 29), 30 o 31 días. Si la fecha no es válida se muestra un mensaje de error y se vuelve a leer los datos. El algoritmo termina cuando la fecha ingresada es válida.

5. ARREGLOS

Es menester
conocerse a sí mismo;
si esto no basta
para hacernos hallar la verdad,
por lo menos sirve
para que arreglemos nuestra vida;
y nada hay más justo que eso.
Pascal.

Cuando se piensa en un algoritmo como el diseño de una solución a un problema haciendo uso de un programa de computador, es preciso considerar, al menos, cuatro aspectos:

- El procesamiento de los datos: las operaciones para transformar la información de entrada en información de salida.
- El almacenamiento de los datos: cómo se guardan y cómo se accede a ellos, debe tenerse en cuenta el almacenamiento en memoria primaria y en memoria secundaria. Los arreglos son una forma de manejar datos en memoria principal.
- La arquitectura del sistema: tiene que ver con la forma cómo se organizan y cómo interactúan los diferentes componentes del sistema. Parte de este tema se aborda en el siguiente capítulo.

- Interfaz de usuario: cómo interactúa el usuario con el sistema, de qué elementos dispone y cómo se atiende sus requerimientos. Este tema no se aborda directamente en este libro.

Los arreglos son estructuras que permiten agrupar datos para facilitar su manejo en la memoria del computador y realizar operaciones que requieren tener acceso a conjuntos de datos al tiempo, como: buscar y ordenar. Para comprender mejor este concepto considérese los siguientes casos:

Primer caso: se requiere un algoritmo para calcular la nota definitiva de un estudiante a partir de tres notas parciales. En esta situación, como en todos los ejemplos que se han presentado en los capítulos anteriores, basta con declarar tres variables, una para cada nota.

Segundo caso: se necesita un algoritmo para calcular la nota definitiva de un grupo de 40 estudiantes, a partir de tres notas parciales, y presentar el listado en orden alfabético. Podría decirse que se declaran 160 variables para las notas y 40 para los nombres; sin embargo, esto es prácticamente inmanejable. Para casos como este es necesario agrupar los datos del mismo tipo bajo un mismo identificador, de manera que se maneje un arreglo de nombres y un arreglo de notas.

5.1 CONCEPTO DE ARREGLO

Se entiende como arreglo una colección de elementos del mismo tipo*, almacenados en direcciones seguidas de memoria a los cuales se hace referencia con un mismo identificador o nombre,

* La definición de arreglo como colección de datos del mismo tipo (homogéneos) está generalizada por el uso de lenguajes de programación con definición de tipos, en los que es necesario declarar las variables y los arreglos antes de utilizarlos. No obstante, en los lenguajes no tipados los arreglos pueden agrupar elementos de diferente clase.

pero se distinguen entre sí mediante un índice que representa la posición de cada elemento dentro de la estructura.

En este mismo sentido Brassard y Bratley (1997: 167) consideran el término arreglo como sinónimo de matriz, a la que conceptualizan como una estructura de datos con un número fijo de elementos del mismo tipo, los cuales se almacenan en posiciones contiguas en la memoria del computador, siendo así que al conocer el tamaño de los elementos y la dirección del primero se puede calcular con facilidad la posición de cualquier ítem, cuando sea necesario, considerando que se organizan de izquierda a derecha.

También se define un arreglo como un tipo de dato compuesto, conformado por una colección de datos simples o compuestos. De esto se desprende que un elemento de un arreglo puede ser: un número, un carácter, una cadena, un arreglo o un registro.

El uso de arreglos es fundamental cuando se requiere mantener en memoria y operar muchos datos. Por ejemplo: un grupo de estudiantes, una nómina, una factura (incluye varios productos), un inventario, una biblioteca; todos estos conceptos hacen referencia a conjuntos de datos del mismo tipo, en los que se puede realizar operaciones como: buscar, ordenar, sumar.

Entre las características de los arreglos se tiene:

- Son estructuras estáticas; es decir, que una vez definidas no se puede cambiar su tamaño.
- Almacenan datos homogéneos, excepto en lenguajes no tipados
- Los datos se almacenan en memoria ocupando posiciones seguidas, de manera que solo es necesario la referencia al primer elemento
- Tienen un mismo identificador (nombre de variable) que representa a todos los elementos.
- Los elementos individuales se reconocen por el índice o posición que ocupan en el arreglo.

5.2 CLASES DE ARREGLOS

Los arreglos se clasifican de acuerdo con su contenido; en consecuencia, se tienen arreglos numéricos, arreglos de caracteres, arreglos de cadenas, arreglos de registros y arreglos de arreglos.

Cuando un arreglo está conformado por datos simples, como números o caracteres, se dice que es lineal o que tiene una sola dimensión (unidimensional) y se les da el nombre de vectores; éstos tienen sólo un índice. Cuando los elementos del arreglo son también arreglos se dice que es multidimensional, tienen un índice por cada dimensión y se les denomina matrices. En las figuras 82, 83 y 84 se muestran las representaciones gráficas de arreglos de una, dos y tres dimensiones.

Figura 82. Representación gráfica de un vector

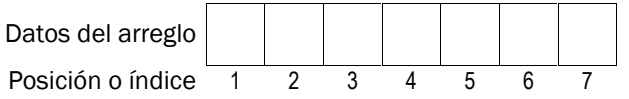


Figura 83. Representación gráfica de una matriz de dos dimensiones

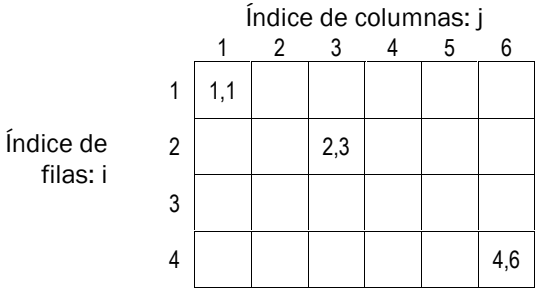
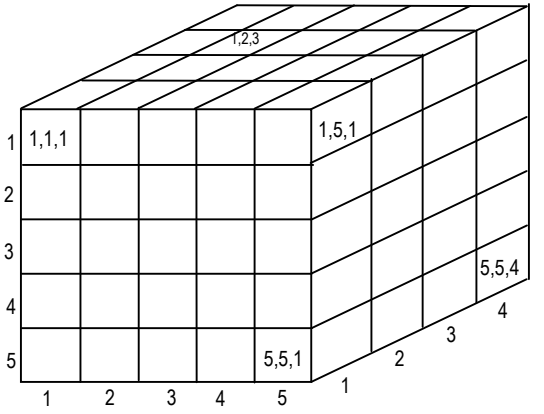


Figura 84. Representación gráfica de una matriz de tres dimensiones



Como se puede apreciar en las gráficas, un vector es una lista cuyos elementos se identifican con un solo índice, mientras que una matriz de dos dimensiones requiere dos índices: uno para filas y otro para columnas; y una matriz de tres dimensiones requiere tres índices ya que la estructura es análoga a un artefacto tridimensional, como si fuera una caja con largo, ancho y alto, dividida en compartimentos igualmente tridimensionales. Teóricamente se pueden definir matrices de dos, tres, cuatro o más dimensiones; no obstante, en la práctica es común utilizar vectores y matrices de dos dimensiones, pero no de tres o más. En lo que sigue de este capítulo se estudia el manejo de vectores y matrices de dos dimensiones.

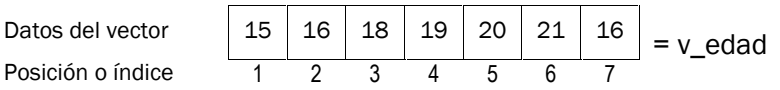
5.3 MANEJO DE VECTORES

Un vector es una estructura de datos estática correspondiente a una colección de datos del mismo tipo que se agrupan bajo un mismo nombre y se diferencias unos de otros por la posición que ocupan en la estructura.

Según Timarán *et al* (2009) un vector es una colección finita y ordenada de elementos. Finita, porque todo vector tiene un número limitado de elementos y ordenada porque se puede determinar cuál es el primero, el segundo y el n-ésimo elemento.

Por ejemplo, si se tienen las edades de siete estudiantes se pueden almacenar en una estructura de tipo vector llamado *v_edad* de siete elementos. Gráficamente este vector se vería como la figura 85.

Figura 85. Representación gráfica de un vector de edades



5.3.1 Declaración

Declarar un vector debe entenderse como el proceso de reservar un espacio en memoria para almacenar un número determinado de datos, los cuales se distinguirán mediante un identificador y un índice, tal como se ha descrito en páginas anteriores.

La forma más sencilla y a la vez más funcional de declarar un vector es la siguiente:

Tipo_dato: identificador[tamaño]

Ejemplos:

Entero: v_edad[7]

Se está declarando un vector de siete elementos de tipo entero, como el que se mostró en la figura 85, lo que equivale a decir que se reserva espacio en memoria para almacenar siete datos enteros en posiciones consecutivas y que se reconocerán con el identificador *v_edad.*

Cadena: v_nombre[30]

Esta declaración reserva memoria para almacenar 30 nombres a los que se reconocerá con el identificador: *v_nombre*.

5.3.2 Acceso a elementos

Para acceder a un elemento de un vector y llevar a cabo operaciones como: asignación, lectura, impresión y conformación de expresiones, es necesario escribir el identificador (nombre) del arreglo y entre corchetes la posición a la que se hace referencia, de la forma:

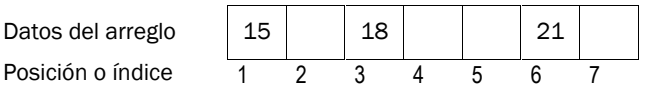
vector[pos]

Ejemplo de declaración y acceso a los elementos de un vector:

Entero: v_edad[7]
v_edad[1] = 15
v_edad[3] = 18
v_edad[6] = 21

Se declara un vector de siete posiciones y se asigna datos en las posiciones 1, 3 y 6. La representación gráfica se muestra en la figura 86.

Figura 86. Vector con datos



Para leer un número y guardarlo en la cuarta posición se escribe:

Leer v_edad[4]

Para imprimir el dato almacenado en la sexta posición se escribe:

Escribir v_edad[6]

De forma similar se puede construir expresiones utilizando los elementos del vector, por ejemplo, si se quiere conocer la diferencia entre las edades almacenadas en las posiciones 1 y 3 se escribe:

Entero: dif
dif = v_edad[1] – v_edad[3]

En síntesis, cuando se escribe el identificador del vector junto con la posición de un elemento éstos se comportan como si se tratara de una variable sobre la que se puede realizar diferentes operaciones.

5.3.3 Recorrido de un vector

Muchas operaciones que se aplican sobre vectores incluyen a todos los elementos, en estos casos es necesario acceder consecutivamente desde la primera hasta la última posición, a esto se le llama recorrido de un vector.

Para recorrer un vector se utiliza una estructura iterativa controlada por una variable que comienza en 1, para hacer referencia a la primera posición, y se incrementa de uno en uno hasta alcanzar el tamaño del arreglo, de la forma:

*Tipo_Dato: vector[n]**
int: i
para i = 1 hasta n hacer
    operación sobre vector[i]
fin para

* n es el tamaño del vector

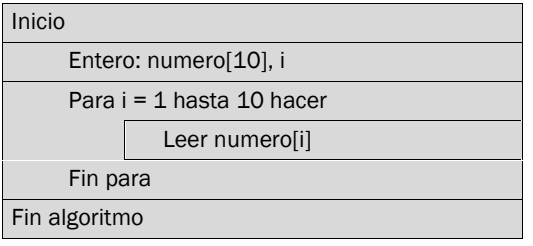
La variable *i* representa la posición de cada uno de los elementos; de este modo, cualquier operación que se defina sobre *vector[i]* al interior del ciclo se llevará a cabo sobre cada uno de los elementos del vector.

Ejemplo 44. Guardar números en vector

Diseñar un algoritmo que lea 10 números y los guarde en un vector.

Para solucionar este ejercicio es necesario utilizar las operaciones que se han explicado en las páginas anteriores: declaración, recorrido y acceso a los elementos de un vector. El diagrama N-S de este algoritmo se presenta en la figura 87.

Figura 87. Diagrama N-S para guardar números en un vector



En este algoritmo se aprecia el uso de una estructura iterativa para recorrer el vector y leer un dato para cada posición, se utiliza la variable de control del ciclo (*i*) como índice para identificar los elementos del arreglo.

Ejemplo 45: Entrada y salida de datos de un vector

Diseñar un algoritmo para leer 10 números, guardarlos en un vector y luego mostrarlos en orden inverso a como se introdujeron.

Una de las ventajas de manejar arreglos es que permiten almacenar conjuntos de datos en memoria y acceder fácilmente a ellos. Los datos que están en memoria pueden ser listados en

cualquier orden, según el extremo por el cual se aborde el recorrido del vector.

El recorrido se hace utilizando la variable de control del ciclo como índice del vector. Si la variable comienza en 1 y se incrementa, se hará un recorrido hacia adelante; mientras que si comienza en 10 y se decrementa se hará un recorrido hacia atrás.

En este algoritmo se declara un vector y se utilizan dos ciclos: el primero hace un recorrido desde la posición 1 hasta la 10 ingresando el dato para cada posición y el segundo hace un recorrido desde la última posición hasta la primera imprimiendo el dato de cada posición. El diagrama N-S de este algoritmo se presenta en la figura 88.

Figura 88. Diagrama N-S para entrada y salida de datos de un vector

```
Inicio
    Entero: v[10], i
    Para i = 1 hasta 10 hacer
        Leer v[i]
    Fin para
    Para i = 10 hasta 1 decrementar 1 hacer
        Escribir v[i]
    Fin para
Fin algoritmo
```

Si al verificar el algoritmo se ingresan los datos: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, el vector se vería como en la figura 89:

Figura 89. Representación gráfica de un vector

2	4	6	8	10	12	14	16	18	20
1	2	3	4	5	6	7	8	9	10

Al recorrer el vector y mostrar los datos comenzando por la décima posición se tendría los datos en este orden: 20, 18, 16, 14, 12, 10, 8, 6, 4, 2.

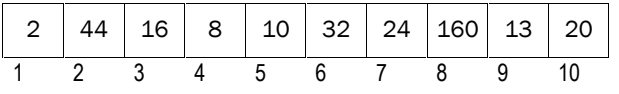
Ejemplo 46: Calcular promedio de números en un vector

Leer 10 números enteros y almacenarlos en un vector. Calcular y mostrar el promedio, en seguida generar un listado con los números menores al promedio y otro con los números mayores a éste.

Leer números y calcular el promedio se puede hacer únicamente utilizando ciclos, pero cuando se solicita que se muestre los números menores y mayores al promedio, por separado, es evidente que se tiene que haber guardado los números para poder acceder a ellos y compararlos con el promedio de los mismos. Aquí es evidente la necesidad de utilizar un vector, en caso contrario sería necesario declarar 10 variables y luego implementar 20 decisiones para comparar los números, esto realmente sería una mala práctica de programación, especialmente si se considera que así como son 10 números podrían ser 100.

Para comprender mejor el ejercicio, supóngase el vector que se muestra en la figura 90.

Figura 90. Representación gráfica del vector numérico



Una vez que se tienen los datos en el vector es necesario calcular el promedio y para ello es preciso calcular la sumatoria, para este caso se tiene:

Suma = 329
Promedio = 32,9

Para generar la lista de los números mayores al promedio se debe comparar cada elemento del vector con el valor obtenido como promedio:

Números mayores al promedio: 44, 160

De la misma forma se obtienen el segundo listado:

Números menores al promedio: 2, 16, 8, 10, 32, 24, 13, 20

Ahora se procede al diseño del algoritmo teniendo en cuenta esta información:

Datos de entrada: número (10 números leídos al interior de un ciclo y guardados en un vector)

Estructura de almacenamiento: vector de 10 elementos

Datos de salida: promedio, números menores al promedio y números mayores al promedio

Proceso: suma = suma + número (en cada posición del vector)
promedio = suma / 10

En este algoritmo se implementan tres ciclos, uno para leer los datos y guardarlos en cada posición del vector y dos más para generar los dos listados de forma independiente.

El algoritmo se presenta en el cuadro 79 en notación de pseudocódigo.

Cuadro 79. Verificación del algoritmo raíz cuadrada

```
1.  Inicio
2.      Entero:  v[10], i, suma = 0
3.      Real:  prom
4.      Para i = 1 hasta 10 hacer
5.          Leer v[i]
6.          suma = suma + v[i]
7.      Fin para
8.      prom = suma / 10
9.      Escribir "Promedio:", prom
10.     Escribir "Mayores al promedio"
11.     Para i = 1 hasta 10 hacer
12.         Si v[i] > prom entonces
13.             Escribir v[i]
14.         Fin si
15.     Fin para
16.     Escribir "Menores al promedio"
17.     Para i = 1 hasta 10 hacer
18.         Si v[i] < prom entonces
19.             Escribir v[i]
20.         Fin si
21.     Fin para
22. Fin algoritmo
```

5.4 MANEJO DE MATRICES

Una matriz es un conjunto de elementos dispuestos a lo largo de m filas y n columnas, conocidas también como diagrama de doble entrada o tabla completa (Gran Enciclopedia Espasa, 2005: 7594).

En programación se define una matriz como una estructura de datos estática que bajo un identificador almacena una colección de datos del mismo tipo, es un arreglo de dos dimensiones, organizado en forma de filas y columnas; y por lo tanto utiliza dos índices para identificar los elementos. Gráficamente una matriz tiene la forma de una tabla de doble entrada como la mostrada en la figura 91.

Figura 91. Representación gráfica de una matriz

		Índice de columnas: j					
		1	2	3	4	5	6
Índice de filas: i	1	1,1	1,2	1,3	1,4	1,5	1,6
	2	2,1	2,2	2,3	2,4	2,5	2,6
	3	3,1	3,2	3,3	3,4	3,5	3,6
	4	4,1	4,2	4,3	4,4	4,5	4,6

5.4.1 Declaración

Para declara una matriz se utiliza la misma sintaxis que para los vectores, con la variación de que se utiliza dos tamaños para indicar la cantidad de filas y la cantidad de columnas. La forma general de declarar una matriz es:

TipoDato: identificador[m][n]

Donde *m* es el número de filas y *n* el número de columnas. El total de posiciones disponibles en la matriz es el producto de la cantidad de filas por la cantidad de columnas: *m* * *n*.

Ejemplos:

Entero: mat[4][6]
Carácter: letras[10][12]

En la primera instrucción se declara la matriz mat de cuatro filas y seis columnas, como la mostrada en la figura 89, con capacidad para contener 24 números enteros. En la segunda, la matriz letras de 10 filas y 12 columnas, lo que reserva un espacio para guardar 120 caracteres.

5.4.2 Acceso a elementos

Para hacer referencia a un elemento de una matriz es necesario especificar el identificador o nombre de la matriz y entre corchetes el índice de filas y el índice de columnas, de la forma:

Identificador [i][j]

Donde *i* hace referencia a la fila y *j* a la columna. Al escribir el identificador y los dos índices se hace referencia a un elemento en particular; por tanto, puede utilizarse como una variable sencilla a la que se le asigna datos y desde la que se toman para conformar expresiones o para enviar a un dispositivo de salida.

Ejemplo:

Entero: m[3][4]
m[1][1] = 2
m[2][2] = 5
m[3][4] = 15

La primera instrucción declara la matriz *m*, la segunda asigna el número 2 a la primera posición de *m*, la tercera asigna el número 5 al elemento ubicado en la intersección entre la fila 2 y la columna 2 y de forma similar, la cuarta almacena el número 15 en la posición 3,4. Gráficamente, el resultado de estas expresiones sería como la figura 92.

Figura 92. Representación gráfica de una matriz con datos

	1	2	3	4
1	2			
2		5		
3				15

5.4.3 Recorrido de una matriz

Algunas operaciones se aplican sobre todos y cada uno de los elementos de la matriz, como: leer y guardar datos, imprimir todos los datos, buscar un elemento o sumar una matriz numérica, en estos casos, al acto de visitar consecutivamente todos los elementos de la estructura se le denomina recorrido. Para recorrer una matriz se precisan dos ciclos anidados: uno para desplazarse por las filas y otro por las columnas.

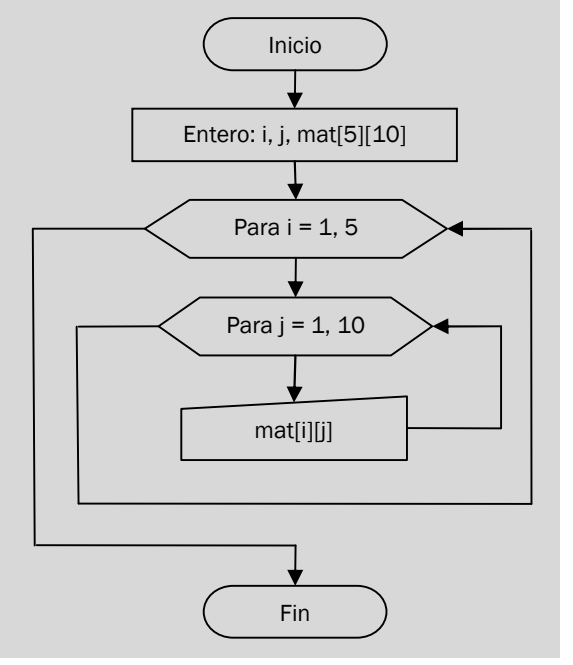
En términos generales, para recorrer una matriz de tamaño $m * n$ se define los ciclos de la forma:

```
Para i = 1 hasta m hacer
    Para un j = 1 hasta n hacer
        Operación sobre el dato
    Fin para
Fin para
```

Ejemplo 47. Llenar una matriz

Para poner en práctica cuanto se ha estudiado respecto a declaración, acceso y recorrido de una matriz, en este ejemplo se declara una matriz de 5 * 10, se recorre posición a posición, se lee y guarda un número en cada una de ellas. El diagrama de flujo de este algoritmo se presenta en la figura 93.

Figura 93. Diagrama de flujo para llenar matriz



En el ejemplo se declara una matriz de 5 filas por 10 columnas para un total de 50 posiciones. Se utiliza un ciclo con la variable *i* para recorrer todas las filas, y estando en cada una de ellas, se utiliza un ciclo con la variable *j* para recorrer cada una de las columnas; es decir, cada una de las posiciones en la fila i-ésima y en cada posición se almacena un valor leído desde el teclado.

Ejemplo 48. Imprimir el contenido de una matriz

Declarar una matriz de 4*6, llenar cada posición con la suma de sus índices y luego mostrar su contenido. Para esto es necesario implementar dos recorridos independientes, uno para cada operación, el primero para llenar la matriz y el segundo para mostrar su contenido. Una vez efectuado el primer recorrido la matriz estará como la que se muestra en la figura 94. El diagrama de N-S de este algoritmo se presenta en la figura 95.

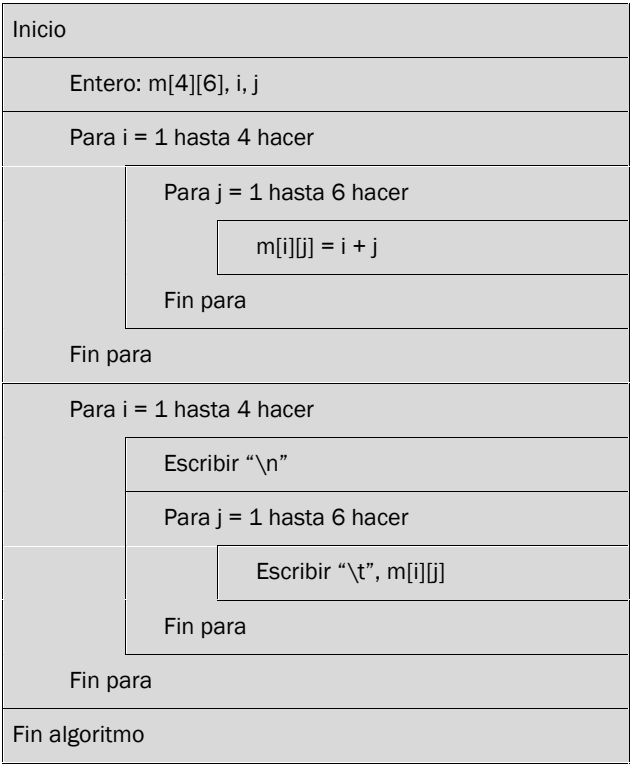
Figura 94. Matriz de 4 * 6

m[i][j]	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10

Este algoritmo implementa dos recorridos de matriz utilizando cuatro ciclos anidados de dos en dos. En el primer recorrido se suman las variables $i + j$ y se guarda el resultado en la posición correspondiente de la matriz; en el segundo recorrido, en cada cambio de fila se imprime "\n", este es un código común en programación que significa salto de línea, es decir: pasar a la línea que sigue, y en cada cambio de columna se imprime código "\t", igualmente común en programación, éste significa introducir una tabulación y se utiliza en este caso para separar los números entre si, seguidamente se imprime el contenido de la celda. De esta forma, al ejecutar este algoritmo se obtiene una salida como ésta:

```
2  3  4  5  6  7
3  4  5  6  7  8
4  5  6  7  8  9
5  6  7  8  9  10
```

Figura 95. Diagrama N-S para imprimir el contenido de una matriz



5.5 MÁS EJEMPLOS CON ARREGLOS

Ejemplo 49. Buscar los datos mayor y menor en un vector

Diseñar un algoritmo para guardar 12 números en un vector y luego identificar el número menor, el número mayor y la posición en que se encuentra cada uno de éstos.

La primera parte de este ejercicio, concerniente a declarar el vector y guardar los números se realiza de igual forma que en los ejercicios anteriores: usando una estructura iterativa. La segunda parte: identificar el número mayor y el número menor, requiere una estrategia de solución.

La estrategia consiste en declarar dos variables: una para mantener la posición del número mayor identificado y otra para la posición del menor. Las dos variables se inicializan en uno para comenzar las comparaciones desde el primer elemento del vector.

Para el diseño de la solución se tiene en cuenta los siguientes datos:

Datos de entrada: número (se lee 12 veces)

Datos de salida: mayor valor, posición del mayor, menor valor y posición del menor

Estructura de almacenamiento: vector de 12 posiciones

Proceso: comparación y asignación

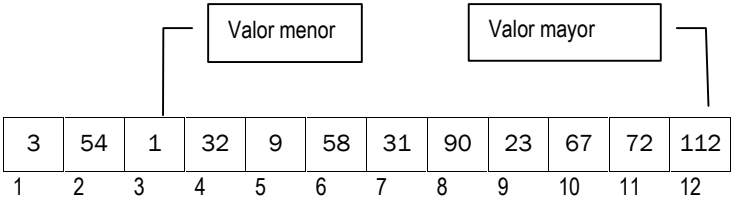
El pseudocódigo de este algoritmo se presenta en el cuadro 80.

Al hacer la verificación del algoritmo e introducir los datos: 3, 54, 1, 32, 9, 58, 31, 90, 23, 67, 72 y 112; se tiene el vector que se muestra en la figura 96.

Cuadro 80. Pseudocódigo para buscar los datos mayor y menor en un vector

```
1.  Inicio
2.      Entero: v[12], i, pmay = 1, pmen = 1
3.      Para i = 1 hasta 12 hacer
4.          Leer v[i]
5.      Fin para
6.      Para i = 1 hasta 12 hacer
7.          Si v[i] > v[pmay] entonces
8.              pmay = i
9.          Si no
10.             Si v[i] < v[pmen] entonces
11.                 pmen = i
12.             fin si
13.         Fin si
14.     Fin para
15.     Escribir "Número mayor:", v[pmay], "en la posición:",pmay
16.     Escribir "Número menor:", v[pmen], "en la posición:",pmen
17. Fin algoritmo
```

Figura 96. Vector con datos del ejemplo 49



La salida obtenida al ejecutar el algoritmo con los datos mencionados es:

```
Número mayor: 112 en posición: 12
Número menor: 1 en posición: 3
```

Ejemplo 50. Recurrencia de un dato en un arreglo

Se tiene una lista de 25 estudiantes y el municipio de origen. Se requiere un algoritmo para guardar la lista en una matriz y consultar los estudiantes que provienen de un municipio determinado.

Este algoritmo se estructura en dos operaciones básicas: la primera consiste en almacenar los datos en una matriz de 25 filas y dos columnas, para lo cual se hace un recorrido leyendo los datos para cada posición; y ubicar los estudiantes del municipio solicitado, esto implica leer el dato a buscar y luego hacer un recorrido en el que se compara el dato de referencia con los datos almacenados en la segunda columna de la matriz, si coinciden se incrementa un contador y se muestra el contenido de la primera columna. Sistematizando la información del ejercicio se tiene:

Datos de entrada: nombres de estudiantes, municipio de procedencia y municipio consultado
Estructura de almacenamiento: matriz de 25 filas y 2 columnas
Datos de salida: nombres de estudiantes y contador
Proceso: comparar y contar

La solución de este ejercicio se presenta en la figura 97.

Ejemplo 51. Diferencia de vectores

Diseñar un algoritmo que almacene datos en dos vectores y luego identifique y muestre los elementos que están en el primero y no están en el segundo.

En este caso, lo primero que se debe hacer es declarar dos vectores y llenarlos con datos, luego se hace un recorrido en el primer vector y por cada elemento de éste se hace un recorrido en el segundo vector buscando las coincidencias del elemento, de esta forma se averigua si el dato se encuentra en los dos arreglos, en cuyo caso se levanta una bandera. Al comenzar cada ciclo interno se inicializa la bandera y si al finalizar el recorrido del segundo vector la bandera se mantiene abajo se muestra el elemento del

primer vector. En la figura 98 se observa gráficamente las comparaciones efectuadas para el primer elemento.

