PROGRAMACIÓN
-Teoría y aplicaciones-

Luis Joyanes Aguilar
Jaime Alberto Echeverri Arias
Gildardo Antonio Orrego Villa
Óscar Hernando Arenas Arenas

UNIVERSIDAD DE MEDELLIN

PROGRAMACIÓN
-TEORÍA Y APLICACIONES-

1a. edición: 2012
2a. edición: 2016
© Universidad de Medellín
© Luis Joyanes Aguilar
© Jaime Alberto Echeverri Arias
© Gildardo Antonio Orrego Villa
© Óscar Hernando Arenas Arenas

ISBN: 978-958-8922-63-8
ISBN-ebook: 978-958-8922-64-5

Editor:
Leonardo David López Escobar
Dirección electrónica: ldlopez@udem.edu.co
Universidad de Medellín. Medellín, Colombia
Cra. 87 No. 30-65. Bloque 20, piso 2.
Teléfonos: 340 52 42 - 340 53 35
Medellín - Colombia

Distribución y ventas:
Universidad de Medellín
e-mail: selloeditorial@udem.edu.co
www.udem.edu.co
Cra. 87 No. 30-65
Teléfono: 340 52 42
Medellín, Colombia

Corrección de estilo:
Lorenza Correa Restrepo
lorenzacorrea@une.net.co

Diseño portada:
Claudia Castrillón Álvarez
claudiadisenografico@gmail.com

Diagramación:
Hernán D. Durango T.
hernandedurango@gmail.com

Impresión:
Xpress Estudio Gráfico y Digital S.A.
Carrera 69 H # 77-40.
Teléfono: 6020808.
Bogotá - Colombia

Todos los derechos reservados.
Esta publicación no puede ser reproducida, ni en todo ni en parte, por ningún medio inventado o por inventarse, sin el permiso previo y por escrito de la Universidad de Medellín.
Hecho el depósito legal.

Agradecimientos

A mis alumnos, a mis colegas, profesores y maestros y a mis lectores de todas mis obras de España y de Latinoamérica, de universidades e institutos tecnológicos y politécnicos, a los que debo mi reconocimiento profesional. Mi agradecimiento eterno.

Luis Joyanes Aguilar

Para la elaboración de este texto he recibido ayuda de los diferentes profesores de la asignatura Fundamentos de Programación, quienes lo largo de su trabajo con cientos de estudiantes de las carreras de ingeniería han logrado adquirir una amplia experiencia en cuanto a las emáticas fundamentales y los puntos críticos en la enseñanza de esta vertida materia. Mi especial agradecimiento a los profesores: Juan David ujillo, Lina Tobón, Juan David Carrasquilla, Héctor Jairo Ortiz, Jesús alaguera, Darío León Valencia, Walter Cano por su aporte fundamental ra el planteamiento de los ejercicios y ejemplos propuestos.

Jaime Alberto Echeverri Arias

n el aliento y ánimo brindados por mi esposa Amparo y mi hija Xiomara, ía inútil poder participar como coautor de este texto. Pero no puedo vidar a mis seres queridos (mis padres, mi suegra; mis hermanos y manas) quienes desde otros lares me enviaron su rayito de luz que iluminaría para hacer mi aporte al proceso algorítmico de este libro.

mi hermano-amigo Ángel y a mis compañeros Oscar Ochoa Villegas, son Gutiérrez y Ricardo Muñoz, mi más sincero agradecimiento.

Gildardo Antonio Orrego Villa

i madre Rosalba, a quien le debo todo en mi vida. A mi familia, Manue- Sandra, su existencia ha iluminado la mía. Mi eterno agradecimiento profesores Guillermo González y Jairo Ortiz.

Óscar Hernando Arenas Arenas

Contenido

Prólogo 15
Introducción 17

Capítulo 1
Introducción al diseño de algoritmos

1.1. Presentación 19
1.2. Concepto de algoritmo 19
1.3. Metodología para el proceso enseñanza-aprendizaje en el diseño de algoritmos 21
1.4. Lenguajes utilizados en la solución algorítmica de problemas 22
.5. Etapas para la solución algorítmica de problemas 23

Capítulo 2
Estructura de un algoritmo

1. Presentación 27
2. Palabra reservada 27
Tipos de datos 28
2.3.1. Dato numérico 29
2.3.2. Dato lógico 30
2.3.3. Dato carácter 30
2.3.4. Dato cadena 30
2.3.5. Dato arreglo 31
Operadores 31
Expresiones 40
Concepto de variable y constante 43
Estructura de un algoritmo en seudocódigo 45
Prueba de escritorio 49

Capítulo 3
Estructuras de programación

Presentación 53
Estructuras de selección 54

3.2.1 Estructura de selección simple ... 54
3.2.2 Estructura de selección doble ... 56
3.2.3 Estructuras de selección anidadas ... 58
3.3. Estructuras de repetición ... 73
3.3.1. Estructura de repetición 'para' ... 74
3.3.2. Estructura de repetición 'mientras' ... 83

Capítulo 4
Vectores

4.1. Presentación ... 101
4.2. Concepto de vector ... 101
4.3. Operación con vectores ... 102
4.3.1. Operaciones de escritura y lectura ... 102
4.3.2. Operaciones sobre todos los elementos de un arreglo ... 104
4.4. Ordenamiento de un vector ... 111
4.5. Búsqueda en un vector ... 117

Capítulo 5
Matrices

5.1. Presentación ... 123
5.2. Concepto de matriz ... 123
5.3. Declaración de matrices ... 123
5.4. Referencia a los elementos de una matriz ... 124
5.5. Recorridos en una matriz ... 126

Capítulo 6
Subalgoritmos

6.1. Presentación ... 133
6.2. Definición ... 134
6.3. Parámetros ... 135
6.3.1. Parámetros según su ubicación ... 135
6.3.2. Parámetros según su función ... 135
6.4. Funciones ... 135
6.4.1. Definición de funciones externas ... 136
6.4.2. Elementos del encabezado de las funciones: ... 136
6.4.3. Elementos del cuerpo de las funciones ... 137
6.5. Procedimientos ... 142
Bibliografía ... 155

Índice de tablas

2.1. Funciones matemáticas básicas ... 28
2.2. Operadores aritméticos y lógicos ... 32
2.3. Prioridad de operadores ... 33
2.4. Tabla de verdad para la conjunción y la disyunción ... 36
2.5. Tabla de verdad del operador lógico negación ... 36
2.6. Tipo de dato, variables y valores posibles ... 44
7. Tipo de dato, constantes y valor ... 45
2.8. Explicación de los pasos del algoritmo del ejemplo 2.13 ... 48

Índice de figuras

1.1. Mapa conceptual algoritmo 20
1.2. Lenguajes usados en la solución de problemas mediante el computador 22
2.1. Mapa conceptual tipo de dato 29
2.2. Representación alternativa de las operaciones módulo y división entera entre enteros positivos 42
2.3. Estructura de un algoritmo especificado mediante seudocódigo 45
2.4. Resultado de ejecutar el algoritmo del ejemplo 2.22 47
3.1. Mapa conceptual estructuras de repetición 54
3.2. Proceso para extraer las cifras entero 86
4.1. Representación gráfica de un vector 102
4.2. Estado del vector después de modificarse el valor de la segunda posición 103
4.3. Estado del vector después de modificarse el valor de la tercera posición 104
4.4. Estrategia del método burbuja 112
4.5. Estrategia del método de selección 113
4.6. Estrategia del método de inserción 115
5.1. Representación gráfica de la matriz creada en el ejemplo 5.1 124
5.2. Estado de la matriz m después de la asignación 125
5.3. Valores de la matriz m del algoritmo 5.3 125
5.4. Recorrido de una matriz por filas 127
5.5. Recorrido de una matriz por columnas 127
6.1. Invocación de subalgoritmos 134
6.2. Mapa conceptual subalgoritmo 134

Autores

LUIS JOYANES AGUILAR

Catedrático de Lenguajes y Sistemas Informáticos de la Universidad Pontificia de Salamanca. Doctor ingeniero en Informática por la Universidad de Oviedo y doctor en Sociología por la Universidad Pontificia de Salamanca; licenciado en Ciencias Físicas por la Universidad Complutense de Madrid, con Grado en Electrónica; licenciado de Enseñanza Superior Militar especialidad de Artillería (empleo Teniente Coronel en la Reserva) por la Academia General Militar de Zaragoza. Doctor Honoris Causa por la Universidad Privada Antenor Orrego (UPAO) de Trujillo (Perú); doctor Honoris Causa por la Universidad San Martín de Porres de Lima (Perú).

Investigador asociado internacional y profesor visitante de la Universidad de Medellín (Colombia) desde noviembre de 2014. Profesor visitante e investigador internacional de la Universidad Católica de Santiago de Guayaquil, desde agosto de 2014.

Ha escrito más de 100 artículos profesionales y científicos, muchos de ellos en revistas indexadas en índices tales como JCR, Scopus, DBLP, Latindex. Dialnet, etc. Es autor de más de 40 libros de Informática e Ingeniería de Sistemas en las editoriales multinacionales McGraw-Hill y Alfaomega. Sus últimos libros publicados son: *Programación en C, C++, Java y UML (2ª ed.)* y *Fundamentos generales de programación*, en la editorial McGraw-Hill de México; *Sistemas de información en la empresa (2015); Big Data (2014)* y *Computación en la nube (2013)*, en la editorial Alfaomega de México.

En diciembre de 2014 fue elegido por el correspondiente Patronato (Consejo), PRESIDENTE de la Fundación de I+D de Software Libre (Fidesol) de Granada (España). Es miembro del Grupo de Investigación de "Ética en la nube, FFI2013-46908-R: Ciencia, tecnología y sociedad: Problemas políticos y éticos de la computación en nube como nuevo paradigma sociotécnico" de la Facultad de Filosofía de la Universidad Complutense de Madrid.

JAIME ALBERTO ECHEVERRI ARIAS

M. Sc. Ingeniería de Sistemas, se desempeña como profesor de tiempo completo de la Universidad de Medellín. Ha escrito varios artículos publicados en revistas científicas. Ha trabajado en varios eventos nacionales relacionados con redes neuronales, reconstrucción tridimensional, funciones de base radial, procesamiento digital de imágenes, recuperación de información a partir de imágenes. Actualmente participa en varios proyectos de investigación en áreas como bases de datos, repositorios de información, procesamiento de información, programación web, entre otros.

Ha sido coautor de varios libros publicados, entre ellos, *Sistemas computacionales*, y ha participado con diferentes capítulos de algunos libros editados en el Sello Editorial Universidad de Medellín.

Actualmente es el investigador líder del grupo de investigación ARKADIUS que tiene como líneas de investigación Automatización, Ingeniería del Software e Inteligencia Artificial. Ha sido profesor en Ingeniería de Sistemas de las siguientes asignaturas: Lenguajes de Programación, Fundamentos de Programación, Estructura de Datos, Bases de Datos, Análisis de Algoritmos, Computación Gráfica, Programación Web. Actualmente es coordinador del área de algoritmia de la Universidad de Medellín

GILDARDO ANTONIO ORREGO VILLA

Matemático e ingeniero de Sistemas de la Universidad de Antioquia y especialista en Edumática de la Fundación Universitaria Autónoma de Colombia.

Docente de cátedra de la Escuela de Ingeniería de Antioquia, sirve las asignaturas de informática en Ingeniería Biomédica, y Matemáticas Discretas, en Ingeniería Informática. Además, es catedrático de la Universidad de Medellín en la Facultad de Ingeniería de Sistemas con las asignaturas Fundamentos de Programación, Matemáticas Discretas y Matemáticas Especiales.

Ha realizado varios diplomados en la Universidad de Medellín entre otros, "Metodología de la investigación" con el que comenzó la investigación "Situaciones que obstaculizan el diseño de algoritmos computacionales" con estudiantes de los primeros niveles de Ingeniería de Sistemas y carreras afines de esta universidad.

Publicó varios textos, entre ellos, *Introducción a los computadores, al sistema operativo DOS y al Word*, el cual fue utilizado en varios colegios de Antioquia a principios de los años 90, y *Diseño de algoritmos*, que ha servido como texto guía para informática en la Escuela de Ingeniería de Antioquia.

ÓSCAR HERNANDO ARENAS ARENAS

Ingeniero de Sistemas de la Universidad Nacional de Colombia (Sede Medellín) y estudiante de la Especialización de Ingeniería de Software de la Universidad de Medellín.

Docente de cátedra de la Universidad de Medellín, la Escuela de Ingeniería de Antioquia y el Politécnico Grancolombiano (Sede Medellín). Es profesor de los cursos Fundamentos de Programación, Estructuras de Datos, Lenguajes de Programación, Programación Orientada a Objetos y Análisis y Verificación de Algoritmos.

Ha participado en proyectos de investigación en la Universidad de Medellín desarrollando aplicaciones de computación gráfica en 3D y aplicaciones móviles para dispositivos Android.

Ha realizado los diplomados Evaluación de Aprendizajes, Escribir para Publicar y Formación en Ambientes Virtuales de Aprendizaje FAVA en la Universidad de Medellín.

Prólogo

El desarrollo tecnológico actual ha hecho de la programación de computadores, un campo esencial en la mayoría de las áreas de la ciencia. Es difícil encontrar un campo de aplicación en el cual la ciencia de la computación no tenga herramientas que faciliten las tareas cotidianas en ese campo. Los programas de las carreras de ingeniería de Ambiental, Civil, Financiera, Sistemas y Telecomunicaciones de la Universidad de Medellín contemplan dentro de su plan de estudios la asignatura Fundamentos de Programación, con el fin de brindar los conceptos necesarios para desarrollar la habilidad de diseñar programas para computadores. En las clases teóricas se trabaja con seudocódigo.

Un seudocódigo se puede entender como una descripción informal, que brinda detalles de alto nivel de un algoritmo, que emplea convenios estructurales de un lenguaje particular, sin la rigidez de la sintaxis de estos, ni a la fluidez del lenguaje corriente. Que permite codificar un programa con mayor velocidad que en cualquier lenguaje de programación, con la misma validez semántica. (Oviedo, 2002).

En este texto se utilizará el lenguaje de programación Visual Basic. NET. Dicho lenguaje es uno de los más populares en el mundo por su facilidad para diseñar aplicaciones cada vez más robustas; permite crear aplicaciones para Windows de una forma sencilla. La palabra visual hace referencia a la forma en que se van diseñando las aplicaciones, y al aspecto gráfico que toman los diferentes objetos en el momento de ejecutar las aplicaciones; la palabra NET hace referencia al medio donde se ejecutarán las aplicaciones diseñadas.

La necesidad sentida en las universidades y politécnicos de contar con un lenguaje que permita de manera rápida poner en práctica los conceptos teóricos tratados en las clases de *Fundamentos de Programación* y *Programación Orientada a Objetos* (para estudiantes de Ingeniería de Sistemas) ha motivado la redacción de este texto. Su propósito es acercar a

los estudiantes a una herramienta potente y fácil de utilizar en un tiempo reducido, con el fin de probar los algoritmos diseñados en clase y validarlos de acuerdo con los requerimientos impuestos. Se pretende con este texto servir de guía a los estudiantes de *Fundamentos de Programación* de la Universidad de Medellín y de otras universidades para que puedan poner en práctica los conceptos tratados en la clase teórica.

Medellín, julio 2015

Luis Joyanes Aguilar
Jaime Alberto Echeverri Arias
Gildardo Antonio Orrego Villa
Óscar Hernando Arenas Arenas

Introducción

Diseñar algoritmos es una actividad exigente; es una disciplina que permite al programador situarse al frente de la solución de un problema de manera organizada y hábil, evitando una serie de errores de análisis y lógicos que posiblemente no le dejarán tener un buen desempeño. Entre otras, las debilidades que tiene el ser humano para desarrollar la solución de problemas y, en particular, en el diseño de algoritmos son: pereza mental, distracción y desconcentración, y falta de análisis, creatividad, sentido común, dedicación y planeación lógica.

Al trabajar la disciplina de la programación, las cualidades que fortalecen a un programador son: el análisis lógico, la creatividad, el entusiasmo, el dinamismo, la concentración, la atención, la dedicación, el sentido común, la memoria, el orden y el planeamiento lógico.

En cada capítulo del texto se exponen conceptos de los diferentes temas con ejemplos y problemas resueltos que le ayudarán a visualizar diferentes maneras de construir algoritmos. Al final de estos, se presentan problemas propuestos con los cuales el lector se podrá ejercitar para adquirir la habilidad en el diseño de algoritmos.

Tanto los estudiantes de Tecnología de Desarrollo de Software como los de Ingeniería tendrán la posibilidad de conocer los principios para el diseño de algoritmos que les faciliten una aproximación a los lenguajes de alto nivel y les ayuden en el razonamiento lógico-matemático para la solución de problemas por medio del uso de herramientas informáticas.

Capítulo 1

Introducción al diseño de algoritmos

1.1. PRESENTACIÓN

Actualmente todas las áreas de la tecnología utilizan herramientas informáticas que permiten agilizar y organizar la información y, por lo tanto, optimizar los recursos empleados en labores cotidianas dentro de las organizaciones.

Diseñar algoritmos se ha convertido en una de las disciplinas más solicitadas en el mundo. Tareas como el diseño y manejo de datos, construcción de *software* para aplicaciones específicas y su respectivo manejo hacen que las tecnologías en desarrollo de *software* tenga una importancia preponderante para el desarrollo de un país, tal como quedó claro en el *cluster* de Tecnologías de la Información y la Comunicación.

1.2. CONCEPTO DE ALGORITMO

Un algoritmo se define como una serie lógica y finita de pasos para resolver un problema. Diariamente ejecutamos algoritmos en nuestra cotidianidad, por ejemplo: los pasos que seguimos para desplazarnos a nuestro lugar de trabajo, las diferentes actividades realizadas para preparar un delicioso arroz se pueden clasificar como algoritmos.

Diseñar algoritmos es una actividad exigente; es una disciplina que permite al programador situarse al frente de la solución de un problema de manera organizada y hábil, evitando una serie de errores lógicos que posiblemente no le dejarán tener un buen desempeño. Entre otras, las debilidades que tiene el ser humano para desarrollar la solución de problemas y en particular, en el diseño de algoritmos son: pereza mental, distracción, desconcentración; falta de análisis, poca creatividad, carecer de sentido común. La planeación lógica facilita la elaboración, puesta a punto y ejecución de algoritmos.

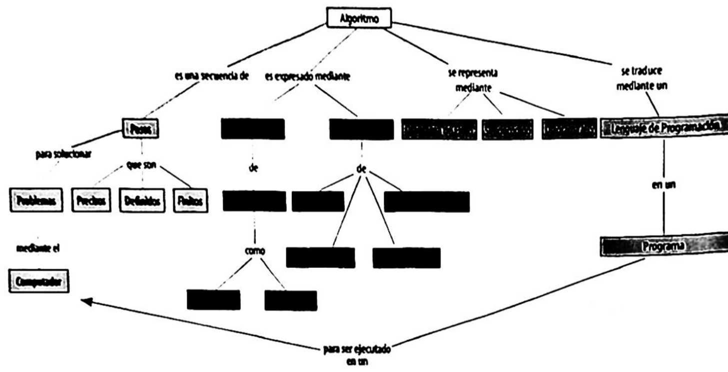


Figura 1.1. Mapa conceptual algoritmo

Para trabajar en la disciplina de la programación, las cualidades que fortalecen a un programador son el análisis lógico, la creatividad, e entusiasmo, el dinamismo, la concentración, la atención, la dedicación, el sentido común, la memoria, el orden y el planeamiento lógico. El desarrollo de algoritmos lleva varias décadas aplicándose en los computadores; pe aún no ha alcanzado el nivel de evolución lo suficientemente alto com el que tienen otras ciencias y disciplinas, tales como las matemáticas, l medicina, la física y la química. Sin embargo, ha servido de apoyo pa el desarrollo de estas, y a la par con las ciencias, la disciplina de diseñ algoritmos ha continuado su proceso de desarrollo tratando de supli las complejas necesidades humanas y sirviendo de apoyo a las dem ciencias.

El presente capítulo tiene como fin primordial inducir y orientar l conceptos básicos para el diseño de algoritmos, los cuales se podría codificar para obtener programas computacionales. En efecto, algunc problemas planteados y los algoritmos diseñados estarán especificad mediante un lenguaje que pueda ser entendido por un computador. Co se parte de lo simple a lo complejo, no es necesario que el lector teng alguna fundamentación o conocimientos previos de programación pa diseñar un algoritmo.

El diseño de algoritmos considera los avances tecnológicos, pues medida que avanza la informática, también progresan las herramient de programación. Entre otras herramientas se conocen: diagramas d

flujo, diagramas rectangulares, seudocódigo y las modernas herramientas de lenguajes para el modelamiento tales como UML (Unified Modeling Language). Dichas herramientas se pueden utilizar según la necesidad; además, permiten que se pueda pasar de una herramienta a otra, sin temor a errores de consistencia; todo depende de la necesidad para saber qué herramienta utilizar. Es precisamente como el carpintero, que dependiendo de lo que necesita hacer, utiliza la herramienta adecuada que le facilite el trabajo. Sin embargo, este texto utilizará como herramienta de programación el seudocódigo, porque facilita la codificación para diferentes lenguajes.

En cada capítulo se presentan conceptos de los diferentes temas con ejemplos y problemas resueltos que le ayudarán a visualizar diferentes maneras de construir algoritmos. Al final de estos se proponen problemas con los cuales el lector se podrá ejercitar para adquirir la habilidad en el diseño de algoritmos.

Los problemas propuestos están pensados de tal manera que sirvan como base y ejercitación para otras asignaturas relacionadas con la programación de computadoras, como: lenguajes de programación, estructuras de datos, entre otras. De ahí la importancia de desarrollar la solución a dichos problemas.

1.3. METODOLOGÍA PARA EL PROCESO ENSEÑANZA-APRENDIZAJE EN EL DISEÑO DE ALGORITMOS

El estudiante en su proceso de aprendizaje presenta dificultades para el diseño de algoritmos. Para mejorar el desarrollo de estos procesos, se sugiere que tenga en cuenta las siguientes actividades:

1. Aprenda del error. Recuerde que "el que no se equivoca es porque nada ha hecho". El error del compañero de aula no debe causar burla, sino atención para no ir a cometer la misma equivocación.
2. Hacer los problemas propuestos al igual que los resueltos, buscando soluciones más eficientes o al menos iguales a las planteadas; sugerir problemas con soluciones algorítmicas (sin memorizar la solución).
3. Hacer énfasis en el problema por resolver, mas no en el lenguaje de programación. Olvidarse del "cacharreo" en el computador.
4. Reflexionar sobre el problema, analizar ejemplos similares.