Figura 97. Diagrama N-S recurrencia de datos en una matriz

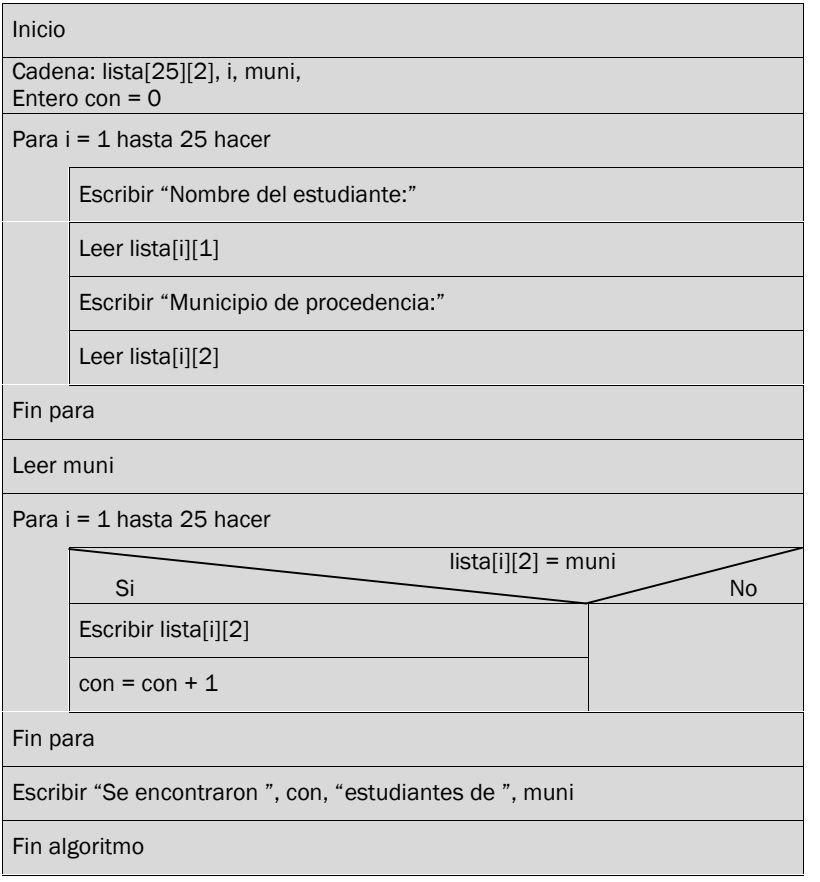
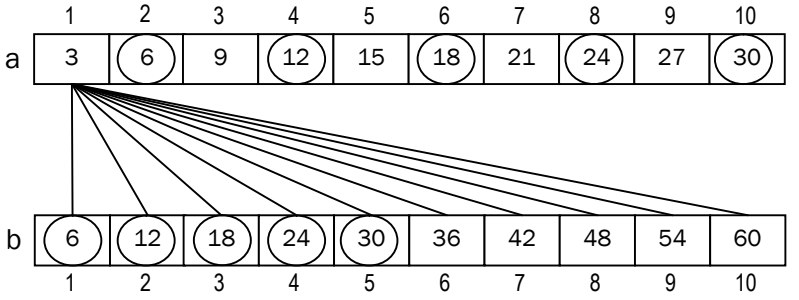
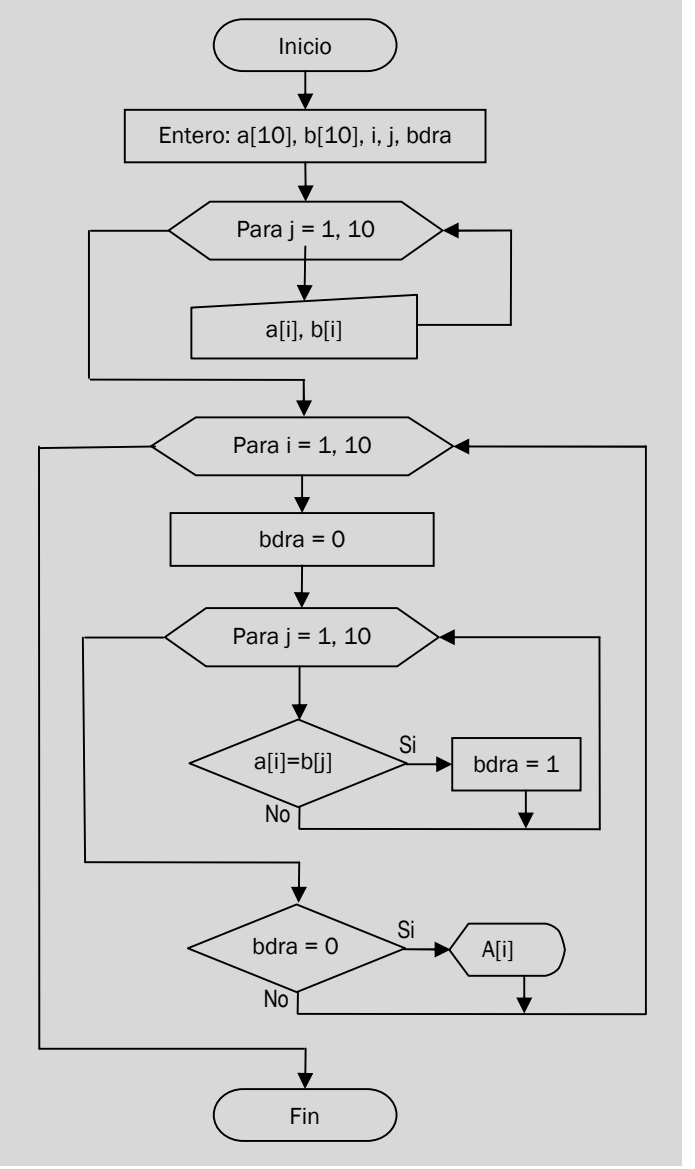


Figura 98. Diferencia de vectores



En este ejemplo el primer vector contiene los primeros 10 múltiplos de 3 y el segundo los primeros 10 múltiplos de 6. Las líneas muestran las comparaciones que se hacen desde el primer elemento del vector 1 con todos los elementos del vector 2, luego se hará lo propio con el segundo elemento y con el tercero, hasta completar el recorrido del primer vector. Los círculos muestras los elementos que durante los recorridos se identifican como comunes a los dos vectores, de esta manera los elementos del primer arreglo que no están encerrados con círculo corresponden a la diferencia: 3, 9, 15, 21, 27. Es decir, los múltiplos de 3 que no son múltiplos de 6. El diagrama de flujo de la solución a este ejercicio se presenta en la figura 99.

Figura 99. Diagrama de flujo del algoritmo diferencia de vectores

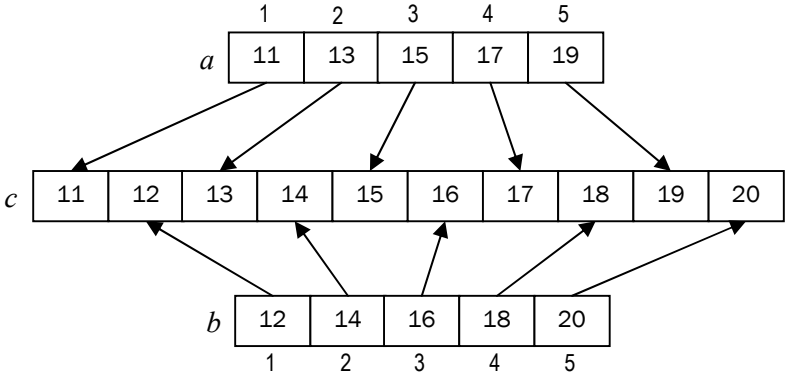


Ejemplo 52. Intercalar vectores

Dados dos vectores a y b de n elementos se requiere un algoritmo para generar un tercer vector (c) con los datos de los dos primeros intercalados. El nuevo vector tiene un tamaño de $2n$ posiciones.

Como ejemplo tómese dos vectores (a y b) de cinco elementos e intercálense sus elementos para crear el vector c de 10 elementos como se muestra en la figura 100.

Figura 100. Intercalación de vectores



En la solución de este ejercicio se declaran los tres arreglos, los dos primeros se llenan con datos registrados por el usuario y el tercero es el resultado de intercalar los dos primeros. La lectura y almacenamiento de datos se hace para los dos vectores dentro de un mismo ciclo, el intercalado se hace con un segundo ciclo. El algoritmo se muestra en el cuadro 81.

Cuadro 81. Pseudocódigo para intercalar vectores

```
1.  Inicio
2.      Entero a[5], b[5], c[10], i=1, x=1
3.      Mientras i <= 5 hacer
4.          Leer a[i], b[i]
5.          i = i + 1
6.      Fin mientras
7.      Mientras i <= 5 hacer
8.          c[x] = a[i]
9.          x = x + 1
10.         c[x] = b[i]
11.         x = x + 1
12.         i = i + 1
13.     Fin mientras
14. Fin algoritmo
```

Ejemplo 53. Sumatoria de filas y columnas de una matriz

Dada una matriz de *m*n* diseñar un algoritmo para sumar cada una de las filas y guardar los resultados en un vector llamado *sumafila*, sumar cada una de las columnas y guardar los resultados en el vector *sumacol*, finalmente mostrar los dos vectores.

Considérese una matriz de cuatro filas por seis columnas, esto implica que el vector *sumafila* deberá tener cuatro posiciones y el vector *sumacol* seis posiciones, como se muestra en la figura 101.

En este ejemplo se supone que la matriz se llena con los números consecutivos del 1 al 24. En el algoritmo que se presenta en la figura 102 los datos los digita el usuario.

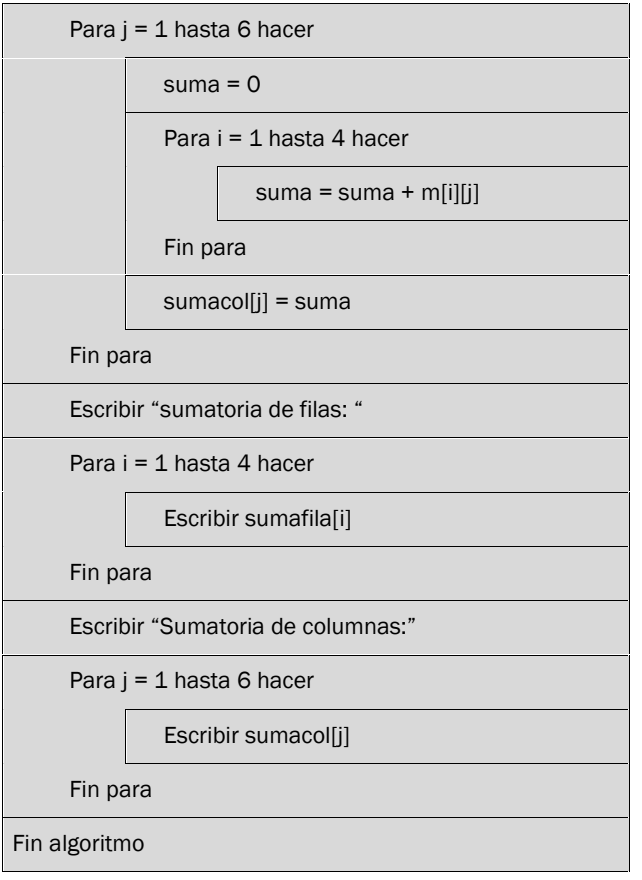
Figura 101. Sumatoria de filas y columnas de una matriz

	1	2	3	4	5	6	
1	1	2	3	4	5	6	21
2	7	8	9	10	11	12	57
3	13	14	15	16	17	18	93
4	19	20	21	22	23	24	129
	40	44	48	52	56	60	

Figura 102. Diagrama N-S para sumatoria de filas y columnas de una matriz

```
Inicio
    Entero: m[4][6], sumafila[4], sumacol[6], i, j, suma
    Para i = 1 hasta 4 hacer
        Para j = 1 hasta 6 hacer
            Leer m[i][j]
        Fin para
    Fin para
    Para i = 1 hasta 4 hacer
        suma = 0
        Para j = 1 hasta 6 hacer
            suma = suma + m[i][j]
        Fin para
        sumafila[i] = suma
    Fin para
```

Figura 100. (Continuación)



Para sumar las filas se precisa hacer un recorrido con dos ciclos anidados, el externo controla el índice de filas y el interno el de columnas, el acumulador se inicializa antes de comenzar la ejecución del ciclo interno y se guarda su valor al terminar este ciclo. Para sumar las columnas se procede de forma similar, pero teniendo en cuenta que el ciclo externo corresponde al índice de columnas y el interno a filas.

Al ejecutar el algoritmo e introducir los números del 1 al 24 los resultados serían:

```
Sumatoria de filas: 21 57 93 129
Sumatoria de columnas: 40 44 48 52 56 60
```

Ejemplo 54. Diagonal principal de una matriz

Diseñar un algoritmo para mostrar y sumar la diagonal principal de una matriz cuadrada.

Antes de diseñar el algoritmo, en la figura 103 se identifica gráficamente la diagonal principal en una matriz de 5 * 5.

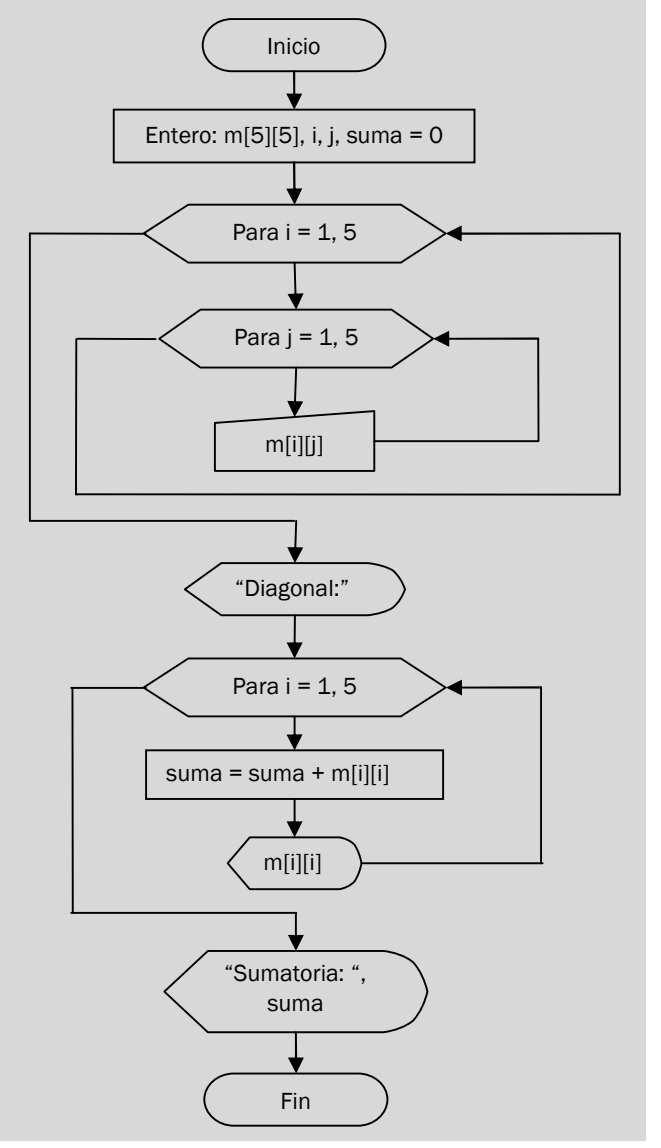
Figura 103. Diagonal principal de una matriz

	1	2	3	4	5
1	**1**	2	3	4	5
2	6	**7**	8	9	10
3	11	12	**13**	14	15
4	16	17	18	**19**	20
5	21	22	23	24	**25**

Observando la gráfica se encuentra que los elementos de la diagonal principal tienen el mismo índice en filas y columnas: *m[1][1], m[2][2], m[3][3], m[4][4] y m[5][5]*. De esto se concluye que para recorrer la diagonal no se requieren dos ciclos, basta con uno, y la variable de control del ciclo se la utiliza como índice de filas y de columnas.

En el diagrama de flujo de la figura 104 se implementan dos ciclos anidados para leer números y guardarlos en la matriz, luego se implementa un ciclo para recorrer la diagonal, en cada iteración se muestra el elemento y se lo adicional al acumulador suma.

Figura 104. Diagrama de flujo del algoritmo diferencia de vectores



Ejemplo 55. Procesar notas utilizando arreglos

Se requiere un algoritmo para procesar las notas de un grupo de 40 estudiantes, se cuenta con tres notas parciales para cada estudiante y se desea conocer la nota definitiva, la que se obtiene por promedio. Después de procesar las notas de todos los estudiantes se desea conocer el promedio del grupo.

Para solucionar este ejercicio se requiere utilizar un vector y una matriz. En el vector se almacenan los nombres de los estudiantes y en la matriz las notas parciales y la definitiva. En consecuencia, el vector será de 40 posiciones y la matriz de 40 filas por cuatro columnas. En la figura 105 se muestra el modelo de la estructura de datos para 6 estudiantes.

Figura 105. Arreglos para procesar notas

Vector para nombres

1	Carlos Salazar
2	Carmen Bolaños
3	Margarita Bermúdez
4	Sebastián Paz
5	Juan Carlos Díaz
6	Verónica Marroquín

Matriz para notas

	1	2	3	4
1	3,5	2,5	4,0	3,3
2	4,2	3,8	4,5	4,2
3	2,0	4,5	4,8	3,8
4	2,5	3,5	4,4	3,5
5	3,4	3,5	3,8	3,6
6	4,0	4,6	3,3	4,0
	Parcial 1	Parcial 2	Parcial 3	Definitiva

Aunque se manejan dos estructuras de datos: un vector y una matriz, los datos pueden mantener su integridad a través del índice del vector y el índice de filas. En la matriz las columnas corresponden a las notas, las tres primeras a las notas parciales y la última a la nota definitiva. En este orden de ideas, los datos se leen así: Carlos Salazar tiene 3,5, 2,5 y 4,0 en las calificaciones

parciales y su nota definitiva es 3,3. De igual manera en las demás filas.

Como cada fila en el vector y en la matriz hace referencia a un mismo estudiante, las iteraciones se hacen estudiante por estudiante, por lo tanto sólo se utiliza un ciclo y dentro de éste se lee el nombre y las notas parciales, se calcula la nota definitiva, se muestra y se acumula para posteriormente calcular el promedio del curso. El algoritmo se presenta en notación de pseudocódigo en el cuadro 82.

Cuadro 82. Pseudocódigo para procesar notas utilizando arreglos

```
1.  Inicio
2.      Cadena: nombres[40]
        Real: notas[40][4], suma = 0, prom
        Entero: con
3.      Para con = 1 hasta 40 hacer
4.          Escribir "Nombre: \t "
5.          Leer nombres[con]
6.          Escribir "Nota parcial 1: \t "
7.          Leer notas[con][1]
8.          Escribir "Nota parcial 2: \t "
9.          Leer notas[con][2]
10.         Escribir "Nota parcial 3: \t "
11.         Leer notas[con][3]
12.         notas[con][4] = (notas[con][1] + notas[con][2] + notas[con][3])/3
13.         Escribir "Definitiva: \t", notas[con][4]
14.         suma = suma + notas[con][4]
15.     Fin para
16.     prom = suma / 40
17.     Escribir "Promedio del curso: \t", prom
18. Fin algoritmo
```

Verificando el algoritmo y procesando las notas que se utilizaron en el modelo de la figura 105 se tiene como salida:

```
Nombre:  Carlos Salazar
Nota parcial 1: 3,5
Nota parcial 2: 2,5
Nota parcial 3: 4,0
Definitiva:    3,3

Nombre:  Carmen Bolaños
Nota parcial 1: 4,2
Nota parcial 2: 3,8
Nota parcial 3: 4,5
Definitiva:    4,2

…              …

Promedio del grupo:  3,7
```

5.6. EJERCICIOS PROPUESTOS

Solucionar los siguientes ejercicios

1. Dados dos vectores numéricos diseñar un algoritmo que identifique y muestre los números que tienen en común.

2. Diseñar un algoritmo para sumar dos vectores numéricos

3. Diseñar un algoritmo para trasponer una matriz cuadrada

4. Dadas dos matrices: $A[m][n]$ y $B[n][m]$, diseñar un algoritmo para comprobar si la matriz B es la traspuesta de A.

5. Diseñar un algoritmo para calcular el producto de un número por un vector numérico de 10 elementos.

6. Diseñar un algoritmo para multiplicar dos vectores

7. Diseñar un algoritmo para insertar un dato en un vector en una posición escogida por el usuario, si la posición está ocupada los datos se desplazan a la derecha para dar espacio al nuevo dato. Si el vector está lleno no se inserta el dato y se muestra un mensaje.

8. Se tiene un vector de caracteres en el que se ha almacenado una frase. Se requiere un algoritmo que determine si la frase es un palíndromo*.

9. Diseñar un algoritmo para hacer circular los datos de un vector: todos los elementos se desplazan una posición y el último elemento pasa a ocupar la primera posición.

10. Diseñar un algoritmo para determinar si dos matrices son iguales; es decir, verificar si para todos los elementos se cumple que $A[i][j] = B[i][j]$

11. Diseñar un algoritmo para determinar si dos matrices contienen los mismos elementos aunque estos no se presenten en el mismo orden.

12. Diseñar un algoritmo para sumar dos matrices

13. Diseñar un algoritmo para determinar si una matriz es simétrica

14. Diseñar un algoritmo para calcular el producto de un número por una matriz

15. Diseñar un algoritmo para calcular el producto de dos matrices

16. Dada la matriz $Letras[20][20]$ que está completamente llena de caracteres del alfabeto calcular la frecuencia absoluta y la frecuencia relativa para cada una de las vocales.

* Palindromo: palabra o frase que se lee igual de izquierda a derecha, que de derecha a izquierda. (Circulo Enciclopedia Universal, 2006:1670)

17. Para facturar el servicio de energía la empresa Energía Para Todos cuenta con una lista de usuarios almacenada en un vector y la lista de lecturas del mes anterior en un segundo vector. Se requiere un algoritmo que lea el valor del kW, tome la lectura actual de cada usuario y la registre en un tercer vector, calcule el consumo del mes por diferencia de lecturas y muestre para cada usuario: nombre, consumo y valor a pagar.

6. SUBPROGRAMAS

Un ordenador digital
es como el cálculo.
Puede dividir un problema
en partes tan pequeñas
como desee.
Van Doren

A lo largo de este documento se han propuesto y explicado numerosos ejemplos; no obstante, estos han sido pensados para ilustrar un tema en particular y no tienen la complejidad de los problemas reales, en los que la solución puede incluir miles, cientos de miles o millones de líneas de código fuente.

Los programas diseñados para solucionar problemas reales de procesamiento de datos, que atienden conjuntos grades de requisitos y responden a múltiples restricciones, aquellos que Booch(1994: 4) denomina software de dimensión industrial no pueden emprenderse como una sola pieza, pues su construcción, corrección y mantenimiento resultaría una tarea demasiado difícil y costosa.

La estrategia más difundida para reducir la complejidad se conoce como *divide y vencerás*. Ésta consiste en descomponer el problema que se quiere resolver en un determinado número de subproblemas más pequeños, resolver sucesiva e independientemente todos los subproblemas, luego combinar las soluciones

obtenidas para, de esta manera, solucionar el problema original (Brassard y Bratley, 1997: 247), (Aho, Hopcroft y Ullman, 1988: 307). De ésta forma, un programa está conformado por varios subprogramas.

Un subprograma o subrutina implementa un algoritmo diseñado para una tarea particular, por tanto no puede constituirse en una solución a un problema por sí mismo, sino que es llamado desde otro programa o subprograma.

En Oviedo (2002) se puede identificar algunas ventajas de la utilización de subprogramas, como son: la fragmentación del programa en módulos proporciona mayor claridad y facilidad para distribuir el trabajo entre los miembros del equipo de desarrollo, facilidad para escribir, probar y corregir el código, estructura lógica del programa más comprensible y la posibilidad de ejecutar repetidas veces el subprograma.

Considerando la naturaleza de la tarea que el subprograma realiza y si devuelve o no datos al programa que lo llama, los subprogramas se catalogan como funciones o procedimientos (Oviedo, 2002).

6.1. FUNCIONES

El concepto de función en el campo de la programación fue tomado de las matemáticas y consiste en establecer una relación entre dos conjuntos: origen y espejo, donde a cada elemento del primero le corresponde uno del segundo (Gran Enciclopedia Espasa, 2005: 5140). En términos de Joyanes (1996: 166), una función es una operación que toma uno o más valores y genera un nuevo valor como resultado. A los valores que entran a la función se les denomina argumentos. En este mismo sentido, Timarán *et al* (2009: 57) define una función como: un subprograma que recibe uno o más parámetros, ejecuta una tarea específica y devuelve un único valor como resultado, al programa o función que la invocó.

Una función se representa de la forma:

$y = f(x)$
$f(x) = 3x - 2$

Y se lee y depende de x o y es una función de x y significa que para obtener el valor de y es necesario reemplazar x por un valor (argumento) y realizar la operación que indica la función. Por ejemplo, si se resuelve la función con $x = 2$, se obtiene $y = 4$.

Una característica importante de las funciones es que su resultado depende exclusivamente del valor o los valores que recibe como argumentos. Si se toma la función del Máximo Común Divisor, ésta requiere dos argumentos y el resultado que genere será diferente para cada par de argumentos.

Sea $f(x,z)$ el Máximo Común Divisor de x y z. Entonces:

$f(12,18) = 6$
$f(40,60) = 20$

Las funciones se clasifican en dos tipos: internas y externas.

Funciones internas. Son aquellas que están definidas en el lenguaje de programación, se distribuyen en forma de librerías* o de APIs† y se utilizan como base para el desarrollo de los programas. Estas atienden necesidades generales y suelen agruparse dependiendo del dominio en que pueden ser utilizadas, por ejemplo: matemáticas, cadenas, fechas, archivos.

Funciones externas. También se conocen como funciones definidas por el usuario y son todas aquellas escritas por el

* Es una colección de programas y subprogramas normilizados y probados con los que se puede resolver problemas específicos (Lopezcano, 1998: 322).

† API es una sigla de las palabras en Inglés Application Programming Interface, en Español: Interfaz para programación de aplicaciones o conjunto de herramientas para desarrollar un programa.

programador para atender necesidades particulares en sus aplicaciones.

6.1.1. Diseño de una Función

Una función tiene dos partes: la definición y la implementación, algunos autores, como Joyanes (1996: 167) y Galve *et al* (1993: 30), las denominan cabecera y cuerpo respectivamente. En un programa, por ejemplo en lenguaje C, la definición de la función le proporciona al compilador información sobre las características que tiene la función cuya implementación aparecerá más adelante. Esto ayuda a verificar la corrección del programa.

Definición de la función. Ésta incluye tres elementos que son explícitos en las notaciones pseudocódigo y diagramas de Nassi-Shneiderman, más no en los diagramas de flujo, estos son:

a. El tipo de retorno
b. El identificador o nombre de la función
c. La lista de parámetros

Hasta hace algunos años era común definir una función de la forma:

Función identificador(parámetros): tipo_retorno

Ejemplo:

Función factorial(entero n): entero

Esta instrucción define la función llamada factorial la que recibe un número entero como parámetro y lo almacena en la variable *n* y devuelve como resultado un número entero.

Actualmente y para mantener la similitud con los lenguajes de programación actuales se definen de la forma:

Tipo_retorno identificador(parámetros)

Ejemplos:

Entero factorial(entero n)
Real potencia(entero base, entero exponente)
Boobleano esprimo(entero n)

Se ha suprimido la palabra reservada función y se coloca el tipo de retorno al comenzar la definición. Esta es la forma como se definirán las funciones en lo que sigue del libro.

Parámetros. Dado que una función se especializa en desarrollar una tarea, que por lo general es un cálculo, no se puede esperar que también se encargue de la lectura e impresión de datos; por eso, es necesario suministrarle los datos que ésta requiere para desarrollar el cálculo, a estos datos que se le entregan se les denomina parámetros.

En este orden de ideas, los parámetros son los datos que se le envían a la función para que desarrolle la tarea para la que fue diseñada. Por ejemplo, si se diseña una función para calcular el factorial, es necesario pasarle como parámetro el número para el cual se desea encontrar el factorial; si se diseña una función para calcular una potencia, es preciso que se le proporcione la base y el exponente.

La declaración de parámetros forma parte de la definición de la función, se trata de colocar entre paréntesis los tipos y los identificadores de los datos que se entregarán a la función. Estas declaraciones se constituyen en variables locales de la función. En la definición:

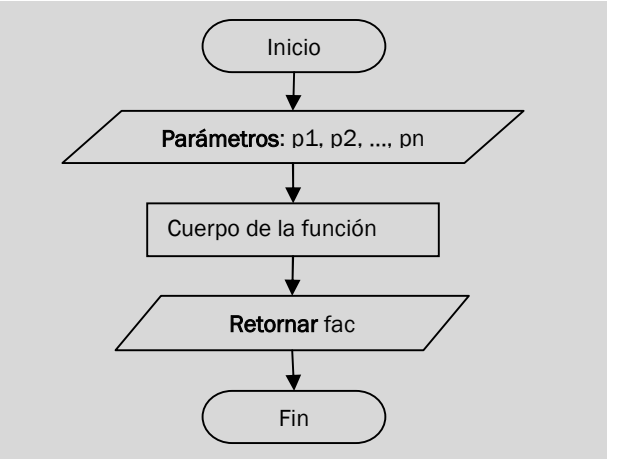
Real potencia(entero base, entero exponente)

Se está expresando que la función potencia recibirá dos datos de tipo entero y los almacenará en las variables: *base* y *exponente*.

En diagrama de flujo, una función no tiene una definición explícita. No obstante, en este documento, se la diferencia de un

programa en cuanto incluye un símbolo de entrada/salida para recibir los parámetros y otro para devolver el resultado, como se aprecia en la figura 106.

Figura 106. Diagrama de flujo de una función



Si bien la definición es fundamental por cuando le da existencia e identidad a la función, la implementación es lo que permite que la función desarrolle una operación.

Implementación. Es el conjunto de líneas de código o instrucciones que procesan los datos recibidos como parámetros y producen el resultado esperado por el programa que invoca la función. El cuerpo de la función comienza después de la definición y termina con la palabra reservada *fin* seguido del nombre de la función, pero en la penúltima línea debe aparecer la instrucción *Retornar o Devolver* seguida de un valor o una expresión, como se puede apreciar en los ejemplos 54 y 55.

Ejemplo 54. Función sumar

Diseñar una función para sumar dos números enteros

Este ejemplo tiene como propósito mostrar la estructura de una función, haciendo evidente su definición, parámetros, implementación y retorno.

La función sumar recibe dos parámetros enteros, los suma y devuelve el resultado, como se muestra en el cuadro 83.

Cuadro 83. Pseudocódigo de la función sumar

```
1   Entero sumar(entero a, entero b)
2       Entero resultado
3       resultado = a + b
4       Retornar resultado
5   Fin sumar
```

Obsérvese que la función se define de tipo entero, lo que significa que el dato que devolverá será de este tipo; en consecuencia, la variable local que se declara para almacenar el resultado es de tipo entero. En cuanto a los parámetros, aunque son del mismo tipo es necesario especificar para cada uno que son de tipo entero, ya que no siempre ocurre así.

Las variables que se declaran en el cuerpo de una función son de ámbito local y por tanto sólo se puede acceder a ellas dentro de la función. En cuanto la ejecución termina, las variables dejan de existir.

Ejemplo 55. Función factorial

Diseñar una función para calcular en factorial de un número entero positivo. Se sabe que el factorial de 0 es 1 y el factorial de cualquier número n mayor 0 es el producto de los números entre 1 y n.

La función factorial requiere un parámetro: el número, y devuelve otro número: el factorial, en su implementación incluye un ciclo en el que se calcula el producto desde 1 hasta *n*, como se muestra en el pseudocódigo del cuadro 84.

Cuadro 84. Pseudocódigo de la función factorial

```
1   Entero factorial(entero n)
2       Entero fac = 1, con
3       Para con = 1 hasta n hacer
4           fac = fac * con
5       Fin para
6       Devolver fac
7   Fin factorial
```

6.1.2. Invocación de una Función

La invocación o llamada a una función se hace escribiendo su nombre y la lista de parámetros que ésta requiere, de la forma:

Nombre_función(parámetros)

Ejemplo:

Sumar(123, 432)
Factorial(5)

Al ejecutarse un algoritmo o un programa, en el momento en que se encuentra la invocación de una función, el control de ejecución pasa a la función. En la ejecución de la función, la primera tarea que se realiza es la declaración de las variables locales y su inicialización con los datos proporcionados como parámetros. Por eso, es muy importante tener en cuenta que los argumentos escritos al invocar la función y la lista de parámetros definidos en el diseño de la función deben coincidir en tipo y en cantidad, en caso

contrario se presentará un error y la ejecución de la función se cancelará.

En el momento en que la ejecución de la función encuentra la palabra *retornar* el control vuelve exactamente a la línea donde se invocó la función con el resultado del cálculo efectuado.

El resultado devuelto por una función puede almacenarse en una variable, puede ser enviado directamente a un dispositivo de salida o puede utilizarse como argumento para otra función, como se muestra en las expresiones *a, b* y *c*, respectivamente.

a. *x = factorial(8)*
b. *Escribir "Factorial de 8:", factorial(8)*
c. *sumar(factorial(3), factorial(4))*

Ejemplo 56. Operaciones aritméticas

Diseñar el algoritmo principal y las funciones necesarias para realizar las operaciones aritméticas: suma, resta, multiplicación y división de dos números.

Para solucionar este ejercicio es necesario diseñar cuatro funciones, una para cada operación, y un programa principal que invoque la función que corresponda según el requerimiento del usuario. Como cada función se encarga exclusivamente de hacer el cálculo, el programa principal debe encargarse de leer los números, la operación a realizar y mostrar el resultado.

En la implementación de los programas en algunos lenguajes es necesario definir las funciones antes de invocarlas. Para desarrollar buenas prácticas de programación, en este ejercicio y en los que siguen, se diseña primero las funciones o procedimientos y al final el programa principal.

Las funciones y el programa para este ejercicio se presentan en los cuadros 85, 86, 87 y 88. La función sumar ya se diseño y aparece en el cuadro 83 por ello no se incluye.

Cuadro 85. Pseudocódigo de la función restar

```
1   Entero restar(entero a, entero b)
2       Entero diferencia
3       resultado = a – b
4       Retornar diferencia
5   Fin restar
```

Cuadro 86. Pseudocódigo de la función multiplicar

```
1   Entero multiplicar(entero a, entero b)
2       Entero producto
3       resultado = a * b
4       Retornar producto
5   Fin multiplicar
```

Cuadro 87. Pseudocódigo de la función dividir

```
1   Entero dividir(entero a, entero b)
2       Real cociente = 0
3       Si b != 0 entonces
4           cociente = a / b
5       Fin si
6       Retornar cociente
7   Fin dividir
```

Cuadro 88. Pseudocódigo del algoritmo operaciones aritméticas

```
1   Inicio
2       Entero x, y, opc
3       Leer x, y
4       Escribir "1. Sumar  2. Restar 3. Multiplicar 4. Dividir"
5       Leer opc
6       Según sea opc hacer
7           1: Escribir "sumatoria = ", sumar(x,y)
8           2: Escribir "Diferencia = ", restar(x,y)
9           3: Escribir "Producto = ", multiplicar(x,y)
10          4:  Si y = 0 entonces
11                  Escribir "Error, divisor = 0"
12              Si no
13                  Escribir "Cociente = ", dividir(x,y)
14              Fin si
15      Fin según sea
16  Fin suma
```

En las líneas 7, 8, 9 y 13 se realiza el llamado a las funciones previamente definidas. Aunque en la función dividir incluye un condicional para evitar el error de dividir sobre 0, la validación de los datos se hace en el algoritmo principal (línea 10) ya que la función sólo puede retornar un valor: el cociente, no un mensaje de error.

6.1.3. Más ejemplos de funciones

Ejemplo 57. Función potencia

La potenciación es una función matemática que consta de una base y un exponente: a^n. El resultado es el producto de multiplicar la base por sí misma tantas veces como indique el exponente, como se explicó en el ejemplo 40.

Esta función requiere dos parámetros: la base y el exponente. En su implementación se incluye un ciclo para calcular el producto desde uno hasta el valor absoluto de n y una decisión para determinar si el exponente es negativo, en cuyo caso el resultado es uno sobre el producto obtenido.

$$a^{-n} = \frac{1}{a^n}$$

En el cuadro 89 se diseña la función para obtener el valor absoluto de un número, en el cuadro 90 la función potencia y en el 91 el algoritmo principal que invoca a potencia.

Cuadro 89. Pseudocódigo de la función valor absoluto

```
1   Entero valorabsoluto (entero n)
2       Si n < 0 entonces
3           n = n * -1
4       Fin si
5       Retornar n
6   Fin valorabsoluto
```

Cuadro 90. Pseudocódigo de la función potencia

```
1   Real potencia(entero a, entero n)
2       Entero p= 1, x
3       Para x = 1 hasta valorabsoluto(n) hacer
4           p = p * a
5       Fin para
6       Si n < 0 entonces
7           p = 1 / p
8       Fin si
9       Retornar p
10  Fin potencia
```

Cuadro 91. Pseudocódigo del algoritmo principal para potenciación

```
1   Inicio
2       Entero b, e
3       Leer b, e
4       Escribir "Potencia = ", potencia(b,e)
5   Fin algoritmo
```

En este ejemplo se puede apreciar cómo el uso de funciones facilita la solución del problema. En este caso, el algoritmo principal se encarga de leer los datos y mostrar el resultado, la función *potencia()* se encarga de calcular el resultado y la función *valorabsoluto()* facilita que se pueda calcular tanto con exponente positivo como con negativo.

Para verificar que una función produce el resultado esperado se procede de igual manera que para los demás algoritmos: se ejecuta paso a paso y se registra los datos que se guardan en las variables. Para verificar la corrección de la solución general se coloca cada variable y cada función como una columna para registrar el resultado que devuelve, como se aprecia los cuadros 92 y 93.

Cuadro 92. Verificación función potencia

Ejecución	a	n	x	Valorabsoluto(n)	p
1	4	3	 1 2 3	3	1 4 16 64
2	3	-4	 1 2 3 4	4	1 3 9 27 81 1/81

Cuadro 93. Verificación del algoritmo para calcular una potencia

Ejecución	b	e	Potencia(b,e)	Salida
1	4	3	64	Potencia = 64
2	3	-4	1/81	Potencia = 1/81

Ejemplo 58. Función Interés simple

Diseñar una función para calcular el interés simple de una inversión.

El interés simple es el beneficio que se obtiene por invertir un determinado capital. En esta operación intervienen tres elementos: el capital invertido, la tasa de interés o razón por unidad de tiempo y los periodos de tiempo pactados en la inversión.

Por ejemplo: Juan presta 10 millones de pesos a Carlos por un periodo de seis meses, Carlos se compromete a pagar una tasa de interés del 2% mensual al final de los seis meses. Cuánto recibirá Juan por concepto de interés?

La fórmula para calcular el interés simple es:

$$I = C * R * T$$

Donde:

I: interés simple
C: capital invertido
R: tasa de interés o razón, es un porcentaje
T: tiempo

Retomando los datos del ejemplo se tiene:

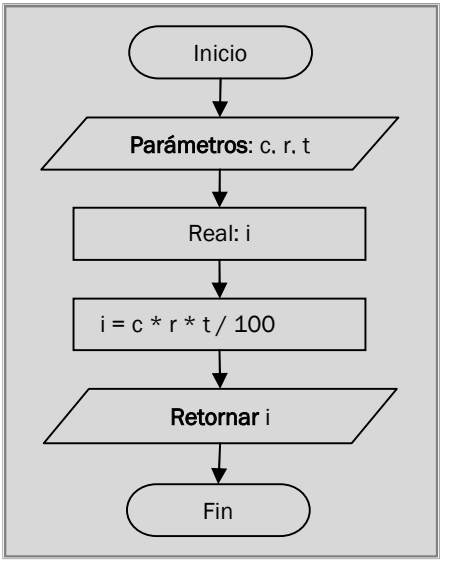
C = 10.000.000
R = 2%
T = 6

En consecuencia:

I = 10.000.000 * 2 * 6 / 100 = 1.200.000

Ahora hay que expresar la fórmula del interés simple como una subrutina o función, como se presenta en notación de diagrama de flujo en la figura 107.

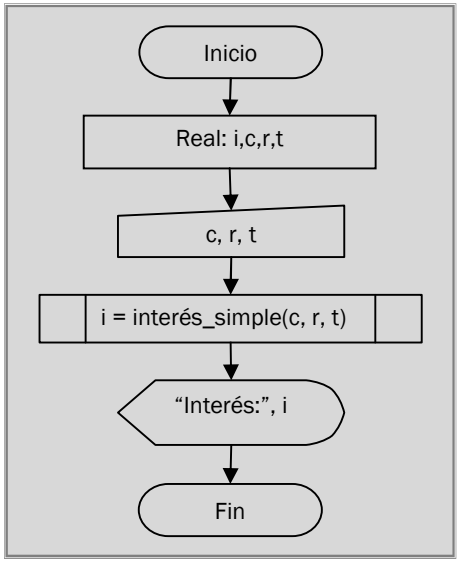
Figura 107. Diagrama de flujo de la función interés simple



En la figura 108 se presenta el diagrama de flujo del algoritmo principal para calcular el interés simple de una inversión. Este se encarga de leer los datos, invocar la función, recibir el resultado y finalmente presentarlo al usuario.

En este ejemplo se utiliza una variable i tanto en el algoritmo del programa principal como en la función. Cabe resaltar que se trata de dos variables independiente, debido a que se declaran como variables locales en cada algoritmo y por tanto no existe la una en el ámbito de la otra.

Figura 108. Diagrama de flujo de algoritmo interés simple



Los resultados de la verificación paso a paso de éste algoritmo se presentan en el cuadro 94.

Cuadro 94. Verificación solución para calcular interés simple

Ejecución	C	r	T	i	Salida
1	10000000	2%	6	1200000	Interés: 1200000
2	20000000	1,5%	12	3600000	Interés: 3600000

Ejemplo 59. Función nota definitiva

En la universidad Buena Nota las notas definitivas de cada asignatura se obtienen promediando tres notas parciales, donde la primera y la segunda equivalen al 30% cada una y la tercera al 40% de la nota final. Se requiere una función para hacer dicho cálculo.

Según el planteamiento anterior, si un estudiante obtiene las notas:

Nota 1 = 3,5
Nota 2 = 4,3
Nota 3 = 2,5

Su nota definitiva será:
Nota definitiva = 3,5 * 30/100 + 4,3 * 30/100 + 2,5 * 40/100
Nota definitiva = 3,3

En general, la nota se obtiene mediante la aplicación de la fórmula:

Nota definitiva = nota 1 * 30/100 + nota 2 * 30/100 + nota 3 * 40/100

La función para realizar esta tarea se presenta en el cuadro 95.

Cuadro 95. Pseudocódigo de la función nota definitiva

```
1   Real notadefinitiva(real n1, real n2, real n3)
2       Real nd
3       nd = n1 * 30/100 + n2 * 30/100 + n3 * 40/100
4       Retornar nd
5   Fin notadefinitiva
```

Ejemplo 60. Función área de un triángulo

El área de un triángulo es equivalente a la mitad del producto de su base por su altura.

Área = base * altura / 2

Por tanto, la función para calcular el área del triángulo requiere que se le proporcione la base y la altura. El pseudocódigo de esta función se presenta en el cuadro 96.

Cuadro 96. Pseudocódigo de la función área de un triángulo

```
1   Real areatriangulo(real base, real altura)
2       Real area
3       area = base * altura / 2
4       Retornar area
5   Fin areatriangulo
```

Ejemplo 61. Función año bisiesto

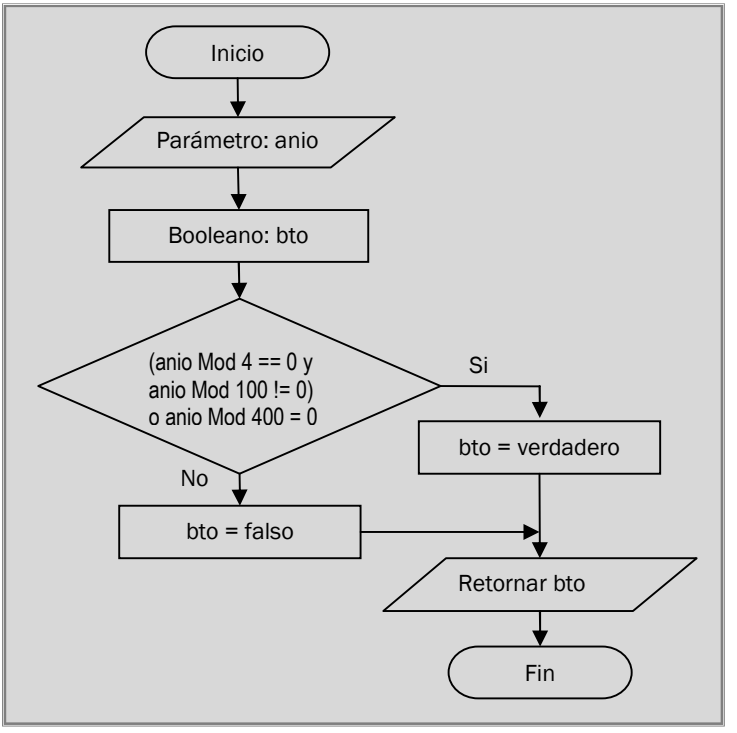
En el calendario gregoriano se cuenta con un año bisiesto* cada cuatro años, excepto el último de los centenarios que no son divisibles para 400. Diseñar una función para determinar si un año es bisiesto.

Tómese, por ejemplo, el año 1700. Es divisible para 4 y también para 100, pero no es divisible de forma exacta para 400, por tanto no es bisiesto. Mientras que 1904 es divisible para 4, pero no es divisible para 100, por tanto si es bisiesto; 2000 es divisible para 4, para 100 y también para 400, en consecuencia es bisiesto.

La condición para que un año sea bisiesto es: que sea divisible para 4 y no para cien ó que sea divisible para 400. Es decir, cualquier año divisible para cuatro que no sea el último del centenario es bisiesto, pero si es el último del centenario debe ser divisible para 400. La función se presenta en la figura 109.

* Un año bisiesto es aquel que cuenta con 366 días a diferencia de los otros que tienen 365, éste día adicional se incluye en el mes de febrero que en dicho caso registra 29 días.

Figura 109. Diagrama de flujo de la función año bisiesto



Ejemplo 62. Función contar caracteres

Diseñar una función que reciba como parámetros un arreglo de caracteres y un carácter. La función debe contar la cantidad de veces que aparece el carácter en el arreglo y devolver dicho contador.

Por ejemplo, si el arreglo está conformado por los caracteres: a, b, c, d, e, d, c, b, a y el carácter a buscar es b, éste aparece dos veces en el arreglo.

La función debe hacer un recorrido del vector y en cada posición comparar su contenido con el carácter de referencia, si estos

coinciden incrementa el contador. El problema es que para el recorrido parece necesario conocer el tamaño del vector y no se cuenta con dicho dato.

Algunos lenguajes de programación cuentan con una función interna que devuelve el tamaño de un vector, algunos otros reportan un valor falso o nulo cuando se intenta acceder a una posición que no existe en el vector. En este caso para evitar cualquier posibilidad de error, se incluye un tercer parámetro que corresponde al tamaño del vector, de esta manera se puede hacer el recorrido sin dificultad. Esta función se presenta en el cuadro 97.

Cuadro 97. Pseudocódigo de la función contar caracteres

```
1  Real contarcaracteres(caracter v[], caracter car, entero n)
2      Entero i, con = 0
3      Para i = 1 hasta n hacer
4          Si car = v[i] entonces
5              con = con + 1
6          Fin si
7      Fin para
8      Retornar con
9  Fin contarcaracteres
```

Ejemplo 63. Sumar días a una fecha

Se requiere una función para sumar a una fecha (año, mes, día) un número determinado de días.

Supóngase que se tiene la fecha: 2000/07/01 y se le suman 5 días, la fecha obtenida será: 2000/07/06, en este caso sólo se suma los días, pero si se tiene la fecha 2005/12/28 y se le suman 87 días, ¿cuál será la nueva fecha? ¿Qué operaciones realizar?

Para realizar esta suma, lo primero por hacer es extraer los años y los meses que hay en el número de días, con lo que se obtiene un

dato en formato de fecha, con el propósito de sumar días con días, meses con meses y años con años.

Años = 87 / 365 = 0
Meses = 87 – (años * 365) / 30 = 2
Días = 87 - (años * 365) – (meses * 30) = 27

De esto se desprende que la operación a realizar es:
2005/12/28 +
0000/02/27

Ahora se suman los días: 28 + 27 = 55, como se obtiene un número mayor a 30 se lo separa en meses y días: 1 mes y 25, se registran los días y se acarrea 1 para sumarlo con meses.

Se suman los meses: 12 + 2 + 1 = 15, al obtener un número mayor a 12 se lo discrimina en años y meses: 1 año y 3 meses, se registra los 3 meses y se acarrea 1 para sumarlo con años.

Se suman los años: 2005 + 0 + 1 = 2006.

Entonces la solución es:

2005/12/28 +
0000/02/27

2006/03/25

Estas operaciones se deben expresar en forma algorítmica y colocarse en una función que puede ser invocada cuantas veces sea necesario. Algo muy importante a tener en cuenta es que por definición una función devuelve un valor y en este caso se requiere devolver tres (año, meses y días). La solución consiste en agrupar los datos mediante un vector, donde la primera posición corresponderá al año, las segunda a los meses y la tercera a los días, luego se establece el vector como parámetro y de igual

manera la función devolverá un vector como resultado. El pseudocódigo de esta función se presenta en el cuadro 98.

Cuadro 98. Pseudocódigo de la función sumar días a fecha

```
1   Entero[] sumardiasafecha(Entero f[], Entero días)
2       Entero aux[3], nf[3]
3       aux[1] = días / 365
4       días = días Mod 365
5       aux[2] = días / 30
6       aux[3] = días Mod 30
7       nf[3] = f[3]+aux[3]
8       nf[2] = f[2]+aux[2]
9       nf[1] = f[1]+aux[1]
10      Si nf[3] > 30 entonces
11          nf[3] = nf[3]-30
12          nf[2] = nf[2] + 1
13      Fin si
14      Si nf[2] > 12 entonces
15          nf[2] = nf[2]-12
16          nf[1] = nf[1] + 1
17      Fin si
18      Retornar nf[]
19  Fin sumardiasafecha
```

Esta función recibe un vector de tres número enteros y una variable como parámetros, internamente declara dos vectores, un auxiliar para discriminar en años, meses y días el contenido de la variable recibida y el segundo para guardar la fecha que se genera como resultado. Inicialmente se suman directamente las posiciones de los vectores, luego se verifica la validez de los días y meses, y si éstos no son válidos se hace el ajuste correspondiente.

En el cuadro 99 se evidencia el comportamiento de los vectores al ejecutar la función.

Cuadro 99. Verificación de la función sumar días a fecha

Vector f[]			dias	Vector aux[]			Vector nf[]		
f[1]	f[2]	f[3]		aux[1]	aux[2]	aux[3]	nf[1]	nf[2]	nf[3]
2005	12	28	87	0	2	27	2005	14	55
								15	25
							2006	3	25

6.1.4 Ejercicios propuestos

Diseñar funciones para los siguientes planteamientos

1. Conocer la nota mínima que un estudiante debe obtener en el examen final de una materia para aprobarla con nota mínima (3.0), a partir de las notas obtenidas en dos pruebas parciales. Se sabe que cada parcial equivale al 30% y el examen final al 40% de la nota definitiva.

2. Calcular el perímetro de un rectángulo

3. Calcular el área de un círculo

4. Calcula la longitud de una circunferencia

5. Calcular el volumen de un cilindro

6. Determinar la cantidad de papel que se requiere para forrar completamente una caja

7. Establecer la velocidad de un atleta en k/h sabiendo que tardó x minutos en dar una vuelta al estadio cuyo recorrido tiene n metros.

8. Restar una cantidad de días a una fecha en formato año/mes/día

9. Sumar una cantidad de años, meses y días a una fecha

10. Obtener la diferencia entre dos fechas, expresada en años, meses y días.

11. Calcular el valor futuro de una inversión con interés compuesto

12. Determinar si un número es primo

13. Averiguar si un número n forma parte de la serie Fibonacci

14. Evaluar una fecha e informar si es válida o no. Se debe tener en cuenta los meses que tienen 28, 30 y 31 días y los años bisiestos.

15. Calcular el precio de venta de un producto, aplicando el 30% de utilidad sobre el costo y el 16% de impuesto al valor agregado (IVA).

16. Calcular el mínimo común múltiplo de dos números

17. Calcular la sumatoria de los primeros n números

18. Invertir el orden de los elementos de un vector

6.2. PROCEDIMIENTOS

Un procedimiento es un conjunto de líneas de código que se escriben por separado del algoritmo principal con el propósito de solicitar su ejecución desde diversas partes del algoritmo o programa, esto hace más organizado y entendible el código, más fácil de escribir, de corregir y de mantener.

Los procedimientos al igual que las funciones cumplen una tarea específica, pero a diferencia de éstas, no están diseñados para realizar cálculos y por tanto no devuelven ningún valor

El diseño de un procedimiento incluye su definición y su implementación, en la definición se le asigna un nombre y se especifican los parámetros que recibirá, en la implementación se escriben las instrucciones detalladas que le permitirán cumplir su tarea.

La forma general de diseñar un procedimiento es:

```
Nombre_procedimiento(parámetros)
    Instrucciones
Fin nombre_procedimiento
```

Ejemplos:

```
Imprimir(entero a, entero b, entero c)
    Escribir a
    Escribir b
    Escribir c
Fin imprimir
```

Si en el cuerpo del procedimiento se declaran variables, éstas tienen ámbito local.

Para invocar un procedimiento se escribe su nombre y la lista de parámetros, teniendo en cuenta que éstos deben corresponder en número y tipo con la lista de parámetros de la definición del procedimiento. En el momento en que la ejecución de un algoritmo encuentra la invocación de un procedimiento pasa a ejecutar las instrucciones de éste y cuando termina vuelve a la línea que sigue a la invocación.

Ejemplo 64. Procedimiento tabla de multiplicar

Se requiere un algoritmo para generar tablas de multiplicar para el número que el usuario ingrese, la ejecución termina cuando se ingresa el 0.

En este ejercicio se pueden observar dos tareas: por una parte, administrar las entradas, esto es, leer el número y decidir si se genera la tabla o se termina la ejecución del algoritmo, y repetir esta operación hasta que se ingrese el 0; por otra, generar la tabla de multiplicar para el número ingresado. La segunda tarea puede ser delegada a un procedimiento

Este procedimiento recibe como parámetro el número para el que se requiere la tabla y se encarga de llevar a cabo las iteraciones y mostrar el producto en cada una de ellas. El pseudocódigo de este procedimiento se muestra en el cuadro 100.

Cuadro 100. Procedimiento tabla de multiplicar

```
1   Tablamultiplicar(Entero n)
2       Entero i
3       Para i = 1 hasta 10 hacer
4           Escribir n, "*", i, " =", n*i
5       Fin para
6   Fin tablamultiplicar
```

El algoritmo principal, desde dónde se invoca este procedimiento, se presenta en el cuadro 101. El llamado al procedimiento se hace desde la línea 5.

Cuadro 101. Pseudocódigo del algoritmo para generar tablas de multiplicar

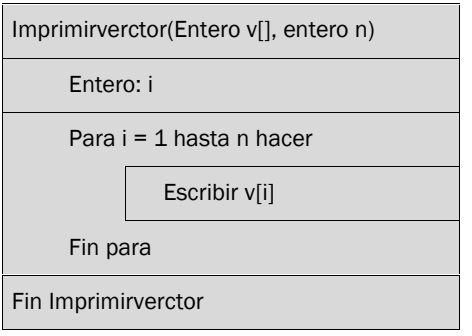
```
1   Inicio
2       Entero n
3       Hacer
4           Leer n
5           Tablamultiplicar(n)
6       Mientras n != 0 hacer
7   Fin algoritmo
```

Ejemplo 65. Procedimiento imprimir vector

Diseñar un procedimiento que se pueda invocar cada vez que se desee listar los datos de un vector.

Este procedimiento debe recibir como parámetros el vector y el número de elementos que contiene, hacer un recorrido y mostrar cada elemento. El diagrama N-S de este procedimiento se muestra en la figura 110.

Figura 110. Diagrama N-S para el procedimiento imprimir vector



Ejemplo 66. Procedimiento escribir fecha

Diseñar un procedimiento que reciba tres números enteros correspondientes a día, mes y año y muestre la fecha en forma de texto.

Este procedimiento debe averiguar el nombre del mes y luego anteponer y adicionar la proposición “de”. Por ejemplo, si recibe los datos: día = 11, mes = 2 y año = 2012 debe escribir: “11 de febrero de 2012”. El pseudocódigo de este procedimiento se presenta en el cuadro 102.

Cuadro 102. Pseudocódigo del procedimiento escribir fecha

```
1   Escribirfecha(entero d, entero m, entero a)
2       Cadena nmes
3       Según sea m hacer
4           1: nmes = “enero”
5           2: nmes = “febrero”
6           3: nmes = “marzo”
7           4: nmes = “abril”
8           5: nmes = “mayo”
9           6: nmes = “junio”
10          7: nmes = “julio”
11          8: nmes = “agosto”
12          9: nmes = “septiembre”
13          10: nmes = “octubre”
14          11: nmes = “noviembre”
15          12: nmes = “diciembre”
16      Fin según sea
17      Escribir d, “ de “, nmes, “ de “, a
18  Fin Escribirfecha
```

Ejemplo 67. Procedimiento mostrar una sucesión

Diseñar una procedimiento para generar los primeros n términos de la secuencia generada por la expresión: $x^2 - x$.

El procedimiento consiste en hacer un ciclo desde 1 hasta n y aplicar la expresión. Los resultados forman la secuencia:

0, 2, 6, 12, 20, 30, ...

El pseudocódigo se muestra en el cuadro 103.

Cuadro 103. Pseudocódigo del procedimiento mostrar sucesión

```
1   mostrarsecuencia(entero n)
2       Entero x, t
3       Para x = 1 hasta n hacer
4           t = x * x – x
5           Escribir x
6       Fin para
7   Fin mostrarsecuencia
```

7. BÚSQUEDA Y ORDENAMIENTO

El que no sabe
por qué camino llegar al mar
debe buscar
el río por compañero.
Plauto

Búsqueda y ordenamiento constituyen dos tareas fundamentales en la administración de datos, especialmente si se trata de grandes volúmenes. Las dos operaciones se desarrollan con base en un dato de referencia al que comúnmente se llama clave. La búsqueda permite encontrar un elemento particular en el conjunto, mientras que el ordenamiento consiste en ubicar los datos atendiendo a un criterio de manera que sea más fácil encontrar el elemento que se requiere o identificar las relaciones entre los datos.

En este capítulo se estudian y explican detalladamente los algoritmos de búsqueda y ordenamiento, apoyados con ejemplos y gráficas que facilitan su comprensión.

7.1 ALGORITMOS DE BÚSQUEDA

La búsqueda es una operación de vital importancia cuando se manipulan grandes conjuntos de datos donde localizar un elemento no es tarea fácil, como afirman Lopez, Jeder y Vega (2009: 129).

Para buscar un dato en un vector existen dos métodos: búsqueda secuencial o lineal y búsqueda binaria. El primero es más

fácil de implementar pero puede tomar más tiempo, el segundo es más eficiente, pero requiere que el vector esté ordenado.

7.1.1. Búsqueda Lineal

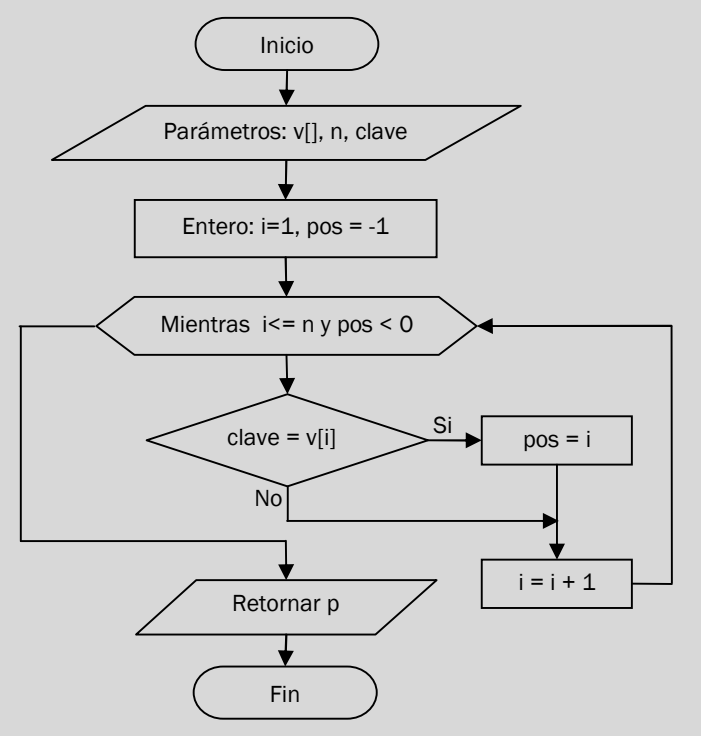
Este método consiste en tomar un dato clave que identifica al elemento que se busca y hacer un recorrido a través de todo el arreglo comparando el dato de referencia con el dato de cada posición.

Supóngase que se tienen una lista de estudiantes y se desea ubicar al que se identifica con el número 27844562. La búsqueda consiste en comparar, dicho número con la identificación de cada estudiante de la lista. La búsqueda terminará en el evento de encontrar una coincidencia en los números o si al llegar al final de la lista no se encontró identificación igual al número buscado, en cuyo caso se concluye que el dato no existe en el vector.

En la figura 111 se presenta el diagrama del flujo de una función con los pasos generales para hacer una búsqueda secuencial en un vector.