1.4. LENGUAJES UTILIZADOS EN LA SOLUCIÓN ALGORÍTMICA DE PROBLEMAS

Los problemas que resolveremos serán expresados en lenguaje natural, ayudados, ocasionalmente, de notación matemática para simplificar el enunciado. Como plantearemos soluciones algorítmicas a los problemas usaremos exclusivamente el lenguaje de especificación seudocódigo para expresar la serie de pasos que resuelven el problema. Como el seudocódigo no es estándar y existen tantas versiones como profesores o autores, propondremos aquí nuestro propio seudocódigo que seguiremos consistentemente en todos los ejemplos presentados. El seudocódigo es una herramienta de programación que facilita tanto la lectura como la escritura de programas. Las instrucciones que lo componen se escriben normalmente con palabras tomadas del inglés, pero aquí las traduciremos al español. El seudocódigo es un lenguaje diseñado para especificar o describir algoritmos.

Figura 1.2. Lenguajes usados en la solución de problemas mediante el computador

Una vez diseñado el algoritmo lo traduciremos en un programa mediante un lenguaje de programación de alto nivel para ejecutarlo en un computador. Por tanto, tendremos la solución a un problema expresada mediante dos lenguajes diferentes: en seudocódigo y en algún lenguaje de alto nivel.

Los lenguajes de programación están diseñados para ser leídos y escritos por humanos y ejecutados por computadoras. Se dividen en lenguajes de bajo y de alto nivel. Esta clasificación no hace referencia a que existen lenguajes que son inferiores en calidad a otros. Los lenguajes de bajo nivel están estrechamente relacionados con la estructura física y lógica del procesador en el cual se ejecutan sus instrucciones, y los de alto nivel son más cercanos a la capacidad cognitiva de los humanos. El lenguaje máquina y el lenguaje ensamblador son lenguajes de bajo nivel. Algunos lenguajes de alto nivel son: C, C++ C#, Python, Java, JavaScript, Matlab, Perl, PHP, Fortran y Lisp.

Un programa es una secuencia de instrucciones que se ejecutan en un computador y producen un resultado. Los programas se escriben o desarrollan mediante los lenguajes de programación.

1.5. ETAPAS PARA LA SOLUCIÓN ALGORÍTMICA DE PROBLEMAS

Para solucionar problemas con la ayuda del computador se tienen dos etapas principales:

ETAPA 1: RESOLUCIÓN ALGORÍTMICA DEL PROBLEMA

La primera etapa es la más importante acción que debe realizar para solucionar un problema de manera algorítmica, ya que corresponde a la etapa de diseño de un algoritmo. Se lleva a cabo en los siguientes pasos:

Lectura exhaustiva del problema. Corresponde a una de las partes fundamentales para la solución correcta del problema; porque permite comprender lo que se está pidiendo. Para tal fin, al leer el enunciado debe entender claramente qué está pidiendo en dicho problema y, por lo tanto, no debe hacer más de lo que le piden, pero tampoco menos. No debe comenzar a diseñar el algoritmo hasta tanto no haya entendido el problema.

Análisis del problema. Es la tarea más importante de todo el proceso de programación, ya que en ella se conocen exactamente los datos que se deben usar para obtener y mostrar los resultados pedidos. Esta etapa requiere imaginación y creatividad por parte del programador. Tiene como finalidad determinar cuidadosamente qué tipo de información se necesita producir; por consiguiente, debe realizarse una lectura reflexiva. Esta fase exige un estudio exhaustivo del problema, para definirlo de manera precisa. Debe tenerse muy claro cuál es el problema y el objetivo. Tener claro el objetivo nos va a permitir obtener, entre otros los siguientes beneficios:

a) Saber lo que nos piden o hacia dónde vamos.
b) Saber cómo alcanzar lo que nos piden.
c) Saber hasta dónde debemos llegar.

En esta etapa es donde el programador se da cuenta de si el problema está bien definido, si las especificaciones de entrada y salida han sido o no descritas. En síntesis, se logra:

a) Definir el problema.

b) Determinar los datos de entrada: Información necesaria para resolver el problema.

c) Identificar el proceso: la secuencia de operaciones que permitirán obtener el resultado pedido a partir de los datos de entrada.

d) Identificar los datos de salida (resultados finales de los cálculos).

Recomendación. Para comprender el problema lea varias veces su enunciado y responda las siguientes preguntas:

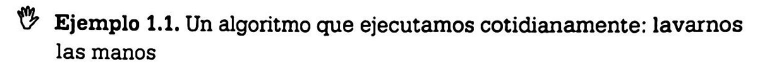
- ¿Qué datos me dan en el enunciado del problema?
- ¿Cuál es la pregunta que me da el problema?
- ¿Qué debo lograr?
- ¿Es toda la información útil?
- ¿Cuál es la incógnita del problema?
- ¿Es posible organizar la información?
- ¿Se pueden agrupar los datos por categorías?
- ¿Es posible trazar una figura o diagrama del problema?

Diseño del algoritmo. Una vez conocidos los datos de entrada, los datos de salida y el proceso necesario para generar los resultados a partir de los datos de entrada, se procede a especificar el algoritmo mediante seudocódigo.

Verificación manual del algoritmo. Esta fase permite determinar si el algoritmo que hemos diseñado logra el objetivo propuesto; de no ser así se debe corregir el algoritmo hasta lograr que satisfaga el objetivo propuesto. Consiste en hacer una simulación que comprueba si lo que se ha diseñado en el problema produce los resultados esperados correctamente. En efecto, se usan datos significativos de entrada o auxiliares, los cuales se anotan en una hoja de papel y con ellos se hace el seguimiento paso a paso a dicho algoritmo, hasta obtener los valores resultantes. De tal manera, escriba todas las variables utilizadas en el algoritmo y siga paso a paso los diferentes cambios de las variables en cada instrucción. Este proceso se conoce como "prueba de escritorio".

ETAPA 2: IMPLANTACIÓN DE LA SOLUCIÓN EN EL COMPUTADOR

En esta etapa el algoritmo diseñado en la etapa anterior se traduce a un programa, mediante un lenguaje de programación de alto nivel (codificación), y se ejecuta en un computador para verificar si es correcta la solución o se debe corregir.



Ejemplo 1.1. Un algoritmo que ejecutamos cotidianamente: lavarnos las manos

1. **inicio**
2. Abrir la llave del lavamanos
3. Remojar las manos con agua
4. Cerrar la llave
5. Aplicar jabón en las manos
6. Frotar las manos
7. Abrir la llave
8. Enjuagar
9. Cerrar la llave
10. **fin**

Con respecto al algoritmo anterior se pueden anotar varias cosas:

- Es posible que otra persona ejecute esta actividad en más o en menos pasos; igual ocurre con los programas o algoritmos computacionales: un programador (desarrollador) experto puede diseñar algoritmos con menos pasos y posiblemente más eficientes que los algoritmos diseñados por novatos.
- Los algoritmos se delimitan por las instrucciones inicio – fin
- Los pasos no necesariamente deben tener un orden preciso; es posible que otra persona obtenga el mismo resultado con pasos distintos o en otro orden.

Ejemplo 1.2. Hacer una solicitud de préstamo en un banco:

1. **inicio**
2. El banco verifica que el cliente tenga una cuenta
3. Verifica que no esté moroso
4. Si el cliente está moroso entonces se rechaza la solicitud
5. En caso contrario se acepta la solicitud
6. **fin**

Ejercicio 1.1. Proponga algoritmos para las siguientes actividades:

1. Hacer el cambio de un bombillo.
2. Echar combustible a un vehículo.
3. Adquirir una revista.
4. Entrar a una casa que está con llave.
5. Arrancar un vehículo.
6. Estudiar para un examen.
7. Viajar en avión.
8. Parquear un vehículo.
9. Ir de casa al centro de estudios.
10. Abrir una ventana.
11. Tomar una fotografía
12. Diseñe un algoritmo que calcule el área de un triángulo.

CAPÍTULO 2

Estructura de un algoritmo

2.1. PRESENTACIÓN

La descripción escrita de un algoritmo contiene diversos elementos y convenciones que facilitan expresar y comprender la solución planteada a un problema. Dicha descripción la vamos a realizar mediante el lenguaje denominado seudocódigo. Los elementos frecuentemente utilizados en la especificación de algoritmos son: palabras reservadas, valores, variables, constantes, operadores, expresiones, instrucciones y subalgoritmo.

2.2. PALABRA RESERVADA

Es una secuencia de caracteres que tienen un significado preciso en el seudocódigo o en un lenguaje de programación. Por tanto, a una palabra reservada no se le puede cambiar el significado durante el diseño de un algoritmo. Por ejemplo, una palabra reservada no puede ser utilizada como el nombre de una variable, constante o subalgoritmo. Las palabras reservadas que emplearemos en la especificación de algoritmos son:

- algoritmo, variables, constantes, entero, real, lógico, cadena, verdadero, falso, mod, div, y, o, no, inicio, fin, si-entonces-si _ no-fin _ si, para-hasta-paso-fin _ para, mientras-hacer-fin _ mientras, pi, funcion, devolver, procedimiento.

Un subalgoritmo es una secuencia de declaraciones e instrucciones que llevan a cabo una tarea específica. Se dividen en funciones y procedimientos. En la tabla 2.1 se listan las funciones que utilizaremos en el diseño de algoritmos.

Tabla 2.1. Funciones matemáticas básicas

Nombre función	*Seudocódigo*	*Forma de uso*
Raíz cuadrada	raiz2(x)	Recibe un número y retorna su raíz cuadrada
Seno	sen(x)	Recibe un ángulo en grados y retorna su seno
Coseno	cos(x)	Recibe un ángulo en grados y retorna su coseno
Tangente	tan(x)	Recibe un ángulo en grados y retorna su tangente
Logaritmo natural	ln(x)	Recibe un número positivo y retorna su logaritmo natural
Exponencial	exp(x)	Recibe un número positivo y retorna su exponencial
Aleatorio	aleatorio()	No recibe valor y retorna un número aleatorio entre cero y uno
Aleatorio	aleatorio(x)	Recibe un número entero positivo x. Retorna un número aleatorio entre uno y x
Piso	piso(x)	Recibe un valor numérico x y lo devuelve redondeado hacia abajo al entero más cercano
Techo	techo(x)	Recibe un valor numérico x y lo devuelve redondeado hacia arriba al entero más cercano
Redondear	redondear(x)	Recibe un valor numérico x y lo devuelve redondeado al entero más cercano

Utilizaremos los procedimientos:

- **muestre():** Empleado para indicar el envío de datos (información) a un dispositivo de salida del computador y ser visualizados por el usuario. Muestra o imprime valores, constantes o variables.
- **lea():** Se utiliza para indicar que el usuario debe ingresar datos a un algoritmo por medio de algún dispositivo de entrada del computador. Los datos ingresados se deben asignar a las variables del algoritmo.

2.3. TIPOS DE DATOS

Es importante que al comenzar la etapa de diseño del algoritmo se dedique buena parte al análisis de los datos, y en este caso a la declaración del tipo de dato al cual pertenece cada variable (como se verá más adelante).

Los tipos de datos determinan el conjunto de valores que pueden tomar los objetos con los cuales operan los computadores y el conjunto de operaciones que se pueden aplicar a dichos valores.

Los tipo de datos se pueden clasificar en datos simples (entero, real, lógico y carácter) y datos compuestos (cadenas, arreglos, listas, árboles, grafos). En un tipo de dato simple solo se puede almacenar un valor a la vez, mientras que en los tipos datos compuestos se puede almacenar más de un valor a la vez pero del mismo tipo de dato simple.

Figura 2.1. Mapa conceptual para tipo de dato

2.3.1. *Dato numérico*

Se dice que un dato es de tipo numérico si está en un intervalo con valores numéricos. Ese conjunto puede estar formado por enteros o reales.

Tipo numérico entero: Si el valor del dato no tiene componentes fraccionarias o decimales se dice que es de tipo entero. Dichos valores pueden ser positivos, negativos o cero. Por ejemplo, la edad, la cantidad de personas, animales o cosas en algún sitio.

Tipo numérico real: Si el valor al cual representa el dato tiene componentes fraccionarias o decimales se dice que es de tipo real. Dichos valores pueden ser positivos, negativos o cero. Ejemplo, la estatura en metros, el peso en kilogramos.

2.3.2. *Dato lógico*

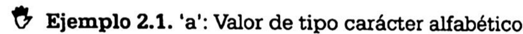
Se denomina dato de tipo lógico a aquel que puede tomar solo uno de dos valores, verdadero o falso. Este tipo de dato se utiliza para representar alternativas o decisiones y comparaciones, de las cuales se puede decir que es cierta o falsa.

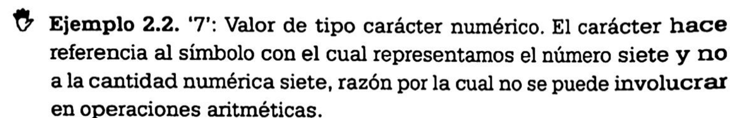
2.3.3. *Dato carácter*

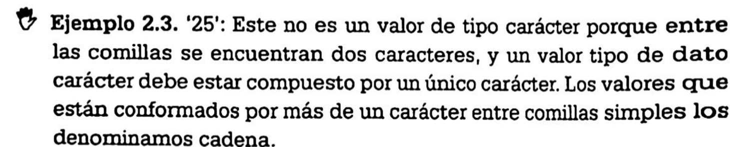
Hace referencia a cada uno de los símbolos que reconoce una computadora y que están dispuestos en el teclado. Se clasifican en

- Caracteres alfabéticos: a, b, c,..., z, A, B, C,..., Z
- Caracteres numéricos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Caracteres especiales: #, $, ', %, &, /, +, -, !, ?, \, *, ;, {,|, (,), <, >, ~,

Los caracteres los encerramos o delimitamos entre comillas simples. Las comillas no hacen parte del carácter y tienen como propósito evitar confundirlo con el nombre de una variable, constante o subalgoritmo.


Ejemplo 2.1. 'a': Valor de tipo carácter alfabético


Ejemplo 2.2. '7': Valor de tipo carácter numérico. El carácter hace referencia al símbolo con el cual representamos el número siete y no a la cantidad numérica siete, razón por la cual no se puede involucrar en operaciones aritméticas.


Ejemplo 2.3. '25': Este no es un valor de tipo carácter porque entre las comillas se encuentran dos caracteres, y un valor tipo de dato carácter debe estar compuesto por un único carácter. Los valores que están conformados por más de un carácter entre comillas simples los denominamos cadena.

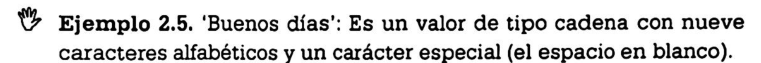
2.3.4. *Dato cadena*

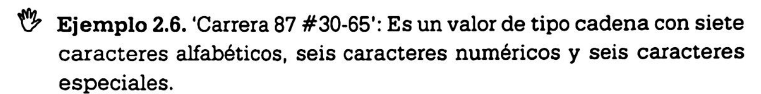
Es una secuencia finita de dos o más valores de tipo carácter, encerrados entre comillas simples. Estas comillas no hacen parte de la cadena, solamente la delimitan y nos permiten diferenciarla del nombre de una variable, constante o subalgoritmo.

Las cadenas se pueden reconocer también como arreglos de caracteres. En este punto se recomienda ir al siguiente enlace:

http://www.youtube.com/watch?v=5CBoxm _ L38Y

Ejemplo 2.4. 'María': Es un valor de tipo cadena con cinco caracteres alfabéticos.


Ejemplo 2.5. 'Buenos días': Es un valor de tipo cadena con nueve caracteres alfabéticos y un carácter especial (el espacio en blanco).


Ejemplo 2.6. 'Carrera 87 #30-65': Es un valor de tipo cadena con siete caracteres alfabéticos, seis caracteres numéricos y seis caracteres especiales.

2.3.5. *Dato arreglo*

Es una colección finita y ordenada de valores que pertenecen al mismo tipo de dato simple. En el capítulo cinco profundizaremos en el estudio de este tipo de dato.

Aquí no estudiaremos los tipos de datos compuestos dinámicos puesto que hacen parte de la asignatura Estructuras de Datos. Nos centraremos en los tipos de datos simples y los tipos de datos compuestos estáticos.

2.4. OPERADORES

Son aquellos símbolos que indican operaciones a realizar. Las operaciones pueden ser aritméticas, lógicas o asignación.

En la tabla 2.2 se listan los operadores aritméticos, relacionales y lógicos utilizados en el diseño de algoritmos. Allí se considera que X y Y son operandos de tipo numérico, y P y Q son proposiciones.

Una proposición es una afirmación que puede ser verdadera o falsa. La afirmación se puede expresar en lenguaje natural o mediante lenguaje matemático. En el diseño de algoritmos expresaremos las proposiciones mediante notación matemática.

Tabla 2.2. Operadores aritméticos y lógicos

Tipo de operador	*Nombre*	*Símbolo*	*Ejemplo*	*Lectura*
Aritmético	Suma	+	X + Y	X más Y
	Resta	-	X - Y	X menos Y
	Multiplicación	*	X * Y	X por Y
	División	/	X / Y	X dividido Y
	Potencia	^	X ^ Y	X elevado a la Y
	Módulo	mod	X mod Y	X módulo Y
	División entera	div	X div Y	X división entera Y
Relacional	Igual	==	X == Y	X igual a Y
	Diferente	<>	X <> Y	X diferente de Y
	Menor que	<	X < Y	X menor que Y
	Menor o igual que	<=	X <= Y	X menor o igual que Y
	Mayor que	>	X > Y	X mayor que Y
	Mayor o igual que	>=	X >= Y	X mayor o igual que Y
Lógico	Negación	~	~P	Negación de P
	Conjunción	y	P y Q	P y Q
	Disyunción	o	P o Q	P o Q

Cada operador tiene su correspondiente prioridad la cual hay que tener en cuenta en el momento de evaluar una expresión. La prioridad nos indica el orden en que se deben efectuar las operaciones cuando una expresión está compuesta por diferentes operadores. En la tabla 2.3 se listan los operadores ordenados de mayor a menor prioridad.

Tabla 2.3. Prioridad de operadores

Prioridad	*Operador*
1	( )
2	^
3	~, - (unarios)
4	*, /, mod, div
5	+, -
6	<,<=,>,>=
7	==, <>
8	y
9	o
10	=

Según la tabla 2.3, si una expresión contiene otras expresiones encerradas entre paréntesis, estas se evalúan primero, según su prioridad.

Asociatividad: Cuando en una expresión existen operadores con igual precedencia o prioridad, estos se evalúan de izquierda a derecha.

Operadores aritméticos

En el planteamiento de soluciones algorítmicas emplearemos los operadores básicos de la aritmética: suma, resta, multiplicación, división, potencia, módulo y división entera. Con los primeros cinco operadores aritméticos estamos familiarizados, pero no tanto con los operadores módulo (**mod**) y división entera (**div**). A continuación definiremos estos dos operadores y veremos cómo utilizarlos.

En este texto vamos a considerar que ambos operadores operan solo sobre números enteros y generan como resultado un número entero.

División entera: Es el cociente entero de dividir dos números enteros. Supongamos que A y B son dos números enteros, entonces

A **div** B = piso(A / B)

Donde la función *piso* indica que se toma el número entero menor más cercano al valor A / B.

Ejemplo 2.7. Evalúe la expresión

17 **div** 5

Solución:

17 div 5 = piso(17 / 5) = piso(3.4) = 3

Al dividir 17 entre 5 nos da como resultado 3.4 y tomamos el menor entero más cercano a este número. Por tanto, 17 div 5 da como resultado **3.**

Observe que la división entera indica el número de veces enteras o exactas que un número está en otro. El 5 está 3 veces en el 17.

Ejemplo 2.8. Evalúe la expresión

28 div 10

Solución:

28 div 10 = piso(28 / 10) = piso(2.8) = 2

El diez está dos veces en el veintiocho.

Ejemplo 2.9. Evalúe la expresión

12 div –7

Solución:

12 div –7 = piso(–12 / 7) = piso(–1.7143) = –2

El resultado de la división de los dos enteros es -1.7143 y el entero menor más cercano a él es –2. Así, 12 **div** -7 = –2

Módulo: Es el residuo entero de dividir dos números enteros. Supongamos que A y B son dos números enteros, entonces

A mod B = A – B * piso(A / B)

Corresponde al residuo de la división entera que es un entero no negativo, menor que el divisor.

Ejemplo 2.10. Evalúe la expresión

12 mod 5

Solución:

12 **mod** 5 = 12 – 5 * piso(12 / 5) = 12 – 5 * piso(2.4) =
12 – 5 * 2 = 12 – 10 = 2

El residuo de dividir 12 entre 5 en los enteros es 2.

Ejemplo 2.11. Evalúe las expresiones

4 mod 9 y 4 mod -9

Solución:

4 **mod** 9 = 4 – 9 * piso(4 / 9) = 4 – 9 * piso(0.4444)
= 4 – 9 * 0 = 4 – 0 = **4**

4 **mod** –9 =–4 **mod** 9= –4 –9*piso(–4 / 9)
=–4– 9 * piso(–0.4444)=–4 – 9 *(–1)
=–4+9= **5**

Operadores lógicos

En el diseño de algoritmos utilizaremos solo tres operadores lógicos: negación, conjunción y disyunción.

Negación: opera sobre un único valor de verdad; típicamente el valor de verdad de una proposición, devolviendo el valor de verdad verdadero si la proposición es falsa, y falso si la proposición es verdadera.

La conjunción: opera sobre dos valores de verdad, y evalúa al valor de verdad verdadero cuando ambas proposiciones son verdaderas, y falso en cualquier otro caso.

La disyunción: opera sobre dos valores de verdad, y evalúa al valor de verdad falso cuando ambas son falsas, y verdadero en cualquier otro caso.