La función recibe como parámetros el vector y la clave del elemento a buscar, recorre el vector hasta que se encuentre el elemento, en cuyo caso la posición se registra en la variable *pos*, o hasta que se llegue al final del vector. Al finalizar el algoritmo devuelve la variable *pos*, si el dato fue encontrado ésta contiene la posición que ocupa en el vector, en caso contrario, reporta el valor menos uno (-1) indicando que el dato no está en el vector.

Figura 111. Diagrama de flujo del algoritmo de búsqueda lineal

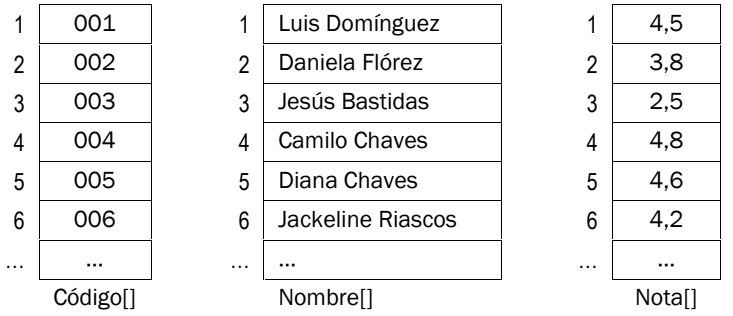


7.1.2. Ejemplos de búsqueda lineal

Ejemplo 68. Buscar un estudiante

Un profesor guarda los datos de sus estudiantes en tres vectores: código, nombre y nota, como se muestra en la figura 112. Se requiere un algoritmo para consultar la nota de un estudiante a partir de su código.

Figura 112. Datos de estudiantes almacenados en vectores



Haciendo uso de la función *búsquedalineal()* y pasándole como parámetros el vector *código*, el tamaño del mismo y el código del estudiante, se puede conocer en qué posición de los vectores se encuentran los datos del estudiante. El algoritmo se presenta en la figura 113, en éste se supone la existencia de un procedimiento para llenar los datos en los vectores.

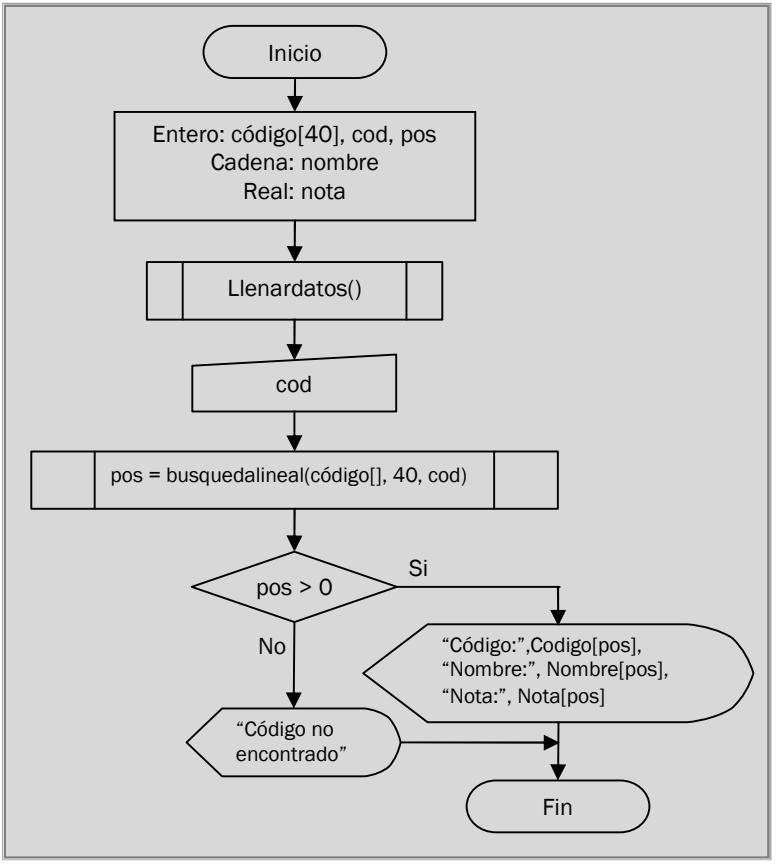
Ejemplo 69. Consultar saldo

En un hipotético cajero electrónico, cuando un cliente solicita el saldo de su cuenta desliza la tarjeta, el lector toma el número de cuenta y el sistema busca dicho número en la base de datos. Si lo encuentra reporta el saldo, si no muestra un mensaje de error. Suponiendo que la base de datos consiste en tres vectores: *número de cuenta, titular* y *saldo*. Diseñar un algoritmo para consultar el saldo de una cuenta.

En este algoritmo se lee el número de la cuenta y se invoca la función *busquedalineal()* proporcionándole el vector con los números de cuenta y el dato leído, esta función hace la búsqueda y retorna la posición dónde se encuentran los datos correspondientes a dicha cuenta. Si el dato devuelto por la función es un entero positivo se muestran los datos de todos los vectores, si es -1 significa que la cuenta no existe y se muestra un mensaje

informando de esta situación. El pseudocódigo de este algoritmo se presenta en el cuadro 104

Figura 113. Diagrama de flujo del algoritmo de buscar estudiante



Cuadro 104. Pseudocódigo del algoritmo consultar saldo

```
1   Inicio
        Entero num, pos, cuenta[1000]
2       Cadena: titular[1000]
        Real: saldo[1000]
3       Llenardatos()
4       Leer num
5       pos = busquedalineal(cuenta[], 10000, num)
6       Si pos > 0 entonces
7           Escribir "Cuenta:", cuenta[pos]
8           Escribir "Titular: ",titular[pos],
9           Escribir "Saldo: ",saldo[pos]
10      Si no
11          Escribir "Número de cuenta no encontrado"
12      Fin si
13  Fin algoritmo
```

7.1.3. Búsqueda Binaria

Este método es más eficiente que la búsqueda secuencial pero sólo se puede aplicar sobre vectores o listas de datos ordenados, como lo confirman López, Jeder y Vega (2009: 130)

En la búsqueda binaria no se hace un recorrido de principio a fin, sino que se delimita progresivamente el espacio de búsqueda hasta llegar al elemento buscado. La primera comparación se hace con el elemento de la mitad del arreglo, si aquel no es el dato buscado, se decide si buscar en la mitad inferior o en la mitad superior según la clave sea menor o mayor del elemento de la mitad. Se toma como espacio de búsqueda la mitad del vector que corresponda y se procede de igual forma, se compara con el elemento del centro, si ese no es el que se busca, se toma un nuevo espacio de búsqueda correspondiente a la mitad inferior o superior del espacio anterior, se compara nuevamente con el elemento del centro, y así

sucesivamente hasta que se encuentre el elemento o el espacio de búsqueda se haya reducido un elemento.

En la búsqueda binaria el número de elementos que conforman el campo de búsqueda se reduce a la mitad en cada iteración, siendo:

n para la primera iteración (n = tamaño del vector),
$n/2$ para la segunda,
$n/2^2$ en la tercera y
$n/2^3$ para la cuarta.

En general, el espacio de búsqueda en la i-ésima iteración está conformado por $n/2^{i-1}$ elementos.

Supóngase un vector v de 20 elementos ordenado de forma ascendente, como se muestra en la figura 114, en el que se busca el número 10. (clave = 10).

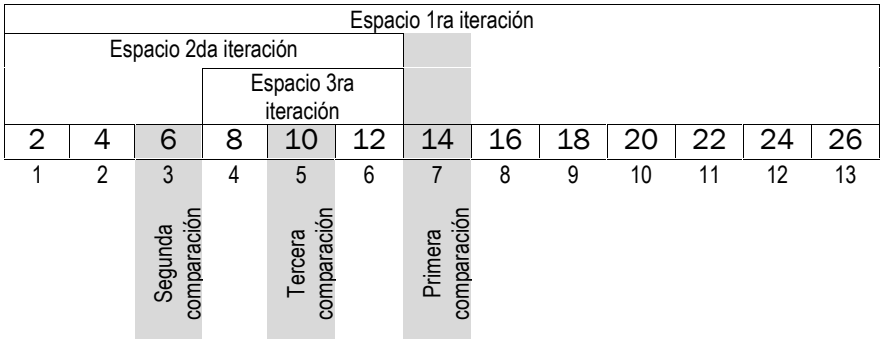
La primera iteración toma como espacio de búsqueda el vector completo y se ubica en el elemento de la mitad. Para determinar cuál es elemento de la mitad se suma el índice del extremo inferior con el índice del extremo superior del espacio de búsqueda y se divide para dos.

$(1 + 13) / 2 = 7$

Se averigua si la clave está en el elemento del centro:

$clave = v[7]?$
$10 = 14?$

Figura 114. Diagrama de flujo del algoritmo de búsqueda lineal



Como el dato no está en posición 7 se define un nuevo espacio de búsqueda correspondiente a la mitad inferior del vector, dado que la clave (calve = 10) es menor que el elemento del centro *(v[7] = 14)*. El espacio para la segunda comparación está entre las posiciones 1 y 6.

Se calcula nuevamente el centro:

(1 + 6) / 2 = 3

Ahora el elemento central es el de la posición 3, se averigua si ese es el dato que se busca:

clave = v[3]?
10 = 6?

Como en esta posición tampoco está el dato buscado se hace una nueva búsqueda, esta vez se toma la mitad superior del espacio, ya que la clave es mayor que el dato del centro: 10 > 6.

El nuevo espacio de búsqueda comprende los elementos entre las posiciones 4 y 6. Se calcula nuevamente el elemento del centro:

(4 + 6) / 2 = 5

Se compara la clave con el elemento de la posición central

clave = v[5]?
10 = 10?

Los valores son iguales, en esta iteración el dato fue encontrado. La búsqueda termina. Fueron necesarias 3 comparaciones para encontrar el elemento.

La función para realizar la búsqueda binaria recibe como parámetros el vector con los datos, el tamaño del vector y el dato que se busca o clave de búsqueda. El pseudocódigo de ésta se muestra en el cuadro 105.

Cuadro 105. Pseudocódigo de la función búsqueda binaria

```
1   Entero busquedabinaria(entero v[], entero n, entero clave)
2       Entero inf = 1, sup = n, centro, pos = -1
3       Mientras inf <= sup y pos < 0 hacer
4           centro = (inf + sup) / 2
5           Si v[centro] = clave entonces
6               pos = centro
7           Si no
8               Si clave > v[centro] entonces
9                   inf = centro + 1
10              Si no
11                  sup = centro – 1
12              Fin si
13          Fin si
14      Fin mientras
15      Retornar pos
16  Fin busquedabinaria
```

La función declara cuatro variables locales, dos para mantener los límites del espacio de búsqueda: *inf* y *sup*, estas comienzan en 1

y n respectivamente, lo que permite que la primera iteración se aplique sobre todo el vector, la variable centro se actualiza en cada iteración y su propósito es identificar el dato ubicado en el centro del espacio de búsqueda para compararlo con la clave; finalmente, la variable *pos* se destina a almacenar la posición del elemento en caso de hallarlo, se inicializa en -1 lo que indica que el dato no ha sido encontrado. El ciclo tiene dos condiciones: que el límite inferior sea menor que el superior, es decir que haya un espacio dónde hacer la búsqueda y que el dato no haya sido encontrado. En el momento en que el dato se localiza la variable *pos* toma un valor positivo y el ciclo deja de ejecutarse.

Al terminar el ciclo se devuelve la variable *pos*, si tiene un valor positivo, éste corresponde a la posición dónde se encuentra el dato buscado, pero si contiene -1 significa que el dato no está en el vector.

7.1.4. Ejemplos de búsqueda binaria

Ejemplo 70. Número ganador

La lotería de Mundo Rico registra los números de los billetes vendidos y el lugar dónde se vendió, en orden ascendente. En el momento de jugar la lotería, al conocer el número ganador se requiere conocer inmediatamente en qué lugar fue vendido. Diseñar un algoritmo para realizar esta consulta.

Los datos se guardan en una estructura conformada por dos vectores como se muestra en la figura 115. Dado que los números están ordenados se puede aplicar el algoritmo de búsqueda binaria, éste encontrará el número mucho más rápido que si se utilizara la búsqueda lineal.

Figura 115. Número de billetes de lotería y lugar de venta

Las operaciones a realizar en este ejercicio son: leer el número ganador, pasarlo a la función de búsqueda y mostrar los resultados. La función proporciona la posición dónde se encuentra el dato o en su defecto informará que éste no se encuentra, lo que se interpreta como número no vendido. La lectura de datos y la impresión de resultados son acciones que debe realizar el algoritmo principal. Ver cuadro 106.

Cuadro 106. Pseudocódigo del algoritmo número ganador

```
1   Inicio
2     Entero numero[1000], lugar[1000], ganador, pos
3     Llenardatos()
4     Leer ganador
5     pos = busquedabinaria(numero[], 1000, ganador)
6     Si pos> 0 entonces
7         Escribir “El numero: “, ganador, “ fue vendido en: “, lugar[pos]
8     Si no
9         Escribir “El número no fue vendido”
10    Fin si
11  Fin algoritmo
```

En la línea 3 del algoritmo se invoca un procedimiento para llenar los datos, lo que garantiza que los vectores contienen datos, no obstante la implementación de éste no se incluye. En la línea 5 se invoca a la función busquedabinaria() enviando como parámetros el vector que contiene los número, el tamaño del vector y el número ganador.

Ejemplo 71. Historia clínica

En la clínica Curatodo se cuenta con un índice de historias clínicas mediante dos vectores, en el primero se registra el número de historia, en el segundo la identificación del paciente. Se requiere un algoritmo que lea la identificación del paciente y reporte el número de la historia.

Como se ha visto en los ejemplos anteriores, tanto de búsqueda secuencial como binaria, una vez que cuenta con la función ésta se puede utilizar para cualquier vector. En este caso, el algoritmo consiste básicamente en leer el número de identificación del paciente, invocar la función y mostrar los datos del vector en la posición reportada por la función o en su defecto un mensaje de "identificación no encontrada". El algoritmo se presenta en el cuadro 107

Cuadro 107. Pseudocódigo del algoritmo historia clínica

```
1   Inicio
2     Entero identificación[1000], historia[1000], numcc, pos
3     Llenardatos()
4     Leer numcc
5     pos = busquedabinaria(identificación[], 1000, numcc)
6     Si pos> 0 entonces
7         Escribir "La historia del paciente es: ", historia[pos]
8     Si no
9         Escribir "No se encontró historia para el paciente"
10    Fin si
11  Fin algoritmo
```

7.1.5. Ejercicios propuestos

1. Se cuenta con la información del directorio telefónico almacenada en una matriz de 3 columnas, en la primera está el apellido, en la segunda el nombre y en la tercera el número telefónico. Se requiere un algoritmo para buscar el número de teléfono de una persona tomando como clave su apellido y nombre.

2. La lista de libros existentes en una biblioteca se encuentra almacenada en una matriz de 5 columnas, en la primera está el autor, en la segunda el título, en la tercera la editorial, en la cuarta el año y en la última, la signatura. Se solicita diseñar el algoritmo y la función de búsqueda por título.

3. El día de elecciones, en cada mesa se cuenta con un listado de las personas que votarán en dicha mesa. Los datos están guardados en dos vectores, en el primero la cédula y en el segundo el nombre. Se requiere un algoritmo que ingrese la cédula de una persona e informe si puede sufragar en ese puesto de votación.

4. La nomina de la empresa BuenSalario está registrada en dos vectores y una matriz. En el primer vector se guarda el número de cédula y en el segundo el nombre, en la matriz se guardan los siguientes datos: sueldo básico, deducciones, neto a pagar. Se requiere un algoritmo que lea la cédula de un empleado y muestre los datos de su nómina.

5. En el almacén Todocaro se cuenta con los datos: número de factura, nombre del cliente, fecha de facturación y valor de la factura, almacenados en vectores. Se desea un algoritmo que lea el número de factura y muestre los demás datos.

6. La empresa de comunicaciones Línea Segura registra el origen, el destino, la fecha, la hora y la duración de todas las llamadas que hacen sus usuarios, para esto hace uso de

cinco vectores. Se requiere un algoritmo para saber si desde un teléfono determinado se ha efectuado alguna llamada a un destino y en caso afirmativo conocer la fecha, la hora y la duración de la llamada.

7. La compañía de aviación Vuelo Alto mantiene la información de todas sus rutas de vuelo en una base de datos con información como: lugares de origen y destino, horario, valor; para ello utiliza una matriz en la que cada columna está destinada a uno de los datos mencionados. Diseñar un algoritmo para averiguar si existe un vuelo entre un origen y un destino introducidos por el usuario y en caso de existir obtener información sobre el mismo.

7.2 ALGORITMOS DE ORDENAMIENTO

Los datos de un vector, una lista o un archivo están ordenados cuando cada elemento ocupa el lugar que le corresponde según su valor, un dato clave o un criterio (en este documento sólo se aborda el ordenamiento de vectores). Si cada elemento del vector diferente del primero es mayor que los anteriores, se dice que está ordenado ascendentemente, mientras que si cada elemento diferente del primero es menor que los anteriores, está ordenado de forma descendente.

En un pequeño conjunto de datos el orden en que estos se presenten puede ser irrelevante, pero cuando la cantidad aumenta el orden toma importancia. Qué tan útil podría ser un diccionario si no estuviera ordenado? O el directorio telefónico?

Baase y Van Gelder (2002: 150) mencionan que el ordenamiento de datos fue una de las principales preocupaciones de las ciencias de la computación, prueba de ello es la cantidad de métodos de ordenamiento que se han diseñado. En este capítulo se explican los algoritmos más utilizados para este propósito.

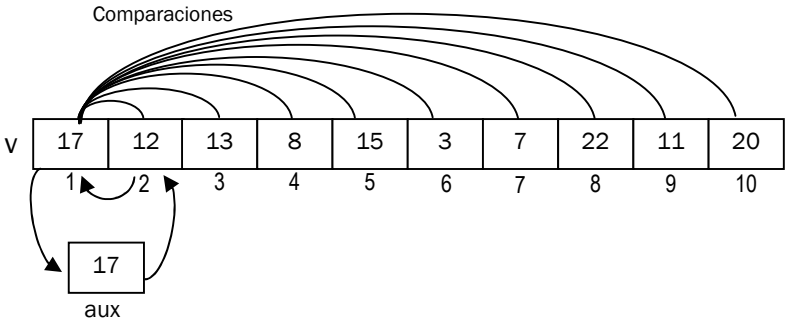
7.2.1 Algoritmo de intercambio

Este algoritmo no es el más eficiente, pero es muy didáctico y por ende es común su utilización en los primeros cursos programación. Consiste en tomar cada elemento y compararlo con los que están a su derecha, cada vez que se identifica un par de elementos que no cumplen el criterio de ordenamiento que se está aplicando se intercambian. En cada iteración se encuentra el elemento que corresponde a una posición.

Considérese el vector v de la figura 116, para ordenar los números de forma ascendente se toma el primer elemento *v[1]* y se compara con *v[2]*. Si *v[1]* menor que *v[2]* éstos están en orden,

pero si *v[1] > v[2]* están desordenados y es necesario intercambiarlos.

Figura 116. Algoritmo de intercambio comparaciones del primer elemento



Como se aprecia en la figura, para intercambiar dos elementos de un vector es necesario utilizar una variable auxiliar y realizar tres asignaciones:

aux = v[1]
v[1] = v[2]
v[2] = aux

Después de este intercambio se compara *v[1]* con *v[3],* si no están ordenados se procede a hacer el intercambio, luego *v[1]* con *v[4]* y así sucesivamente hasta *v[1]* con *v[n].* Explicaciones más detalladas de este proceso se encuentran en Joyanes (2000: 248) y en Chaves (2004: 235).

Para comparar el primer elemento del vector con todos los que le siguen y ubicar en la primera posición el valor que corresponde en el vector ordenado se ejecuta las instrucciones del cuadro 108.

Después de ejecutar el ciclo y encontrar el valor que corresponde a la primera posición del vector ordenado se procede a hacer lo mismo para la segunda posición, luego la tercera y así

sucesivamente hasta la penúltima. Dado que para cada posición es necesario un recorrido por los elementos de la derecha, es necesario utilizar dos ciclos anidados.

Cuadro 108. Ordenamiento por intercambio ubicación del primer elemento

```
1   Para j = 1 hasta n hacer
2       Si v[1] > v[j] entonces
3           aux = v[1]
4           v[1] = v[j]
5           v[j] = aux
6       Fin si
7   Fin para
```

En síntesis, el método de intercambio cosiste en implementar un recorrido para el vector donde para cada elemento se hace un segundo ciclo en el que se compara éste con todos los elementos que le siguen y cuando no estén ordenados se hace un intercambio. Cada iteración del ciclo externo ordena una posición del vector. En el cuadro 109 se presenta una función para ordenar un vector de enteros aplicando éste método. La función recibe como parámetro un vector de tipo entero no ordenado y el tamaño del mismo, aplica el algoritmo de ordenamiento y devuelve el vector ordenado.

El primer ciclo comienza en el primer elemento y termina en el penúltimo, el segundo ciclo comienza en el elemento que sigue al elemento a ordenar (determinado por la variable del primer ciclo) y termina en la última posición

Cuadro 109. Función ordenar vector por el método de intercambio

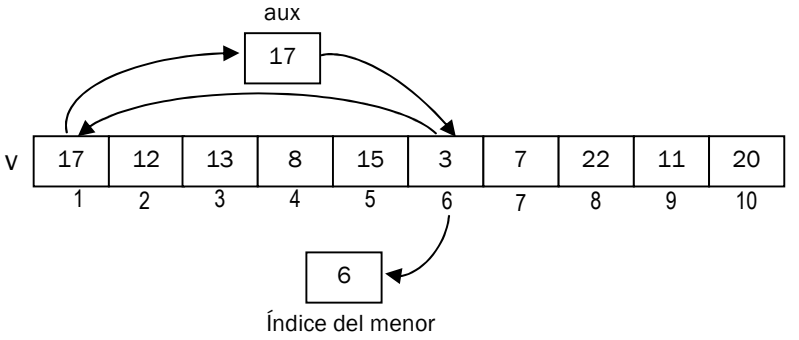
```
1   Entero[] intercambio(entero v[], Entero n)
2       Entero i, j, aux
3       Para i = 1 hasta n-1 hacer
4           Para j = i+1 hasta n hacer
5               Si v[i] > v[j] entonces
6                   aux = v[i]
7                   v[i] = v[j]
8                   v[j] = aux
9               Fin si
10          Fin para
11      Fin para
12      Retornar v[]
13  Fin intercambio
```

7.2.2 Algoritmo de Selección

Este método es similar al anterior en cuanto a los recorridos del vector y las comparaciones, pero con un número menor de intercambios. En el método de intercambio cada vez que se comparan dos posiciones y éstas no están ordenadas se hace el intercambio de los datos, de manera que en un mismo recorrido puede haber varios intercambios antes de que el número ocupe el lugar que le corresponde en el arreglo ordenado. En el método de selección, en cada recorrido se identifica el elemento que pertenece a una posición y se hace un solo intercambio.

Considérese el vector de la figura 117, donde se muestra el intercambio efectuado para la primera posición.

Figura 117. Algoritmo de selección – intercambio del primer elemento



Se declara una variable para guardar la posición del menor y se inicializa en 1 para comenzar las comparaciones en la primera posición, se hace un recorrido desde la segunda posición hasta la última y cada vez que se encuentra un elemento menor al que indica la variable se la actualiza.

Estando en la posición i del vector v, el recorrido para identificar el elemento que corresponde a dicha posición en el vector ordenado ascendentemente se lleva a cabo con las instrucciones que se presentan en el cuadro 110.

Cuadro 110. Recorrido para seleccionar el i-ésimo elemento

```
1    posmenor = i
2    Para j = i+1 hasta n hacer
3        Si v[posmenor] > v[j] entonces
4            posmenor = j
5        Fin si
6    Fin para
7    Aux = v[i]
8    v[i] = v[posmenor]
9    v[posmenor] = aux
```

Como se aprecia en el pseudocódigo del cuadro 110, el recorrido coloca el índice del menor elemento encontrado en la variable *posmenor* y al finalizar el recorrido se hace el intercambio entre la posición *i* y la posición que indica *posmenor*.

Ahora bien, para ordenar todo el vector solo hay que hacer que la variable *i* se mueva desde el primer elemento hasta el penúltimo, como se muestra en el cuadro 111, donde se presenta la función para ordenar un vector aplicando el algoritmo de selección. Esta función recibe un vector no ordenado como parámetro y lo devuelve ordenado.

Cuadro 111. Función ordenar vector por el método de selección

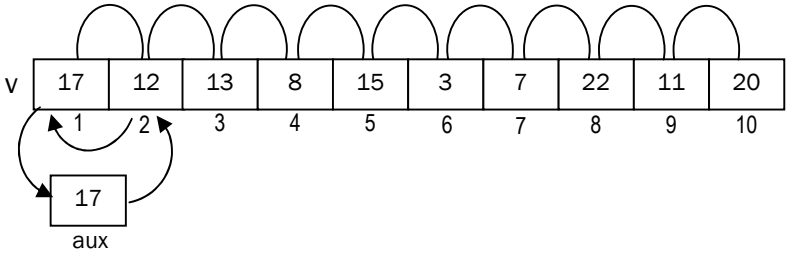
```
1   Entero[] seleccion(entero v[], entero n)
2       Entero: i, j, posmenor, aux
3       Para i = 1 hasta n-1 hacer
4       posmenor = i
5           Para j = i+1 hasta n hacer
6               Si v[posmenor] > v[j] entonces
7                   posmenor = j
8               Fin si
9           Fin para
10          aux = v[i]
11          v[i] = v[posmenor]
12          v[posmenor] = aux
13      Fin para
14      Retornar v[]
15  Fin selección
```

7.2.3 Algoritmo de la burbuja

Por la forma como se hacen los recorridos y las comparaciones éste es uno de los algoritmos de ordenamiento más fáciles de comprender y de programar, pero como explica Joyanes(2000: 252) no es recomendable su utilización en el desarrollo de software, por ser el de menor eficiencia.

La técnica de ordenamiento conocida como burbuja o burbujeo consiste en comparar el primer elemento con el segundo y si no cumplen el criterio de orden que se está aplicando se intercambian, acto seguido se pasa a comparar el segundo elemento con el tercero y si no están ordenados se intercambian también, luego se compara el tercero con el cuarto y después el cuarto con el quinto y así sucesivamente hasta comparar los elementos *v[n-1]* con *v[n]*. Como se ilustra en el vector de la figura 118, en un recorrido se hacen *n-1* comparaciones y todos los intercambios necesarios a que haya lugar.

Figura 118. Algoritmo de la burbuja – comparaciones del primer recorrido



En el primer recorrido del vector, dado que se comparan cada elemento con el adyacente, si se aplica un orden ascendente el valor más alto se desplazará paso a paso hacia la derecha llegando hasta la última posición, pero los demás valores quedarán en desorden en las demás posiciones. En el cuadro 112 se presentan las instrucciones para un recorrido.

Cuadro 112. Recorrido para burbujear un elemento

```
1    Para j = 1 hasta n-1 hacer
2        Si v[j] > v[j+1] entonces
3            aux = v[j]
4            v[j] = v[j+1]
5            v[j+1] = aux
6        Fin si
7    Fin para
```

Para ordenar completamente el vector, considerando que en una comparación se ordenan dos elementos y en un recorrido se lleva un elemento hasta su posición ordenada, es necesario realizar *n-1* recorridos.

Para mejorar el desempeño de este algoritmo se pueden reducir el número de comparaciones y en algunos casos la cantidad de recorridos.

Si se tienen en cuenta que cada recorrido toma el dato más grande y lo desplaza hacia la derecha ocurre que el primer recorrido lleva el dato más grande a la posición *n*, el segundo recorrido lleva el segundo dato más grande a la posición *n-1*, el tercero el que sigue a la posición *n-1*. De esto se desprende que en el segundo recorrido la comparación $v[n-1] > v[n]$ es inútil pues ya se sabe que en la última posición está el elemento más grande. También ocurre que en el tercer recorrido, la comparación $v[n-2] > v[n-1]$ y $v[n-1] > v[n]$ no aportan nada, pues las últimas posiciones se van ordenando en cada recorrido. En conclusión, todos los recorridos no necesitan llegar hasta *n-1*, el segundo va hasta *n-2*, el tercero hasta *n-3* y así sucesivamente. En un vector con muchos elementos el número de comparaciones que se reduce es importante. En el cuadro 113 se muestra el recorrido mejorado para la i-ésima iteración.

Cuadro 113. Recorrido mejorado para burbujear el i-ésimo elemento

```
1    Para j = i hasta n-i hacer
2        Si v[j] > v[j+1] entonces
3            aux = v[j]
4            v[j] = v[j+1]
5            v[j+1] = aux
6        Fin si
7    Fin para
```

Por otra parte puede ocurrir que un vector quede completamente ordenado en una cantidad de recorridos menor a n-1 y por ende los últimos recorridos sean innecesarios. Para evitar esta pérdida de tiempo se vigila que en cada recorrido hayan intercambios, si en el i-ésimo recorrido no tuvo lugar ninguno significa que el vector ya está ordenado y no es conveniente hacer los recorridos faltantes. Para ello se puede utilizar un contador de intercambios o un conmutador (bandera) que cambia de estado al efectuar un intercambio. En el cuadro 114 se presenta la función para ordenar un vector aplicando el algoritmo de la burbuja con las dos mejoras analizadas.

7.2.4 Algoritmo de Inserción

Este método consiste en ordenar los elementos del arreglo progresivamente, comenzando por los primeros y avanzando hasta ordenarlos todos. Dado el elemento de una posición i (i comienza en 2 y va hasta el fin del arreglo) se guarda en una variable auxiliar y ésta se compara con el elemento que está a la izquierda; si no están ordenados, el elemento de la izquierda se desplaza hacia la derecha. Se vuelve a comparar la variable auxiliar con el elemento que sigue por la izquierda. Todo elemento que no esté ordenado se desplaza una posición a la derecha, hasta que se encuentre uno que si esté en orden o se llegue al inicio del vector. Al terminar este

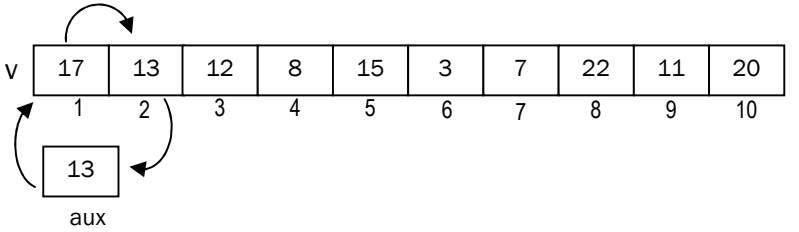
recorrido se deposita, en el espacio que quedó libre, el elemento contenido en la variable auxiliar.

Cuadro 114. Función ordenar vector por el método de la burbuja

```
1.  Entero[] burbuja(entero v[], entero n)
2.      Entero i = 1, j, aux
        Lógico: cambio = verdadero
3.      Mientras i < n-1 y cambio = verdadero hacer
4.          cambio = falso
5.          Para j = 1 hasta n-i hacer
6.              Si v[j] > v[j+1] entonces
7.                  aux = v[j]
8.                  v[j] = v[j+1]
9.                  v[j+1] = aux
10.                 cambio = verdadero
11.             Fin si
12.         Fin para
13.         i = i + 1
14.     Fin mientras
15.     Retornar v[]
16. Fin burbuja
```

Considérese el vector de la figura 119, para ordenarlo ascendentemente se copia a la variable auxiliar el contenido de la segunda posición, luego se compara con el primer elemento, como no están ordenados, el valor de la posición 1 se desplaza a la posición 2 y como no hay más elementos por la izquierda se coloca en la posición 1 el contenido de la variable auxiliar, de esta manera las dos primeras posiciones del vector ya están en orden: $v[1] = 13$ y $v[2] = 17$.

Figura 119. Algoritmo de inserción – intercambio para el segundo elemento



Ahora se toma el elemento de la tercera posición, se lo copia a la variable auxiliar y se compara con el elemento de la izquierda, como no están ordenados se desplaza el contenido de la posición dos a la posición tres. Se compara con el dato de la posición uno y se encuentra que este no está ordenado, se desplaza hacia la derecha. Como no hay más elementos por la izquierda se deposita en la posición uno el contenido de la variable auxiliar.

En resumen, se trata de tomar el i-ésimo elemento y ubicarlo en el lugar que le corresponde en el vector ordenado, para ello se libera el espacio que éste ocupa copiándolo a una variable auxiliar y se desplaza hacia la derecha los elementos mayores a éste, de manera que se abre un espacio justo en la posición que debe quedar. Las instrucciones del cuadro 115 mueven el i-ésimo elemento a la posición que le corresponde en el vector ordenado.

Para ordenar todo el vector solo hay que repetir las instrucciones del cuadro 115 para cada elemento desde la segunda hasta la última posición, como se muestra en el cuadro 116.

Cuadro 115. Instrucciones para insertar el i-ésimo elemento en la posición que le corresponde

```
1    j = i – 1
2    aux = v[i]
3    Mientras v[j] > aux y j > = 1 hacer
4        v[j+1] = v[j]
5        j = j -1
6    Fin mientras
7    v[j+1] = aux
```

Cuadro 116. Función ordenar vector por el método de inserción

```
1   Entero[] insercion(entero v[], entero n)
2       Entero: aux, i , j
3       Para i = 2 hasta n hacer
4           j = i – 1
5           aux = v[i]
6           Mientras v[j] > aux y j >= 1 hacer
7               v[j+1] = v[j]
8               j = j -1
9           Fin mientras
10          v[j+1] = aux
11      Fin para
12      Retornar v[]
13  Fin inserción
```

Este algoritmo puede mejorar su eficiencia cambiando el método de búsqueda que se aplican en la parte izquierda del vector. Teniendo en cuenta que el subvector de la izquierda de cada elemento está ordenado, se puede utilizar búsqueda binaria en vez de búsqueda secuencial, de esta forma se reduce el tiempo que el algoritmo tarda en encontrar el lugar que le corresponde al elemento. Esta reducción en el tiempo de búsqueda puede ser

significativa en vectores con muchos elementos. Hernández *et al* (2001: 53) expresan este cambio en los siguientes pasos:

a. Tomar un elemento en la posición i
b. Buscar de forma binaria su lugar en las posiciones anteriores
c. Mover los elementos restantes hacia la derecha
d. Insertar el elemento

7.2.5 Algoritmo de Donald Shell

Es un algoritmo de inserción con saltos decrecientes diseñado por Donald Shell* y reconocido generalmente por el nombre de su creador. Funciona de forma similar al algoritmo de inserción, pero a diferencia de éste no mueve los elementos una posición, sino varias posiciones a la vez, de manera que los elementos llegan más rápido a su destino. Este método es muy adecuado para vectores con gran cantidad de datos.

En este algoritmo, se toma un elemento y se lo compara con los que están a su izquierda, si se busca un orden ascendente, los elementos mayores al de referencia se mueven a la derecha y éste se ubica en el lugar que le corresponde, pero no se compara con el elemento que está inmediatamente a la izquierda, sino con el que se encuentra x posiciones atrás. A x se le llama salto, en la primera iteración se dan saltos de la mitad del tamaño del vector, de manera que se comparan los elementos de la mitad izquierda con los elementos de la derecha y cada intercambio implica pasar de un lado al otro.

* Donald Shell trabajó en la división de ingeniería de *General electric*, se doctoró en matemáticas en la Universidad de Cincinnati en 1959 y en ese mismo año publicó el algoritmo que hoy lleva su nombre con el título *A high-speed sorting procedure* en *Communications of the ACM.*

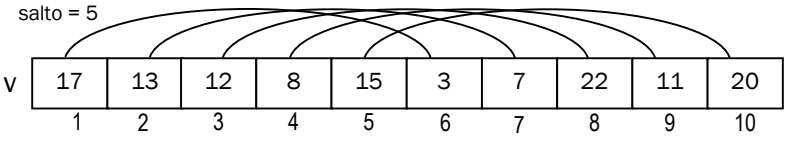
Para la segunda iteración el salto se reduce a la mitad y para la tercera nuevamente a la mitad y así sucesivamente hasta llegar a hacer comparaciones de uno en uno.

Para analizar el funcionamiento de este algoritmo considérese un vector de 10 elementos. Entonces:

Salto = 10/2 = 5

En este caso, en la primera iteración las comparaciones se hacen con una diferencia de 5 posiciones, *v[1]* con *v[6]*, *v[2]* con *v[7]* y así sucesivamente, como se muestra en la figura 120.

Figura 120. Algoritmo de Shell – primera iteración



En la primera iteración cada elemento del lado izquierdo sólo se compara con uno del lado derecho ya que el salto divide el vector en dos partes.

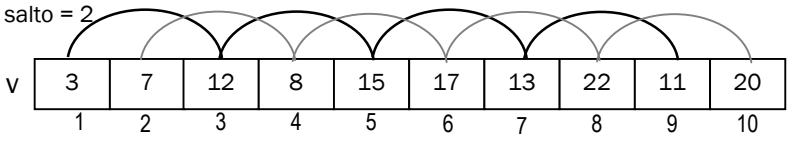
Comparación	Resultado	Intercambio
V[1] > v[6]	Verdadero	Si
V[2] > v[7]	Verdadero	Si
V[3] > v[8]	Falso	No
V[4] > v[9]	Falso	No
V[5] > v[10]	Falso	No

Para la segunda iteración se actualiza el salto tomando como valor la cuarta parte del tamaño del vector, lo que implica que desde algunos elementos se pueden dar varios saltos a la izquierda.

salto = salto/2
salto = 5/2 = 2

En la figura 121 se muestra el vector después de la primera iteración y las comparaciones que se harán en la segunda iteración.

Figura 121. Algoritmo de Shell – primera iteración



En la segunda iteración, en el ejemplo que se está analizando, salto = 2, en consecuencia el ciclo se ejecuta desde el tercer elemento hasta el último. La variable *i* se inicializa en 1 y la variable *aux* toma el tercer elemento (12). La primera comparación es entre *v[1]* y *v[3] (12 < 3 = falso)*, si los elementos están en orden la variable *seguir* toma el valor falso, el ciclo *mientras* terminar y se procede a realizar las iteraciones para el elemento siguiente (*v[4]*) y así sucesivamente.

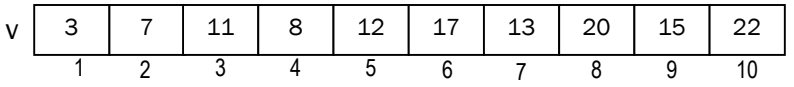
En el cuadro 117 se muestra el pseudocódigo para una iteración cualquiera. En este algoritmo, cada elemento se compara con los que están a su izquierda, el ciclo *mientras* se encarga del recorrido hacia la izquierda, siempre que hayan elementos y el dato que se pretende insertar sea menor que el indicado por el índice *i*. Si se encuentra que el elemento de la posición *i* es menor que el de *j* no es necesario seguir hacia la izquierda, pues éstos ya están en orden.

Cuadro 117. Algoritmo de Shell – una iteración

```
1   Para j = (1 + salto) hasta n hacer
2       i = j – salto
3       aux = v[j]
4       seguir = verdadero
5       Mientras i > 0 y seguir = verdadero hacer
6           Si aux < v[i] entonces
7               v[i + salto] = v[i]
8               i = i – salto
9           Si no
10              seguir = falso
11          Fin si
12      Fin mientras
13      v[i + salto] = aux
14  Fin para
```

En la figura 122 se muestra el orden de los elementos después de la segunda iteración con saltos de dos en dos.

Figura 122. Algoritmo de Shell – primera iteración



En este ejemplo, debido a que se utilizó un vector pequeño (10 elementos), para la tercera iteración los saltos son de uno en uno y después de ésta el vector está completamente ordenado. Arreglos con mayor cantidad de elementos requieren más iteraciones.

En el cuadro 118 se presenta una función para ordenar un vector aplicando el algoritmo de Shell.

Cuadro 118. Función ordenar vector por el método de Shell

```
1   Entero[] shell(entero v[], entero n)
2       Entero: aux, salto, i , j
        Lógico: seguir
3       salto = n/2
4       Mientras salto >= 1 hacer
5           Para j = (1 + salto) hasta n hacer
6               i = j – salto
7               aux = v[j]
8               seguir = verdadero
9               Mientras i > 0 y seguir = verdadero hacer
10                  Si aux < v[i] entonces
11                      v[i + salto] = v[i]
12                      i = i – salto
13                  Si no
14                      seguir = falso
15                  Fin si
16              Fin mientras
17              v[i + salto] = aux
18          Fin para
19          Salto = salto/2
20      Fin mientras
21      Retornar v[]
22  Fin Shell
```

7.2.6 Algoritmo de Ordenamiento Rápido

Este algoritmo es considerado el más rápido de los estudiados en este capítulo. Fue diseñado por Tony Hoare* y se basa en la técnica *dividir para vencer* de donde se desprende que ordenar dos listas pequeñas es más rápido y más fácil que ordenar una grande (Joyanes, 2000: 405).

El fundamento del algoritmo *quicksort* es la partición del vector o la lista en tres de menor tamaño distribuidas a la izquierda, al centro y a la derecha. La del centro contiene el valor utilizado como referencia para particionar el vector y por tanto un solo elemento.

En términos generales el algoritmo de ordenamiento rápido consta de las siguientes operaciones:

1. Seleccionar un elemento de referencia al que se denomina pivote
2. Recorrer el arreglo de izquierda a derecha buscando elementos mayores al pivote
3. Recorrer el arreglo de derecha a izquierda buscando elementos menores al pivote
4. Intercambiar cada elemento menor al pivote por uno mayor al pivote, de manera que los valores menores queden a la izquierda, los mayores a la derecha y el pivote al centro.
5. Realizar las mismas operaciones con cada una de las sublistas hasta que se tengan sublistas de un elemento.
6. Reconstruir la lista concatenando la sublista de la izquierda, el pivote y la sublista de la derecha.

Aunque este algoritmo puede desarrollarse de manera iterativa es mucho más fácil de comprender e implementar utilizando

* Su nombre completo es Charles Antony Richard Hoare, estudio en la Universidad de Oxford y en la Universidad Estatal de Moscú. Desarrolló el lenguaje algol 60 y diseño el algoritmo de ordenamiento rápido (Quicksort), trabajó en la Universidad de Queen y en el Laboratorio de Computación de la Universidad de Oxford.

recursividad, tema que se explica en el capítulo siguiente. En el cuadro 119 se presenta el pseudocódigo para particionar el vector.

Cuadro 119. Función para particionar un vector

```
1   Entero[] particionar(entero v[], entero inf, entero sup)
2       Entero: aux, pivote = v[inf], m = inf, n = sup+1
3       Mientras m <= n hacer
4           Hacer
5               m = m + 1
6           Mientras v[m] < pivote
7           Hacer
8               n = n – 1
9           Mientras v[n] > pivote
10          Si m < n entonces
11              Aux = v[m]
12              v[m] = v[n]
13              v[n] = aux
14          Fin si
15      Fin mientras
16      v[inf] = v[n]
17      v[n] = pivote
18      Invocación recursiva
19      Retornar v[]
20  Fin particionar
```

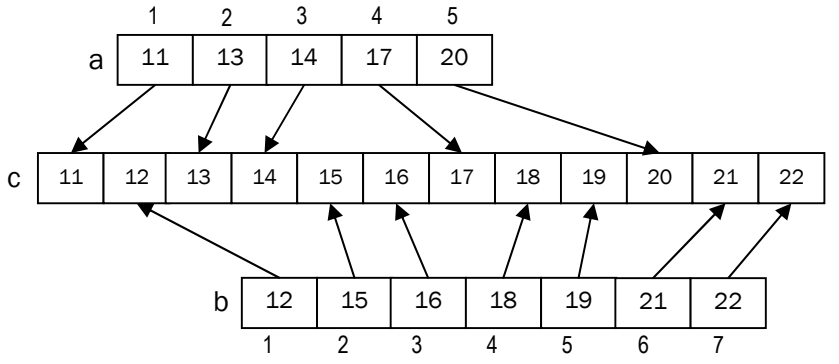
La versión recursiva de este algoritmo se explica detalladamente en la sección 8.7

7.2.7 Fusión de vectores ordenados

Esta operación recibe otros nombres como intercalación (Joyanes, 1996: 343) y mezcla (López et al, 2009: 135). Se trata de unir los elementos de dos vectores o listas ordenadas y formar un tercer vector o lista también ordenados, en donde se aprovecha el orden previo de las entradas para conseguir un conjunto ordenado en menor tiempo.

En la figura 123 se muestra dos vectores de diferente tamaño y su fusión en un tercer vector ordenado.

Figura 123. Fusión de vectores ordenados



El algoritmo para fusionar los dos vectores ordenados consiste en un ciclo para recorrer los dos vectores de entrada al tiempo y en cada iteración seleccionar el menor elemento para pasarlo al nuevo vector. En el arreglo que se toma el elemento se incrementa el índice mientras que en el otro permanece constante hasta que un elemento sea pasado al nuevo vector. El ciclo termina cuando alguno de los vectores haya sido copiado en su totalidad y los elementos restantes del otro vector se adicionan en el orden en que

están. En el cuadro 120 se presenta la función para realizar esta tarea.

Cuadro 120. Función para fusionar vectores ordenados

```
1   Entero[] fusionar(entero a[], entero b[], entero m, entero n)
2       Entero: c[m+n], i= 1, j = 1, k = 1, x
3       Mientras i <= m y j <= n hacer
4           Si a[i] <= b[j] entonces
5               c[k] = a[i]
6               i = i + 1
7           Si no
8               c[k] = b[j]
9               j = j + 1
10          Fin si
11          k = k + 1
12      Fin  mientras
13      Si i <= m entonces
14          Para  x = i hasta m hacer
15              c[k] = a[x]
16              k = k + 1
17          Fin para
18      Fin si
19      Si j <= n entonces
20          Para  x = j hasta n hacer
21              c[k] = b[x]
22              k = k + 1
23          Fin para
24      Fin si
25      Retornar c[]
26  Fin fusionar
```

7.2.8 Otros algoritmos de Ordenamiento

Además de los seis algoritmos de ordenamiento presentados en esta sección existen muchos más, algunos más eficientes que los aquí estudiados y por tanto más complejos, por lo que exigen conocimientos más avanzados en ciencias de la computación. A continuación se mencionan algunos pensando en los lectores que desean profundizar en este tema.

Mergesort: es un algoritmo recursivo que aplica el principio *divide y vencerás* de forma similar a *Quicksort*. Consiste en particionar el vector o la lista en dos partes iguales, se ordena recursivamente cada una de ellas y luego se unen mediante el proceso conocido como fusión de sucesiones ordenadas o mezcla de vectores ordenados.

Heapsort: este algoritmo combina las ventajas de Quicksort y *Mergesort*. Utiliza una estructura de datos denominada *heap* (montón) que es un árbol binario completo del que se eliminan algunas hojas de extrema derecha.

Ordenamiento por base: este método consiste en distribuir los elementos del conjunto a ordenar en varios subconjuntos atendiendo algún criterio, se ordena cada subconjunto aplicando cualquier método de ordenamiento y luego se vuelven a unir conservando el orden. Por ejemplo, supóngase que se tiene un conjunto grande de tarjetas de invitación y se desea organizarlas alfabéticamente por el nombre de su destinatario. Se puede formar 26 subconjuntos según la letra con que comienza el nombre, luego se ordena cada montón utilizando, por ejemplo, el método de inserción y finalmente se unen los grupos comenzando por el de la letra A.

7.2.9 Un ejemplo completo

Ejemplo 72. Lista de calificaciones

Un profesor maneja la lista de estudiantes y calificaciones mediante arreglos: en un vector de tipo entero guarda los códigos, en uno de tipo cadena, los nombres; y en una matriz de cuatro columnas mantiene las notas: tres parciales y la definitiva.

Entre las operaciones que el profesor realiza se tienen: ingresar los nombres de los estudiantes, ingresar las notas en el momento en que estas se generan, calcular la definitiva y listar las calificaciones; en ocasiones consulta las notas de un estudiante, en otras debe corregir alguna nota. El listado se presenta ordenado según alguno de estos criterios: orden alfabético por apellido u orden descendente por nota definitiva. Diseñar el algoritmo para realizar las tareas mencionadas.

Según la descripción del problema, la estructura de datos utilizada es la que se muestra en la figura 124.

Figura 124. Arreglos para guardas planilla de calificaciones

Vector códigos

1	101
2	102
3	102
4	104
5	105
6	106
…	…
33	133
34	134
35	-1
…	-1

Vector nombres

1	Salazar Carlos
2	Bolaños Carmen
3	Bermúdez Margarita
4	Paz Sebastián
5	Díaz Juan Carlos
6	Marroquín Verónica
…	…
33	Hernández María
34	Ordóñez Miguel
35	
…	

Matriz notas

	1	2	3	4
1	3,5	2,5	4,0	3,3
2	4,2	3,8	4,5	4,2
3	2,0	4,5	4,8	3,8
4	2,5	3,5	4,4	3,5
5	3,4	3,5	3,8	3,6
6	4,0	4,6	3,3	4,0
…	…	…	…	…
33	3,2	4,0	4,6	3,9
34	4,5	3,8	3,0	3,8
35				
…				
	Parcial 1	Parcial 2	Parcial 3	Definitiva

En este ejercicio se hace una breve descripción de los requisitos que debe satisfacer la aplicación que se pretende diseñar. Hasta el momento ya se sabe el qué hacer, ahora se debe pensar en el cómo. De eso se ocupan los párrafos siguientes.

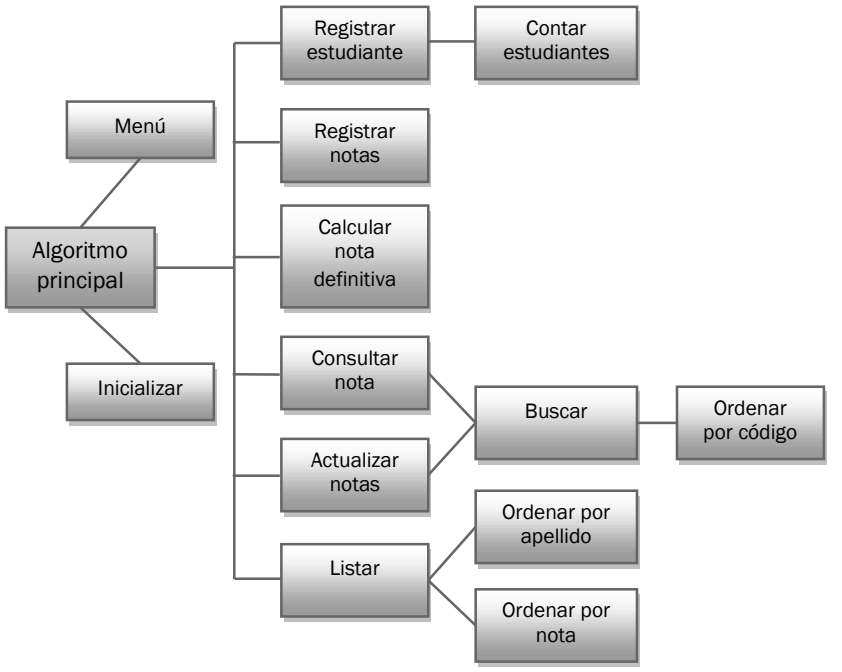
Este problema es muy complejo para intentar solucionarlo con un solo algoritmo, es necesario dividirlo en módulos o subrutinas que se puedan diseñar más fácilmente. ¿Cómo separar las tareas?

Una forma de hacerlo es a partir de las tareas que el usuario necesita que el programa realice, lo que técnicamente se conoce como requisitos, estos son: ingresar los estudiantes, por tanto se requiere un procedimiento o una función para ingresar estos datos; después de calificar los exámenes, el profesor debe registrar las notas, en consecuencia se requiere un procedimiento para ingresar calificaciones; otro procedimiento para consultar las calificaciones de un estudiante y otro para modificarlas en caso de ser necesario y finalmente; para generar los listados se requiere también un procedimiento y dado que los listados se presentan ordenados, también un procedimiento para ordenar los datos en los arreglos.

Una cuestión importante a tener en cuenta es que los algoritmos que se han estudiado operan sobre un número determinado de datos. En este ejercicio no se conoce el número de estudiantes que conforman el curso. La estrategia para solucionar casos en los que no se conoce la cantidad de espacio requerido es: analizar el dominio del problema y declarar los arreglos considerando el caso en que más espacio se requiera. Tratándose de una planilla de calificaciones, es probable que no sean más de 40 estudiantes, pero podría ocurrir que excepcionalmente hubiera más de 50, así que para evitar que alguno se quede sin registrar es mejor considerar el mayor espacio requerido y declarar los arreglos con dicho tamaño. Entonces, en este ejemplo, se reserva espacio para 65 estudiantes.

En la figura 125 se presenta un diagrama con los procedimientos y funciones a diseñar para resolver este ejercicio.

Figura 125. Diagrama de procedimientos y funciones



Ahora bien, dado que se reserva espacio suficiente para un grupo muy grande, en la mayoría de los casos el curso será más pequeño y parte de los arreglos quedará vacía. Esto implica que, en procura de la eficiencia de los algoritmos, se debe hacer los recorridos de los arreglos únicamente hasta donde hay datos. La estrategia aplicada en este caso es inicializar el vector de códigos con -1 en cada posición, de manera que al ingresar estudiantes este valor cambia por el código del estudiante y para saber cuántos registros se tienen basta con diseñar una función para contar los elementos con valor diferente de -1. En el cuadro 121 se presenta el procedimiento para inicializar el vector de códigos y en el cuadro 122 la función para conocer el número de estudiantes registrados.

Cuadro 121. Función para inicializar vector

```
1   inicializar()
2         Entero: i
3         Para i = 1 hasta 65 hacer
4             codigo[i] = -1
5         Fin  para
6    Fin inicializar
```

En este procedimiento, al igual que la función que sigue, no se definen parámetros, esto indica que el vector al que acceden se define como global en el algoritmo principal.

Cuadro 122. Función para contar estudiantes

```
1   Entero contarestudiantes()
2         Entero: i=0
3         Mientras código[i+1] != -1 hacer
4             i = i + 1
5         Fin  mientras
6         Retornar i
7    Fin contarestudiantes
```

Ahora se diseñan los procedimientos y funciones para implementar los requisitos de la aplicación. En el cuadro 123 se presenta el procedimiento para registrar los nombres de los estudiantes.

Cuadro 123. Función para registrar estudiantes

```
1   registrarestudiantes()
2       Entero: i
3       Caracter: rta
4       i = contarestudiantes()
5       Hacer
6           i = i + 1
7           Escribir "Código: "
8           Leer codigo[i]
9           Escribir "Nombre: "
10          Leer nombre[i]
11          Escribir "Mas estudiantes s/n?  "
12          Leer rta
13      Mientras rta = 'S' o rta = 's'
14  Fin registrarestudiantes
```

En el procedimiento registrar estudiantes se hace uso de la función *contarestudiantes()*, esto permite que se registren los nuevos estudiantes al final de la lista. En el cuadro 124 se presenta el procedimiento para registrar las notas. Éste recibe como parámetro el parcial al que pertenece la calificación (primero, segundo o tercero), presenta el nombre del estudiante, lee la nota y la guarda en la columna que corresponde.

Una vez se hayan registrado las tres notas se calcula la nota definitiva. El procedimiento para calcular la nota definitiva se presenta en el cuadro 125.

Cuadro 124. procedimiento para registrar calificaciones

```
1    registrarnotas (Entero: j)
2          Entero: i,n
3          n = contarestudiantes()
4          Para i = 1 hasta n hacer
5              Escribir "Código: ", código[i]
6              Escribir "Nombre: ", nombre[i]
7              Escribir "Nota ", j, ":"
8              Leer notas[i][j]
9          Fin para
10   Fin registrarnotas
```

Cuadro 125. Procedimiento para calcular nota definitiva

```
1    Calcularnotadefinitiva ()
2          Entero: i,n
3          n = contarestudiantes()
4          Para i = 1 hasta n hacer
5              Notas[i][4] = (notas[i][1] + notas[i][2] + notas[i][3])/3
6          Fin para
7    Fin calcularnotadefinitiva
```

Cuando ya se han ingresado los datos, hay tres operaciones que se pueden realizar: consultar, modificar y reportar. Para las tres operaciones es conveniente que los datos estén ordenados. Antes de proceder a escribir el procedimiento para consultar una nota se diseña uno para ordenar los vectores tomando como referencia el código del estudiante y aplicando el método de selección. Esto permitirá aplicar búsqueda binaria en las operaciones de consulta y modificación. Este se presenta en el cuadro 126.

Cuadro 126. Procedimiento ordenar vector por el método de selección

```
1   ordenarporcodigo()
        Entero i, j, posmenor, aux, n
2       Real: v[4]
        Cadena: nom
3       n = contarestudiantes()
4       Para i = 1 hasta n-1 hacer
5           posmenor = i
6           Para j = i+1 hasta n hacer
7               Si codigo[posmenor] > codigo[j] entonces
8                   posmenor = j
9               Fin si
10          Fin para
11          aux = codigo[i]
12          codigo[i] = codigo[posmenor]
13          codigo[posmenor] = aux
14          nom = nombre[i]
15          nombre[i] = nombre[posmenor]
16          nombre[posmenor] = nom
17          V[1] = notas[i][1]
18          V[2] = notas[i][2]
19          V[3] = notas[i][3]
20          V[4] = notas[i][4]
21          notas[i][1] = notas[posmenor][1]
22          notas[i][2] = notas[posmenor][2]
23          notas[i][3] = notas[posmenor][3]
24          notas[i][4] = notas[posmenor][4]
25          notas[posmenor][1] = v[1]
26          notas[posmenor][2] = v[2]
27          notas[posmenor][3] = v[3]
28          notas[posmenor][4] = v[4]
29      Fin para
30  Fin ordenarporcodigo
```

En este algoritmo, entre las líneas 11 y 28 se hace el intercambio de datos, en los dos vectores y en la matriz. Para mover la fila de la matriz se utiliza un vector de 4 elementos.

Para consultar las notas de un estudiante es necesario realizar una búsqueda con base en su código. Considerando que ya se desarrolló el procedimiento para ordenar los datos, se puede aplicar la búsqueda binaria sobre el vector de códigos. En el cuadro 127 se presenta el algoritmo de la función búsqueda binaria adaptado a la lógica que se está aplicando en este ejercicio, éste devuelve la posición que le corresponde al estudiante y con el índice se accede a los demás arreglos.

Cuadro 127. Función búsqueda binaria para código de estudiante

```
1   busquedabinaria(Entero cod)
2       Entero inf, sup, centro, pos = -1
3       Inf = 1
4       sup = contarestudiantes()
5       Ordenarporcodigo()
6       Mientras inf <= sup y pos < 0 hacer
7           centro = (inf + sup) / 2
8           Si v[centro] = cod entonces
9               pos = centro
10          Si no
11              Si cod > v[centro] entonces
12                  inf = centro + 1
13              Si no
14                  sup = centro – 1
15              Fin si
16          Fin si
17      Fin mientras
18      Retornar pos
19  Fin busquedabinaria
```

El algoritmos de la búsqueda binaria se explica detalladamente en la sección 7.1.2.

De esta manera, el procedimiento para consultar las notas de un estudiantes solo tiene que realizar tres tareas: leer el código, buscarlo y acceder a los datos. Este procedimiento se muestra en el cuadro 128.

Cuadro 128. Procedimiento consultar notas

```
1   consultar()
2       Entero: cod, pos
3       Escribir "Codigo del estudiante: "
4       Leer cod
5       pos = búsquedabinaria(cod)
6       Si pos > 0 entonces
7           Escribir "Codigo:", código[pos]
8           Escribir "Nombre:", nombre[pos]
9           Escribir "Nota 1:", notas[pos][1]
10          Escribir "Nota 2:", notas[pos][2]
11          Escribir "Nota 3:", notas[pos][3]
12          Escribir "Nota definitiva:", notas[pos][4]
13      Si no
14              Escribir "Codigo ", cod, "no encontrado"
15      Fin si
16  Fin consultar
```

En algunas ocasiones el docente necesitará cambiar alguna calificación, para ello se ofrece el procedimiento modificar, éste realiza las siguientes operaciones: lee el código del estudiante, lo busca, si lo encuentra muestra los datos, lee el nuevo dato y lo guarda en la columna que corresponda. El pseudocódigo de este procedimiento se muestra en el cuadro 129.

Cuadro 129. Procedimiento modificar notas

```
1   modificar()
2       Entero: cod, pos, opc
3       Escribir "Codigo del estudiante: "
4       Leer cod
5       pos = búsquedabinaria(cod)
6       Si pos > 0 entonces
7           Escribir "Codigo:", código[pos]
8           Escribir "Nombre:", nombre[pos]
9           Escribir "Nota 1:", notas[pos][1]
10          Escribir "Nota 2:", notas[pos][2]
11          Escribir "Nota 3:", notas[pos][3]
12          Escribir "Nota definitiva:", notas[pos][4]
13          Escribir "Ingrese la nota a modificar (1, 2 o 3): "
14          Leer opc
15          Según sea opc hacer
16              1: leer notas[pos][1]
17              2: leer notas[pos][2]
18              3: leer notas[pos][1]
19          Fin según sea
20          Notas[pos][4] = (Notas[pos][1]+ Notas[pos][2]+ Notas[pos][3])/3
21      Si no
22              Escribir "Código ", cod, "no encontrado"
23      Fin si
24  Fin modificar
```

Ahora se procede a diseñar los subalgoritmos para generar el reporte de calificaciones, éste puede estar ordenado alfabéticamente por nombre o descendentemente por la nota definitiva con el propósito de que aparezcan primero las notas más altas. Al ordenar los datos es preciso tener en cuenta que los intercambios se deben hacer en los tres arreglos. En el cuadro 130 se presenta el pseudocódigo para organizar los datos alfabéticamente aplicando el algoritmo de intercambio directo.