Las tablas de verdad: Permiten determinar el valor de verdad de una proposición (enunciado que puede ser verdadero o falso) compuesta, para cada combinación de valores de verdad que se pueda asignar a sus componentes.

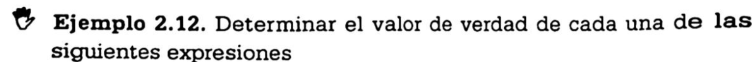
La tabla 2.4 y la tabla 2.5 resumen los valores de verdad de la disyunción (representada por **o** que representa el conector lógico **O**, en inglés OR), de la conjunción (representada por **y** que representa el conector lógico **Y**, en inglés AND) y la negación (representada por ~ que representa operador lógico **NO**, en inglés NOT)

Tabla 2.4. Tabla de verdad para la conjunción y la disyunción. (V: verdadero, F: Falso)

P	*Q*	*P y Q*	*P o Q*
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Tabla 2.5. Tabla de verdad del operador lógico negación

P	~P
V	F
F	V


Ejemplo 2.12. Determinar el valor de verdad de cada una de las siguientes expresiones

```
2 + 3 * 9 + 4 ^ 2 - 7 * 4 / 2
```

De acuerdo con la tabla 2.3 la operación que se debe efectuar en primer lugar es la exponenciación 4 ^ 2 que da como resultado 16. Así, obtenemos la siguiente expresión que es equivalente a la anterior y resulta de reemplazar 4 ^ 2 por 16.


```
2 + 3 * 9 + 16 - 7 * 4 / 2
```

Ahora, en la expresión tenemos como operadores de mayor prioridad a la multiplicación y a la división. Como estos dos operadores tienen el mismo nivel de prioridad entonces evaluamos estos operadores en el orden que aparezcan al recorrer la expresión de izquierda a derecha. Así, la segunda operación a realizar es la multiplicación 3 * 9 y obtenemos la expresión equivalente:

```
2 + 27 + 16 - 7 * 4 / 2
```

Luego, efectuamos el producto 7 * 4 para llegar a la expresión

```
2 + 27 + 16 - 28 / 2
```

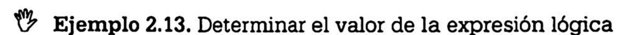
Donde efectuamos la división 28 / 2 y queda la expresión

```
2 + 27 + 16 - 14
```

En esta expresión equivalente solo están los operadores suma y resta. Estos operadores tienen el mismo nivel de prioridad y, por tanto, debemos evaluarlos de izquierda a derecha.

```
29 + 16 - 14
45 - 14
31
```

Entonces podemos concluir que la expresión aritmética 2 + 3 * 9 + 4 ^ 2 - 7 * 4 / 2 evalúa o es equivalente al valor 31.


Ejemplo 2.13. Determinar el valor de la expresión lógica

```
(A > B) o (B > A)
```

Considerando que A=10 y B=3.

Para evaluar la expresión primero debemos reemplazar las variables por su valor en la expresión dada:

(10 > 3) o (3 > 10)

De acuerdo con la tabla 2.3 las expresiones entre paréntesis son las que primero se deben evaluar, y las evaluamos de izquierda a derecha. Al evaluar la proposición del primer paréntesis 10 > 3 obtenemos como resultado el valor de verdad verdadero, ya que es cierto que diez es mayor que tres, y obtenemos

verdadero o (3 > 10)

Ahora, evaluamos la proposición 3 > 10 del ultimo paréntesis obtenemos el valor de verdad falso, puesto que tres no es mayor que diez. La expresión resultante es

verdadero o falso

El último operador en la expresión es la disyunción, y de acuerdo con la tabla 2.4 el resultado es verdadero. Observe que debido a que la primera proposición es verdadera no es necesario indagar por la segunda, y la proposición completa es verdadera.

Ejemplo 2.14. Determinar el valor de verdad las siguiente expresión

~((A >= B) o (B < A)) y (C + B > A)

Considerando que A= 23, B = 3 y C = 18.

Primero debemos reemplazar las variables por sus respectivos valores y a continuación tener en cuenta el orden de prioridad dado en la tabla 2.3. Para simplificar, indicaremos la operación a realizar subrayándola en cada expresión equivalente.

~((23 >= 3) o (3 < 23)) y (18 + 3 > 23)
~(verdadero o (3 < 23)) y (18 + 3 > 23)
~(verdadero o verdadero) y (18 + 3 > 23)
~verdadero y (18 + 3 > 23)
falso y (18 + 3 > 23)
falso y (21 > 23)
falso y falso
falso

Ejemplo 2.15. Determinar el valor de verdad en la siguiente expresión

A mod B mod 2 >= 1

Considerando que A=23 y B=3

Primero debemos reemplazar las variables por sus respectivos valores y a continuación tener en cuenta el orden de prioridad dado en la tabla 2.3.

23 mod 3 mod 2 >= 1
2 mod 2 >= 1
0 >= 1
falso

Ejemplo 2.16. Determinar el valor de verdad en la siguiente expresión

((A mod 10) > (B mod 8)) y ((C + B) div 10) < 8

Considerando que A=23, B=3 y C=18.

Primero debemos reemplazar las variables por sus respectivos valores y a continuación tener en cuenta el orden de prioridad dado en la tabla 2.3.

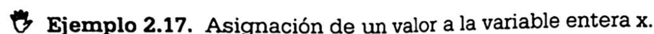
((23 **mod** 10) > (3 **mod** 8)) **y** ((18 + 3) **div** 10) < 8
(3 > (3 **mod** 8)) **y** ((18 + 3) **div** 10) < 8
(3 > 3) **y** ((18 + 3) **div** 10) < 8
falso **y** ((18 + 3) **div** 10) < 8
falso **y** (21 **div** 10) < 8
falso **y** 2 < 8
falso **y** falso
falso

OPERADOR DE ASIGNACIÓN

Es el símbolo con el cual se indica que se le está dando un valor a una variable o a una constante. El símbolo para esta operación será el carácter '='. La sintaxis de la operación de asignación es:

<variable o constante> = <Expresión>

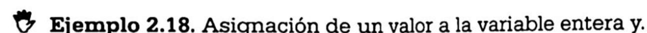
Al lado izquierdo del operador de asignación siempre debe aparecer el nombre de una variable o una constante, y en el lado derecho una expresión.



Ejemplo 2.17. Asignación de un valor a la variable entera x.

x = 23

significa que a la variable *x* se le está asignando el valor 23.



Ejemplo 2.18. Asignación de un valor a la variable entera y.

y = 3 + 7

Según la precedencia de operadores en la expresión el operador de mayor prioridad es la suma, y el de menor, la asignación. Por tanto, la variable *y* se le está asignando el valor 10.

El nombre correcto de este operador, en el desarrollo de algoritmos programación, es operador de asignación y no se debe confundir con el operador matemático de igualdad.

2.5. EXPRESIONES

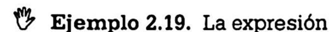
Combinación válida de valores, variables, constantes, operadores, paréntesis y funciones. La sintaxis básica es:

Operando 1 [operador] Operando 2

Clasificación de expresiones

Las expresiones pueden clasificarse en expresiones aritméticas y lógicas.

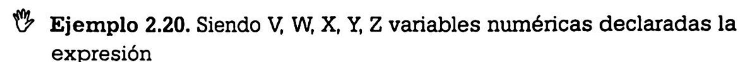
Expresión aritmética: Es un conjunto de variables o constantes numéricas separadas o no por operadores aritméticos. Este tipo de expresión evalúa a un valor numérico.



Ejemplo 2.19. La expresión

3 * 8 mod 5 + 24 / 2

evalúa al valor numérico 16.

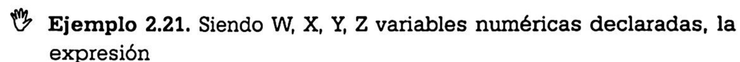


Ejemplo 2.20. Siendo V, W, X, Y, Z variables numéricas declaradas la expresión

(X - Y) * (W + Z) – V

evalúa a un valor numérico.

Expresión lógica: Es un conjunto de variables o constantes lógicas separadas o no por operadores lógicos o también expresiones aritméticas separadas por operadores relacionales. Este tipo de expresión evalúa a un valor lógico.



Ejemplo 2.21. Siendo W, X, Y, Z variables numéricas declaradas, la expresión

(X - Y) > (W + Z)

produce un valor lógico, debido a que el operador relacional *mayor que* es el último que se evalúa.

Ejemplo 2.22. Siendo P, Q, R, S variables lógicas declaradas, la expresión

((~ Q o R) y (P y ~ R)) o ~ S

evalúa a un valor lógico.

Ejemplo 2.23. Determinar el valor asignado a la variable **X en la** expresión

X = Y mod W + Z div W

si se sabe qué Y=8, Z=10, W=3

Solución: Debemos recordar que la operación **mod** entrega el **residuo** de una división entera, y la operación **div** entrega el cociente **de la** división entera.

Primero debemos reemplazar las variables por sus respectivos **valores** y a continuación tener en cuenta el orden de prioridad dado en la **tabla** 2.3. Las operaciones de mayor prioridad son el módulo y la **división** entera, y la operación de menor prioridad es la asignación. Como **el** modulo y la división entera tienen el mismo orden de prioridad **los** evaluamos izquierda a derecha.

X = 8 **mod** 3 + 10 **div** 3

X = 2 + 10 **div** 3

x = 2 + 3

x = 5

8 | 3	10 | 3
2 | 2	1 | 3
MOD DIV	MOD DIV

Figura 2.2. Representación alternativa de las operaciones módulo y división entera entre enteros positivos.

Ejercicio 2.1. De acuerdo a los valores de las variables A = 40 y B = **15**, determine en cada una de las siguientes expresiones **el** valor asignado a la variable X.

1. X = (A **mod** 5) / (B **div** 3)
2. X = (A + B) **mod** 5
3. X = (A - B) * 3 – (A + B)
4. X = (A - B) * 3 – A **div** B

CUESTIONARIO DE RECORDACIÓN

- ¿Qué entiende por precedencia de operadores?
- ¿Cuál (es) son los operadores que tienen mayor precedencia?
- ¿Cuál es la diferencia entre operando y operador?
- Para los siguientes cuatro ítems responda: ¿Cuál es el valor de la expresión, si a=8, b=5 y c=2?
  - ((a **mod** c) + (b **div** 2)) * c
  - (a + b) ^ c
  - a **mod** b **div** c * (a **div** c) + b
  - b + (a **div** (b **div** c)) * (a **mod** c) - c

  Recuerde que el ^2 = 9 esto es, tres al cuadrado.

2.6. CONCEPTO DE VARIABLE Y CONSTANTE

Cuando en la vida real se dice que "las cosas cambian o no con el tiempo", en lo concerniente a la lógica de programación podremos hablar de "variables" o "constantes", respectivamente.

Una variable es un campo de memoria al que puede cambiarse su contenido o valor durante el transcurso de la ejecución del algoritmo cuantas veces sea necesario, pero que en un momento determinado puede poseer el mismo valor. Una constante es aquel campo de memoria que nunca cambia de contenido o valor durante la ejecución del algoritmo; se dice entonces que es una "Constante". Una variable o una constante corresponden, respectivamente, a un espacio físico de memoria que cambia o no durante la ejecución del algoritmo.

La utilización de variables y constantes en el desarrollo de un algoritmo exige decir primero el tipo de dato (entero, real, carácter, etc.) que almacenarán para establecer los valores que se le pueden asignar y el conjunto de operaciones aplicables. La diferencia entre una variable y otra radica precisamente en su contenido y en el tipo de su contenido.

Para nombrar variables se debe seguir estas reglas:

1. El nombre debe ser mnemotécnico, es decir, que al leerlo fácilmente se entienda o al menos pueda intuirse su significado.

2. Evitar que sean muy cortos o muy largos.
3. El nombre no debe llevar caracteres especiales, excepto el guión bajo.
4. No usar la letra ñ o la tilde.
5. El nombre siempre debe empezar con un carácter alfabético en minúscula; pero si es compuesto, debe llevar la segunda palabra carácter alfabético en mayúscula (sin espacio en blanco o con guión bajo).

En la tabla 2.6 se indica la forma de nombrar variables, y en la tabla 2.7 la de nombrar constantes.

Tabla 2.6. Tipo de dato, variables y valores posibles

Tipo de dato	*Nombre variable*	*Valores posibles*
cadena	colorCabello	'Negro', 'castaño oscuro',' rubio', 'blanco'
lógico	valorDeVerdad	falso, verdadero
entero	edad	6, 7, 8, 23, 45
cadena	sexo	'Femenino', 'Masculino'
entero	estaturaPersona	160, 180, 190, 203
real	areaCirculo	23.93, 9.58

Para nombrar constantes se deben seguir estas reglas:

1. El nombre debe ser mnemotécnico, es decir, que al leerlo fácilmente se entienda o al menos se pueda intuir el significado del valor al cual hace referencia.
2. Evitar que sean muy cortos o muy largos.
3. El nombre debe ser en mayúscula sostenida.
4. El nombre siempre debe empezar con un carácter alfabético
5. Si el nombre está compuesto por varias palabras estas se deben de separar con el guión bajo.
6. El nombre no debe llevar caracteres especiales, excepto el guión bajo.
7. No usar la letra ñ o la tilde.

Tabla 2.7. Tipo de dato, constantes y valor

Tipo de dato	*Nombre constante*	*Valor posible*
entero	ALTURA_MINIMA	160
real	TEMPERATURA_MAXIMA	37.4
caracter	OPERACION_SUMA	'+'

2.7. ESTRUCTURA DE UN ALGORITMO EN SEUDOCÓDIGO

Para el diseño de algoritmos vamos a utilizar el lenguaje denominado seudocódigo y la siguiente estructura.

```
algoritmo <NombreDelAlgoritmo>
variables
   <tipoDeDato 1>:>Lista de variables 1>
   <tipoDeDato 1>:>Lista de variables 1>
      ⋮
      <tipoDeDato n>:>Lista de variables n>
inicio
   <instrucción 1>
   <instrucción 2>
      ⋮
   <instrucción 2>
fin
```

Figura 2.3. Estructura de un algoritmo especificado mediante seudocódigo

Las palabras reservadas siempre las escribiremos en minúscula sostenida y negrilla, y seguiremos la tabulación o indentación mostrada en la figura 2.3.

La palabra reservada **algoritmo** indica que el texto siguiente es la solución a un problema escrito en forma de algoritmo. El nombre dado al algoritmo debe seguir las reglas para nombrar variables. La sección de variables es usada para declarar las variables que se utilizarán en la solución; se debe indicar el tipo de dato y el nombre de cada variable. La

palabra reservada **inicio** señala el punto a partir del cual se empezarán a especificar los pasos que llevarán a la solución del problema, y la palabra **fin** señala la culminación de los pasos y que, por consiguiente, se ha resuelto el problema. Entre las etiquetas **inicio** y **fin** se escriben todos los pasos, en forma de instrucciones, con los cuales resolveremos el problema.

Ejemplo 2.24. Declaración de variables

```
1.  algoritmo AsignacionValores
2.  variables
3.      entero: a, b, c
4.  inicio
5.      a = 10
6.      b = 3
7.      c = (a + b) mod 2
8.      a = c + b
9.  fin
```

En la línea cinco del algoritmo AsignacionValores, a la variable *a* se le asigna el valor 10 y, luego, en la línea ocho, se le asigna el valor resultante de sumar c + b, es decir, el valor 4. Una vez se asigna este último valor a la variable *a*, el valor que estaba antes desaparece y es reemplazado por el nuevo. Durante la ejecución del algoritmo el valor almacenado en la variable *a* cambia (varia).

Ejemplo 2.25. Diseñe un algoritmo que permita sumar dos números.

```
1.  algoritmo SumarDosNumeros
2.  variables
3.      real: a, b, c
4.  inicio
5.      muestre('SUMAR DOS NUMEROS')
6.      muestre('Entre el primer número:')
7.      lea(a)
8.      muestre('Entre el segundo número:')
9.      lea(b)
10.     c = a + b
11.     muestre('La suma es:', c)
12. fin
```


Cuando se ejecuta el algoritmo, en pantalla (dispositivo de salida) solo se ven los mensajes que se envíen con el procedimiento *muestre* y el computador espera a que una persona (usuario) ingrese un valor mediante el teclado cada vez que se ejecute el procedimiento *lea*. Las instrucciones que están entre las palabras reservadas **inicio** y **fin** producen los efectos mostrados en la siguiente figura.

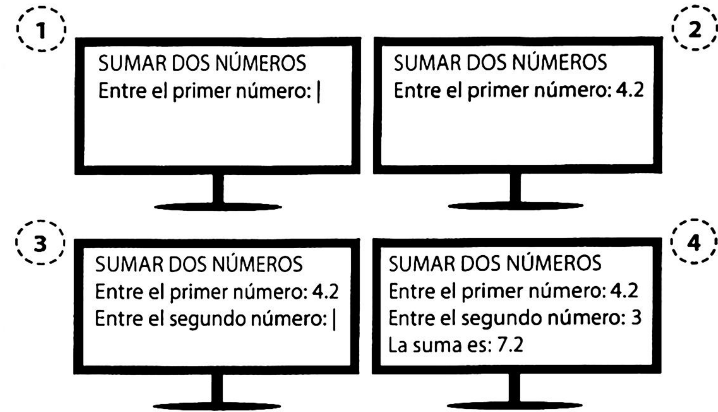


Figura 2.4. Resultado de ejecutar el algoritmo del ejemplo 2.25

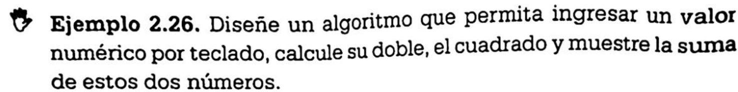
A continuación se explican dichos efectos:

1. En el monitor se observan los mensajes "SUMAR DOS NÚMEROS" y "Entre el primer número", porque el procedimiento *muestre* los envió. Solo se muestran esos dos mensajes porque luego se ejecuta el procedimiento *lea* para poder asignarle un valor a la variable *a*, y dicho procedimiento hace que el computador espere hasta que se ingrese un valor y se presione la tecla Enter.

2. El usuario ingresa un valor numérico (4.2) y a continuación presiona la tecla Enter. De esta manera a la variable *a* se le asigna el número 4.2

3. Se muestra el mensaje "Entre el segundo número: " y el algoritmo se detiene hasta que el usuario ingrese un número y presione la tecla Enter.

4. El usuario ingresa el número 3 y a continuación presiona la tecla Enter. La variable *b* toma el valor de 3. Luego, el algoritmo calcula la suma de los valores almacenados en las variables *a* y *b* y asigna el resultado a la variable *c*. Observe que el cálculo de la suma y la asignación del resultado a la variable *c* no se perciben por la pantalla. El usuario desconoce el proceso mediante el cual se obtuvo el resultado, incluso desconoce el nombre de las variables. Por último, se muestra el mensaje que indica el resultado de la suma y termina el algoritmo.



Ejemplo 2.26. Diseñe un algoritmo que permita ingresar un valor numérico por teclado, calcule su doble, el cuadrado y muestre la suma de estos dos números.

```
1.  algoritmo CalcularDobleYCuadrado
2.  variables
3.      real: numero, doble, cuadrado, suma
4.  inicio
5.      muestre('Entre un número:')
6.      lea(numero)
7.      doble = 2 * numero
8.      cuadrado = numero * numero
9.      suma = doble + cuadrado
10.     muestre('La suma de su doble más su cuadrado es', 
        suma)
11. fin
```

Tabla 2.8. Explicación de los pasos del algoritmo del ejemplo 2.26

Línea	*Discusión*
5	La instrucción muestre le indica al usuario que debe ingresar un número; esta instrucción sirve de guía para el usuario
6	El número que el usuario ingresa por teclado se asigna a la variable número
7	A la variable doble se le asigna el resultado de multiplicar la variable numero por dos
8	A la variable cuadrado se le asigna el resultado de multiplicar la variable número por sí misma
9	A la variable suma se le asigna el resultado de sumar las variables doble y cuadrado
10	Se le muestra al usuario el resultado de la suma realizada en el paso anterior

Ejemplo 2.27. Diseñe un algoritmo que calcule el área de un triángulo.

```
1.  algoritmo Área triángulo
2.  variables
3.      real: base, altura, area
4.  inicio
5.      muestre('CALCULAR EL ÁREA DE UN TRIÁNGULO')
6.      muestre('Entre la base:')
7.      lea(base)
8.      muestre('Entre la altura:')
9.      lea(altura)
10.     area = base * altura / 2
11.     muestre('El área es:', area)
12. fin
```

2.8. PRUEBA DE ESCRITORIO

Es una herramienta útil para entender qué hace un algoritmo o para verificar que un algoritmo soluciona una problema correctamente sin necesidad de escribir el programa equivalente y ejecutarlo.

Para hacer una prueba de escritorio se deben listar en columnas independientes cada una de las variables presentes en un algoritmo y a medida que se recorren los pasos del algoritmo se listan en la columna correspondiente los valores que toman las variables.

Ejemplo 2.28. Prueba de escritorio

```
1.  algoritmo EjemploPruebaEscritorio
2.  variables
3.      entero: a, b, c
4.  inicio
5.      a = 5
6.      b = 6
7.      c = (a + b) * 10 mod 4
8.      b = 15 div a
9.      muestre(a, b, c)
10. fin
```

Solución: Para hacer la prueba de escritorio iniciamos listando los nombres de las variables del algoritmo 2.28 en columnas independientes.

a	b	c
5	6	2
	3	

En la segunda fila están listados los valores que se asignan por primera vez a las variables cuyos nombres están en la primera fila. En la tercera fila solo está el valor para la variable *b*, esto significa que durante el algoritmo a las variables *a* y *c* solo se les asignó un valor mientras que a la variable *b* se le asignaron dos valores, primero el 6 y luego el tres, y ahora solo conserva el último.