Cuadro 130. Procedimiento para ordenar los datos alfabéticamente por intercambio directo

```
1   ordenarporapellido()
        Entero i, j, aux,n, cod
2       Cadena: nom
        Real v[4]
3       n = contarestudiantes()
4       Para i = 1 hasta n-1 hacer
5           Para j = i+1 hasta n hacer
6               Si nombre[i] > nombre[j] entonces
7                   cod = codigo[i]
8                   codigo[i] = codigo[j]
9                   codigo[j] = cod
10                  nom = nombre[i]
11                  nombre[i] = nombre[j]
12                  nombre[j] = nom
13                  V[1] = notas[i][1]
14                  V[2] = notas[i][2]
15                  V[3] = notas[i][3]
16                  V[4] = notas[i][4]
17                  notas[i][1] = notas[j][1]
18                  notas[i][2] = notas[j][2]
19                  notas[i][3] = notas[j][3]
20                  notas[i][4] = notas[j][4]
21                  notas[j][1] = v[1]
22                  notas[j][2] = v[2]
23                  notas[j][3] = v[3]
24                  notas[j][4] = v[4]
25              Fin si
26          Fin para
27      Fin para
28  Fin ordenarporapellido
```

Para ordenar la planilla según la nota definitiva se adapta el algoritmo de Shell, como se aprecia en el cuadro 131.

Cuadro 131. Procedimiento para ordenar datos descendentemente aplicando algoritmo de Shell

```
1   Ordenarpornota()
2       Entero: aux, salto, i , j, cod, n
        Logito: seguir
        Cadena: nom
        Real: v[4]
3       n = contarestudiantes()
4       salto = n/2
5       Mientras salto >= 1 hacer
6           Para j = (1 + salto) hasta n hacer
7               i = j – salto
8               cod = codigo[j]
9               nom = nombre[j]
10              V[1] = notas[j][1]
11              V[2] = notas[j][2]
12              V[3] = notas[j][3]
13              V[4] = notas[j][4]
14              seguir = verdadero
15              Mientras i > 0 y seguir = verdadero hacer
16                  Si V[4] < notas[i][4] entonces
17                      codigo[i + salto] = codigo[i]
18                      nombre[i + salto] = nombre[i]
19                      notas[i + salto][1] = notas[i][1]
20                      notas[i + salto][2] = notas[i][2]
21                      notas[i + salto][3] = notas[i][3]
22                      notas[i + salto][4] = notas[i][4]
23                      i = i – salto
24                  Si no
25                      seguir = falso
26                  Fin si
```

Cuadro 131. (continuación)

```
27                  Fin mientras
28                  codigo[i + salto] = cod
29                  nombre[i + salto] = nom
30                  notas[i + salto][1] = v[1]
31                  notas[i + salto][2] = v[2]
32                  notas[i + salto][3] = v[3]
33                  notas[i + salto][4] = v[4]
34              Fin para
35              Salto = salto/2
36          Fin mientras
37      Fin ordenarpornota
```

Si se quisiera imponer un orden ascendente bastaría con cambiar el signo *menor que* por *mayor que* en la línea 16.

En el procedimiento *listar()* que se presenta en el cuadro 132, se solicita al usuario que seleccione en qué orden desea la lista, luego se invoca uno de los dos procedimientos de ordenamiento ya diseñados y posteriormente se genera el listado.

Cuadro 132. Procedimiento para listar estudiantes y calificaciones

```
1   Listar ()
2       Entero: orden, i, n
3       n = contarestudiantes()
4       Escribir "Generar listado ordenado por:"
5       Escribir "1. Nombre  2. Nota definitiva"
6       Leer orden
7       Si orden = 1 entonces
8           Ordenarporapellido()
9       Si no
```

Cuadro 132. (continuación)

```
10          Ordenarpornota()
11      Fin si
12      Escribir "código \t Nombre \t Nota1 \t Nota2 \t Nota3 \t Nota def."
13      Para i = 1 hasta n hacer
14          Escribir código[i], "\t", nombre[i], "\t", nota[i][1], "\t", nota[i][2],
                "\t", nota[i][3], "\t", nota[i][4]
15      Fin para
16  Fin listar
```

Continuando con el algoritmo de solución al problema planteado, ya se tienen varios procedimientos y algunas funciones para desarrollar las tareas específicas, ahora se requiere una función que interactúe con el usuario y reporte al algoritmo principal la tarea que él desea ejecutar. Para ello se diseña un menú y una función que lo muestra y captura la selección, el pseudocódigo de ésta se presenta en el cuadro 133.

Cuadro 133. Función menú

```
1   Entero menu()
2       Entero: opción
3       Escribir "Menu"
4       Escribir "1. Registrar estudiantes"
5       Escribir "2. Registrar calificaciones"
6       Escribir "3. Calcular nota definitiva"
7       Escribir "4. Consultar notas de un estudiante"
8       Escribir "5. Actualizar notas de un estudiante"
9       Escribir "6. Imprimir reporte"
10      Escribir "7. Terminar
11      Hacer
12          Leer opción
13      Mientras opción < 1 u opción > 7
14      Retornar opción
15  Fin menú
```

Finalmente, en el cuadro 134 se presenta el algoritmo principal, el que controla la ejecución de todos los procedimientos y funciones que se han diseñado previamente.

Cuadro 134. Algoritmo principal

```
1   Inicio
2       Global Entero: codigo[65]
        Entero opc, nnota
        Global Cadena: nombre[65]
        Global Real: notas[65][4]
3       Inicializar()
4       Hacer
5           opc = menú()
6           Según sea opc hacer
7               1: registrarestudiantes()
8               2: Escribir "Cuál nota desea ingresar. (1, 2 o 3)?"
9                  Leer nnota
10                 Registrarnotas(nnota)
11              3: calcularnotadefinitiva()
12              4: consultar()
13              5: modificar()
14              6: listar()
15          Fin según sea
16      Mientras opc != 7
17  Fin algoritmo
```

Así se concluye este pequeño ejercicio de diseño de un programa para solucionar un problema de baja complejidad como es el administrar las calificaciones de un curso. No obstante, ha permitido mostrar la aplicación de todos los conceptos estudiados a lo largo de los capítulos anteriores, incluyendo manejo de arreglos, búsqueda y ordenamiento de datos. De manera que puede ser tomado como referencia para desarrollar los ejercicios que se plantean a continuación.

7.2.10 Ejercicios propuestos

1. La biblioteca Lecturas Interesantes desea administrar sus materiales mediante un programa de computador y ha solicitado que se le diseñe un algoritmo para este propósito. Se desea almacenar los datos de los libros, como son: materia, titulo, editorial, autor y año. Se desea ingresar todos los libros y poder adicionar las nuevas adquisiciones en la medida en que se hacen; hacer consultas con base en autor, título o ISBN, de igual manera generar listados ordenados por autor o por título.

2. La empresa Buena Paga desea un algoritmo para procesar la información de su nómina. Se pretende almacenar en arreglos los datos: nombre del empleado, cargo, sueldo básico, días trabajados, deducciones y sueldo del mes. Entre las tareas a realizar están: ingreso de datos, modificación de datos, procesamiento de la nómina, consultas y generación de reportes.

3. Un algoritmo que mediante matrices mantenga los datos: nombre, dirección y teléfono de los profesores de la universidad y permita adicionar datos, modificar, listar en orden alfabético y buscar por nombre.

8. RECURSIVIDAD

El juicio de los hombres entendidos
descubre por las cosas claras las oscuras,
por las pequeñas las grandes,
por las próximas las remotas
y por la apariencia la realidad.
Séneca

Recursividad o recursión, como también se le llama, es un concepto que se aplica de manera especial en las matemáticas y en la programación de computadores; y de forma similar en numerosas situaciones de la vida cotidiana. En matemáticas se habla de *definición inductiva* para referirse a los numerosos métodos que hacen uso de la recursividad; mientras que en programación, este tema, se aplica a los algoritmos en el momento de la especificación, y a las funciones en la implementación (Joyanes, 1999).

La utilización del concepto de recursividad en los lenguajes de programación se la debe al profesor John McCarthy, del *Massachusetts Institute of Tenology* (MIT), quien recomendó su inclusión en el diseño de Algol60 y desarrolló el lenguaje Lisp, que introdujo estructuras de datos recursivas junto con procedimientos y funciones recursivas (Baase, 2002).

La recursividad es una alternativa a la utilización de estructuras iterativas. Un algoritmo recursivo ejecuta cálculos repetidas veces mediante llamados consecutivos a sí mismo. Esto no significa que los algoritmos recursivos sean más eficientes que los iterativos,

incluso, en algunos casos, los programas pueden requerir más tiempo y memoria para ejecutarse, pero este costo puede ser compensado por una solución más intuitiva y sustentada matemáticamente en una prueba por inducción.

La mayoría de los lenguajes de programación actuales permiten la implementación de algoritmos recursivos, mediante procedimientos o funciones. Para el caso de lenguajes que no permiten su implementación directa, ésta puede simularse utilizando estructuras de datos de tipo pila.

La recursividad es un concepto que se aplica indistintamente a algoritmos, procedimientos y programas. No obstante, en este libro se aborda desde la perspectiva del diseño de algoritmos.

8.1 LA RECURSIVIDAD Y EL DISEÑO DE ALGORITMOS

En el ámbito del diseño de algoritmo se define como recursividad la técnica de plantear la solución de un problema invocando consecutivamente el mismo algoritmo con un problema cada vez menos complejo, hasta llegar a una versión con una solución conocida. Por ejemplo, para calcular el factorial de un número.

Ejemplo 73. Función recursiva para calcular el factorial de un número

Se conoce como factorial de un número al producto de los enteros comprendidos entre 1 y *n* inclusive, y se representa mediante: *n!*

De manera que:

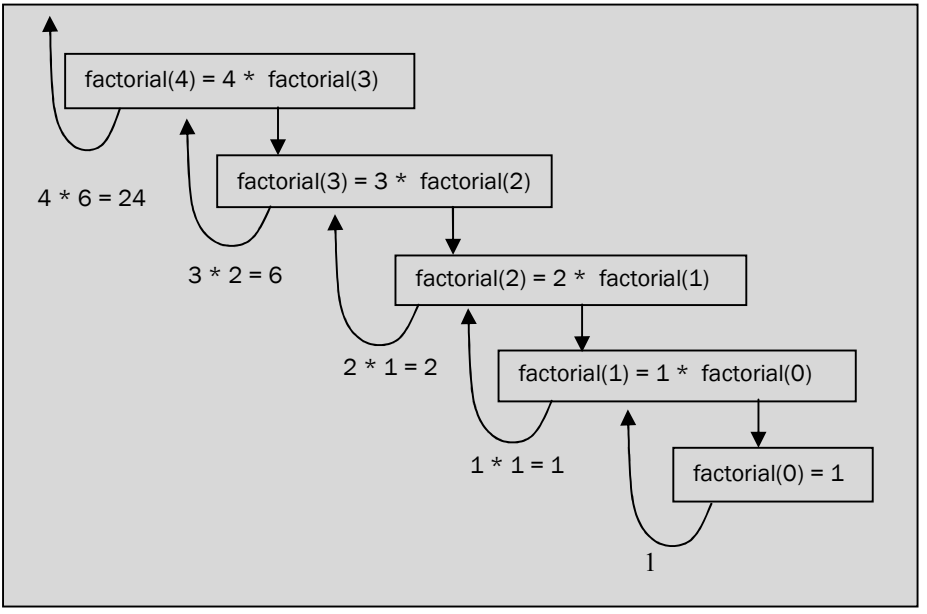
$$n! = 1 * 2 * 3 * 4 * \ldots * (n-2) * (n - 1) * n$$

También se sabe que el factorial está definido para el caso de que $n = 0$, cuyo resultado es 1. Es decir, $0! = 1$.

De esto se desprende que para calcular el factorial de 4 (4!) se calculará el producto de 4 por el factorial de 3 (3!); para calcular el factorial de 3 (3!), el producto de 3 por el factorial de 2 (2!); para calcular el factorial de 2 (2!), 2 por factorial de 1 (1!); para calcular factorial de 1 (1!), 1 por el factorial de 0. Ahora, como se conoce que el factorial de 0 es 1 (0! = 1), entonces el problema está resuelto. Más adelante se desarrolla completamente este ejercicio.

Como se aprecia en la figura 126, la recursividad consiste en escribir una función que entre sus líneas o sentencias incluyen un llamado a sí misma con valores diferentes para sus parámetros, de manera que progresivamente llevan a la solución definitiva.

Figura 126. Esquema de una función recursiva



La recursividad es otra forma de hacer que una parte del código se ejecute repetidas veces, pero sin implementar estructuras iterativas como: *mientras, para* o *hacer mientras.*

8.2 ESTRUCTURA DE UNA FUNCIÓN RECURSIVA

Se ha mencionado que consiste en implementar funciones que se invocan a sí mismas. Esto implica que la función debe tener una estructura tal que le permita invocarse a sí misma y ejecutarse tantas veces como se requiera para solucionar el problema, pero a la vez, sin hacer más invocaciones de las estrictamente necesarias, evitando que se genere una secuencia infinita de llamadas a sí misma. Cuando una función recursiva especifica en su definición cuándo autoinvocarse y cuándo dejar de hacerlo se dice que aquella está correctamente definida (Lipschutz, 1993)

Una función recursiva bien definida ha de cumplir las dos condiciones siguientes:

1. Debe existir un *criterio base* cuya solución se conoce y que no implica una llamada recursiva. En el caso de utilizar una función recursiva para calcular el factorial, el criterio base indica que factorial de 0 es 1 (0! = 1). Es decir, si la función recibe como parámetro el 0 ya no necesita invocarse nuevamente, simplemente retorna el valor correspondiente, en este caso el 1.

   factorial(n) = ? (requiere llamada a la función recursiva)
   factorial(0) = 1 (no requiere llamado a la función)

2. Cada vez que la función se invoque a sí misma, directa o indirectamente, debe hacerlo con un valor más cercano al *criterio base*. Dado que el criterio base es un caso particular del problema en el que se conoce la solución esto indica que con cada llamado a la función recursiva se estará acercando a la solución conocida. Continuando con el ejemplo de la función

factorial, se ha determinado que el *criterio base* es *0! = 1*, entonces, cada vez que se invoque la función deberá utilizarse como parámetro un valor más cercano a 0.

El factorial de un número es el producto del número por el factorial del número menor, hasta llegar al factorial de 0 que es constante. Con base en esto se puede afirmar que está definido para cualquier entero positivo, así:

0! = 1	
1! = 1 * 1 = 1	1! = 1 * 0!
2! = 2 * 1 = 2	2! = 2 * 1!
3! = 3 * 2 * 1 = 6	3! = 3 * 2!
4! = 4 * 3 * 2 * 1 = 24	4! = 4 * 3!
5! = 5 * 4 * 3 * 2 * 1 = 120	5! = 5 * 4!

Por lo tanto, la solución del factorial se puede expresar como:

a. Si $n = 0 \rightarrow n! = 1$
b. Si $n > 0 \rightarrow n! = n * (n - 1)!$

Y esta es la definición correcta de una función recursiva, donde *a* es el criterio base, es decir, la solución conocida, y *b* es la invocación recursiva con un argumento más cercano a la solución. Expresado en forma de función matemática se tiene:

Siendo *f(n)* la función para calcular el factorial de *n*, entonces:

$$f(n) = \begin{cases} 1 & \text{Si } n = 0 \\ n * f(n-1) & \text{Si } n > 0 \end{cases}$$

El pseudocódigo para esta función se presenta en el cuadro 135.

Cuadro 135. Función recursiva para calcular el factorial de un número

```
1.  Entero factorial(entero n)
2.      Si n = 0 entonces
3.          Retornar  1
4.      Si no
5.          Retornar (n * factorial(n – 1))
6.      Fin si
7.  Fin factorial
```

En la definición se puede apreciar claramente el cumplimiento de las dos condiciones de que trata esta sección. En las líneas 2 y 3 se examina el cumplimiento del *criterio base*, para el cual se conoce la solución (0! = 1).

```
2.      Si n = 0 entonces
3.          Retornar 1
```

Si aun no se ha llegado a dicho criterio, es necesario invocar nuevamente la función recursiva, pero con un valor menor, como se aprecia en las líneas 4 y 5.

```
4.  Si no
5.      Retornar (n * factorial(n – 1))
```

8.3 EJECUCIÓN DE FUNCIONES RECURSIVAS

La ejecución de una función recursiva requiere que dinámicamente se le asigne la memoria suficiente para almacenar sus datos. Por cada ejecución se reserva espacio para los parámetros, las variables locales, las variables temporales y el valor devuelto por la función. El código se ejecuta desde el principio con los nuevos datos. Cabe aclarar que no se hace copia del código recursivo en cada marco de activación. (Schildt, 1993).

El espacio en que se ejecuta cada invocación de una función recursiva se denomina *Marco de activación* (Baase, 2002). Este marco, además de proporcionar espacio para guardar las variables con que opera la función, también, proporciona espacio para otras necesidades contables, como la dirección de retorno, que indica la instrucción que se ha de ejecutar una vez salga de la función recursiva. Así, se crea un marco de referencia en el que la función se ejecuta únicamente durante una invocación.

Ahora bien, como la función será invocada un número indeterminado de veces (depende del valor de los parámetros) y por cada invocación se creará un marco de activación, el compilador deberá asignar una región de la memoria para la creación de la *pila de marcos*. A este espacio se hace referencia mediante un registro llamado *apuntador de marco*, de modo que mientras se ejecuta una invocación de la función, se conoce dónde están almacenadas las variables locales, los parámetros de entrada y el valor devuelto.

Una ejecución a mano que muestra los estados de los marcos de activación se denomina *rastreo de activación* (Baase, 2002) y permite analizar el tiempo de ejecución de la función y comprender como funciona realmente la recursividad en el computador.

Cada invocación activa tiene un marco de activación único. Una invocación de función está activa desde el momento en que se entra en ella hasta que se sale, después de haber resuelto el problema que se le paso como parámetro. Todas las activaciones de la función recursiva que están activas simultáneamente tienen marcos de activación distintos. Cuando se sale de una invocación de función su marco de activación se libera automáticamente para que otras invocaciones puedan hacer uso de ese espacio y la ejecución se reanuda en el punto donde se hizo la invocación de la función.

Para ilustrar mejor estas ideas, en la figura 127 se presenta el rastreo de activación para la función factorial.

Figura 127. Rastreo de activación para la función factorial

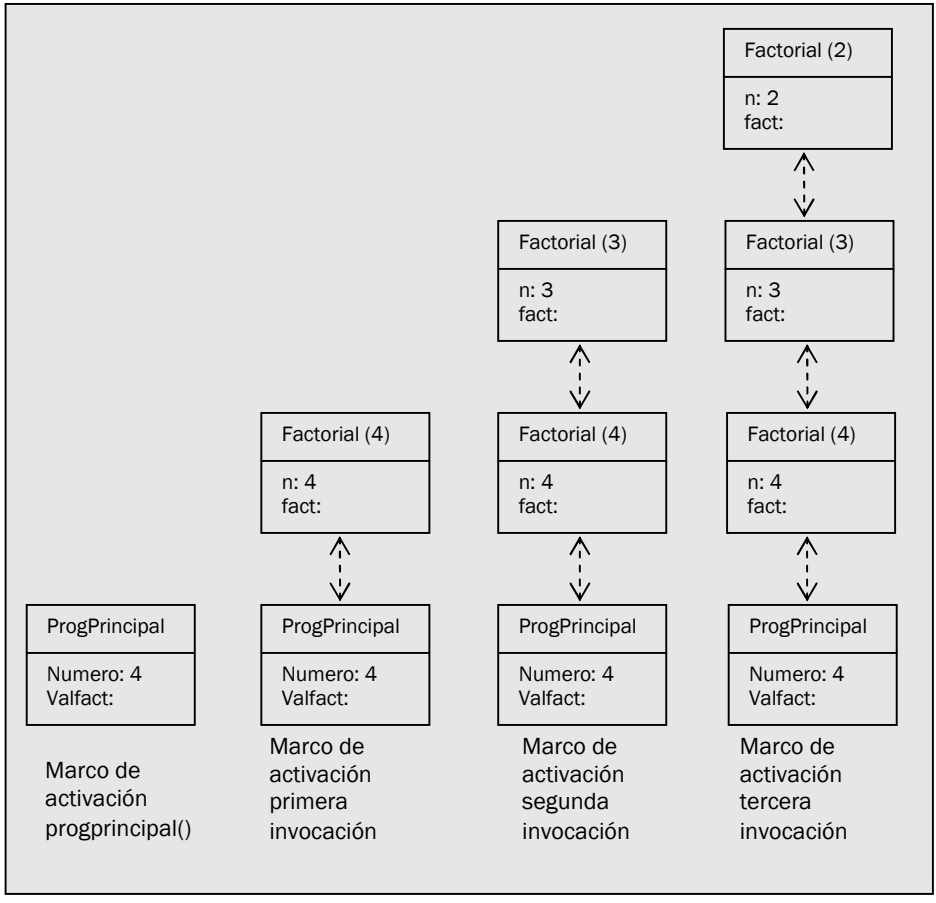
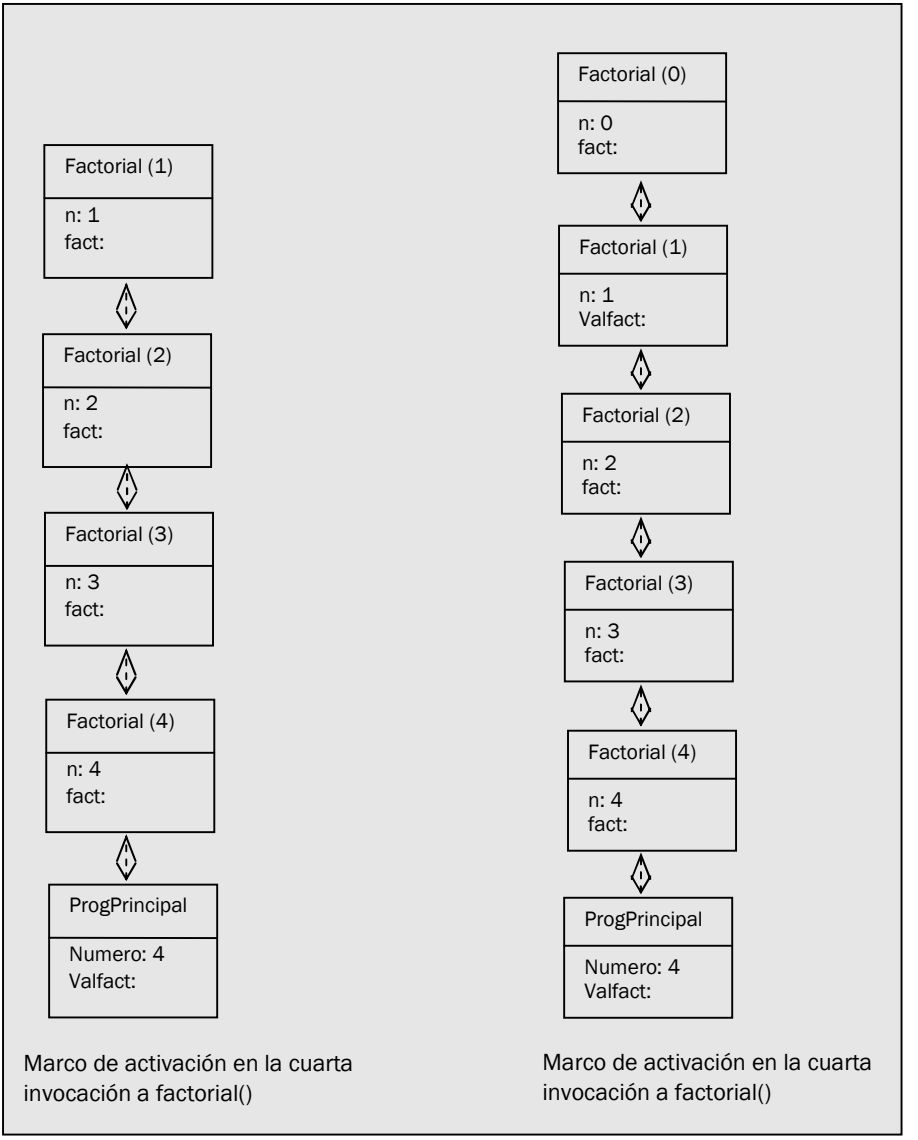


Figura 127. (Continuación)



Una vez que la función ha encontrado el criterio base, es decir, un caso con solución no recursiva, éste marco devuelve a quien lo invocó el valor determinado para dicho criterio y libera el espacio ocupado; en este caso, el último marco retorna 1 para el marco anterior y desaparece. Y continúa un proceso regresivo en que cada marco de activación desarrolla sus operaciones utilizando el valor devuelto por la función recursiva invocada y retorna el valor obtenido al marco que lo invocó.

8.4 ENVOLTURAS PARA FUNCIONES RECURSIVAS

Es común que las funciones recursivas se concentren en la búsqueda de un valor o en el desarrollo de una tarea concreta, como calcular el factorial, invertir una cadena, encontrar un valor en una estructura de datos u ordenar una lista y no se ocupen de otras tareas que no requieren ser recursivas como leer el dato a buscar, comprobar la validez de un argumento o imprimir los resultados.

Es decir, cuando se programa utilizando recursividad se encontrará que hay algunas tareas que no requieren un tratamiento recursivo y que pueden estar antes o después de la ejecución de la función recursiva. Para manejar estas tareas se definen programas, procedimientos o funciones *envoltura*.

Se denomina *envoltura* a los procedimientos no recursivos que se ejecutan antes o después de la invocación de una función recursiva y que se encargan de preparar los parámetros de la función y de procesar los resultados (Baase, 2002).

En el ejemplo del factorial se utiliza como envoltura un algoritmo que se encarga de leer el número que se utiliza como argumento de la función, invocar la función y mostrar el resultado. El seudocódigo se presenta en el cuadro 136.

Cuadro 136. Pseudocódigo del programa principal para calcular el factorial

```
1.    Inicio
2.        Entero: num
3.        Leer  num
4.        Imprimir "factorial de", num, " = ", factorial(num)
5.    Fin algoritmo
```

8.5 TIPOS DE RECURSIVIDAD

La recursividad puede presentarse de diferentes maneras y dependiendo de ellas se han establecido al menos cuatro tipos diferentes de implementaciones recursivas.

Recursividad simple: se presenta cuando una función incluye un llamado a sí misma con un argumento diferente. Ejemplo de este tipo de recursividad es la función factorial().

Este tipo de recursividad se caracteriza porque puede pasarse fácilmente a una solución iterativa.

Recursividad múltiple: el cuerpo de una función incluye más de una llamado a la misma función, por ejemplo, la función para calcular un valor de la serie Fibonacci (esta función se explica en la sección 8.7).

Recursividad anidada: se dice que una función recursiva es anidada cuando entre los parámetros que se pasan a la función se incluye una invocación a la misma. Un ejemplo de recursividad anidada es la solución al problema de Ackerman*.

* Wilhelm Ackerman (29 de marzo 1896 - 24 de diciembre 1962) matemático alemán, concibió la función doblemente recursiva que lleva su nombre como un ejemplo de la teoría computacional y demostró que no es primitiva recursiva.

Recursividad cruzada o indirecta: en este tipo de recursividad, el cuerpo de la función no contiene un llamado a sí misma, sino a otra función; pero, la segunda incluye un llamado a la primera. Puede ser que participen más de dos funciones. Este tipo de implementaciones también se conoce como *cadenas recursivas*. Como ejemplo de este tipo de recursividad se tiene la función para validar una expresión matemática.

8.6 EFICIENCIA DE LA RECURSIVIDAD

En general, una versión iterativa se ejecutará con más eficiencia en términos de tiempo y espacio que una versión recursiva. Esto se debe a que, en la versión iterativa, se evita la información general implícita al entrar y salir de una función. En la versión recursiva, con frecuencia, es necesario apilar y desapilar variables anónimas en cada campo de activación. Esto no es necesario en el caso de una implementación iterativa donde los resultados se van reemplazando en los mismos espacios de memoria.

Por otra parte, la recursividad es la forma más natural de solucionar algunos tipos de problemas, como es el caso de: ordenamiento rápido (*QuickSort*), las torres de Hanoy, las ocho reinas, la conversión de prefijo a posfijo o el recorrido de árboles, que aunque pueden implementarse soluciones iterativas, la solución recursiva surge directamente de la definición del problema.

El optar por métodos recursivos o iterativos es, más bien, un conflicto entre la eficiencia de la máquina y la eficiencia del programador (Langsam, 1997). Considerando que el costo de la programación tiende a aumentar y el costo de computación a disminuir, no vale la pena que un programador invierta tiempo y esfuerzo desarrollando una complicada solución iterativa para un problema que tiene una solución recursiva sencilla, como tampoco vale la pena implementar soluciones recursivas para problemas que pueden solucionarse fácilmente de manera iterativa. Muchos de los

ejemplos de soluciones recursivas de este capítulo se presentan con fines didácticos más no por su eficiencia.

Conviene tener presente que la demanda de tiempo y espacio extra, en las soluciones recursivas, proviene principalmente de la creación de los espacios de activación de las funciones y del apilamiento de resultados parciales, de manera que éstas pueden optimizarse reduciendo el uso de variables locales. A la vez que las soluciones iterativas que utilizan pilas pueden requerir tanto tiempo y espacio como las recursivas.

8.7 EJEMPLOS DE SOLUCIONES RECURSIVAS

Ejemplo 74. Sumatoria recursiva

Diseñar una función recursiva para calcular la sumatoria de los primeros n números enteros positivos, de la forma:

$$1 + 2 + 3 + 4 + 5 + 6 + \ldots + (n-1) + n$$

Se define la función $f(n)$ donde n puede ser cualquier número mayor o igual a 1 ($n >= 1$).

Se conoce que al sumar 0 a cualquier número éste se mantiene. De ahí se desprende que la sumatoria de cero es cero. Para cualquier otro entero positivo, la sumatoria se calcula adicionando su propio valor a la sumatoria del número inmediatamente inferior. Así:

$$f(0) = 0$$
$$f(1) = 1 + f(0)$$
$$f(2) = 2 + f(1)$$
$$f(3) = 3 + f(2)$$
$$f(4) = 4 + f(3)$$
$$f(n) = n + f(n-1)$$

De esto se desprende que

a. Si $n = 0 \rightarrow f(n) = 0$
b. Si $n > 0 \rightarrow f(n) = n + f(n-1)$

De esta manera se plantea una solución recursiva donde *a* representa el criterio base y *b* la invocación recursiva de la función.

$$f(n) = \begin{cases} 0 & \text{Si } n = 0 \\ n + f(n-1) & \text{Si } n > 0 \end{cases}$$

El pseudocódigo de la función sumatoria se presenta en el cuadro 137.

Cuadro 137. Función recursiva para calcular la sumatoria de un número

```
1.  Entero sumatoria(entero n)
2.      Si n = 0 entonces
3.          Retornar 0
4.      Si no
5.          Retornar  (n + sumatoria(n – 1))
6.      Fin si
7.  Fin sumatoria
```

Ejemplo 75. Potenciación recursiva

Se define como cálculo de una potencia a la solución de una operación de la forma x^n, donde x es un número entero o real que se conoce como base y n es un entero no negativo conocido como exponente.

La potencia x^n es el producto de n veces x, de la forma:

$$x^n = x * x * x * \ldots * x \ (n \text{ veces } x)$$

Por las propiedades de la potenciación se tiene que cualquier número elevado a 0 da como resultado 1 y que cualquier número elevado a 1 es el mismo número.

$$x^0 = 1$$
$$x^1 = x$$

Para este ejercicio se extiende la primera propiedad, también, para el caso de x = 0. Aunque normalmente se dice que este resultado no está definido, para explicar recursividad esto es irrelevante.

Para proponer una solución recursiva es necesario considerar la potencia x^n en función de una expresión más cercana a la identidad: $x^0 = 1$.

Se propone el siguiente ejemplo:

$2^0 = 1$	
$2^1 = 2$	$2^1 = 2 * 2^0 = 2$
$2^2 = 4$	$2^2 = 2 * 2^1 = 2 * 2 = 4$
$2^3 = 8$	$2^3 = 2 * 2^2 = 2 * 4 = 8$
$2^4 = 16$	$2^4 = 2 * 2^3 = 2 * 8 = 16$
$2^5 = 32$	$2^5 = 2 * 2^4 = 2 * 16 = 32$

Como se aprecia en la columna de la derecha, cada potencia puede plantearse utilizando la solución de la potencia anterior, con exponente menor.

En consecuencia:

a. Si $n = 0 \rightarrow x^n = 1$
b. Si $n > 0 \rightarrow x^n = x * x^{n-1}$

Donde a es el criterio base y b es la invocación recursiva.

En el cuadro 138 se presenta el pseudocódigo para calcular recursivamente cualquier potencia de cualquier número.

Cuadro 138. Función recursiva para calcular una potencia

```
1.  Entero potencia(entero x, entero n)
2.      Si(n = 0)
3.          Retornar 1
4.      Si no
5.          Retornar (x * potencia(x, n-1))
6.      Fin si
7.  Fin potencia
```

En las líneas 2 y 3 se soluciona el problema cuando se presenta el criterio base, en caso contrario, se debe hacer una invocación recursiva de la función, según las líneas 4 y 5.

Ejemplo 76. Serie Fibonacci

La serie Fibonacci tiene muchas aplicaciones en ciencias de la computación, en matemáticas y en teoría de juegos. Fue publicada por primera vez en 1202 por un matemático italiano del mismo nombre, en su libro titulado *Liberabaci* (Brassard, 1997).

Fibonacci planteó el problema de la siguiente manera: supóngase que una pareja de conejos produce dos descendientes cada mes y cada nuevo ejemplar comienza su reproducción después de dos meses. De manera que al comprar una pareja de conejos, en los meses uno y dos se tendrá una pareja, pero al tercer mes se habrán reproducido y se contará con dos parejas, en el mes cuatro solo se reproduce la primera pareja, así que el número aumentará a tres parejas; en el mes cinco, comienza la reproducción de la segunda pareja, con lo cual se obtendrán cinco parejas, y así sucesivamente.

Si ningún ejemplar muere, el número de conejos que se tendrán cada mes está dado por la sucesión: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …, que corresponde a la relación que se muestra en el cuadro 139:

Cuadro 139. Términos de la serie Fibonacci

Mes (*n*)	0	1	2	3	4	5	6	7	8	9	…
Pares de conejos *f(n)*	0	1	1	2	3	5	8	13	21	34	…

Cuya representación formal es:

a. f(0) = 0
b. f(1) = 1
c. f(n) = f(n-1) + f(n-2)

Es decir, en el mes 0 aun no se tiene ningún par de conejos, en el mes uno se adquiere la primera pareja, en el mes dos se mantiene la misma pareja comprada, porque aun no comienzan a reproducirse. Pero a partir del mes tres hay que calcular la cantidad de parejas que se tienen considerando la condición de que cada par genera un nuevo par cada mes, pero solo comienza a reproducirse después de dos meses.

De esta manera, el problema está resuelto para el mes cero y para el mes uno, estos se toman como criterio base (*a, b*) y a partir del segundo mes se debe tener en cuenta los meses anteriores. Esto genera una invocación recursiva de la forma:

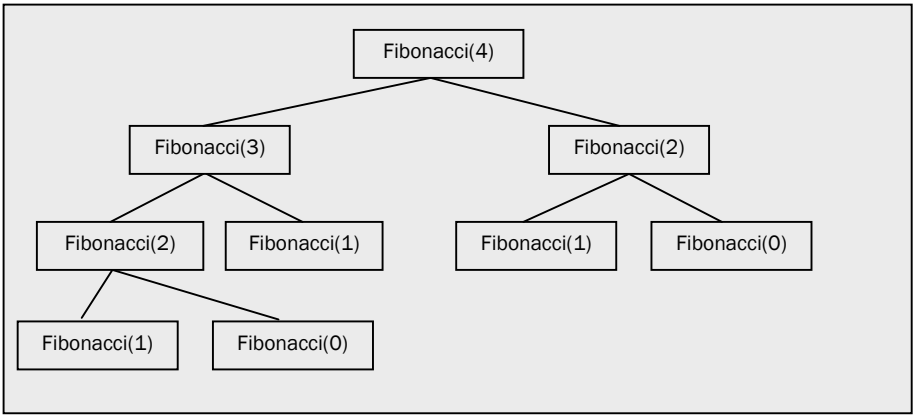
f(n) = f(n-1) + f(n-2) para *n >= 2*.

En el pseudocódigo que se presenta en el cuadro 140 se aprecia que cada activación tiene definidos dos criterios bases (líneas 2) y, si aun no se ha llegado a estos, en cada marco de activación se generan dos nuevas invocaciones recursivas (línea 5). En la figura 128 se presenta el árbol que se forma al calcular el valor de la serie para el número 4.

Cuadro 140. Función recursiva para calcular el n-ésimo termino de la serie Fibonacci

```
1.    Entero Fibonacci(Entero n)
2.        Si (n = 0) o (n = 1) entonces
3.            Retornar n
4.        Si no
5.            Retornar (Fibonacci(n – 1) + Fibonacci(n – 2))
6.        Fin si
7.    Fin Fibonacci
```

Figura 128. Marcos de activación de la serie Fibonacci



Ejemplo 77. Máximo común divisor

El Máximo Común Divisor (MCD) entre dos o más números es el mayor de los divisores comunes. Se conocen dos métodos para encontrar el MCD entre dos números: el primero consiste en descomponer cada número en sus factores primos, escribir los números en forma de potencia y luego tomar los factores comunes con menor exponente y el producto de estos será el MCD; el segundo, es la búsqueda recursiva en la que se verifica si el segundo número es divisor del primero, en cuyo caso será el MCD,

si no lo es se divide el segundo número sobre el módulo del primero sobre el segundo y así sucesivamente con un número cada vez más pequeño que tiende a 1 como el divisor común a todos los números. Este último se conoce como algoritmo de Euclides.

Por ejemplo, si se desea encontrar el MCD de 24 y 18, aplicando el algoritmo de Euclides, se divide el primero sobre el segundo (24/18= 1, residuo = 6), si la división es exacta se tiene que el MCD es el segundo número, pero si no lo es, como en este caso, se vuelve a buscar el MCD entre el segundo número (18) y el módulo de la división (6) y de esa manera hasta que se lo encuentre. Si los dos números son primos se llegará hasta 1.

Si $f(a,b)$ es la función que devuelve el Máximo Común Divisor de a y b, entonces se tiene que:

$c = a \bmod b$

a. $Si\ c = 0 \rightarrow f(a,b) = b$
b. $Si\ c > 0 \rightarrow f(a,b) = f(b,c)$

La función recursiva para calcular el MCD de dos números aplicando el algoritmo de Euclides se presenta en el cuadro 141.

Cuadro 141. Función recursiva para calcular el MCD

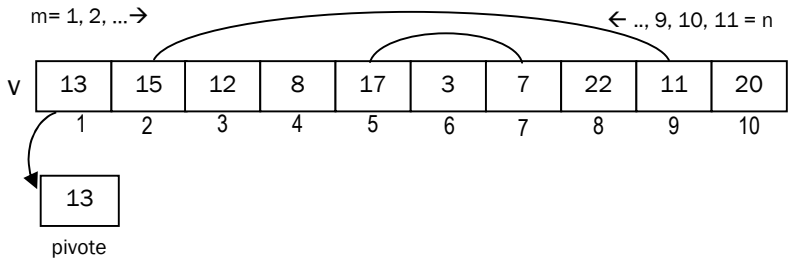
```
1.    Entero mcd(Entero a, Entero b)
2.       Entero c
3.       c = a Mod b
4.       Si c = 0 entonces
5.             Retornar b
6.       Si no
7.             Retornar mcd(b, c)
8.       Fin si
9.    Fin mcd
```

Ejemplo 78. Algoritmo de ordenamiento rápido recursivo

Este algoritmo recursivo de ordenamiento de vectores o listas se basa en el principio *dividir para vencer*, que aplicado al caso se traduce en que es más fácil ordenar dos vectores pequeños que uno grande. Se trata de tomar un elemento del vector como referencia, a éste se le llama *pivote*, y dividir el vector en tres partes: los elementos menores al pivote, el pivote y los elementos mayores al pivote.

Para particionar el vector se declara dos índices y se hacen dos recorridos, uno desde el primer elemento hacia adelante y otro desde el último elemento hacia atrás. El primer índice se mueve hasta encontrar un elemento mayor al pivote, mientas que el segundo índice se mueve hasta encontrar un elemento menor al pivote, cuando los dos índices se han detenido se hace el intercambio y se continúa los recorridos. Cuando los dos índices se cruzan se suspende el particionamiento. Como ejemplo, considérese el vector de la figura 112 y aplíquesele las instrucciones del cuadro 142.

Figura 129. Recorridos para particionar un vector

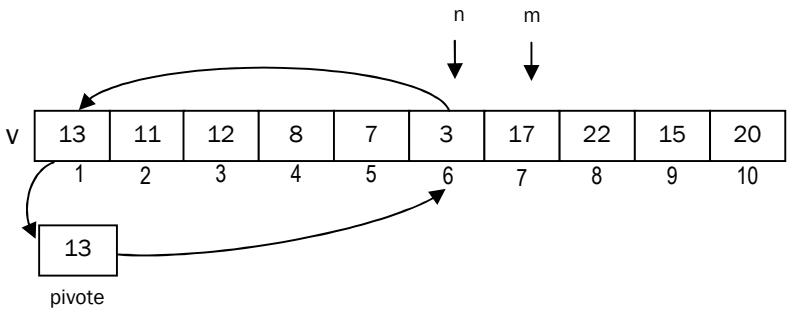


Cuadro 142. Recorridos para particionar un vector

```
1   Entero m = 1, n = 11, pivote = v[1]
2   Mientras m <= n hacer
3         Hacer
4               m = m + 1
5         Mientras v[m] < pivote
6         Hacer
7               n = n – 1
8         Mientras v[n] > pivote
9         Si m < n entonces
10              aux = v[m]
11              v[m] = v[n]
12              v[n] = aux
13        Fin si
14  Fin mientras
```

Después de ejecutar las instrucciones del cuadro 142 el vector quedaría como se muestra en la figura 130.

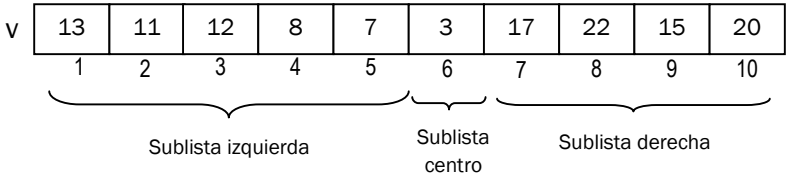
Figura 130. Recorridos para particionar un vector



Después de pasar los datos mayores al pivote al lado derecho y los menores al lado izquierdo se intercambia el pivote con el dato

indicado por el índice que recorre el vector de derecha a izquierda (n), como lo muestran las flechas en la figura 130. Después de dicho intercambio, los elementos quedan en el orden que se muestra en la figura 131.

Figura 131. Orden del vector después de la primera partición



Aunque las divisiones son puramente formales, ahora se tienen tres sublistas:

lista-izquierda = {3, 11, 12, 8, 7}
lista-centro = {13}
lista-derecha = {17, 22, 15, 20}

Como se aprecia en las sublistas, los datos se han movido de un lado del pivote al otro, pero no están ordenados. Por ello, el siguiente paso en el algoritmo *Quicksort* consiste en tomar cada una de las sublistas y realizar el mismo proceso, y continuar haciendo particiones de las sublistas que se generan hasta reducir su tamaño a un elemento.

En el cuadro 143 se presenta la función recursiva *Quicksort* con el algoritmo completo para ordenar un vector aplicando éste método.

Cuadro 143. Función recursiva Quicksort

```
1   Entero[] Quicksort(entero v[], entero inf, entero sup)
2       Entero: aux, pivote = v[inf], m = inf, n = sup+1
3       Mientras m <= n hacer
4           Hacer
5               m = m + 1
6           Mientras v[m] < pivote
7           Hacer
8               n = n – 1
9           Mientras v[n] > pivote
10          Si m < n entonces
11              Aux = v[m]
12              v[m] = v[n]
13              v[n] = aux
14          Fin si
15      Fin mientras
16      v[inf] = v[n]
17      v[n] = pivote
18      Si inf < sup entonces
19          v[] = Quicksort(v[], inf, n-1)
20          v[] = Quicksort(v[], n+1,sup)
21      Fin si
22      Retornar v[]
23  Fin Quicksort
```

La función *Quicksort* recibe un vector de enteros, en este ejemplo, y dos valores: *inf* y *sup*, correspondientes a las posiciones que delimitan el conjunto de elementos a ordenar. La primera invocación a la función se hará con todo el vector, enviando como parámetros el índice del primero y del último elemento (1 y n), pero en las que siguen el número de elementos se reducen significativamente en cada llamada, pues corresponde a las sub-listas que se generan en cada partición.

En esta versión de la función se toma como pivote el primer elemento, pero esto no necesariamente tiene que ser así, se puede diseñar una función para buscar un elemento con un valor intermedio que permita una partición equilibrada de la lista.

8.8 EJERCICIOS PROPUESTOS

Diseñar las soluciones recursivas para las necesidades que se plantean a continuación

1. Mostrar los números de 1 a n
2. Generar la tabla de multiplicar de un número
3. Sumar los divisores de un número
4. Determinar si un número es perfecto
5. Calcular el producto de dos números sin utilizar el operador *, teniendo en cuenta que una multiplicación consiste en sumar el multiplicando tantas veces como indica el multiplicador.
6. Calcular el Mínimo Común Múltiplo (MCM) de dos números.
7. Convertir un número decimal a binario
8. Convertir un número binario a decimal
9. Determinar si una cadena es palíndromo
10. Mostrar los primeros n términos de la serie Fibonacci
11. Invertir el orden de los dígitos que conforman un número
12. Determinar si un número es primo.

9. EL CUBO DE RUBIK

Si es que dormido estoy,
o estoy despierto
si un muerto soy
que sueña que está vivo
o un vivo soy
que sueña que está muerto.
Nuestros sueños se cumplen
siempre que tengamos paciencia...
W. Stekel

El cubo de Rubik o cubo mágico, como se lo llama comúnmente, es un rompecabezas mecánico diseñado por el arquitecto y escultor húngaro Ernö Rubik y patentado en 1975 en Hungría. Inicialmente, se utilizó este juego como instrumento didáctico en algunas escuelas de Hungría, se consideraba que por la atención y el esfuerzo mental que requiere para armarlo era un buen ejercicio para la mente.

Desde los primeros años de difusión de este objeto, los jugadores han medido sus habilidades para armarlo en el menor tiempo. En 1982 se celebró el primer campeonato mundial en Budapest.

En el año 1976 se patentó en Japón un invento similar al cubo de Rubik, su autor fue el ingeniero Teruthos Ishige.

9.1 DESCRIPCIÓN DEL CUBO

Es un rompecabezas formado por 26 piezas que se mueven sobre un dispositivo mecánico ubicado en el centro del cubo, oculto a la vista del usuario, como se observa en la figura 132.

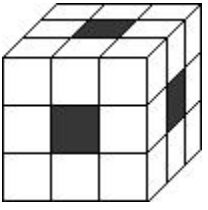
Se han producido muchas versiones del cubo de Rubik utilizando números, letras, figuras y colores. La más difundida es la de los seis colores, uno para cada cara. El juego consiste en ordenar las piezas de tal manera que el cubo tenga un solo color en cada cara.

Figura 132. Cubo de Rubik

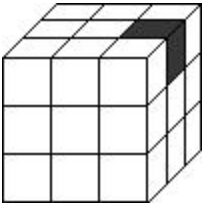
Aparentemente cada una de las 26 piezas que forman el cubo es, también, un pequeño cubo; sin embargo no lo son, se tienen tres tipos diferentes: centro, aristas y vértice (esquina), como se muestran en la figura 133.

Figura 133. Piezas que forman el cubo de Rubik

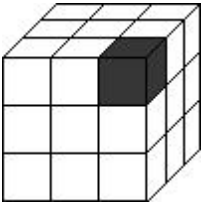
a. Piezas centrales



b. Arista



c. Vértice o esquina

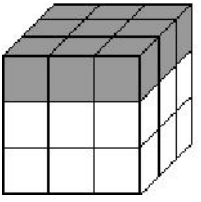


Los centros tienen solo una cara y el color de esta pieza es el punto de referencia para ordenar el cubo, estas piezas están fijas al dispositivo interno y por tanto no cambian de posición; las aristas tienen dos caras con diferente color y los vértices tienen tres.

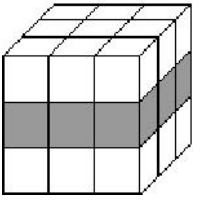
El posicionamiento de las piezas se logra mediante sucesivos giros de los módulos que conforman el cubo. Para efecto de facilitar la compresión de los movimientos necesarios para armar el cubo, en este documento se proponen los módulos de la figura 134.

Figura 134. Módulos del cubo de Rubick

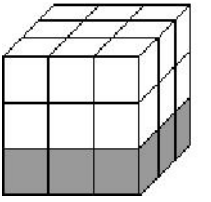
a. Módulo superior



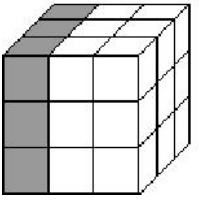
b. Módulo central



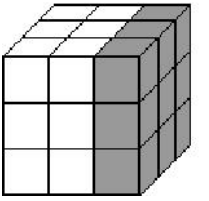
c. Módulo inferior



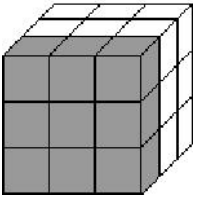
d. Módulo izquierdo



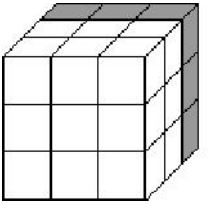
e. Módulo derecho



g. Módulo frontal



h. Módulo posterior



9.2 SOLUCION ALGORÍTMICA

La solución que se propone aquí consiste en armar el cubo en el siguiente orden.

1. Armar módulo superior
2. Armar módulo central
3. Armar módulo inferior

En el módulo superior y en el inferior es necesario ubicar y ordenar tanto las aristas como los vértices, mientras que en el central sólo se tienen que ordenar aristas. La principal dificultad radica en que en cada paso se debe cuidar de no deshacer lo que se ha hecho en los anteriores. Más adelante, en este mismo capítulo, se presentan algoritmos detallados para lograr este propósito, todo lo que se requiere es atención, paciencia y perseverancia en la aplicación de los mismos.

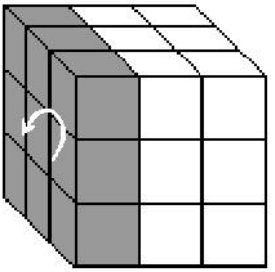
9.3.1 Consideraciones preliminares

Antes de presentar las secuencias que permiten armar el cubo es necesario especificar cómo deben efectuarse los movimientos y los nombres que se utiliza para describirlos.

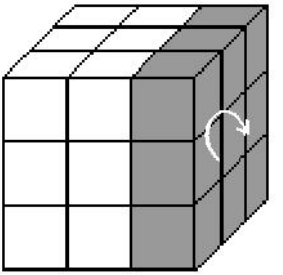
Cada módulo puede efectuar dos tipos de movimientos: los módulos derecho e izquierdo puede girar hacia adelante o hacia atrás; los módulos superior, inferior, frontal y posterior pueden girar hacia la derecha o hacia la izquierda, como se indica en los cubos de la figura 135. El módulo central no gira, a menos que se gire todo el cubo, en cuyo caso no cambiarán de posición las piezas.

Figura 135. Movimientos de los módulos del cubo de Rubick

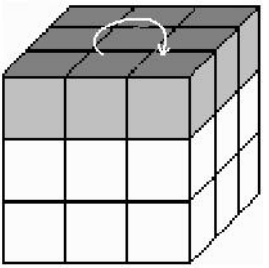
a. Módulo izquierdo, giro hacia adelante



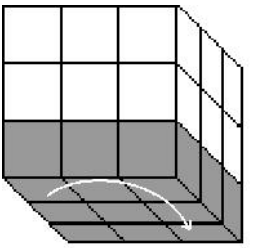
b. Módulo derecho, giro hacia adelante



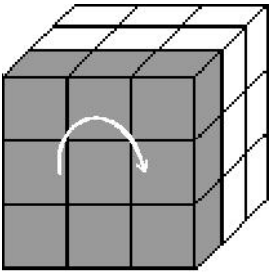
c. Módulo superior, giro hacia la derecha



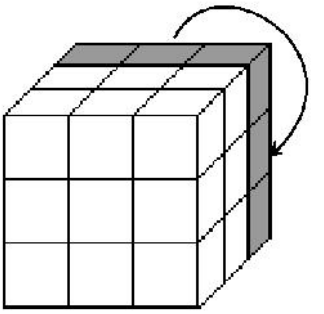
d. Módulo inferior, giro hacia la derecha



e. Módulo frontal, giro hacia la derecha



f. Módulo posterior, giro hacia la derecha



Cada giro que se indique debe efectuarse de 90 grados; es decir, que una esquina pasa a ocupar la posición de la siguiente en el orden que corresponda según el tipo de giro. Cuando se requiera hacer un giro de 180 grados se escribirá dos veces.

El algoritmo para armar el cubo se presenta de forma modular; es decir que está conformado por varios procedimientos que son llamados en el momento que se necesitan. Esto facilita la comprensión de las secuencias, ya que cada una cambia de posición una o dos piezas.

En la solución que se presenta se utilizan las secuencias más fáciles de comprender y de aplicar, no las más cortas, esto para que el lector no se desanime. Cuando haya adquirido confianza en la ejecución del algoritmo y haya aprendido cómo cambian de posición las piezas, podrá introducir algunos cambios para armar el cubo con un número menor de movimientos.

Para armar el cubo es necesario desarrollar una combinación de giros de los diferentes módulos que lo conforman. Cada giro se indica citando el nombre del módulo, seguido del tipo de giro a realizar. Ejemplo:

Superior → *derecha*: indica que el módulo superior debe girarse hacia la derecha.

Derecho → *adelante*: significa que se debe girar el módulo derecho hacia adelante.

9.3.2 Algoritmo principal para armar el cubo de Rubik

Como se ha mencionado en páginas anteriores, el cubo de Rubik es un rompecabezas, así que armarlo no es más que un juego, un entretenimiento; no obstante, dado el número de piezas y la cantidad de combinaciones de movimientos que se pueden generar, aprender a armarlo por el método de prueba y error puede tomar bastante tiempo y muchas personas podrían desanimarse al

ver que cada vez que intentan poner en orden una pieza se desordenan las que había ordenado previamente.

Para facilitar esta actividad se presentan, en forma algorítmica, las secuencias de movimientos necesarios para armarlo. Como la lista es extensa se opta por aplicar la estrategia *dividir para vencer* y se diseñan procedimientos para armar cada parte del cubo. El algoritmo principal se presenta en el cuadro 144.

Cuadro 144. Algoritmo principal cubo de Rubik

```
1.  Inicio
2.      Seleccionar y posicionar el centro de referencia
3.      Armar el módulo superior
4.      Armar el módulo central
5.      Armar el módulo inferior
6.  Fin algoritmo
```

Cada uno de los pasos de este algoritmo se explica y se refinan en lo que sigue de este capítulo.

9.3.3 Seleccionar y posicionar un centro de referencia

Para proceder a armar el cubo lo primero que se hace es seleccionar un color de referencia para evitar confundirse cuando el cubo gira y cambia de posición. Dado que las piezas del centro no cambian de posición, son éstas las que determinan el color de cada una de las caras y una de ellas servirá de referencia durante todo el proceso.

Siguiendo esta lógica, lo primero que se debe hacer es escoger uno de los seis colores y ubicar hacia arriba la cara que tiene dicho color en el centro. De esta manera, ya está definido cuál es el módulo superior para proceder a ordenar sus piezas en el siguiente paso.

Por ejemplo, si se toma como color de referencia el blanco, se comienza a armar el módulo que tiene en el centro la pieza de este color y se ubica hacia arriba. Cada vez que se ejecute alguno de los sub-algoritmos para armar cualquier módulo, la pieza central de color blanco debe estar hacia arriba.

El centro de referencia se mantiene durante todo el proceso de armar el cubo, pero antes de iniciar cualquier procedimiento es importante prestar atención al color del centro frontal para no cambiar de posición el cubo por error.

9.3.4 Armar el módulo superior

Este módulo se arma en dos fases: primero se ordenan las aristas haciendo uso del procedimiento *ordenar_arista_superior()*, luego se ordenan los vértices, para ello se invoca el procedimiento *ordenar_vértice_superior()*. El procedimiento principal se presenta en el cuadro 145.

Cuadro 145. Procedimiento para armar el módulo superior

```
1.  Armar_módulo_superior()
2.      Entero: arista, vértice
3.      // ciclo para ordenar las aristas del módulo superior
4.      Para arista = 1 hasta 4 hacer
5.          Si arista no está ordenada entonces
6.              Ordenar_arista_superior()
7.          Fin si
8.          Girar cubo hacia la izquierda
9.      Fin para
10.     // ciclo para ordenar los vértices del módulo superior
11.     Para vértice = 1 hasta 4 hacer
12.         Si vértice no está ordenado entonces
13.             Ordenar_vértice_superior()
14.         Fin si
```

Cuadro 145. (Continuación)

```
15.          Girar cubo hacia la izquierda
16.      Fin para
17.  Fin Armar_módulo_superior
```

En este procedimiento, mediante un ciclo, se examina cada una de las aristas del cubo, si ésta está ordena se pasa a la siguiente, pero si no lo está se invoca al procedimiento diseñado para ordenar aristas superiores. Luego se realiza la misma operación con los vértices.

Una arista está ordenada si el color de cada una de sus caras es el mismo color de la pieza central del cubo en la cara correspondiente. Por ejemplo, si el color de la pieza del centro en la cara superior es blanco y el color de la pieza central en la cara frontal es rojo, la arista superior frontal estará ordena si el color superior es blanco y el frontal es rojo, en caso contrario no está ordenada y será necesario invocar al procedimiento *Ordenar_arista_superior()*. Un vértice está ordenado si cada una de sus caras corresponde con el color de la pieza central en la cara respectiva, en caso contrario es necesario invocar el procedimiento *Ordenar_véritice_superior()*.

Ordenar las aristas del módulo superior

En esta fase del proceso se colocan las aristas del módulo superior de tal manera que los colores de cada una correspondan con el color del centro superior o color de referencia y con el color del centro de cada una de las caras del cubo según la posición que ocupan.

El procedimiento *Ordenar_arista_superior()*, que se presenta en el cuadro 146, está diseñado para colocar la pieza correcta en la posición *frontal-superior*, para ello se realizan tres operaciones fundamentales:

1. Encontrar la arista cuyos colores correspondan al centro superior y al centro frontal;
2. Mover la arista de su posición actual a la posición *frontal-inferior* sin importar la orientación.
3. Llevar la arista desde la posición *frontal-inferior* a la *frontal-superior* y orientarla.

Cuadro 146. Procedimiento para ordenar aristas superiores

```
1.  Ordenar_arista_superior()
2.      Identificar posición actual de la arista a ordenar
3.      // mover arista a la posición frontal inferior
4.      Según sea posición actual hacer
5.          Posición actual = Modulo posterior: arista superior
6.              Posterior → derecha
7.              Posterior → derecha
8.              Inferior → derecha
9.              Inferior → derecha
10.         Posición actual = Módulo derecho: arista posterior
11.             Derecho → adelante
12.             Inferior → izquierda
13.             Derecho → atrás
14.         Posición actual = Módulo derecho: arista superior
15.             Derecho → adelante
16.             Derecho → adelante
17.             Inferior → izquierda
18.         Posición actual = Módulo derecho: arista frontal
19.             Frontal → derecha
20.             Posición actual = Módulo izquierdo: arista posterior
21.             Izquierdo → adelante
22.             Inferior → derecha
23.             Izquierdo → atrás
24.         Posición actual = Módulo izquierdo: arista superior
25.             Izquierdo → adelante
```

Cuadro 146. (Continuación)

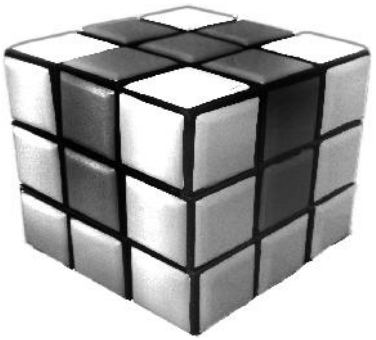
```
26.                 Izquierdo → adelante
27.                 Inferior → derecha
28.             Posición actual = Módulo izquierdo: arista frontal
29.                 Frontal → izquierda
30.             Posición actual = Módulo Frontal: arista superior
31.                 Frontal → derecha
32.                 Frontal → derecha
33.             Posición actual = Módulo inferior: arista posterior
34.                 Inferior → izquierda
35.                 Inferior → izquierda
36.             Posición actual = Módulo inferior: arista derecha
37.                 Inferior → izquierda
38.             Posición actual = Módulo inferior: arista izquierda
39.                 Inferior → derecha
40.         Fin según sea
41.         // pasar la pieza de la posición frontal-inferior a la frontal-
               superior
42.         Si color_frontal de arista frontal-inferior =  color de referencia
                                                            entonces
43.             Inferior → derecha
44.             Derecho → adelante
45.             Frontal → izquierda
46.             Derecho → atrás
47.         Si no
48.             Frontal → izquierda
49.             Frontal → izquierda
50.         Fin si
51.     Fin ordenar_arista_superiores
```

La ejecución de este procedimiento ordena la arista *superior-frontal*, para ordenar todas las aristas del módulo superior es

necesario ejecutarlo cuatro veces, como se especifica en el procedimiento *Armar_módulo_superior()*.