Ejemplo 2.29. La empresa CLV requiere un programa que permita calcular el salario semanal de un empleado. La empresa paga a sus empleados $18.000 pesos por hora y le descuenta un 8 % por pagos a salud sobre el salario base; le hace, además, un descuento del 5 % por retención en la fuente sobre el salario después de descontar la salud.

```
Algoritmo:
1.  algoritmo SalarioNeto
2.  variables
3.      entero: horasLaboradas
4.      real: reteFuente, salud, salario
5.  inicio
6.      muestre('Ingrese el número de horas laboradas:')
7.      lea(horasLaboradas)
8.      salario = 18000 * horasLaboradas
9.      salud = 0.08 * salario
10.     salario = salario - salud
11.     reteFuente = 0.05 * salario
12.     salario = salario - reteFuente
13.     muestre('Salario neto:', salario)
14. fin
```


Ejercicio 2.2.

1. Escriba las siguientes expresiones algebraicas en forma de expresiones aritméticas en seudocódigo:
2. Diseñe un algoritmo que calcule el área y el perímetro de un rectángulo.
3. Diseñe un algoritmo que calcule el área y el perímetro de un trapecio.
4. Diseñe un algoritmo que lea un número y escriba su cuadrado.
5. Diseñe un algoritmo que calcule el área superficial y el volumen de un cilindro.
6. Diseñe un algoritmo que calcule el volumen de un cubo.
7. Diseñe un algoritmo que calcule el volumen de una esfera.
8. Diseñe un algoritmo que calcule el volumen de una pirámide.
9. Diseñe un algoritmo que lea el precio de un artículo y muestre el resultado de restarle al precio el 15 %.
10. Diseñe un algoritmo que lea el precio de un artículo y el porcentaje de descuento que tiene el artículo. Muestre el nuevo precio y el descuento.
11. Diseñe un algoritmo que convierta una temperatura leída en grados Celsius a grados Fahrenheit.

CUESTIONARIO DE RECORDACIÓN 2.2

1. ¿Qué es una prueba de escritorio?
2. ¿Para qué se realiza una prueba de escritorio?
3. ¿Cómo se lleva a cabo una prueba de escritorio?
4. ¿En qué consiste la operación de asignación?
5. ¿Cuál es la diferencia entre una variable entera y una real?

Capítulo 3

Estructuras de programación

3.1. PRESENTACIÓN

La solución a un problema se puede dividir en acciones elementales o instrucciones, usando un número limitado de estructuras de control básicas y sus posibles combinaciones.

El trabajo de investigación de Bohm & Jacopini demostró que todos los programas podían ser escritos con tres estructuras básicas de control:

- Secuenciales: las instrucciones en un algoritmo se ejecutan en el orden en que se escriben, de arriba hacia abajo.
- Selección o decisión: permiten determinar si una secuencia de instrucciones se ejecutan o no dependiendo del valor de verdad de una expresión lógica.
- Repetición: permiten que una misma secuencia de instrucciones se ejecuten varias veces siempre y cuando se cumpla una condición planteada como expresión lógica.

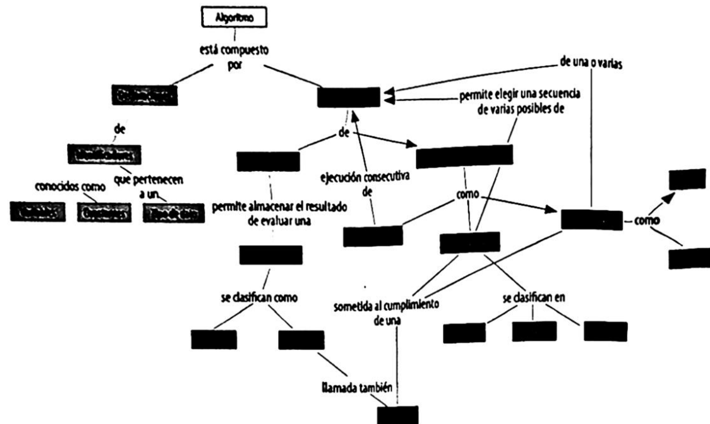


Figura 3.1. Mapa conceptual estructuras de repetición

3.2. ESTRUCTURAS DE SELECCIÓN

Esta estructura depende de una decisión lógica, llamada condici de la cual surgen dos posibles alternativas de secuencia: una de e cuando al evaluarse la condición es verdadera y la otra cuando es fa Finalmente, se unifican las salidas de las alternativas, cumpliendo así la característica de cada estructura: producir una sola salida. En el ar de la programación esta estructura es más conocida como "Condiciona

3.2.1. *Estructura de selección simple*

Esta estructura se encarga de evaluar una expresión lógica llam condición y en caso de verificarse (ser verdadera) realiza unas accione y en caso contrario se ignoran esas acciones. Es decir, esta estructura utiliza para determinar si una secuencia de instrucciones se debe ejec o no durante la ejecución de un algoritmo.

Sintaxis de la estructura condicional simple en seudocódigo:

```
si <expresionLogica> entonces
    <Instrucciones>
fin_si
```


Esta estructura usa las palabras reservadas **si**, **entonces** y **fin_si**. Si la expresión lógica o condición <expresionLogica> evalúa a verdadero entonces las instrucciones que se encuentran entre las palabras reservadas **entonces** y **fin_si** se ejecutan. Si la expresión lógica evalúa a falso, el algoritmo salta a la palabra reservada **fin_si** y las instrucciones <Instrucciones> son ignoradas. Sin importar cuál sea el caso, el algoritmo continúa ejecutando las instrucciones que se encuentren después del **fin_si**. El algoritmo sigue el flujo secuencial de ejecución.

Ejemplo 3.1. Diseñe un algoritmo que calcule la raíz cuadrada de un número positivo.

Solución:

```
1.  algoritmo RaizCuadrada
2.  variables
3.      real: numero, r
4.  inicio
5.      muestre('CALCULAR RAÍZ CUADRADA DE UN NÚMERO
        NO NEGATIVO')
6.      muestre('Entre un número:')
7.      lea(numero)
8.      si numero >= 0 entonces
9.          r = raiz2(numero)
10.         muestre('La raíz cuadrada es:', r)
11.     fin_si
12. fin
```

En la línea 7 del algoritmo RaizCuadrada el usuario asigna un valor a la variable *numero* mediante el teclado. Como la variable es real y no controlamos el valor que el usuario ingrese, decidimos utilizar una estructura de selección y verificar primero si el valor ingresado es no negativo. Alguien puede decidir ingresar un número negativo y producir un error porque los números negativos no tienen raíz cuadrada definida en los números reales. Así, con la estructura de selección de la línea 8 garantizamos que solo calcularemos la raíz cuadrada del número ingresado, y mostraremos el resultado de la operación, si es mayor o igual a cero. En caso contrario, el algoritmo salta al **fin_si**, y termina el algoritmo sin mostrar ningún resultado o mensaje.

3.2.2. *Estructura de selección doble*

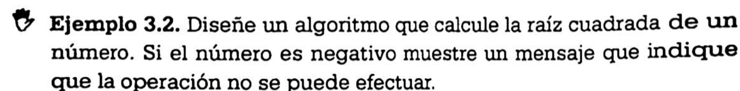
Esta estructura es similar a la de selección simple. La única diferencia es que incorpora una palabra reservada para contemplar el caso en que la condición evalúa a falso. Es decir, si la expresión lógica evalúa a verdadero se ejecuta una secuencia de instrucciones, y si evalúa a falso se ejecuta otro conjunto de instrucciones diferente.

Sintaxis de la estructura condicional doble en seudocódigo:

```
si <expresionLogica> entonces
    <Instrucciones 1>
si_no
    <Instrucciones 2>
fin_si
```

Esta estructura usa las palabras reservadas **si**, **entonces**, **si_no** y **fin_si**. Si la expresión lógica <expresionLogica> evalúa a verdadero entonces solo se ejecutan las instrucciones que se encuentran entre las palabras reservadas **entonces** y **si_no**. Si la expresión lógica evalúa a falso solo se ejecutan las instrucciones que se encuentran entre las etiquetas **si_no** y **fin_si**. Sin importar cuál sea el valor de la condición, el algoritmo continuará ejecutando las instrucciones que se encuentren después del **fin_si**.

Es claro que la estructura de selección doble permite determinar cuál secuencia de instrucciones, de dos posibles, debe ejecutarse. Se ejecuta una secuencia o la otra, pero nunca se ejecutaran las dos secuencialmente durante el desarrollo de un algoritmo.



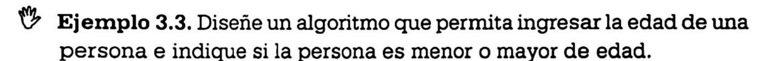
Ejemplo 3.2. Diseñe un algoritmo que calcule la raíz cuadrada de un número. Si el número es negativo muestre un mensaje que indique que la operación no se puede efectuar.

```
1.  algoritmo RaizCuadrada2
2.  variables
3.      real: numero, r
4.  inicio
5.      muestre('CALCULAR RAÍZ CUADRADA')
```



```
6.      muestre('Entre un número: ')
7.      lea(numero)
8.      si numero >= 0 entonces
9.          r = raiz2(numero)
10.         muestre('La raíz cuadrada es: ', r)
11.     si_no
12.         muestre('Error: En los reales no hay raíz cuadrada
            de números negativos')
13.     fin_si
14. Fin
```

Si el número que se ingresa en la línea 7 es mayor o igual a cero la condición de la estructura de selección de la línea 8 evalúa a verdadero y se ejecutan las instrucciones de las líneas 9 y 10 pero si la condición evalúa a falso solo se ejecuta la instrucción de la línea 12 para mostrar el mensaje de error. Para un valor dado de la variable *numero*, solo se ejecutará una de las dos secuencias de instrucciones. Cuál se ejecuta depende del valor de verdad de la condición.



Ejemplo 3.3. Diseñe un algoritmo que permita ingresar la edad de una persona e indique si la persona es menor o mayor de edad.

Solución: En nuestro país se considera que una persona es mayor de edad si tiene 18 años o más, y es menor de edad en caso contrario. Por tal motivo, necesitamos una variable de tipo entero para almacenar la edad ingresada por el usuario y posteriormente comparar su contenido con el valor 18. Si el valor almacenado en la variable edad es mayor o igual que 18 el usuario es mayor de edad.

```
1.  algoritmo MenorMayorDeEdad
2.  variables
3.      entero: edad
4.  inicio
5.      muestre('Entre su edad:')
6.      lea(edad)
7.      si edad >= 18 entonces
8.          muestre('Usted es mayor de edad')
```

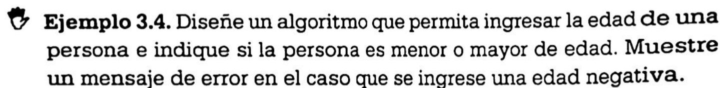
```
9.      si_no
10.         muestre('Usted es menor de edad')
11.     fin_si
12. fin
```

¿Qué sucede si en el algoritmo MenorMayorDeEdad se ingresa un valor negativo? En este caso el algoritmo dará una respuesta incorrecta, porque al evaluar la condición dará como resultado el valor de verdad falso y, por tanto, se mostrará el mensaje que indica que es menor de edad. Como no existen edades negativas y un algoritmo siempre debe dar una respuesta correcta para todos los posibles casos debemos controlar el evento en que alguien asigne un valor negativo a la variable edad en la línea 6. ¿Cómo controlar ese evento? Lo mostraremos en la siguiente sección.

3.2.3. *Estructuras de selección anidadas*

Las estructuras de selección anidadas permiten elegir, de varias secuencias de instrucciones posibles, cuál secuencia se debe ejecutar.

Este tipo de estructura se crea escribiendo estructuras de selección simple o doble dentro de estructuras de selección simple o doble. Esta anidación se puede hacer tantas veces como sea necesario.



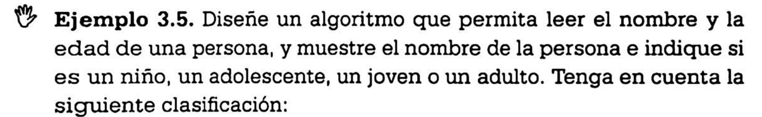
Ejemplo 3.4. Diseñe un algoritmo que permita ingresar la edad de una persona e indique si la persona es menor o mayor de edad. Muestre un mensaje de error en el caso que se ingrese una edad negativa.

```
1.  algoritmo MenorMayorDeEdad2
2.  variables
3.      entero: edad
4.  inicio
5.      muestre('Entre su edad:')
6.      lea(edad)
7.      si edad >= 0 entonces
8.          si edad >= 18 entonces
9.              muestre('Usted es mayor de edad')
10.         si_no
```



```
11.             muestre('Usted es menor de edad')
12.         fin_si
13.     si_no
14.         muestre('Error: No hay edades negativas')
15.     fin_si
16. fin
```

En la línea 7 primero se verifica si el valor asignado a la variable edad en la línea 6 es mayor o igual a cero. Solo si la edad es mayor igual a cero tiene sentido averiguar si la persona es menor o mayor de edad; por eso, si la condición evalúa a verdadero preguntamos si la edad es mayor o igual a 18 y así mostrar el mensaje apropiado como lo hicimos en el ejemplo 3.3. Si la condición de la línea 7 evalúa a falso significa que se ingresó un número negativo y, por tanto, mostramos el mensaje de error de la línea 14. Observe que cuando la condición de la línea 7 evalúa a falso jamás se evaluará la condición de línea 8.



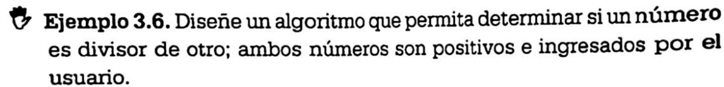
Ejemplo 3.5. Diseñe un algoritmo que permita leer el nombre y la edad de una persona, y muestre el nombre de la persona e indique si es un niño, un adolescente, un joven o un adulto. Tenga en cuenta la siguiente clasificación:

Niño:	Menor que 14 años
Adolescente:	Entre 15 y 17 años
Joven:	De 18 a 25 años
Adulto:	Mayor de 25 años.

Solución: Después de leer la edad ésta debe compararse con los valores dados en los diferentes rangos y, según el caso, mostrar el mensaje correspondiente y el nombre de la persona.

```
Algoritmo:
1.  algoritmo RangoEdades
2.  variables
3.      entero: edad
4.      cadena: nombre
5.  inicio
```

```
6.      muestre ('Ingrese su nombre:')
7.      lea (nombre)
8.      muestre ('Ingrese su edad:')
9.      lea (edad)
10.     si edad <= 14 entonces
11.         muestre (nombre, 'eres un niño')
12.     si_no
13.         si edad < 18 entonces
14.             muestre (nombre, 'eres un adolescente')
15.         si_no
16.             si edad < 25 entonces
17.                 muestre (nombre, 'eres un joven')
18.             si_no
19.                 muestre (nombre, 'eres un adulto')
20.             fin_si
21.         fin_si
22.     fin_si
23. fin
```



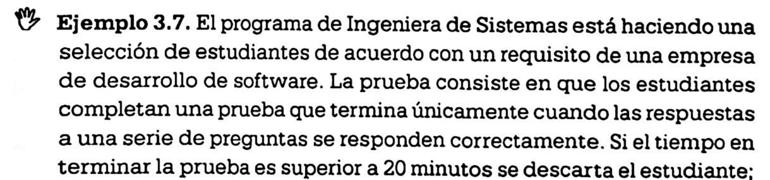
Ejemplo 3.6. Diseñe un algoritmo que permita determinar si un número es divisor de otro; ambos números son positivos e ingresados por el usuario.

Análisis: Debemos entender lo que se pide, ¿Qué significa entonces que un número sea divisor de otro? Sabemos que 2 es divisor de 14, ¿Qué condición cumplen estos números? Si dividimos a 14 entre 2 obtenemos como resultado 7 y ¿cuánto sobra? o sea, ¿Cuál es el resultado de aplicar la operación módulo a 14 y a 2?, 14 **mod** 2 = 0, esto es, el residuo de dividir 14 entre 2 es cero. Cuando aplicamos la operación módulo a dos números, si el resultado es cero significa que el segundo número es divisor del primer número o que el primero es múltiplo del segundo, entonces debemos aplicar la operación módulo para saber si un número divide a otro.

Para el caso presente el usuario ingresa los dos números, luego determinamos cuál de ellos es el mayor y finalmente preguntamos el mayor es divisible por el menor. La secuencia a seguir será:

1. Leer los dos números
2. Determinar cuál es el mayor
3. Preguntar si el mayor es divisible por el menor
4. En caso afirmativo si es divisor
5. En caso contrario no es divisor

```
Algoritmo:
1.  algoritmo Divisor
2.  variables
3.      entero: numero1, numero2, menor, mayor
4.  inicio
5.      muestre('Ingrese el primer número:')
6.      lea(numero1)
7.      muestre('Ingrese el segundo número:')
8.      lea(numero2)
9.      si numero1 > numero2 entonces
10.         mayor = numero1
11.         menor = numero2
12.     si_no
13.         mayor = numero2
14.         menor = numero1
15.     fin_si
16.     si mayor mod menor == 0 entonces
17.         muestre('El número', menor, 'es divisor de', mayor)
18.     si_no
19.         muestre('El número', menor, 'no es divisor de', mayor)
20.     fin_si
21. fin
```

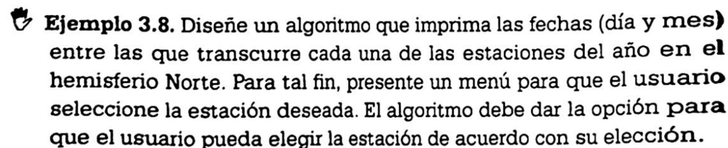


Ejemplo 3.7. El programa de Ingeniera de Sistemas está haciendo una selección de estudiantes de acuerdo con un requisito de una empresa de desarrollo de software. La prueba consiste en que los estudiantes completan una prueba que termina únicamente cuando las respuestas a una serie de preguntas se responden correctamente. Si el tiempo en terminar la prueba es superior a 20 minutos se descarta el estudiante;

si es menor a 15 minutos es aceptado, y si el tiempo está entre 15 y 20 minutos se la hace una segunda prueba, y se toma el promedio de las dos pruebas; si el promedio es menor a 17 minutos se acepta; en caso contrario, se rechaza.

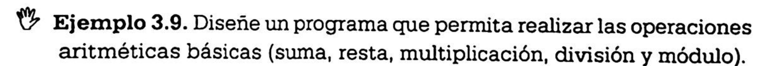
Solución:

```
1.  algoritmo SeleccionEstudiantes
2.  variables
3.      entero: tiempoPrimeraPrueba, tiempoSegundaPrueba
4.      real: promedio
5.  inicio
6.      muestre('Ingrese el tiempo en que respondió la primera prueba:')
7.      lea(tiempoPrimeraPrueba)
8.      si tiempoPrimeraPrueba < 20 entonces
9.          si tiempoPrimeraPrueba < 15 entonces
10.             muestre('Estudiante aceptado')
11.         si_no
12.             muestre('Ingrese el tiempo en que respondió la segunda
        prueba:')
13.             lea(tiempoSegundaPrueba)
14.             promedio = (tiempoPrimeraPrueba + tiempoSegundaPrueba)
                / 2
15.             si promedio < 17 entonces
16.                 muestre('Estudiante aceptado')
17.             si_no
18.                 muestre('Estudiante descartado')
19.             fin_si
20.         fin_si
21.     si_no
22.         muestre('Estudiante descartado')
23.     fin_si
24. fin
```


Ejemplo 3.8. Diseñe un algoritmo que imprima las fechas (día y mes) entre las que transcurre cada una de las estaciones del año en el hemisferio Norte. Para tal fin, presente un menú para que el usuario seleccione la estación deseada. El algoritmo debe dar la opción para que el usuario pueda elegir la estación de acuerdo con su elección.

Solución:

```
1.  algoritmo Estaciones
2.  variables
3.      entero: opcion
4.  inicio
5.      muestre ('MENU')
6.      muestre ('1. Primavera')
7.      muestre ('2. Verano')
8.      muestre ('3. Otoño')
9.      muestre ('4. Invierno')
10.     muestre ('Ingrese el número de la opción deseada:')
11.     lea (opcion)
12.     si opcion == 1entonces
13.         muestre ('22 de marzo hasta 21 de junio')
14.     si_no
15.         si opcion == 2 entonces
16.             muestre ('22 de junio hasta 21 de agosto')
17.         si_no
18.             si opcion == 3 entonces
19.                 muestre ('22 de septiembre hasta 21 de diciembre')
20.             si_no
21.                 si opcion == 4 entonces
22.                     muestre ('22 de diciembre hasta 21 de marzo')
23.                 si_no
24.                     muestre ('Opción no valida')
25.                 fin_si
26.             fin_si
27.         fin_si
28.     fin_si
29. fin
```


Ejemplo 3.9. Diseñe un programa que permita realizar las operaciones aritméticas básicas (suma, resta, multiplicación, división y módulo).

Solución:

Análisis del problema: En este caso el usuario ingresa dos números y de acuerdo con el objetivo del problema debe realizar la operación

aritmética seleccionada. El usuario ingresará el número uno para la suma, dos para la resta, tres para la multiplicación, cuatro para la división y cinco para el módulo.

Variables

- Variables de entrada: operación, numero1, numero2
- Variable de salida: resultado

Macroalgoritmo

- Leer numero1, numero2 y operación.
- Comparar el valor de la variable *operación*
- De acuerdo con el valor de la variable operación seleccionada realizar la operación
- Mostrar el resultado

```
Algoritmo:
1.  algoritmo OperacionesAritmeticaBasicas
2.  variables
3.    entero: numero1, numero2, operacion
4.    real: resultado
5.  inicio
6.    muestre('Ingrese el primer número:')
7.    lea(numero1)
8.    muestre('Ingrese el segundo número:')
9.    lea(numero2)
10.   muestre ('1. OPERACIONES ARITMÉTICAS BÁSICAS')
11.   muestre ('1. Suma')
12.   muestre ('2. Resta')
13.   muestre ('3. Multiplicación')
14.   muestre ('4. División')
15.   muestre ('5. Módulo')
16.   muestre('Ingrese el número de la operación aritmética básica:')
17.   lea(operacion)
18.   si operación == 1 entonces
19.     resultado = numero1 + numero2
20.   si_no
21.     si operación == 2 entonces
22.       resultado = numero1 - numero2
23.     si_no
```



```
24.       si operación == 3 entonces
25.         resultado = numero1 * numero2
26.       si_no
27.         si operación == 4 entonces
28.           si numero2 <> 0 entonces
29.             resultado = numero1 / numero2
30.           si_no
31.             muestre('Error: División por cero')
32.           fin_si
33.         si_no
34.           si operación == 5 entonces
35.             resultado = numero1 mod numero2
36.           si_no
37.             muestre('Error: Operación inválida')
38.           fin_si
39.         fin_si
40.       fin_si
41.     fin_si
42.   fin_si
43.   muestre('El resultado es:', resultado)
44. fin
```

Ejemplo 3.10. En la Universidad de Medellín la evaluación de los cursos de pregrado está compuesta por dos seguimientos, un parcial y un examen final. Cada uno con un valor del 25 %.

El estudiante cuyo acumulado al momento de tener registrado el 75 % de su evaluación sea igual o superior a 3.15, es decir, una nota mínima de 4.2, quedará eximido del examen final, en cuyo caso la nota final de la asignatura será igual al acumulado del 75 % registrado; alternativamente quien no deseare optar por la eximición lo podrá hacer previa solicitud por escrito dirigida a la Decanatura o Jefatura de Programa (Acuerdo 72 de 15 de julio de 2013, Reforma al Reglamento Académico y Disciplinario).

Diseñe un algoritmo que lea las notas de un estudiante de la Universidad de Medellín correspondientes a los dos seguimientos y al parcial del curso de Fundamentos de Programación y muestre la nota acumulada actual y la nota mínima que debe obtener en el examen

final para aprobar el curso. En caso que el estudiante no necesite nota para el final muestre un mensaje que indique que el estudiante ya aprobó el curso.

```
Algoritmo:
1.  algoritmo NotaEnExamenFinal
2.  variables
3.      real: seguimiento1, seguimiento2, parcial, final, notaAcumulada,
    definitiva
4.  inicio
5.      muestre('CALCULAR NOTA DE ESTUDIANTE PARA EL EXAMEN FINAL')
6.      muestre('Ingrese la nota del primer seguimiento:')
7.      lea(seguimiento1)
8.      muestre('Ingrese la nota del parcial:')
9.      lea(parcial)
10.     muestre('Ingrese la nota del segundo seguimiento:')
11.     lea(seguimiento2)
12.     notaAcumulada = (seguimiento1 + parcial + seguimiento2) / 4
13.     muestre('La nota acumulada es:', notaAcumulada)
14.     si notaAcumulada >= 3.15 entonces
15.         muestre('Está eximido de presentar el examen final. ¡Felicitaciones!')
16.         definitiva = (seguimiento1 + parcial + seguimiento2) / 3
17.         muestre('Su nota definitiva es:', definitiva)
18.     si_no
19.         si notaAcumulada >= 3.0 entonces
20.             muestre('Ya aprobó el curso. Felicitaciones!')
21.             muestre('Pero no está eximido de presentar el examen final.')
22.         si_no
23.             final = 4 * (3.0 – notaAcumulada)
24.             muestre('En el final debe obtener una nota mínima de',
    final, 'para aprobar el curso.')
25.         fin_si
26.     fin_si
27. fin
```


CUESTIONARIO DE RECORDACIÓN 3.1

- ¿Cuántas posibles salidas tiene una estructura de selección?
- ¿La instrucción muestre es de salida o de ingreso de información?

PREGUNTAS FRECUENTES

– ¿Cuál es la diferencia entre variable y constante?

Respuesta: Una variable puede cambiar su contenido durante la ejecución de un algoritmo o programa, la constante mantiene su valor.

– ¿Se puede declarar una variable de tipo simple que almacene simultáneamente un valor entero y un valor real?

Respuesta: No. Las variables simples solamente pueden almacenar una a la vez.

– ¿Es posible reemplazar una estructura de selección múltiple por condicionales anidados?

Respuesta: Sí. Es posible hacer el reemplazo por los condicionales anidados que también permiten establecer varias opciones de flujo del algoritmo.

– ¿Es obligatorio que cada sentencia condicional **si** ... **entonces** tenga la sentencia **si_no**?

Respuesta: No. Hay casos en los que solo se requiere preguntar si se cumple la condición, pero no interesa conocer el caso contrario.

– ¿Cuál es el tipo de dato que ocupa más espacio en la memoria de un computador?

Respuesta: los datos tipo real, dentro de los tipos de datos simples, y los arreglos bidimensionales, dentro de los tipos de datos compuestos estáticos.

ACTIVIDADES EVALUATIVAS 3.1

CUESTIONARIO EN LÍNEA 3.1

Nombre: variables, constantes y expresiones

Modalidad: individual

Descripción de la actividad: Siga los pasos indicados en el fragmento de algoritmo dado abajo y responda las preguntas de selección múltiple con única respuesta.

Fragmento de algoritmo:

```
inicio
x = -6
y = 10
z = (y - x) * 3 mod 7
y = z div 10
x = y + z
fin
```

Preguntas de selección múltiple con única respuesta

1. El valor de x al final de este algoritmo es:
   a. 7
   b. 16
   c. 6
   d. 8

2. El valor de z al final de este algoritmo es:
   a. 6
   b. 10
   c. –8
   d. –6

3. El valor de y al final de este algoritmo es:
   a. 0
   b. 6
   c. 8
   d. 10

4. ¿Cuántas variables intervienen en el algoritmo anterior?
   a. 3
   b. 2
   c. 4
   d. 5

5. ¿Cuál sería el tipo idónea que deberían tener las variables?
   a. Real
   b. Lógico
   c. Entero
   d. Cadena

6. ¿Qué valores puede entregar la operación **mod** 10?
   a. Valores enteros menores a 10
   b. Valores reales positivos
   c. Enteros entre 10 y 20
   d. Valores entre 0 y 5 reales o enteros

7. Indique los valores de a, b, c en el siguiente segmento de algoritmos.

```
1.  inicio
2.   a = 2
3.   b = 0
4.   c = 0
5.   b = a^3 + (b+1)^2
6.   si (c > 0) o (b > 10) entonces
7.     a = a * 10 + 8
8.     b = a * 2
9.   fin_si
10. fin
```

   ¿Es posible que algunas de las instrucciones del anterior algoritmo no se ejecuten?

8. En caso que en el ítem anterior responda afirmativamente especifique las líneas que no se ejecutan y explique por qué no se ejecutan.

9. Determine el valor de verdad (falso o Verdadero) de las siguientes expresiones teniendo presente que A = 3, B = verdadero y C = 2.31

   a) ((A > 4) **o** (A < -2)) **y** (~ (B) **o** (C <= 2))

   b) (B **o** (A+8) < 1) **y** (~(B) **o** C=2.31 **o** A < 10)

10. ¿Cuál es el tipo de la variable B en el punto anterior?
    a) Real
    b) Entero
    c) Carácter
    d) Lógico

11. ¿Cuál es el tipo de la variable C en el punto 10?
    a) Real
    b) Entero
    c) Carácter
    d) Lógico

12. ¿En un algoritmo es posible declarar una variable tipo verdadero?

13. ¿Cuál es la diferencia entre constante y variable?

14. ¿Qué tipo de variable debe seleccionarse para almacenar la edad de una persona? ¿Por qué?

15. Si se quiere almacenar el nombre completo de una persona ¿qué tipo de variable se debe seleccionar?

16. Dada la siguiente expresión:

    J = b + ((a*b – 3*c ^2) ^ (½)) / ((2*a) / (3*x) + 4*y)

    ¿Cuál es la quinta operación que debe realizarse?
    a) 2 * a
    b) 1 / 2
    c) 3 * c
    d) C ^ 2
    e) 4 * y

17. El orden de prioridad de los operadores lógicos es:
    a) Negación, Conjunción, Disyunción
    b) Conjunción, Negación, Disyunción
    c) Disyunción, Negación, Conjunción
    d) Negación, Disyunción, Conjunción

18. Una de las siguientes afirmaciones es falsa, indique cuál.
    a. En la estructura *CONDICIONAL*, se puede omitir el **si_no**, pero nunca el **entonces**.
    b. La condición en la estructura *CONDICIONAL*, puede ser simple o compuesta
    c. En las estructuras de selección anidadas siempre se cumple: "último ***si*** *en abrir último* ***si*** *en cerrar*"
    d. En los *CONDICIONALES* siempre se tiene que indicar una condición

ACTIVIDAD EVALUATIVA 3.2: sentencias condicionales

Modalidad: Individual

Descripción de la actividad: Diseñe los algoritmos propuestos:

1. Determinar el promedio de cinco números enteros ingresados por el usuario.

2. El usuario ingresa cinco números enteros; el algoritmo debe determinar cuáles son los números más cercanos. Por ejemplo, si el usuario ingresa los números 3, 7, 21, 10 y 8 el programa debe indicar que los más cercanos son 7 y 8.

3. Dada una cantidad en pesos múltiplo de 1000 determinar el menor número de billetes de cada nominación (1000, 2000, 5000, 10.000, 20.000, y 50.000) para obtener esa cifra. Ejemplo: para obtener $237.000 se necesitan 4 billetes de $50.000 uno de $20.000, uno de $10.000, uno de $5000 y uno de $2000.

4. Diseñe un algoritmo para determinar el mayor, el menor y el promedio de tres números ingresado por el usuario.

5. Diseñe un algoritmo que permita determinar si tres valores ingresados por el usuario conforman un triángulo rectángulo.

6. Diseñe un algoritmo que lea tres números enteros determine el promedio de los números e indique cuál es el número más cercano al promedio.

7. Diseñe un algoritmo que permita ingresar un número e indique: la cantidad de cifras que tiene el número, la suma de las cifras del número y el número invertido.

8. Diseñe un algoritmo para calcular el promedio de tres notas ingresadas; calcule y visualice su promedio; si este es menor a 4.0 que ingrese una nueva anota, donde esta nota reemplazará a la nota más baja, calcule y visualice el nuevo promedio.

9. Una empresa trabaja únicamente dos turnos, diurno y nocturno. El valor de la hora diurna es $25.000 y la nocturna $32.000, en caso que el trabajador labore el día domingo la tarifa diurna se incrementa un 2 % del valor de la hora diurna y la tarifa nocturna se incrementa un 10 % del valor de la hora nocturna. Por cada trabajador se conoce el día que laboró, la cantidad de horas laboradas y el turno. Encontrar el total a pagar.

10. Diseñe un algoritmo que lea un número e indique si es negativo o está entre cero y diez inclusive, o entre 11 y 20 inclusive, o entre 21 y 30 inclusive, o entre 31 y 50 inclusive o es mayor que 50.

11. Diseñe un algoritmo que solicite la fecha de su nacimiento y la fecha actual. El algoritmo debe mostrar la cantidad de años, meses y días que ha vivido. Suponga que los meses tienen 30 días.

12. Escriba un algoritmo que lea tres números enteros de un supuesto triángulo, determine si realmente forman un triángulo, e indique su tipo (equilátero, isósceles o escaleno).

    Triángulo: La suma de dos cualesquiera de los lados debe ser mayor que el otro.

    Equilátero: todos los lados son iguales.

    Isósceles: solo dos lados son iguales.

    Escaleno: no tiene dos lados iguales.

13. Diseñe un algoritmo que halle las soluciones de la ecuación cuadrática.

    El algoritmo debe leer los valores a, b y c de la ecuación y mostrar por pantalla las soluciones de la ecuación indicando si son reales o imaginarias.

14. Diseñe un algoritmo que, dada una hora (hh:mm:ss) de 24 horas y el tiempo de cocción, calcule la hora en que estará la comida.

3.3. ESTRUCTURAS DE REPETICIÓN

Esta estructura, también conocida como ciclo o bucle, se utiliza cuando la ejecución de una instrucción o grupo de instrucciones debe repetirse mientras se cumpla alguna condición. Por ejemplo, si se trata de buscar el carácter 'a' en una cadena de caracteres, la instrucción avanzar al siguiente carácter se debe repetir mientras el carácter actual sea distinto de 'a' y no se haya alcanzado el final de la cadena.

Cada evaluación de la condición se denomina paso o iteración de ciclo, y en el momento en que no se cumpla la condición dada, se produce el llamado "rompimiento de control" del ciclo y hasta ese momento se ejecutarán las instrucciones que lo componen.

La condición del ciclo se escribe generalmente en términos de una variable o variables que denominaremos variables de control. Si dicha condición está bien planteada, en algún momento se obtendrá el rompimiento de control del ciclo, en caso contrario podría ocurrir que el ciclo se ejecute indefinidamente, lo cual constituye un serio error de programación.

Una estructura repetitiva o ciclo consta de tres partes básicas:

- La condición del ciclo. Es una proposición escrita en términos de variables, de cuyo valor de verdad depende que el cuerpo del ciclo se ejecute o no. Por ejemplo, en un algoritmo diseñado para sumar una unidad a cada elemento de un vector, la operación repetitiva de sumar 1 se debe ejecutar mientras se cumpla la condición de que no se haya llegado al último elemento del vector; esto se consigue llevando en una variable la cuenta de los elementos ya modificados. La condición es una parte esencial del ciclo y debe analizarse cuidadosamente.

- El cuerpo del ciclo. Contiene la secuencia de instrucciones que deben ser ejecutadas siempre que se cumple la condición.

- La salida del ciclo. Es la señal que se pone para indicar la salida en el momento de terminar la ejecución del cuerpo del ciclo, es decir, cuando ya no se cumpla la condición.

Estudiaremos las estructuras de repetición **para** y **mientras**.

Variables usadas frecuentemente en las estructuras de repetición

- Contador: Variable de tipo entero usada para contar. Se incrementa (o disminuye) en un valor constante en cada iteración del ciclo. **Sintaxis:**

  variableContador = variableContador + valorConstante

- Acumulador: Variable que se usa para almacenar valores numéricos distintos que generalmente se suman (o multiplican) en cada iteración de un ciclo. Sintaxis:

  variableAcumulador = variableAcumulador + variable

- Bandera o centinela: Variable de tipo lógico o entero utilizada en la condición del ciclo mientras para decidir si se itera o no. Es útil cuando no sabemos de antemano el número exacto de veces que se deben ejecutar las instrucciones presentes en el cuerpo del ciclo.

3.3.1. *Estructura de repetición 'para'*

El ciclo 'para' se utiliza cuando se conoce el número exacto de veces que se debe ejecutar una secuencia de instrucciones. El diseñador del algoritmo debe establecer de manera explícita el número de veces que se repetirá el ciclo. Para tal fin, se utiliza una variable contador. Dicha variable se incrementa o decrementa automáticamente en una cantidad fija cada vez que se ejecuta el ciclo. Su sintaxis es:

```
para variableContador = valorInicial hasta valorLimite paso valorPaso
    <Instrucciones>
fin_para
```

La variable controladora *variableContador* del ciclo se inicializa en un valor inicial e irá incrementando o decrementando su valor hasta superar o desbordar el valor límite indicado. La cantidad en que se modifica la variable controladora se denomina paso y puede ser negativa o positiva. Cuando no se especifica el valor del paso se entiende que es de una unidad.

En el **para** se pueden presentar dos casos:

1. El valor inicial asignado a la variable controladora *variableContador* es menor que el valor límite.

   Las instrucciones que están entre las palabras reservadas **para fin_para** se ejecutan siempre y cuando el valor de la variable controladora sea menor o igual que el valor límite y el paso sea positivo.

   Una vez se llegue al **fin_para** la variable controladora se incrementa automáticamente en el valor que indique el paso y se vuelve a verificar si la variable controladora es menor o igual que el valor límite. Si la variable controladora es mayor que el valor límite termina el ciclo y se ejecutan las instrucciones que se encuentren después del **fin_para**.

2. El valor inicial asignado a la variable controladora *variableContador* es mayor que el valor límite.

   Las instrucciones que están entre las palabras reservadas **para** y **fin_para** se ejecutan siempre y cuando el valor de la variable controladora sea mayor o igual que el valor límite y el paso sea negativo. Una vez se llegue al **fin_para** la variable controladora se decrementa automáticamente en el valor que indique el paso y se vuelve a verificar si la variable controladora es mayor o igual que el valor límite. Si la variable controladora es menor que el valor límite termina el ciclo y se ejecutan las instrucciones que se encuentren después del **fin_para**. Para este caso el paso debe ser expresado con un número negativo; de no ser así el las instrucciones en el cuerpo no se ejecutarían ni una vez.

Ejemplo 3.7. Diseñe un algoritmo para sumar los números enteros del uno al cinco.

Solución: Como necesitamos generar los números enteros consecutivos desde uno hasta cinco utilizaremos una variable contadora i que incrementará en una unidad cada vez que se repita el ciclo e iremos sumando los valores que tome en la variable acumuladora *sumatoria*. La variable contadora controlará el ciclo, iniciará en uno y tendrá como valor límite el cinco. Solo cuando termine el ciclo se mostrará el resultado de la sumatoria.

```
1.  algoritmo SumarPrimerosCincoNumerosEnteros
2.  variables
3.      entero: i, sumatoria
4.  inicio
5.      sumatoria = 0
6.      para i = 1 hasta 5
7.          sumatoria = sumatoria + i
8.      fin_para
9.      muestre('La suma es:', sumatoria)
10. fin
```

Hagamos una prueba de escritorio:

i	sumatoria
1	0
2	1
3	3
4	6
5	10
6	15

Ejemplo 3.8: Diseñe un algoritmo que sume los números pares menores o iguales a un valor n ingresado por el usuario.

```
1.  algoritmo SumaNumerosPares
2.  variables
3.      entero: i, n, sumatoria
4.  inicio
5.      muestre('Ingrese un número:')
6.      lea(n)
7.      sumatoria = 0
8.      para i = 2 hasta n paso 2
9.          sumatoria = sumatoria + i
10.     fin_para
11.     muestre('La suma es:', sumatoria)
12. fin
```

Hagamos una prueba de escritorio: Supongamos que un usuario ingrese el número 9.

n	i	sumatoria
9	2	0
	4	2
	6	6
	8	12
	10	20

Ejemplo 3.9. Diseñe un algoritmo que lea un número entero n y muestre la suma de los números enteros de uno a n.

Solución: Este es el mismo problema del ejemplo 3.7, pero ahora vamos a solucionarlo de manera general.

```
1.  algoritmo SumarPrimerosNNumerosEnteros
2.  variables
3.      entero: i, n, sumatoria
4.  inicio
5.      muestre('Ingrese la cantidad de números a sumar:')
6.      lea(n)
7.      sumatoria = 0
8.      para i = 1 hasta n
9.          sumatoria = sumatoria + i
10.     fin_para
11.     muestre('La suma es:', sumatoria)
12. fin
```

Ejemplo 3.10. Diseñe un algoritmo que permita sumar n números ingresados por usuario.

Solución: Como el usuario debe ingresar n números el algoritmo debe solicitar que se ingrese un valor que indique la cantidad de números que se desea sumar y, luego, debe ingresarse cada uno de esos n números.

```
1.  algoritmo SumarNNumeros
2.  variables
3.      entero: i, n
4.      real: numero, sumatoria
5.  inicio
6.      muestre('Ingrese la cantidad de números a sumar: ')
7.      lea(n)
8.      sumatoria = 0
9.      para i = 1 hasta n
10.         muestre('Ingrese un número:')
11.         lea(numero)
```

```
12.          sumatoria = sumatoria + numero
13.      fin_para
14.      muestre('La suma es:', sumatoria)
15. fin
```

Ejemplo 3.11. Diseñe un algoritmo que sume los números enteros que terminen en 7 y estén comprendidos entre m y n. Los valores de m n deben ser ingresados por el usuario.

Solución: Un número entero *x* termina en siete si el resultado de la operación x **mod** 10 es igual a siete.

```
1.  algoritmo SumaNumerosTerminadosEnSiete
2.  variables
3.      entero: i, m, n, sumatoria
4.  inicio
5.      muestre('Ingrese el primer número:')
6.      lea(m)
7.      muestre('Ingrese el segundo número:›)
8.      lea(n)
9.      sumatoria = 0
10.     para i = m hasta n
11.         si i mod 10 == 7 entonces
12.             sumatoria = sumatoria + i
13.         fin_si
14.     fin_para
15.     muestre('La suma es:', sumatoria)
16. fin
```

Hagamos una prueba de escritorio suponiendo que los valores ingresados por el usuario son m = 26 y n = 37.

i	m	n	sumatoria
26	26	37	0
27			27
28			64
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			

El contador empieza en el número menor, o sea en 26, y termina en el número mayor, o sea en 37. Se verifica en cada ejecución del ciclo si el número guardado en la variable contador termina en 7. Los números que cumplen esta propiedad son 27 y 37, por tanto, la variable sumatoria tiene el valor 64 al terminar el ciclo.

Ejemplo 3.12. Haga la prueba de escritorio al siguiente algoritmo.

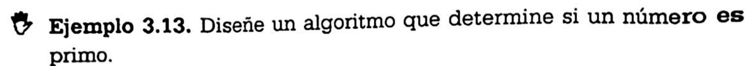
```
1. algoritmo Ejemplo3_12
2. variables
3.     entero: i, a = 10, s = 100
4. inicio
5.     para i = 10 hasta 1 paso -2
6.         s = s - i
7.         a = a - 5
8.     fin_para
9. fin
```

En este caso el ciclo *para* tiene un paso negativo, esto indica que la variable controladora del ciclo disminuye dos unidades en cada iteración hasta alcanzar un valor menor que uno. Como el decremento es de dos unidades los valores de toma i durante la ejecución del algoritmo son: 10, 8, 6, 4, 2 y 0. Cuando alcanza el valor cero finaliza el ciclo.

Prueba de escritorio:

i	s	a
10	100	10
8	90	5
6	82	0
4	76	-5
2	72	-10
0	70	-15

Observe cómo a medida de que ocurren las iteraciones del ciclo los valores de las variables i y a decrementan en dos y cinco, respectivamente, mientras que los valores de s cambian en una cantidad diferente. Por esto, se considera que las variables i y a son contadores y la variable s es un acumulador.

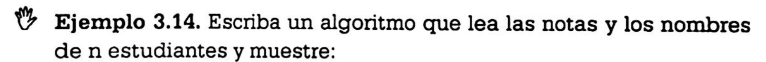


Ejemplo 3.13. Diseñe un algoritmo que determine si un número es primo.