Después de ordenar las aristas superiores el cubo estará como se muestra en la figura 136, en la que se presenta con color gris oscuro la parte del cubo que ya está ordenada.

Figura 136. Módulo superior con aristas superiores ordenadas



Ordenar los vértices del módulo superior

Los vértices (piezas de las esquinas) están formadas por tres colores correspondientes a los colores de las tres caras que convergen en ellos. Si por cada lado el color del vértice es el mismo de la pieza central de la cara, entonces, el vértice está en la posición correcta, en caso contrario se lo debe desplazar hasta el vértice que le corresponde y orientarlo según los colores de las caras.

Los movimientos indicados en el procedimiento *Ordenar_vértice_superior()*, que se presenta en el cuadro 147, permiten ordenar el vértice *frontal–superior–derecho*. Para ello se requieren tres acciones:

1. Identificar la posición actual del vértice que se pretende ordenar;

2. Mover la pieza hasta la posición *frontal–inferior–derecha.* Para este paso se utiliza una estructura de decisión múltiple que incluye una secuencia de movimientos para cada una de las siete alternativas;
3. Mover la pieza desde la esquina *frontal-inferior–derecha* hasta la posición *frontal–superior –derecha*, en este paso se presentan tres posibilidades según la orientación del vértice, éstas se evalúan mediante decisiones anidadas.

Cuadro 147. Procedimiento para ordenar los vértices superiores

```
1.  Ordenar_vértice_superior()
2.      Identificar posición_actual del vértice
3.      // mover pieza a la posición inferior-frontal-derecha
4.      Según sea posición_actual hacer
5.          Posición_actual = módulo superior: vértice posterior-derecho
6.              Derecho → adelante
7.              Inferior → izquierda
8.              Inferior → izquierda
9.              Derecho → atrás
10.             Inferior → derecha
11.         Posición_actual = módulo superior: vértice posterior-izquierdo
12.             Izquierdo → adelante
13.             Inferior → derecha
14.             Inferior → derecha
15.             Izquierdo → atrás
16.         Posición_actual = módulo superior: vértice frontal-derecho
17.             Derecho → atrás
18.             Inferior → izquierda
19.             Derecho → adelante
20.             Inferior → derecha
21.         Posición_actual = módulo superior: vértice frontal-izquierdo
22.             Izquierdo → atrás
23.             Inferior → derecha
24.             Izquierdo → adelante
```

Cuadro 147. (Continuación)

```
25.          Posición_actual = módulo inferior: vértice posterior-derecho
26.              Inferior → izquierda
27.          Posición_actual = módulo inferior: vértice posterior-izquierdo
28.              Inferior → derecha
29.              Inferior → derecha
30.          Posición_actual = módulo inferior: vértice frontal-izquierdo
31.              Inferior → derecha
32.      Fin según sea
33.      // mover y orientar pieza a la posición superior-frontal-derecha
34.      Si color de referencia = color frontal del vértice inferior-derecho
          entonces
35.          Inferior → izquierda
36.          Derecho → atrás
37.          Inferior → derecha
38.          Derecho → adelante
39.      Si no
40.          Si color de referencia = color derecho del vértice inferior-derecho
             entonces
41.              Inferior → derecha
42.              Frontal → derecha
43.              Inferior → izquierda
44.              Frontal → izquierda
45.          Si no
46.              Derecho → atrás
47.              Inferior → derecha
48.              Derecho → adelante
49.              Frontal → derecha
50.              Inferior → derecha
51.              Inferior → derecha
52.              Frontal → izquierda
53.          Fin si
54.      Fin si
55.  Fin ordenar_vértice_superior
```

Este procedimiento se ejecuta cuatro veces, según lo establece la estructura iterativa del procedimiento *Armar_módulo_superior()*, al cabo de las cuales el cubo contará con el primer módulo ordenado, como se muestra en la figura 136.

Figura 136. Cubo ordenado el módulo superior

9.3.5 Armar el Módulo Central

Después de haber ordenado el módulo superior se procede a ordenar las cuatro aristas del módulo central. Cabe anotar que de nada serviría armar este módulo si alguna de las piezas del módulo superior no está en el lugar y con la orientación que le corresponde.

Para ordenar el módulo central se observa la arista inferior del módulo central y se determina si ésta corresponde a la posición derecha o izquierda del módulo central y se procede a moverla. También puede ocurrir que una arista esté en el módulo central, pero no en la posición y orientación correcta, en este caso se la pasa al módulo inferior para luego llevarla a la posición que le corresponda en el módulo central.

El procedimiento del cuadro 148 define un ciclo para ordenar las cuatro aristas y dentro de éste un segundo ciclo para buscar en el

módulo inferior la arista que corresponde a la posición central izquierda o central derecha de cada una de las caras.

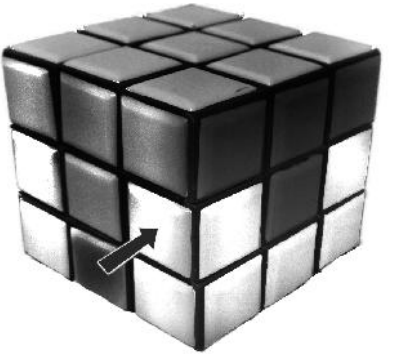
Cuadro 148. Procedimiento para armar el módulo central

```
1.  Armar_módulo_central()
2.      Entero: aristasordenadas, contador
3.      aristasordenadas = aristas ordenadas en el módulos central
4.      // ciclo para ordenar las aristas del módulo central
5.      Mientras  aristasordenadas < 4 hacer
6.      contador = 1
7.          Mientras contador <= 4 hacer
8.              Si color frontal de arista frontal-inferior = color centro-frontal
                Y color inferior de arista frontal-inferior = color centro-derecho
                entonces
9.                  Ordenar_arista_central_derecha()
10.             Si no
11.                 Si color frontal de arista frontal-inferior = color centro-frontal
                    Y color inferior de arista frontal-inferior = color centro-
                    izquierdo entonces
12.                     Ordenar_arista_central_izquierda()
13.                 Fin si
14.             Fin si
15.             Inferior → izquierda
16.             contador = contador + 1
17.         Fin mientras
18.         Si arista frontal-derecha NO está ordenada
            Y color frontal de arista central-derecha != color centro-inferior
            Y color derecho de arista central-derecha != color centro-inferior
            entonces
19.             Mover_arista_central_derecha()
20.         Fin si
21.         Girar cubo hacia la izquierda
22.     Fin para
23. Fin Armar_módulo_central
```

Este procedimiento está diseñado para identificar la arista a mover, ya sea del módulo inferior al central o del central al inferior, una vez localizada una pieza que debe ser movida invoca al procedimiento *Ordenar_arista_central_izquierda()*, *Ordenar_arista_central_derecha()* o *Mover_arista_central_derecha()*, según corresponda a la posición y colores de la arista.

El procedimiento *Ordenar_arista_central_derecha()* se encarga de mover la arista que ocupa posición inferior del módulo frontal a la posición derecha del mismo; o lo que es igual, mueve la arista desde el módulo inferior al módulo central, como se muestra en la figura 137, para ello es necesario que el color frontal de la arista corresponda al color del centro frontal y la cara inferior de la arista sea del mismo color que el centro del módulo derecho. Este procedimiento se presenta en el cuadro 149.

Figura 137. Ordenar arista central derecha



El procedimiento *Ordenar_arista_central_izquierda()* que se presenta en el cuadro 150 cumple una tarea similar al anterior, con la diferencia que mueve la arista inferior al lado izquierdo del módulo frontal.

Cuadro 149. Procedimiento para ordenar la arista central derecha

```
1.  Ordenar_arista_central_derecha()
2.      Inferior → izquierda
3.      Derecho → atrás
4.      Inferior → derecha
5.      Derecho → adelante
6.      Inferior → derecha
7.      Frontal → derecha
8.      Inferior → izquierda
9.      Frontal → izquierda
10. Fin procedimiento
```

Cuadro 150. Procedimiento para ordenar la arista central izquierda

```
1.  Ordenar_arista_central_izquierda()
2.      Inferior → derecha
3.      Izquierdo → atrás
4.      Inferior → izquierda
5.      Izquierdo → adelante
6.      Inferior → izquierda
7.      Frontal → izquierda
8.      Inferior → derecha
9.      Frontal → derecha
10. Fin procedimiento
```

Los dos procedimientos anteriores se utilizan para mover una pieza desde el módulo inferior al central, pero en ocasiones las aristas se encuentran en el módulo central, en posiciones que no les corresponden y es necesario pasarlas al módulo inferior, para lo cual se utiliza el procedimiento *Mover_arista_central_derecha()* que se presenta en el cuadro 151.

Cuadro 151. Procedimiento para mover la arista central derecha

```
1.  Mover_arista_central_derecha()
2.        Derecho: atrás
3.        Inferior: derecha
4.        Derecho: adelante
5.        Inferior: derecha
6.        Frontal: derecha
7.        Inferior: izquierda
8.        Frontal: izquierda
9.  Fin Mover_arista_central_derecha
```

Terminada esta fase, el cubo estará como se muestra en la figura 138.

Figura 138. Cubo con los módulos superior y central ordenados

9.3.6 Armar el Módulo Inferior

El módulo inferior debe armarse cuidadosamente para no desordenar los módulos superior y central. Para facilitar la

comprensión del procedimiento y para evitar que las secuencias de movimientos resulten excesivamente extensas, este módulo se ordena en tres pasos:

1. Se posicionan los vértices inferiores
2. Se orientan los vértices inferiores
3. Se ordenan las aristas inferiores

Al comenzar esta fase debe haber dos vértices que están posicionados aunque no necesariamente orientados, para ubicarlos se gira el módulo inferior, se observa los colores de las tres caras de la pieza y se comparan con los colores de los tres centros de las caras que convergen en dicho vértice. Un vértice está posicionado si tiene los tres colores de las caras, pero sólo está orientado si el color del vértice y el del centro es el mismo en cada cara.

En el algoritmo del cuadro 152 se presenta el algoritmo para armar el módulo inferior. El primer ciclo se ejecutará hasta posicionar los cuatro vértices. Dentro de este está un segundo ciclo anidado que tiene como propósito girar el módulo inferior hasta identificar los dos vértices que ya están posicionados, una vez han sido encontradas se ejecuta un procedimiento para posicionar las dos restantes. En cada ejecución del procedimiento los vértices cambian de posición, pero eso no garantiza que lleguen a la posición que les corresponde, por eso después de cada ejecución es necesario evaluar el estado del módulo y ejecutar nuevamente, tal como lo indica la estructura de repetición.

Cuadro 152. Procedimiento para armar el módulo inferior

```
1.  Armar_Módulo_Inferior()
2.      Mientras vértices_posicionados < 4 hacer
3.          // se identifica las dos que ya están posicionadas
4.          Mientras vértices_posicionados < 2 hacer
5.              Inferior → izquierda
6.          Fin mientras
7.          posicionar_vertices_inferiores()
```

Cuadro 152. (Continuación)

```
8.          Fin mientras
9.          // orientar vértices inferiores
10.         Mientras vértices_orientados < 4 hacer
11.             Orientar_Vértices_Inferiores()
12.         Fin mientras
13.         // Orientar aristas inferiores
14.         Mientras aristas_orientados < 4 hacer
15.             Ordenar_Aristas_Inferiores()
16.         Fin mientras
17.   Fin procedimiento
```

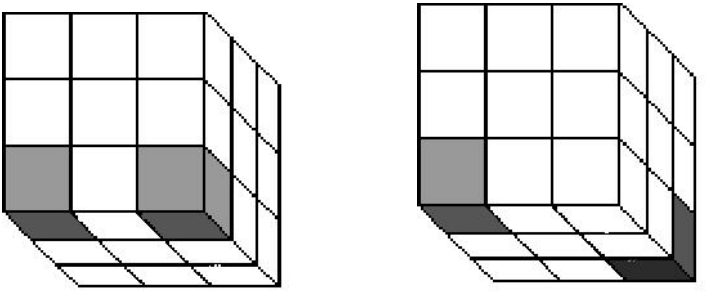
Después de posicionar los vértices se procede a orientarlos, esto es, a hacer que los colores de las caras del vértice y los del centro del cubo correspondan, este procedimiento también se ejecuta hasta que los cuatro vértices estén orientados. Finalmente, se posicionan y orientan las aristas del módulo inferior, para esto se incluye otro ciclo en el algoritmo.

Posicionar vértices inferiores

Al llegar a este punto ya ha identificado cuáles son los dos vértices que están posicionados. Éstos pueden estar adyacentes o en diagonal respecto a sí mismos, como se muestra en la figura 139. Es importante tener en cuenta si están adyacentes o en diagonal, ya que en el segundo caso se requiere un movimiento adicional.

El procedimiento para posicionar los vértices inferiores se presenta en el cuadro 153. Como se mencionó anteriormente, si después de ejecutarlo los cuatro vértices no están posicionados, se debe identificar los dos que si lo están y ejecutar nuevamente el procedimiento.

Figura 139. Posición de los vértices inferiores



Cuadro 153. Procedimiento para posicionar vértices inferiores

```
1.  Posicionar_vértices_inferiores()
2.  Entero aux
3.      Si vértices_posicionados están seguidos entonces
4.          aux = 1
5.          Colocar vértices posicionados en las posiciones Frontal-inferior
            izquierda y frontal-inferior derecha
6.      Si no (si están en diagonal)
7.          aux = 2
8.          Colocar vértices_posicionados en posiciones Frontal-inferior
            derecha y posterior-inferior izquierda
9.      Fin si
10.     Derecho → atrás
11.     Inferior → izquierda
12.     Derecho → adelante
13.     Frontal → derecha
14.     Si aux = 1 entonces
15.         Inferior → derecha
16.     Si no
17.         Inferior → derecha
18.         Inferior → derecha
19.     Fin si
```

Cuadro 153. (Continuación)

```
20.      Frontal → izquierda
21.      Derecho → atrás
22.      Inferior → derecha
23.      Derecho → adelante
24.      Inferior → derecha
25.  Fin algoritmo
```

Orientar los vértices inferiores

En el paso anterior se colocó las piezas esquineras en la posición que les corresponde según sus colores, pero no se las orientó para que los colores de sus caras correspondan con el color de cada una de las caras del cubo.

A estas alturas del proceso, al observar las esquinar inferiores del cubo puede encontrarse alguno de estos casos:

1. Que ningún vértice esté orientado
2. Que solo uno esté orientado
3. Que dos estén orientados y en posiciones seguidas
4. Que dos estén orientados y en diagonal

En cualquiera de los casos, el procedimiento que se presenta en el cuadro 154 se encarga de orientar las caras de los vértices. En esta subrutina se presentan dos secuencias de movimientos: una para situaciones en que ninguno o sólo uno de los vértices esté orientado y otra para cuando hay dos vértices orientados. Si hay vértices orientados se debe girar el cubo hasta que uno de los vértices orientados ocupe la posición inferior izquierda y si hay un segundo, este debe estar en el módulo derecho. Si hay dos vértices orientados, ya sea que estén seguidos o en diagonal, la secuencia de movimientos es la misma, con la diferencia de que si están en diagonal se requiere un movimiento adicional en el módulo inferior.

Cuadro 154. Procedimiento para orientar los vértices inferiores

```
1.  Orientar_vértices_inferiores()
2.      Si vértices orientados < 2 entonces
3.          Si vértices orientados = 1 entonces
4.              Colocar vértice orientado en la posición frontal
                inferior izquierda
5.          Fin si
6.          Derecho → atrás
7.          Inferior → izquierda
8.          Derecho → adelante
9.          Inferior → izquierda
10.         Derecho → atrás
11.         Inferior → derecha
12.         Inferior → derecha
13.         Derecho → adelante
14.         Inferior → derecha
15.         Inferior → derecha
16.     Si no (si dos vértices están orientados)
17.         Colocar un vértice orientado en la posición frontal inferior
                izquierda y el otro en el módulo derecho
18.         Izquierdo → adelante
19.         Superior → derecha
20.         Izquierdo → atrás
21.         Frontal → derecha
22.         Superior → derecha
23.         Frontal → izquierda
24.         Si vértices orientados están seguidos entonces
25.             Inferior → derecha
26.         Si no
27.             Inferior → derecha
28.             Inferior → derecha
29.         Fin si
30.         Frontal → derecha
31.         Superior → izquierda
```

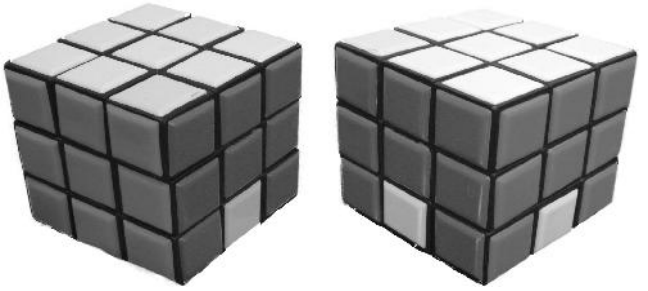
Cuadro 154. (Continuación)

```
32.             Frontal → izquierda
33.             Izquierdo → adelante
34.             Superior → izquierda
35.             Izquierdo → atrás
36.             Inferior → izquierda
37.         Fin si
38.     Fin procedimiento
```

Puede ocurrir que después de ejecutar este procedimiento los vértices no estén orientados, que aunque hayan girado sobre sí mismos los colores aun no correspondan con los colores de las piezas centrales en cada una de las caras. Este procedimiento se ejecuta repetidas veces hasta ordenar los vértices, como lo indica la estructura iterativa del procedimiento *Armar_Módulo_Inferior()*. No obstante, conviene observar atentamente los colores de las piezas, podría ser que por un error en la ejecución de los movimientos alguno de los vértices cambiara de posición en cuyo caso es necesario volver a ejecutar el procedimiento *Posicionar_vértices_inferiores()*.

Al orientar los vértices del módulo inferior solo quedarán faltando las aristas del mismo, como se muestra en la figura 140.

Figura 140. Cubo con vértices inferiores ordenados



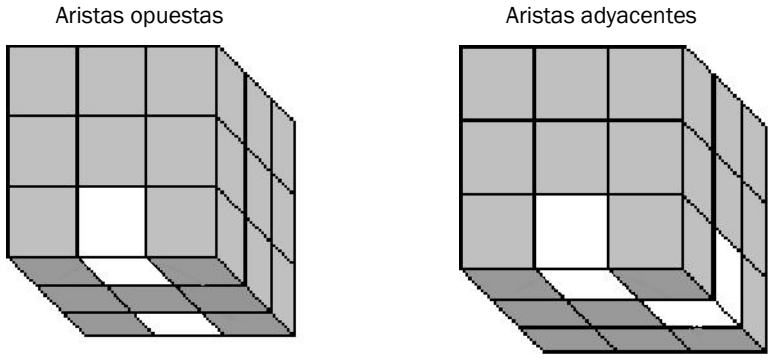
Posicionar y orientar las aristas del módulo inferior

Si los procedimientos anteriores se han ejecutado satisfactoriamente, en este momento sólo restará por ordenar las cuatro aristas del módulo inferior. En la distribución de éstas pueden presentarse los siguientes casos:

1. Ninguna arista está en el lugar correcto
2. Solo una arista está ubicada y orientada
3. Dos aristas están ubicadas y orientadas

Si dos aristas están posicionadas y orientas, éstas pueden estar adyacentes u opuestas, como se muestra en la figura 141. De esto depende la secuencia de movimientos a realizar para llevarlas a la posición final.

Figura 141. Posición de las dos últimas aristas por ordenar



En el caso de que tres aristas estuvieran ordenadas y la cuarta posicionada y no orientada, significa que el cubo se desensambló y al ensamblarlo nuevamente la pieza quedó al revés, y por tanto no tiene solución a menos que se ensamble correctamente. Si mientras se gira los módulos del cubo, accidentalmente una pieza o varias se desencajan es recomendable ensamblar el cubo colocando las piezas de manera que cada cara tenga un solo color.

En el procedimiento del cuadro 155 se presenta la secuencia de movimientos para ordenar las aristas del módulo inferior. Se proponen tres conjuntos de movimientos, el primero se ejecuta cuando ninguna o sólo una arista está posicionada y orientada, el segundo y el tercero cuando dos aristas están posicionadas y ordenadas, dependiendo si están en posiciones adyacentes u opuestas.

Cuadro 156. Procedimiento para posicionar y orientar las aristas del módulo inferior

```
1.  Ordenar_oristas_inferiores()
2.      Si aristas_ordenas < 2 entonces
3.          Si aristas_ordenadas = 1 entonces
4.              Colocar arista ordenada en posición frontal inferior
5.          Fin si
6.          Izquierdo → adelante
7.          Derecho → adelante
8.          Frontal → derecha
9.          Izquierdo → atrás
10.         Derecho → atrás
11.         Inferior → derecha
12.         Inferior → derecha
13.         Izquierdo → adelante
14.         Derecho → adelante
15.         Frontal → derecha
16.         Izquierdo → atrás
17.         Derecho → atrás
18.     Fin si
19.     Si aristas_ordenadas  = 2 entonces
20.         Si aristas ordenadas están en caras adyacentes entonces
21.             Colocar las aristas no orientadas en las posiciones:
                    frontal inferior e izquierda inferior
22.             Izquierdo → adelante
23.             Inferior → izquierda
24.             Superior → derecha
```

Cuadro 156. (Continuación)

25.	Posterior → izquierda
26.	Posterior → izquierda
27.	Inferior → izquierda
28.	Inferior → izquierda
29.	Superior → derecha
30.	Superior → derecha
31.	Frontal → izquierda
32.	Inferior → izquierda
33.	Frontal → derecha
34.	Superior → derecha
35.	Superior → derecha
36.	Inferior → izquierda
37.	Inferior → izquierda
38.	Posterior → izquierda
39.	Posterior → izquierda
40.	Inferior → derecha
41.	Superior → izquierda
42.	Izquierda → atrás
43.	Inferior → derecha
44.	Fin si
45.	Si las aristas orientas están en caras opuestas entonces
46.	Colocar las aristas no orientadas ocupen las posiciones: frontal inferior y posterior inferior
47.	Inferior → derecha
48.	Inferior → derecha
49.	Derecho → adelante
50.	Izquierdo → adelante
51.	Frontal → derecha
52.	Derecho → adelante
53.	Izquierdo → adelante
54.	Superior → derecha

Cuadro 156. (Continuación)

```
55.                 Derecho → adelante
56.                 Izquierdo → adelante
57.                 Posterior → izquierda
58.                 Posterior → izquierda
59.                 Derecha → atrás
60.                 Izquierda → atrás
61.                 Superior → derecha
62.                 Derecha → atrás
63.                 Izquierda → atrás
64.                 Frontal → derecha
65.                 Derecho → atrás
66.                 Izquierdo → atrás
67.             Fin si
68.         Fin si
69.  Fin de algoritmo
```

De igual manera que en los procedimientos anteriores, puede suceder que las piezas no se ordenan aunque cambien de posición o de orientación, es por este motivo que este procedimiento se invoca desde un ciclo, para ejecutarse repetidas veces hasta que se logre el propósito. Las secuencias de movimientos para ordenar el módulo inferior son más largas que las anteriores y mientras se desarrolla habilidad para armar el cubo es probable que se cometan algunos errores en las rutinas y se desordene el módulo central o el superior. Cuando esto ocurra no se desanime, descanse y vuelva a comenzar desde el paso en que se encuentre. Los primeros intentos seguramente no llegarán hasta el final, pero si persiste en su propósito finalmente lo logrará.

Si ya logró armarlo, felicitaciones. Ha demostrado ser paciente y dedicado. Si logró armar el cubo de Rubik puede lograr cualquier meta que se proponga, muchos proyectos, al igual que armar el cubo, requieren más de voluntad y perseverancia que de inteligencia o fuerza.

REFERENCIAS

Aho, A., Hopcroft, J. y Ullman, J. (1988). Estructuras de Datos y Algoritmos. Wilmington: Addison-Wesley Iberoamericana.

Alcalde, E. y García, M. (1992). Metodología de la Programación. Madrid: McGraw-Hill.

Appleby, D. y Vandekopple, J. (1998). Lenguajes de Programación: paradigma y práctica. México: McGraw-Hill.

Baase, S. y Gelder, A. (2002). Algoritmos computacionales. México: Pearson Education.

Becerra, C. (1998). Algoritmos: conceptos básicos. 4 ed. Bogotá: Quimpres.

Booch, G. (1994). Análisis y diseño orientado a objetos. 2 ed. Wilmington: Addison-Wesley.

Brassard G. y Bratley, T. (1997). Fundamentos de algoritmia. España: Prentice Hall.

Bruegge, B. y Dutoit, A. (2002). Ingeniería del software Orientada a Objetos. México: Pearson Education.

Cairó, O. (2005). Metodología de la Programación. México: Alfaomega.

Carretero, J., De Miguel, P., García, F. y Pérez, F. (2001). Sistemas Operativos: una visión aplicada. Madrid: McGraw-Hill.

Círculo Enciclopedia Universal. (2006). Tomo 4. Bogotá: Círculo de lectores.

Comer, T., Leiserson, C. Rivest, R. y Stein, C. (2001). Introduction to Algorithms. Cambridge: McGraw-Hill.

Deitel, H. y Deitel, P. (2004). Como Programar en Java. 5 ed. México: Pearson education

Galve, J., González, J., Sánchez, Á. y Veláquez J. (1993). Algorítmica: diseño y análisis de algoritmos funcionales e imperativos. Wilmington: Addison-Wesley Iberoamericana.

Espasa Calpe (2005). Gran Enciclopedia Espasa. Tomos 8, 9, 13 y 14. Bogotá.

Hermes, H. (1984). Introducción a la Teoría de la Computabilidad: algoritmos y máquinas. Madrid: Tecnos.

Hernández, R., Lázaro, J. C., Dormido, R. y Ros, S. (2001). Estructuras de datos y algoritmos. Madrid: Pearson Education.

Joyanes, L. (1992). Metodología de la Programación. México: McGraw-Hill.

Joyanes, L., Rodríguez, L. y Fernández, M. (1998). Fundamentos de programación: libro de problemas. Madrid: McGraw-Hill.

Joyanes, L. (1996). Fundamentos de Programación: algoritmos y estructuras de datos. 2 ed. Madrid: McGraw-Hill.

Joyanes, L. (2000). Programación en C++: algoritmos, estructuras de datos y objetos. Madrid: McGraw-Hill.

Joyanes, L. y Zahonero, I. (2002). Programación en Java 2: Algoritmos, estructuras de datos y programación orientada a objetos. Madrid: McGraw-Hill

Langsam, Y. y Otros. (1997). Estructuras de datos con C y C++. 2 ed. México: Prentice Hall.

Lipschutz, S. (1993). Estructuras de datos. Meéxico : McGraw-Hill.

López, G., Jeder, I y Vega, A. (2009). Análisis y Diseño de Algoritmos. Buenos Aires: Alfaomega.

Lopezcano, G. (1998). Nuevo Diccionario de la Computación. Tomo 1. Colombia: Zamora Editores.

Martínez, J. y Olvera, J. (2000). Organización y Arquitectura de computadoras. México: Prentice Hall.

Nassi, I. y Shneiderman, B. (1973). Flowchart Techniques for Structure Programming. Notes of the ACM, v. 8, N. 8. p. 12-26, August.

Oviedo, E. (2002). Lógica de la Programación. Bogotá: Eco ediciones. Plasencia, Z. (2008). Introducción a la Informática. Madrid: Anaya multimedia.

Santos, M., Patiño, I. y Carrasco, R. (2006). Fundamentos de programación. Bogotá: Alfaomega Ra-Ma.

Schildt, Herbert. (1993). Aplique turbo C++. Madrid : MacGraw-Hill.

Sommerville, I. (2005). Ingeniería del Software. Madrid: Pearson Education.

Timarán, R., Chaves, A., Checha, J., Jiménez, J. y Ordóñez, H. (2009). Introducción a la programación con Scheme. Pasto: Universidad de Nariño.

LISTA DE FIGURAS

Pág.

LISTA DE CUADROS

Pág.

LISTA DE EJEMPLOS

Pág.

Desde el enfoque de la programación estructurada, el diseño de algoritmos es un paso fundamental en la construcción de programas. Es en esta fase dónde se identifican los datos y las operaciones que sobre ellos se desarrollan, se establecen los cálculos que conducirán al resultado y se disponen todas las acciones en un orden lógico, conformando el diseño de la solución del problema.

Este libro fue preparado con base en la experiencia obtenida a lo largo de 10 años de docencia en el campo de la programación, tiempo en el que el autor ha tenido la oportunidad de observar cómo se enseña y cómo se aprende programación, ha identificado los temas que se comprenden y aplican con facilidad y los que implican mayor esfuerzo. Igualmente ha conformado un profuso repositorio de ejemplos y ejercicios apropiados para cada tema.

Anívar Chaves es ingeniero de sistemas egresado de la Universidad Mariana, especialista en Docencia Universitaria y Magister en Educación de la Universidad de Nariño. Ha sido docente de instituciones de educación superior como: Universidad Mariana, Fundación Universitaria San Martín, Institución Universitaria Cesmag, Universidad de Nariño y Universidad Nacional Abierta y a Distancia.

Es autor del libro Algoritmos: pseudocódigo, diagramas de flujo y diagramas N-S (2004) y coautor de: Internet en las instituciones de educación superior (2007), Introducción a la programación con Scheme (2009) y Un nuevo enfoque en la enseñanza de la programación (2009). Ha publicado artículos en diferentes revistas y ha presentado ponencias en eventos nacionales e internacionales.