Solución: Recordemos que los números primos son los enteros mayores que 1 que son divisibles únicamente por sí mismos y por la unidad. Por ejemplo, 11 es un número primo ya que sus únicos divisores son el 1 y el mismo 11; en cambio, el número 12 no es primo, pues además de tener como divisores al 1 y al mismo 12 es divisible por 2, 3, 4 y 6. Para establecer si el número dado, que llamaremos n, es primo determinaremos cuántos divisores tiene. Si el número de divisores es exactamente 2 indicaremos que el número n es primo. Para contar sus divisores verificaremos si el número n es divisible por cada uno de los enteros menores o iguales que n. Aquí usaremos el hecho de que un entero p es divisible por otro entero q si el residuo de la división en los enteros de p y q es igual a cero. Por tanto, necesitaremos un ciclo que repita la verificación de divisibilidad con cada entero entre 1 y n. Entonces se requiere la variable controladora del ciclo y un contador de divisores, que llamaremos *divisores*, donde iremos acumulando la cantidad de divisores.


```
1.  algoritmo NúmerosPrimos
2.  variables
3.      entero: i, n, divisores
4.  inicio
5.      muestre('Ingrese un entero: ')
6.      lea(n)
7.      divisores = 0
8.      para i = 1 hasta n
9.          si n mod i == 0 entonces
10.             divisores = divisores + 1
11.         fin_si
12.     fin_para
13.     si divisores == 2 entonces
14.         muestre('El número es primo')
15.     si_no
16.         muestre('El número no es primo')
17.     fin_si
18. fin
```

Observación: El anterior algoritmo se puede diseñar de una manera más eficiente efectuando la verificación de divisibilidad solo con los enteros entre dos y el primer entero que supere a la raíz cuadrada del número. Ver ejemplo 3.19.



Ejemplo 3.14. Escriba un algoritmo que lea las notas y los nombres de n estudiantes y muestre:

a) La cantidad de estudiantes que ganaron.

b) El porcentaje de estudiantes que perdieron.

c) El nombre del estudiante con la nota más alta.

d) La nota más alta.

e) La nota más baja.

Solución:

```
1.  algoritmo EstudiantesGanaron
2.  variables
3.      entero: i, n, ganaron
```

```
4.        real: nota, menorNota, mayorNota, porcentajePerdieron
5.        cadena: nombre, estudianteMayorNota
6.    inicio
7.        muestre('CALCULAR ESTUDIANTES GANARON' )
8.        muestre('Ingrese la cantidad de estudiantes: ' )
9.        lea(n)
10.       ganaron = 0
11.       mayorNota = 0.0
12.       menorNota = 5.0
13.       estudianteMayorNota = ''
14.       para i = 1 hasta n
15.           muestre('Ingrese el nombre del estudiante ', i, ': ')
16.           lea(nombre)
17.           muestre('Ingrese la nota: ')
18.           lea(nota)
19.           si nota >= 3.0 entonces
20.               ganaron = ganaron + 1
21.           fin_si
22.           si nota > mayorNota entonces
23.               mayorNota = nota
24.               estudianteMayorNota = nombre
25.           fin_si
26.           si nota < menorNota entonces
27.               menorNota = nota
28.           fin_si
29.       fin_para
30.       porcentajePerdieron = (n - ganaron) / n * 100
31.       muestre('Ganaron: ', ganaron, ' estudiantes')
32.       muestre('Porcentaje perdieron: ', porcentajePerdieron, ' %')
33.       muestre('Estudiante con mayor nota: ', estudianteMayorNota)
34.       muestre('Nota mayor: ', mayorNota)
35.       muestre('Nota menor: ', menorNota)
36.   fin
```


3.3.2. *Estructura de repetición 'mientras'*

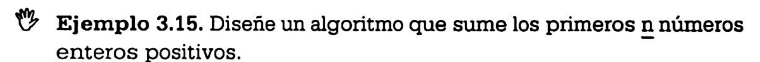
El uso más común del ciclo **mientras** es cuando se desconoce el número exacto de veces que debe ejecutarse una secuencia de instrucciones. El cuerpo del ciclo se ejecuta siempre que la condición sea verdadera. En el momento en que la condición sea falsa el ciclo termina, de modo que es posible que el ciclo no llegue a ejecutarse si la primera vez que se evalúa la condición es falsa. Si se utiliza una variable contador, la cual es modificada en cada ejecución del ciclo, esta funcionará como control del ciclo. Su sintaxis es:

mientras *expresiónLogica* **hacer**

<Instrucciones>

fin_mientras

Si la expresión lógica evalúa a verdadero se ejecutan las instrucciones que están entre las palabras reservadas **hacer** y **fin_mientras**. Una vez ejecutadas las instrucciones y se llegue al **fin_mientras** se vuelve a evaluar la expresión lógica. Dentro las instrucciones debe existir una que eventualmente haga que la expresión lógica evalúe a falso. Si la expresión lógica evalúa a falso, termina el ciclo y se ejecutan las instrucciones que se encuentren después del **fin_mientras**.



Ejemplo 3.15. Diseñe un algoritmo que sume los primeros n números enteros positivos.

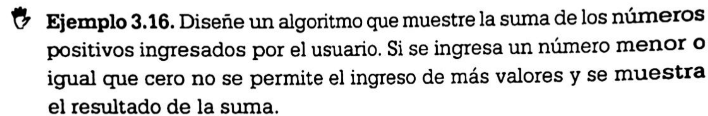
Solución: Este problema ya lo solucionamos empleando un ciclo **para** en el ejemplo 3.9. Ahora lo resolveremos utilizando un ciclo mientras.

```
Algoritmo:
1.   algoritmo SumarPrimerosNNumeros2
2.   variables
3.        entero: i, n, sumatoria
4.   inicio
5.        muestre('Ingrese la cantidad de números a sumar:')
6.        lea(n)
7.        i = 1
8.        sumatoria = 0
9.        mientras i <= n hacer
```

```
10.            sumatoria = sumatoria + i
11.            i = i + 1
12.        fin_mientras
13.        muestre('La suma es:', sumatoria)
14.  fin
```

El ciclo de la línea 9 solo se ejecutará si el valor del contador i es menor o igual que el valor asignado por el usuario a la variable n. Para asegurar que el ciclo termine, dentro de su cuerpo debe existir una instrucción que eventualmente provoque que la condición sea falsa. La instrucción que logra que esto ocurra está en la línea 11. En cada iteración dicha instrucción incrementa en una unidad el valor del contador, y como el valor de n no cambia, eventualmente el valor de i será mayor que el de n y terminará el ciclo.

Observe que si se elimina la instrucción de la línea 11 y el valor de n es mayor que cero el ciclo sería infinito, puesto que la condición sería verdadera cada vez que se evalúe, y jamás se mostraría el resultado de la suma. De ahí la importancia de escribir correctamente la condición y asegurarse de que en el cuerpo del ciclo exista al menos una instrucción que haga que la condición evalúe a falso.



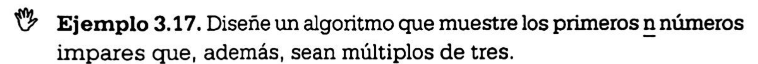
Ejemplo 3.16. Diseñe un algoritmo que muestre la suma de los números positivos ingresados por el usuario. Si se ingresa un número menor o igual que cero no se permite el ingreso de más valores y se muestra el resultado de la suma.

Solución: Como la cantidad de números positivos ingresados depende de la decisión del usuario no podemos saber de antemano cuántos valores van a ser. En este tipo de situaciones es donde es común usar el ciclo mientras. Usaremos dos variables: número y sumatoria. En la variable *número* almacenaremos el valor ingresado por el usuario en cada iteración y la utilizaremos como centinela en la condición del ciclo.

```
Algoritmo:
1.  algoritmo SumarNumerosPositivos
2.  variables
3.      real: numero, sumatoria
```

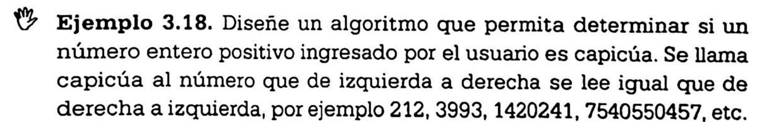


```
4.  inicio
5.      muestre('Ingrese un número: ')
6.      lea(numero)
7.      sumatoria = 0
8.      mientras numero > 0 hacer
9.          sumatoria = sumatoria + numero
10.         muestre('Ingrese un número: ')
11.         lea(numero)
12.     fin_mientras
13.     muestre('La suma es:', sumatoria)
14. fin
```



Ejemplo 3.17. Diseñe un algoritmo que muestre los primeros n números impares que, además, sean múltiplos de tres.

```
1.  algoritmo ImparesMultiplosDeTres
2.  variables
3.      entero: i, n, contador
4.  inicio
5.      muestre('Ingrese un entero: ')
6.      lea(n)
7.      i = 1
8.      contador = 0
9.      mientras contador < n hacer
10.         si i mod 3 == 0 entonces
11.             muestre(i)
12.             contador = contador + 1
13.         fin_si
14.         i = i + 2
15.     fin_mientras
16. fin
```



Ejemplo 3.18. Diseñe un algoritmo que permita determinar si un número entero positivo ingresado por el usuario es capicúa. Se llama capicúa al número que de izquierda a derecha se lee igual que de derecha a izquierda, por ejemplo 212, 3993, 1420241, 7540550457, etc.

Solución: La estrategia de solución será construir un nuevo número, llamado *revés*, formado por las cifras del número dado, pero en orden

inverso. Luego comparamos estos dos números y en caso que **sean** iguales determinaremos que el número dado es capicúa. La extracción de las cifras del número se realiza fácilmente calculando los residuos producidos por la división entera sucesiva del número entre 10. Por ejemplo, si el número es 253, al dividirlo una vez por diez obtenemos como residuo 3. El cociente obtenido que es 25 nuevamente se divide por diez y se obtiene el residuo 5, y así sucesivamente.

La siguiente imagen muestra el proceso que se usará para **extraer las** cifras del número.

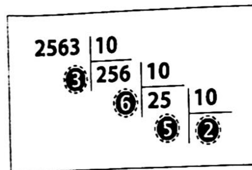


Figura 3.2. Proceso para extraer las cifras entero

Observe que inicialmente se tiene el número 2563 y en cada división queda como residuo una de las cifras de dicho número. En realidad el último residuo es 5, de modo que la primera cifra, es decir 2, debe tomarse del último cociente.

```
1.  algoritmo NumeroCapicua
2.  variables
3.      entero: numero, auxiliar, residuo, reves
4.  inicio
5.      muestre('Ingrese un número entero:')
6.      lea(numero)
7.      auxiliar = numero
8.      reves = 0
9.      mientras auxiliar > 10 hacer
10.         residuo = auxiliar mod 10
11.         reves = reves * 10 + residuo
12.         auxiliar = auxiliar div 10
13.     fin_mientras
14.     reves = reves * 10 + auxiliar
15.     si numero == reves y numero >= 0 entonces
16.         muestre('El número es capicúa.')
```



```
17.     si_no
18.         muestre('El número no es capicúa.')
19.     fin_si
20. fin
```

Mediante la instrucción `residuo = auxiliar mod 10` de la línea 10 se asigna a la variable *residuo* el residuo de la división (numero÷10), es decir, la última cifra del número ingresado por el usuario.

Con la instrucción `reves = reves * 10 + auxiliar` de la línea 11 se agrega una nueva cifra a *revés*.

El proceso de extraer las cifras de número y agregarlas a *reves* se ejecuta repetidamente en el ciclo *mientras*.

Ejemplo 3.19. Algoritmo mejorado para determinar si un número es primo.

Solución: El siguiente algoritmo es más eficiente que el visto anteriormente porque no necesita encontrar todos los posibles divisores de un número para saber si es primo. Este algoritmo solo necesita encontrar un divisor para indicar si es primo o no. La estrategia utilizada aquí consiste en buscar un divisor entre dos y el primer entero que sea mayor que la raíz cuadrada del número n ingresado por el usuario. Si encontramos un divisor entonces el número no es primo.

```
1.  algoritmo NumeroPrimoMejorado
2.  variables
3.      entero: i, n
4.      logico: primo = verdadero
5.      real: r
6.  inicio
7.      muestre('Ingrese un entero:')
8.      lea(n)
9.      i = 2
10.     r = techo(raiz2(n))
11.     mientras primo y i <= r hacer
12.         si n mod i == 0 entonces
13.             primo = falso
14.         si_no
```

```
15.             i = i + 1
16.         fin_si
17.     fin_mientras
18.     si primo y n > 1 entonces
19.         muestre('El número es primo.')
20.     si_no
21.         muestre('El número no es primo.')
22.     fin_si
23. fin
```

Utilice el algoritmo del 3.13 para determinar si el número 100000 es primo y, luego, haga lo mismo con el algoritmo del ejemplo 3.19 para que verifique por qué el segundo es más eficiente. ¿Cuál de los dos algoritmos da la respuesta más rápido?

ACTIVIDAD EVALUATIVA

A. De acuerdo con el siguiente seudocódigo responda las preguntas. Las preguntas son de selección múltiple con única respuesta, es decir, un enunciado con cuatro posibilidades de respuesta de las cuales solo una resuelve correctamente el problema.

```
1. algoritmo ActividadEvaluativa1
2. variables
3.     entero: s = 0, a = 10, i
4. inicio
5.     para i = 1 hasta 10
6.         s = s + i
7.         a = a - 5
8.     fin_para
9. fin
```

Preguntas:

1. El número de veces que se ejecuta la instrucción a = a – 5 en el algoritmo es:
   a) 5 veces
   b) 10 veces
   c) 15 veces
   d) 11 veces

2. El signo de la variable a al final del algoritmo es:
   a) Negativo
   b) Positivo
   c) Cero
   d) No tiene un valor determinado
3. El valor de la variable a al final del algoritmo es
   a) -90
   b) -40
   c) 100
   d) 50
4. El valor de la variable s al final del algoritmo es
   a) 60
   b) 55
   c) 100
   d) 10
5. Haga una prueba de escritorio al siguiente algoritmo y responda las preguntas que se plantean:

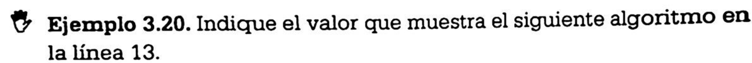
```
1.  algoritmo ActividadEvaluativa2
2.  variables
3.      entero: n, acumulador
4.  inicio
5.      n = 0
6.      acumulador = 0
7.      mientras n >= 0 y acumulador < 30 hacer
8.          n = n + 2
9.          si n mod 3 == 0 entonces
10.             acumulador = acumulador + n * 3
11.         fin_si
12.     fin_mientras
13.     muestre ('Acumulado:', acumulador)
14. fin
```

Preguntas:

1. ¿Cuántas veces se ejecuta el ciclo?
2. ¿Cuántas veces se cumple la condición de la línea 9?
3. ¿Cuál es el valor de la variable acumulador en la cuarta iteración del ciclo?
4. ¿Cuál es el valor de la variable acumulador al terminar el ciclo?
5. ¿Cuál es el valor de la variable n al terminar el ciclo?
6. La condición del ciclo mientras es compuesta está formada por la conjunción de dos condiciones simples n >= 0 y acumulador < 30. El ciclo se puede romper porque ambas condiciones simples o una de ellas dejen de cumplirse. En este caso, ¿por cuál de las dos situaciones se rompe el ciclo?

3.3.3 *Estructuras de repetición anidadas*

Tal como los condicionales, los ciclos también se pueden anidar. En tal caso, el ciclo más interno se procesa primero. Por lo tanto, durante la ejecución de cada iteración del ciclo externo se produce varias veces la ruptura de control del ciclo interno.



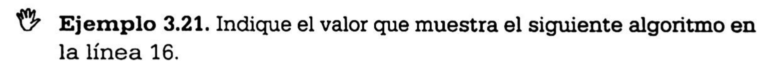
Ejemplo 3.20. Indique el valor que muestra el siguiente algoritmo en la línea 13.

```
1.  algoritmo Ejemplo3_20
2.  variables
3.      entero: a, n = 0, s = 0
4.  inicio
5.      mientras n <= 10 hacer
6.          a = 10
7.          mientras a > 0 hacer
8.              s = s + a
9.              a = a - 4
10.         fin_mientras
11.         n = n + 4
12.     fin_mientras
13.     muestre('La sumatoria es:', s)
14. fin
```


Prueba de escritorio:

n	s	a
0	0	10
4	10	6
8	16	2
12	18	-2
	28	10
	34	6
	36	2
	46	-2
	52	10
	54	6
		2
		-2

Observe que el ciclo interno se ejecuta tres veces para n = 0, n = 4 y n = 8; cada instrucción dentro de este ciclo se ejecuta 9 veces, para los valores de a = 10, a = 6 y a = 2; además, se debe tener presente que para realizar las instrucciones del ciclo externo la condición del ciclo interno debe ser falsa. De la prueba de escritorio se puede concluir que el valor mostrado es el 54.



Ejemplo 3.21. Indique el valor que muestra el siguiente algoritmo en la línea 16.

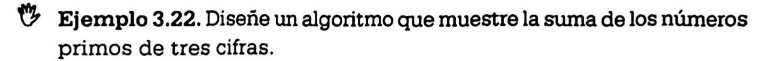
```
1.  algoritmo Ejemplo3_21
2.  variables
3.      entero: i, a
4.      logico: h = falso
5.  inicio
6.      para i = 10 hasta 20 paso 5
7.          a = 10 + i * 3
8.          mientras h == falso hacer
9.              si a mod 3 == 0 entonces
10.                 h = verdadero
11.                 i = 21
```

```
12.            fin_si
13.            a = a + 1
14.        fin_mientras
15.     fin_para
16.     muestre(a)
17. fin
```

Prueba de escritorio:

i	a	h
10	40	falso
21	41	verdadero
26	42	
	43	

Observe que en este caso se tienen dos sentencias cíclicas anidadas, un ciclo **mientras** dentro de un ciclo **para**. El ciclo **para** tiene como variable controladora a i, la cual se incrementa de cinco en cinco, empezando en diez y terminado en 20. Se usa también la variable lógica h (que puede almacenar uno de dos valores: verdadero o falso) Dentro del ciclo mientras se presenta un condicional que verifica si el valor de a es múltiplo de tres, y en tal caso la variable h cambia de estado y romperá el ciclo interno; en caso contrario se incrementa el valor de a en una unidad y nuevamente se verifica la condición de ser divisible por tres. Ahora, dentro del condicional, en la línea 11, también hay una sentencia que tiene como propósito terminar el ciclo **para**, pues si a es múltiplo de 3 a su variable controladora se le asigna un valor (21) que es mayor que el valor límite del ciclo. No obstante, después de la asignación de este valor las instrucciones siguen el flujo de ejecución especificado y cuando llegue a la etiqueta **fin_para** se incrementa automáticamente la variable controladora en cinco unidades. De la prueba de escritorio se puede concluir que el valor mostrado es el 43.



Ejemplo 3.22. Diseñe un algoritmo que muestre la suma de los números primos de tres cifras.

```
Algoritmo:
1.  algoritmo Ejemplo3_22
2.  variables
3.      entero: i, j, divisores, suma = 0
4.  inicio
5.      para i = 100 hasta 999
6.          divisores = 0
7.          para j = 1 hasta i
8.              si i mod j == 0 entonces
9.                  divisores = divisores + 1
10.             fin_si
11.         fin_para
12.         si divisores == 2 entonces
13.             suma = suma + i
14.         fin_si
15.     fin_para
16.     muestre('La suma es:', suma)
17. fin
```

En este algoritmo tenemos dos ciclos **para** anidados; el primero recorre los números de tres cifras, por eso inicia en 100 (que es el primer número de tres cifras) y termina en 999 (que es el último número de tres cifras). La variable controladora de este primer ciclo es la i. El ciclo interno se encarga de contar todos los divisores de cada uno de los números de tres cifras; si determinado número solo tuviera dos divisores (el uno y el mismo número) entonces este es un número primo y se sumaría en la instrucción suma = suma + i; observe que la variable contadora divisores se reinicia antes que comience la ejecución del ciclo interno; esto se hace con el fin de iniciar el conteo de divisores desde cero cada vez que cambia de número de tres cifras.

VICIOS COMUNES EN EL DISEÑO DE ALGORITMOS

- Comenzar a programar sin hacer el suficiente análisis y, lo peor, sin leer ni entender bien el problema. Ahora lo peor: codificar el problema sin realizar el paso anterior del diseño del algoritmo.

- No identificar el tipo de variable (de entrada, de salida, acumulador o contador).
- Dar inicio a la codificación sin tener el correspondiente algoritmo en seudocódigo.
- Escribir de manera desordenada las instrucciones.
- No utilizar las sangrías (tabuladores) en los condicionales, ciclos o luego de inicio.
- Tratar de solucionar a la vez todo lo que el problema pide.
- No usar palabras mnemotécnicas al momento de declarar variables.

ACTIVIDAD EVALUATIVA

A. Construya los algoritmos para resolver los siguientes problemas

1. De una lista de 100 números determine cuáles son pares y cuáles impares, informando con mensajes adecuados.
2. De los primeros 100 números enteros positivos, calcule el promedio de sus números pares y el de sus impares.
3. Imprima los 100 primeros números múltiplos de 3
4. Invierta un número, es decir, que 124 sea 421. Pista: utilizar división entera y módulo.
5. Convierta a horas, minutos y segundos una cantidad de horas con fracción.
6. Calcule el factorial de un número dado por el usuario.
7. Determine los números primos menores que 1000.
8. Determine el máximo común divisor (MCD) de dos números enteros; para tal fin, divida el mayor de los números entre el menor; a continuación divida el número menor (el divisor) por el residuo; continúe este proceso (dividiendo el último divisor por el residuo) hasta que la división sea exacta. El último divisor será el MCD.
9. Haga un algoritmo que calcule el promedio del peso de los hombres y el de las mujeres que ingresan a un recinto. Tenga en cuenta que no se conoce la cantidad de personas. Calcule, además, el porcentaje de los hombres y el de las mujeres que han asistido a ese sitio.

10. Muestre el máximo valor y el mínimo valor de un conjunto de números diferentes de 0, ingresados por el usuario.
11. Muestre el nombre de la persona que tiene la mayor edad y de quien tiene la menor edad, dado que no se conoce la cantidad de personas.
12. Resuelva el problema anterior, mostrando el porcentaje de los mayores de edad y de los menores de edad, respecto al total de personas ingresadas. Además, muestre las edades promedio de los menores de edad y de los mayores de edad.
13. Para un reinado popular se ha presentado una cantidad desconocida de candidatas (se exigen máximo 250 damas). La junta directiva del reinado decide inscribirlas a todas, pero solo selecciona a aquellas cuya estatura oscila entre 165 y 185 cm de estatura, inclusive. Dicha junta pasa dos listados: uno con los nombres y otro con las estaturas dado que el jurado decide seleccionar de estas damas a las más altas (las que están por encima del promedio). Haga un algoritmo que muestre ambos listados y muestre también la estatura promedio de las candidatas seleccionadas por la junta.
14. Diseñe un algoritmo que permita el ingreso de números enteros aleatorios menores o iguales que 200 y determine, si el número es menor que 100, entonces si es o no par y múltiplo o no de 5; ahora, si el número está entre 100 y 200, diga si es o no impar y múltiplo o no de 3.
15. Diseñe un algoritmo que muestre los primeros 100 números primos positivos.
16. Diseñe un algoritmo que permita el ingreso de 20 números enteros aleatorios y determine cuáles son números primos.
17. Diseñe un algoritmo que imprima el cociente, el residuo, el dividendo y el divisor. Dé dos números enteros ingresados por el usuario. No utilice el operador **mod** ni el operador **div**.
18. Diseñe un algoritmo que determine la nota promedio de las cinco materias que ven los 30 estudiantes de un grupo. Además, se quiere saber el código y el nombre de quien tiene el mayor promedio y el porcentaje de los ganadores y el de perdedores, respecto al total de estudiantes del grupo.

19. Diseñe un algoritmo para determinar el nombre del estudiante con la menor edad en un grupo de 20 estudiantes.

20. Diseñe un algoritmo para determinar la cantidad de números capicúas mayores que M y menores que N (M y N ingresados por el usuario).

21. Un club de pesca deportiva realizó un concurso con sus miembros. Por cada trucha atrapada se recolectaron los siguientes datos:

    - Nombre del participante que atrapó la trucha.
    - Longitud de la trucha atrapada (en centímetros).
    - Peso de la trucha atrapada (en gramos).
    - Sexo del participante (F: Femenino, M: Masculino).
    - Categoría del participante (A: 1 a 19 años, B: 20 a 29 años, C: 30 a 39 años, D: 40 años en adelante).

    Diseñe un algoritmo para hallar:

    a) El nombre del ganador del concurso. El ganador del concurso es aquel que atrapó la trucha de mayor longitud y mayor peso.

    b) ¿Quiénes atraparon más truchas, los hombres o las mujeres?

    c) El peso total de las truchas atrapadas por categoría de los participantes.

    d) La longitud promedio de las truchas atrapadas.

22. En un Call Center se atienden n llamadas al día. Cada llamada tiene una duración dada en segundos y se compone de dos tiempos, conversación y documentación. Si una llamada tiene una duración de cero segundos esta debe ser contabilizada como un abandono. Diseñe un algoritmo que permita:

    a) Conocer la duración promedio de todas las llamadas sin incluir las abandonadas.

    b) Saber la cantidad de llamadas abandonadas.

    c) Calcular el tiempo total en el estado de conversación y documentación.

    d) Indicar la llamada con mayor tiempo en conversación.

    e) Indicar la llamada con menor tiempo en documentación.

B. Realice una prueba de escritorio del siguiente algoritmo y responda las preguntas.

```
1.  algoritmo PuntoB
2.  variables
3.      entero: a = 20, s = 0
4.  inicio
5.      mientras a > 0 y s < 15 hacer
6.          s = s + a
7.          a = a - 3
8.      fin_mientras
9.      muestre('La suma es:', s)
10. fin
```

Preguntas:

a) En la condición del mientras se pueden dar dos posibilidades a > 0 y s < 15; si una de las dos no se cumple el ciclo finaliza, ¿Cuál de estas instrucciones se deja de cumplir primero?

b) La variable s actúa como un acumulador. ¿Cuál es su valor al final del algoritmo?

c) ¿Cuál es valor de a al terminar el algoritmo?

d) ¿Cuántas veces se ejecuta la instrucción de la línea 7?

C. Responda las preguntas de acuerdo con los resultados de la ejecución del siguiente algoritmo. Las preguntas son de selección múltiple con única respuesta.

```
1.  algoritmo PuntoC
2.  variables
3.      entero: s = 0, n = 20, c = 0
4.  inicio
5.      mientras c < 3 hacer
6.          si n mod 4 == 0 entonces
```

```
7.              s = s + n
8.              c = c + 1
9.          fin_si
10.         n = n - 1
11.     fin_mientras
12.     muestre('La suma es:', s)
13. fin
```

Preguntas:

1. El valor de n al final de este algoritmo es:
   a) 11
   b) 16
   c) 12
   d) 10

2. ¿Cuantas veces se ejecuta la instrucción de la línea 10?
   a) 5
   b) 10
   c) 8
   d) 9

3. El valor de s al final de este algoritmo es:
   a) 48
   b) 46
   c) 50
   d) 42

4. El valor de c al final de este algoritmo es:
   a) 0
   b) 3
   c) 4
   d) 5

D. Responda las preguntas de acuerdo con los resultados de la ejecución del siguiente algoritmo. Las preguntas son de selección múltiple con única respuesta.

```
1.  algoritmo PuntoD
2.  variables
3.      entero: a = 20, s = 0
4.  inicio
5.      mientras a > 0 hacer
6.          s = s + a
7.          a = a - 3
8.      fin_mientras
9.      muestre('La suma es:', s)
10. fin
```

Preguntas:

1) El valor de a al final de este algoritmo es:
   a) –1
   b) 0
   c) 1
   d) 3

2) ¿Cuántas veces se ejecuta el ciclo?
   a) 5
   b) 6
   c) 7
   d) 8

E. Responda las preguntas de acuerdo con los resultados de la ejecución del siguiente algoritmo. Las preguntas son de selección múltiple con única respuesta.

   Preguntas:

1) El valor final de la variable s es:
   a) 750
   b) 550
   c) 650
   d) 580

2) Los valores de la variable i en la ejecución de este algoritmo son
   a) 10, 20, 30 ..., 100
   b) 10, 11, 12, 13..., 100
   c) 100, 90, 80, ...,10, 0
   d) 100, 90, 80, ..., 10

3) El valor de s cuando i vale 30 es:
   a) 100
   b) 60
   c) 80
   d) 30

PREGUNTAS FRECUENTES:

- ¿Se puede declarar una variable dentro de un ciclo?

  Respuesta: No. Las variables se deben declarar al iniciar el algoritmo; si se declaran dentro de un ciclo, cada vez que se ejecute el ciclo se estará declarando de nuevo la variable. Si esto se hace en un lenguaje de alto nivel se estará haciendo mal uso de un recurso.

- ¿Se pueden reemplazar los ciclos **mientras** por un ciclo **para**?

  Respuesta: No en todos los casos, el ciclo **para** es de tipo cuantitativo, en cambio un ciclo **mientras** puede ser cualitativo o cuantitativo.

- ¿Cuándo utilizo la sentencia **mientras** y cuándo la sentencia **para?**

  Respuesta: es útil emplear **para** cuando de entrada se conoce el número de iteraciones a ejecutar, y el **mientras** en el caso contrario, aunque el ciclo **mientras** puede remplazar el ciclo **para**.

- ¿Cómo se sabe que el ciclo para ha terminado?

  Respuesta: el ciclo **para** tiene implícita una condición, que está dada por el valor tope del ciclo; cuando la variable controladora supera este tope finaliza el ciclo.

- ¿Se pueden anidar más de dos ciclos?

  Respuesta: Sí. Se pueden anidar la cantidad de ciclos necesarios para solucionar el problema. Aunque no hay un límite en el número de ciclos anidados es recomendable no anidar más de tres ciclos, ya que esta situación hace compleja la prueba de escritorio y el control mismo del algoritmo.

Capítulo 4

Vectores

4.1. PRESENTACIÓN

Los arreglos son estructuras de información muy útiles en el desarrollo de algoritmos y en todo lo relacionado con la programación. El uso de arreglos tiene como ventaja que permite almacenar varios valores del mismo tipo en un solo grupo; los almacena y hace referencia a cualquiera de sus elementos de manera sencilla sin invadir los demás datos. Los arreglos pueden ser: de una dimensión y los llamaremos arreglos unidimensionales o vectores, de dos dimensiones que llamaremos arreglos bidimensionales o matrices.

4.2. CONCEPTO DE VECTOR

Un vector es una colección finita y ordenada de valores del mismo tipo de dato. Cada uno de sus elementos puede ser identificado con un valor entero positivo, llamado índice, que indica la posición relativa que ocupan dentro del arreglo. De tal forma que el primer elemento tiene asociado el índice uno, el segundo el dos y así, sucesivamente.

Declaración de vectores:

- Para declarar un arreglo unidimensional se debe especificar su nombre, el tamaño (número máximo de elementos que almacenará) y el tipo de dato de sus elementos.
- El nombre del arreglo debe estar seguido por paréntesis y dentro de estos se expresa el número de elementos del arreglo. El número debe ser un entero mayor que cero.
- El nombre del arreglo debe seguir las mismas reglas que seguimos para nombrar variables.
- Los paréntesis son los que indican que una variable es un tipo de dato arreglo y no un tipo de dato simple.

Representación gráfica de un vector:

En la figura 4.1 se muestra un vector de tipo entero llamado A de tamaño cinco que en cada una de sus posiciones tiene almacenado un valor entero diferente. Es importante comprender la diferencia entre índices (posiciones) y elementos (valores) del arreglo. Los índices del arreglo siempre son enteros mayores o iguales a uno y menores o iguales que el tamaño del arreglo y señalan o indican la posición de un elemento dentro del arreglo. Los elementos del arreglo son, en este caso, los valores enteros que se encuentran almacenados en cada una de las posiciones del arreglo. Por ejemplo, en la posición uno del arreglo está almacenado el valor 12, y en la posición cuatro está almacenado el número 76.

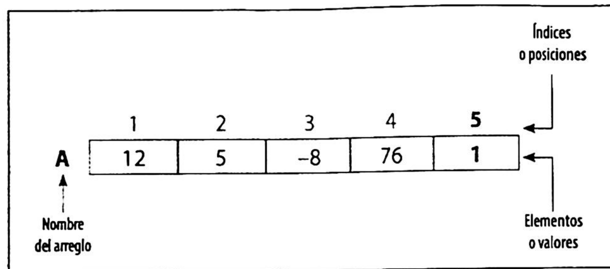


Figura 4.1. Representación gráfica de un vector

4.3. OPERACIÓN CON VECTORES

4.3.1. *Operaciones de escritura y lectura*

Estas operaciones posibilitan la asignación (escritura) de valores a cada una de las posiciones de un vector y acceder (lectura) a los valores del arreglo para realizar operaciones aritméticas o lógicas.

Escritura: Para asignar un valor a una posición x de un arreglo se debe utilizar la siguiente sintaxis.

```
nombreVector(x) = valor
```

Ejemplo 4.1. Asignemos el número 23 a la segunda posición del arreglo unidimensional A, ilustrado en la figura 4.1.

Solución: Para efectuar la asignación debemos usar el nombre del vector, la posición donde deseamos almacenar el valor, el operador de asignación y el valor que queremos almacenar. Este valor se puede expresar con un valor concreto o como el resultado de evaluar una expresión aritmética o lógica. La instrucción de asignación es:

```
A(2) = 23
```

En la figura 4.2 se muestra el nuevo estado del vector. Antes de la asignación en la posición dos estaba el número cinco y después de la asignación el nuevo valor almacenado en la posición dos es el 23. Así, cada vez que asignamos un nuevo valor a una posición de un arreglo, el anterior valor desaparece y ya no es posible recuperarlo, a no ser que se almacene en otra variable. Observe que cuando asignamos un valor a una posición específica de un arreglo solo se modifica esa posición y las demás posiciones permanecen sin cambio.

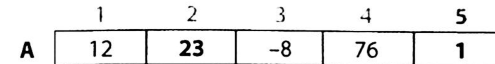


Figura 4.2. Estado del vector después de modificarse el valor de la segunda posición

Si se requiere asignar un valor mediante el teclado a la posición cuatro del vector la instrucción es: lea(A(4)).

Lectura: Para acceder a los elementos de un vector utilizamos el nombre del arreglo, seguido de la posición del elemento entre paréntesis.

Ejemplo 4.2. Asignemos a la tercera posición del arreglo A de la figura 4.2 el resultado de sumar los elementos que se encuentran en la primera y quinta posición.

Solución:

```
A(3) = A(1) + A(5)
```

Después de efectuar la anterior asignación el nuevo elemento en la posición tres es el 13, porque a esa posición le estamos asignando el resultado de sumar el elemento en la posición uno del vector A con el elemento en la posición cinco. En la figura 4.3 se ilustra el nuevo estado del arreglo.

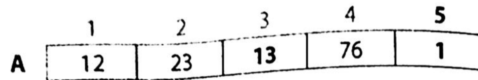


Figura 4.3. Estado del vector después de modificarse el valor de la tercera posición

4.3.2. *Operaciones sobre todos los elementos de un arreglo*

Para realizar una operación sobre un arreglo completo se utilizan estructuras de repetición. Normalmente se usa el ciclo **para** porque conocemos de antemano el número de veces que se debe repetir el ciclo para recorrer completamente el arreglo. Sabemos que los índices de un arreglo comienzan en uno y van hasta el tamaño del vector que especificamos al declararlo. Las operaciones básicas que se realizan sobre un vector son: recorrido, búsqueda y ordenamiento de sus elementos.

Ejemplo 4.3. Diseñe un algoritmo que permita ingresar cien elementos a un arreglo de tipo real y, luego, muestre los valores.

```
Algoritmo:
1.   algoritmo LecturaEscrituraArreglo
2.   variables
3.       entero: i
4.       real: datos(100)
5.   inicio
6.       muestre('INGRESE LOS ELEMENTOS DEL ARREGLO')
7.       para i = 1 hasta 100
8.           muestre('Ingrese el elemento de la posición', i, ': ')
9.           lea(datos(i))
10.      fin_para
11.      para i = 1 hasta 100
12.          muestre('El elemento de la posición', i, 'es:', datos(i))
13.      fin_para
14.  fin
```

Como se ve en el algoritmo en el primer ciclo se permite el ingreso de los cien valores del arreglo y en el segundo ciclo se imprimen los valores. Aunque esas dos operaciones (ingresar y mostrar datos) se pueden realizar en un único ciclo, aquí usamos dos para mostrar claramente la forma en que se recorre un vector.

Ejemplo 4.4. En un arreglo se almacenan las edades de un grupo de diez estudiantes; se quiere determinar la suma de las edades, la edad promedio, la mayor y la menor edad.

```
Algoritmo:
1.   algoritmo PromedioEdades
2.   variables
3.    entero: edad, edades(10), i, mayorEdad = 0, menorEdad = 2000,
 sumatoria = 0
4.    real: edadPromedio
5.   inicio
6.       para i = 1 hasta 10
7.           muestre('Ingrese la edad del estudiante', i,':')
8.           lea(edad)
9.    edades(i)= edad
10.          sumatoria = sumatoria + edades(i)
11.          si edades(i) > mayorEdad entonces
12.              mayorEdad = edades(i)
13.          fin_si
14.          si edades(i) < menorEdad entonces
15.              menorEdad = edades(i)
16.          fin_si
17.      fin_para
18.      edadPromedio = sumatoria / 10
19.      muestre('La edad mayor es', mayorEdad)
20.      muestre('La edad menor es', menorEdad)
21.      muestre('La suma de la edades es', sumatoria)
22.      muestre('El promedio de la edades es', edadPromedio)
23.  fin
```

Tenga en cuenta las siguientes observaciones referentes al anterior algoritmo:

a) La variable mayorEdad la inicializamos en la menor edad posible, en este caso cero, para modificar su valor apenas se ingrese la primera edad. El condicional de la línea 10 solo permite modificar el valor de la variable mayorEdad si se ha ingresado un valor mayor que el contenido en esta variable. Ese condicional actúa como un filtro que solo deja pasar los valores mayores.

b) Igualmente para determinar la menor edad inicializamos la variable menorEdad en un valor que indique una edad suficientemente alta e imposible de alcanzar. En este ejemplo inicializamos la variable edadMenor en 2000. De esta forma garantizamos que la primera vez que se ingrese al ciclo y se lea la primera edad real este valor se asignará a la variable menorEdad y desaparecerá el 2000.

c) Para determinar la suma total de las edades basta con tener una variable acumuladora. En este caso utilizamos para tal fin la variable sumatoria y la inicializamos en cero.

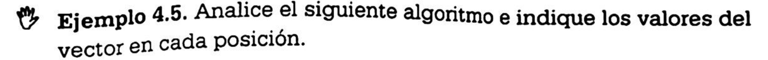
d) Para el promedio de edades tomamos la suma total de las edades y la dividimos por la cantidad de estudiantes, o sea 10. Note que el promedio se calcula por fuera del ciclo, una vez termina su ejecución.

La siguiente prueba de escritorio nos ayuda a comprender las líneas del algoritmo. En la segunda columna de la prueba estas suponiendo que los valores listados son los ingresados por algún usuario y luego son almacenados en el vector edades(i).

i	*edad*	*edades(i)*	*mayorEdad*	*menorEdad*	*sumatoria*	*edadPromedio*
1	18	18	0	200	0	17.1
2	15	15	18	18	18	
3	16	16	19	15	33	
4	19	19	20	14	49	
5	20	20	21		68	
6	14	14			88	
7	15	15			102	
8	18	18			117	
9	15	15			135	
10	21	21			150	
11					171	

Observe cómo cambian los valores de las variables mayorEdad y menorEdad a medida que se ingresan valores para la edad que cumplen

las condiciones de los condicionales en cada iteración. Por ejemplo, en la segunda iteración la edad del estudiante es 15, y la variable menor tenía un valor de 18; como 15 es menor que 18, se cumple la condición y se hace el cambio de valor de la variable, tomado la variable menorEdad el valor 15. Igualmente pasa con la variable mayorEdad en la iteración 10, pues la edad 21 es mayor que el valor 20 que contenía esa variable.



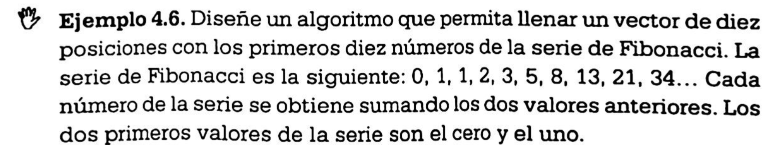
Ejemplo 4.5. Analice el siguiente algoritmo e indique los valores del vector en cada posición.

```
1.  algoritmo Ejemplo4_5
2.  variables
3.      entero: vector(10), i, sumatoria = 0
4.  inicio
5.      para i = 1 hasta 10
6.          vector(i) = i ^ 2 + 3
7.      fin_para
8.  fin
```

En cada posición se almacena el valor de la posición elevado al cuadrado y se le suma tres. Así, por ejemplo, para la posición uno se tiene

$$vector(1) = 1^2 + 3 = 4$$

Y para la posición dos $vector(2) = 2^2 + 3 = 7$, y así sucesivamente. En este caso el vector se llena automáticamente; los valores del vectro serán 4, 7, 12, 19, 28, 39, 52, 67, 84 y 103.



Ejemplo 4.6. Diseñe un algoritmo que permita llenar un vector de diez posiciones con los primeros diez números de la serie de Fibonacci. La serie de Fibonacci es la siguiente: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... Cada número de la serie se obtiene sumando los dos valores anteriores. Los dos primeros valores de la serie son el cero y el uno.

```
Algoritmo:
1.  algoritmo SerieFibonacci
2.  variables
3.      entero: vector(10), i
4.  inicio
```

```
5.        vector(1) = 0
6.        vector(2) = 1
7.        para i = 3 hasta 10
8.            vector(i) = vector(i - 1) + vector(i - 2)
9.        fin_para
10.       muestre('Primeros 10 números de la serie de Fibonacci:')
11.       para i = 1 hasta 10
12.           muestre(vector(i))
13.       fin_para
14.   fin
```

Ejemplo 4.7. En una estación de monitoreo se requiere conocer la temperatura promedio que tuvo el agua de un río el mes pasado. Diariamente se toma la temperatura del río y se almacena el valor en un vector. Se quiere determinar la temperatura más alta y cuántos días la temperatura del agua se mantuvo por debajo de la temperatura promedio. Suponga que todos los meses tienen 30 días.

```
1.    algoritmo Temperaturas
2.    variables
3.        entero: temperaturas(30), i, sumatoria = 0, contador = 0
4.        entero: mayorTemperatura = -10000
5.        real: temperaturaPromedio
6.    inicio
7.        para i = 1 hasta 30
8.            muestre('Ingrese la temperatura del día', i)
9.            lea(temperaturas(i))
10.           sumatoria = sumatoria + temperaturas(i);
11.           si temperaturas(i) > mayorTemperatura entonces
12.               mayorTemperatura = temperaturas(i)
13.           fin_si
14.       fin_para
15.       temperaturaPromedio = sumatoria / 30
16.       para i = 1 hasta 30
17.           si temperaturas(i) < temperaturaPromedio entonces
18.               contador = contador + 1
19.           fin_si
20.       fin_para
21.       muestre('La temperatura mayor es', mayorTemperatura)
22.       muestre('La temperatura promedio es', temperaturaPromedio)
23.       muestre('Las temperaturas menores que la promedio son', contador)
24.   fin
```


Observación: Recuerde que para obtener el promedio de valores, primero se debe tener la suma total de los valores y, posteriormente, se divide por la cantidad de valores sumados.

Ejemplo 4.8. En un arreglo unidimensional se almacena el precio de 1000 artículos. El precio del primer artículo está en la posición uno del arreglo, el precio del segundo artículo está en la posición dos del arreglo, y así sucesivamente. En otro arreglo se almacena la cantidad de artículos que se vendieron. En la primera posición del arreglo se encuentra la cantidad de artículos que se vendieron del primer artículo, en la segunda posición del arreglo se encuentra la cantidad de artículos que se vendieron del segundo artículo, y así sucesivamente. Por ejemplo, para cinco (n = 5) artículos se tendrían los siguientes vectores:

	1	2	3	4	5
precio	2000	500	1200	2500	200

	1	2	3	4	5
cantidad	20	57	12	17	10

Diseñe un algoritmo que muestre:

a) El artículo que más se vendió

b) El artículo que más dinero recaudo por su venta

Solución: Para mostrar el artículo más vendido basta con buscar el valor mayor del arreglo cantidad y almacenar su índice para indicar la posición del artículo. Según el vector cantidad mostrado arriba el artículo que más se vendió fue el 2, porque en esta posición se encuentra el valor mayor del arreglo. Para saber cuál artículo fue el que más dinero recaudó basta con multiplicar el precio de cada artículo por el número de unidades que se vendieron y hallar el mayor de estos productos.

Aunque estos problemas se pueden resolver con un solo ciclo **para** vamos a utilizar dos: uno exclusivamente para asignar los valores a los vectores, y el otro para solucionar los problemas.

```
Algoritmo:
1.  algoritmo VentaArticulos
2.  variables
3.      entero: i, n = 1000, cantidad(n), precio(n), mayorVenta = 0, recaudo
4.      entero: articuloMasVendido, mayorRecaudo = 0,
    articuloMayorRecaudo
5.  inicio
6.      para i = 1 hasta n
7.          muestre('Ingrese el precio del artículo', i, ':')
8.          lea(precio(i))
9.          muestre('Ingrese la cantidad de unidades vendidas del artículo', i)
10.         lea(cantidad(i))
11.     fin_para
12.     para i = 1 hasta n
13.         si cantidad(i) > mayorVenta entonces
14.             mayorVenta = cantidad(i)
15.             articuloMasVendido = i
16.         fin_si
17.         recaudo = precio(i) * cantidad(i)
18.         si recaudo > mayorRecaudo entonces
19.             mayorRecaudo = recaudo
20.             articuloMayorRecaudo = i
21.         fin_si
22.     fin_para
23.     muestre('El articulo más vendido fue el', articuloMasVendido)
24.     muestre('El artículo que más dinero recaudo fue el',
    articuloMayorRecaudo)
25. fin
```


4.4. ORDENAMIENTO DE UN VECTOR

La ordenación consiste en organizar una colección de valores en orden creciente o decreciente, en el caso de los valores numéricos, o en orden alfabético directo o inverso, en el caso de caracteres.

Vamos a ver los tres métodos básicos de ordenar un arreglo de tipo numérico. Estos tres métodos son:

a) Burbuja

b) Selección

c) Inserción

Aunque los tres métodos hacen lo mismo, ordenan un arreglo, la diferencia radica en la estrategia que siguen para hacerlo. El método burbuja es el menos eficiente y el de inserción es el más eficiente.

Método burbuja: La estrategia seguida por este método consiste en comparar sucesivamente pares de elementos vecinos, e intercambiarlos hasta lograr que todos los valores esten ordenados.

Supongamos que se quiere ordenar de menor a mayor el arreglo de cinco elementos mostrado en la figura 4.4 mediante el método burbuja. Las casillas en gris indican los valores que se van a comparar. El método empieza comparando los elementos de las dos primeras posiciones; si el primero es mayor que el segundo los intercambia y pasa a comparar los elementos de la posición dos y tres; si el de la posición dos es mayor que el de la posición tres los intercambia, y así sucesivamente, hasta que compara el penúltimo elemento con el último. De aquí se deduce fácilmente que si el vector tiene tamaño n el número de comparaciones que se hace de esta manera son n – 1. El hacer solo estas n – 1 comparaciones no garantiza que el arreglo queda ordenado. Para garantizarlo estas comparaciones se deben de hacer máximo n – 1 veces también. De ahí que necesitemos dos ciclos anidados para ordenar el arreglo.

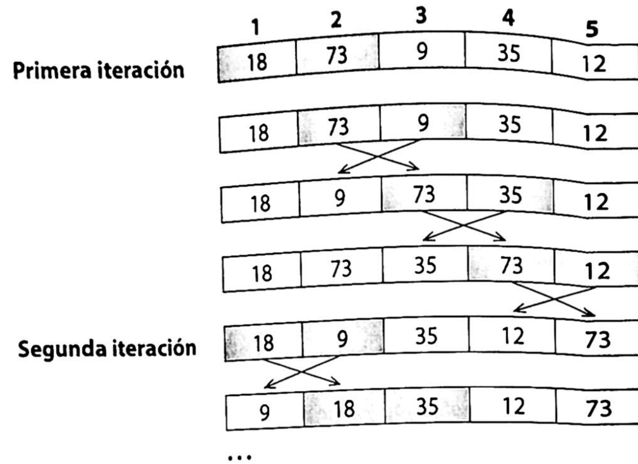


Figura 4.4. Estrategia del método burbuja.

Pero cuando se hacen las primeras n – 1 comparaciones sí se garantiza que en la última posición está el mayor valor presente en el vector. Esto se puede observar claramente en la figura 4.4, donde después de la primera iteración el número 73, que es el valor mayor, está al final del arreglo.

```
Algoritmo:
1.  algoritmo MetodoBurbuja
2.  variables
3.      entero: i, j, n
4.      real: datos(1), auxiliar
5.  inicio
6.      muestre('Ingrese el tamaño del vector:')
7.      lea(n)
8.      datos(n)
9.      para i = 1 hasta n
10.         muestre('Ingrese el elemento de la posición:', i)
11.         lea(datos(i))
12.     fin_para
13.     para i = 1 hasta n - 1
14.         para j = 1 hasta n - 1
15.             si datos(j) > datos(j + 1) entonces
16.                 auxiliar = datos(j)
17.                 datos(j) = datos(j + 1)
18.                 datos(j + 1) = auxiliar
19.             fin_si
20.         fin_para
21.     fin_para
22. fin
```


Método de selección: La estrategia seguida por este método consiste en buscar el elemento menor y ubicarlo en la primera posición. Luego, se busca el segundo elemento menor y se ubica en la segunda posición y, así, sucesivamente hasta lograr que todos los elementos queden ordenados de menor a mayor. En la figura 4.5., las casillas en gris indican los valores que se van a comparar.

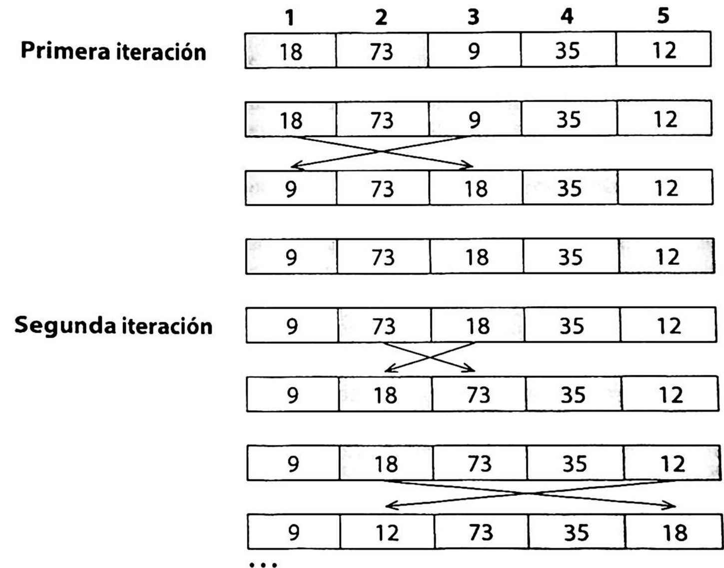


Figura 4.5. Estrategia del método de selección.

```
Algoritmo:
1.  algoritmo MetodoDeSeleccion
2.  variables
3.      entero: i, j, n
4.      real: datos(1), auxiliar
5.  inicio
6.      muestre('Ingrese el tamaño del vector:')
7.      lea(n)
8.      datos(n)
9.      para i = 1 hasta n
10.         muestre('Ingrese el elemento de la posición:', i)
11.         lea(datos(i))
12.     fin_para
13.     para i = 1 hasta n - 1
14.         para j = i + 1 hasta n
15.             si datos(i) > datos(j) entonces
16.                 auxiliar = datos(i)
17.                 datos(i) = datos(j)
18.                 datos(j) = auxiliar
19.             fin_si
20.         fin_para
21.     fin_para
22. fin
```

Observe que en la línea 15 el algoritmo compara el elemento ubicado en la posición i con todos los demás que le siguen y ubica en la primera posición el menor si la condición es verdadera. La primera vez que se ejecuta el ciclo de la línea 13 se le asigna uno a la variable i. La variable controladora j del ciclo interno siempre empieza en un valor mayor en una unidad que i. Por tanto, con la j se genera el índice de los elementos que están después del indicado por la variable i. Así es que se logra encontrar el número menor y ubicarlo en la primera posición, luego, el segundo menor y ubicarlo en la segunda posición, etc.

Método de inserción: La estrategia seguida en este método consiste en tomar los dos primeros elementos del arreglo y ubicar el segundo de manera que estos dos queden ordenados. Luego, se toman los tres

primeros elementos y como los dos primeros ya estan ordenados solo resta ubicar el tercer elemento de manera que los tres queden ordenados y así sucesivamente hasta lograr que todos los elementos queden ordenados.

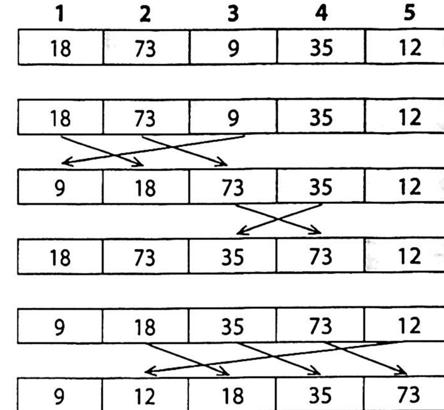


Figura 4.6. Estrategia del método de inserción.

Consideremos que vamos a ordenar el vector de la figura 4.4 mediante el método de inserción. Para entender la estrategia seguida en el diseño del algoritmo nos vamos a apoyar en la figura 4.5. En la figura se puede observar que hay unas posiciones con fondo blanco y otras con fondo gris. Las que están en blanco significa que sus valores ya se consideraron en la ordenación y están ordenadas. Las que están en gris significa que aún no se han tenido en cuenta para la ordenación. Así, en la primera fila de la figura se inicia suponiendo que solo hay un elemento, el 18, y se ignoran los demás. Si pensamos entonces que el vector está compuesto por este único elemento podemos afirmar que el arreglo ya está ordenado y pasamos a considerar que el vector tiene dos elementos, el 18 y el 73 como se muestra en la segunda fila de la figura. Tomamos el 73 como objetivo y procedemos a compararlo con los elementos que están antes que él, en este caso el 18. Como el 18 no es mayor que el objetivo se concluye que los dos elementos están ordenados. Ahora tomamos los tres primeros elementos y el tercer valor será ahora el objetivo. Comparamos el objetivo, es decir, el valor 9, con los valores que le anteceden. Los valores que le anteceden están en las posiciones uno y dos. Como el 73 es mayor que el objetivo copiamos el 73 en la tercera posición, ya que el elemento mayor debe estar al final.

Luego comparamos el valor 18, de la posición uno, con el objetivo. Puesto que 18 es mayor que 9 copiamos el 18 en la segunda posición y nos queda libre la primera para asignar el valor almacenado en el objetivo. Observe que lo que se hizo fue hallar el lugar donde debería estar el número 9 con respecto a los números 18 y 73 que ya estaban ordenados. Con el anterior procedimiento se determinó que el 9 debería estar en la primera posición. Ya tenemos los tres primeros elementos del arreglo ordenados; el paso siguiente es considerar los primeros cuatro elementos del arreglo, teniendo presente que los tres primeros ya están ordenados entre sí, y determinar la ubicación del cuarto elemento (nuevo objetivo) con respecto a los otros tres y, así sucesivamente con los elementos restantes.

```
Algoritmo:
1.  algoritmo MetodoDeInsercion
2.  variables
3.      entero: i, j
4.      real: datos(1), objetivo
5.  inicio
6.      muestre('Ingrese el tamaño del vector:')
7.      lea(n)
8.      datos(n)
9.      para i = 1 hasta n
10.         muestre('Ingrese el elemento de la posición', i,':')
11.         lea(datos(i))
12.     fin_para
13.     para i = 2 hasta n
14.         objetivo = datos(i)
15.         j = i – 1
16.         mientras j >= 1 y datos(j) > objetivo hacer
17.             datos(j + 1) = datos(j)
18.             j = j – 1
19.         fin_mientras
20.         datos(j + 1) = objetivo
21.     fin_para
22. fin
```

Con la variable controladora i del ciclo **para** de la línea 13 recorremos el arreglo e indicamos el objetivo en la línea 14. La variable empieza en

dos porque no tiene sentido que inicie uno, ya que el primer elemento no tiene valores que lo antecedan para efectuar la comparación. En la línea 15 asignamos al contador j el valor de i – 1 porque necesitamos acceder a los elementos que están antes del objetivo. Por tanto, la variable j debe ir disminuyendo en una unidad, siempre y cuando haya elementos en el arreglo (j debe ser mayor que cero) y el elemento en la posición j sea mayor que el objetivo. Precisamente, esta es la condición del ciclo **mientras** de la línea 16 para poder efectuar el desplazamiento de los valores del arreglo para encontrar la posición donde debe ir el objetivo y, así, ir ordenando los valores hasta el objetivo.

4.5. BÚSQUEDA EN UN VECTOR

La búsqueda de un elemento en una colección de datos es una de las actividades básicas cuando se hace uso de computadores. La estrategia de búsqueda se debe elegir de acuerdo con la forma en que esté organizada la colección de datos.

El resultado de la búsqueda de un elemento en un vector será:

a) Si se encuentra el elemento indicaremos su posición

b) Si no se encuentra mostraremos un mensaje que lo indique

Estudiaremos los dos métodos de búsqueda básicos:

a) Búsqueda secuencial

b) Búsqueda binaria

Búsqueda secuencial: Consiste en recorrer todo el vector en orden desde la primera posición y se va comparando cada elemento con el valor buscado. Una vez se encuentra el elemento se deja de recorrer el vector y se indica su posición.

```
Algoritmo:
1.  algoritmo BusquedaSecuencial
2.  variables
3.      entero: i, dato, datos(1), x
4.  inicio
5.      muestre('Ingrese el tamaño del vector:')
6.      lea(n)
```

```
7.      datos(n)
8.      para i = 1 hasta n
9.          muestre('Ingrese el elemento de la posición', i, ':')
10.         lea(dato)
11.         datos(i)=dato
12.     fin_para
13.     muestre('Ingrese el elemento a buscar:')
14.     lea(x)
15.     i = 1
16.     mientras i <= n y datos(i) <> x hacer
17.         i = i + 1
18.     fin_mientras
19.     si i <= n entonces
20.         muestre('El elemento', x, 'está en la posición', i)
21.     si_no
22.         muestre('El elemento', x, 'no está en el vector')
23.     fin_si
24. fin
```

Mediante el ciclo **mientras** se recorre en orden el vector datos. El incremento del índice i del arreglo en la línea 17 solo se efectúa si no se ha sobrepasado el tamaño n del vector y el elemento de la posición es diferente del dato buscado x. Si el elemento no está en el vector el ciclo termina porque la variable controladora será mayor que el tamaño del vector. Si el elemento está en el vector el ciclo termina porque el elemento en la posición i del arreglo no es diferente del dato buscado y el valor de i será menor o igual que el tamaño del vector.

Búsqueda binaria: Este método de búsqueda requiere que los elementos del vector estén ordenados. Aquí garantizaremos que los datos estén ordenados de menor a mayor antes de hacer la búsqueda. La estrategia de este método consiste en comparar el elemento a buscar x con el elemento central del arreglo. Si x es menor que el elemento central entonces el elemento debe estar en la primera mitad del arreglo. Ahora, comparamos el valor x con el elemento central de la primera mitad del arreglo y así sucesivamente hasta encontrar el elemento o determinar que el elemento no está en el arreglo.


```
Algoritmo:
1.  algoritmo BusquedaBinaria
2.  variables
3.      entero: i, j, dato, datos(1), x, bajo, alto, central, objetivo
4.      logico: encontrado = falso
5.  inicio
6.      muestre('Ingrese el tamaño del vector:')
7.      lea(n)
8.      datos(n)
9.      para i = 1 hasta n
10.         muestre('Ingrese el elemento de la posición', i, ':')
11.         lea(dato)
12. datos(i) = dato
13.         objetivo = datos(i)
14.         j = i – 1
15.         mientras j >= 1 y datos(j) > objetivo hacer
16.             datos(j + 1) = datos(j)
17.             j = j - 1
18.         fin_mientras
19.         datos(j + 1) = objetivo
20.     fin_para
21.     muestre('Ingrese el elemento a buscar:')
22.     lea(x)
23.     bajo = 1
24.     alto = n
25.     mientras bajo <= alto y ~encontrado hacer
26.         central = (bajo + alto) div 2
27.         si x < datos(central) entonces
28.             alto = central – 1
29.         si_no
30.             si x == datos(central) entonces
31.                 encontrado = verdadero
32.             si_no
33.                 bajo = central + 1
34.             fin_si
35.         fin_si
36.     fin_mientras
```

```
37.       si encontrado entonces
38.            muestre('El elemento', x, 'está en la posición', central,
   'del vector')
39.       si_no
40.            muestre('El elemento', x, 'no está en el vector')
41.       fin_si
42.  fin
```

Antes de buscar el elemento x en el vector datos se ordena el arreglo mediante el método de inserción. Esta ordenación se realiza entre las líneas 10 y 20. El **mientras** de la línea 25 efectúa la búsqueda binaria. Para tal propósito se usan tres variables fundamentales: bajo, alto y central. Las variables *bajo* y *alto* sirven para delimitar los elementos (subvector) donde se debe buscar, y la variable central se utiliza para indicar el elemento del subvector con el cual se debe hacer la comparación.

Observación: Recuerde que para recorrer un vector, acceder a cada una de sus posiciones, se requiere de una estructura de repetición; normalmente se utiliza el ciclo **para** debido a que conocemos de antemano el tamaño del arreglo.

ACTIVIDAD EVALUATIVA

Resuelva los siguientes ejercicios:

1. Determine los valores de las variables **i** y **j** en el siguiente algoritmo:

```
algoritmo DeterminarValores
variables
      entero: i, j, v(10)
inicio
para i = 1 hasta 10
            v(i) = i
      fin_para
i = 1
      j = 2
      v(i) = j
      v(j + i) = i + j
      i = v(i) + v(j)
      v(3) = 5
      j = v(i) - v(j)
fin
```


2. Diseñe un algoritmo que calcule el promedio de los elementos de un arreglo unidimensional de tipo real.

3. Diseñe un algoritmo que calcule el número de elementos negativos, cero y positivos de un arreglo de reales.

4. Se dispone de un arreglo T de cincuenta números reales distintos de cero. Cree un nuevo arreglo en la que todos sus elementos resulten de dividir los elementos del arreglo T por el elemento T[k], siendo k un valor dado.

5. Se dispone de un arreglo de n elementos. Se desea diseñar un algoritmo que permita insertar el valor x en el lugar k-ésimo del arreglo.

6. Se dispone de n temperaturas almacenadas en un arreglo. Se desea calcular la temperatura promedio y obtener el número de temperaturas mayores o iguales que la promedio.

7. Diseñe un algoritmo que halle el menor valor y el mayor valor de un arreglo de n elementos reales.

8. Dado un arreglo de n elementos de tipo entero, diseñe un algoritmo que calcule de forma independiente la suma de los números pares y la suma de los números impares.

9. Diseñe un algoritmo que calcule cuantas veces se encuentra un entero k en un arreglo de tipo entero.

10. Diseñe un algoritmo que lea tres vectores: el primero almacenará el número de la cédula de n personas, el segundo, sus edades y el tercero, sus estaturas en centímetros. El algoritmo debe mostrar el número de la cédula de la persona más joven y el número de la cédula de la persona más alta.

11. Se tiene un arreglo de diez elementos de tipo entero, con valores entre uno y cincuenta. Cada valor en el arreglo indica el número de asteriscos por fila que se deben mostrar por pantalla. Diseñe un algoritmo que realice dicha tarea.

12. Diseñe un algoritmo que sume dos arreglos del mismo tamaño y almacene y muestre el resultado en otro arreglo. La suma se debe realizar entre elementos con el mismo índice.

13. Diseñe un algoritmo que invierta el orden de los elementos de un arreglo de reales. El primero se vuelve el último; el segundo, el penúltimo, etc.

14. Diseñar un algoritmo que lea un vector de N elementos y mueva todos los elementos una posición a la derecha. El último elemento será el primero.

15. Diseñe un algoritmo que ordene los elementos de un arreglo unidimensional de reales de menor a mayor.

16. Diseñe un algoritmo que calcule el número de veces que se repite la letra a en un arreglo de tipo carácter.

17. Diseñe un algoritmo que concatene dos arreglos de reales.

Capítulo 5

Matrices

5.1. PRESENTACIÓN

El estudio de los arreglos bidimensionales, más conocidos como matrices, lo trataremos en este capítulo, ya que tienen gran utilidad en la solución de problemas que llevan implícitas dos dimensiones.

5.2. CONCEPTO DE MATRIZ

Es una colección finita y ordenada de elementos que pertenecen al mismo tipo de dato dispuestos en forma rectangular. Para determinar la posición de sus elementos debemos utilizar dos índices, uno para indicar la fila y otro para indicar la columna. Al igual que en los vectores, estos índices deben ser de tipo entero mayores que cero y menores que la dimensión correspondiente de la matriz.

5.3. DECLARACIÓN DE MATRICES

Para declarar un arreglo bidimensional se debe especificar su nombre, el número de filas, el número de columnas y el tipo de dato de sus elementos. El tipo de dato debe ser simple (entero, real, lógico o carácter).

El nombre del arreglo debe estar seguido por paréntesis y dentro de estos se deben especificar dos números enteros positivos separados por coma. El primer número indica la cantidad de filas del matriz, y el segundo indica la cantidad de columnas. Las filas se disponen en sentido horizontal y las columnas en sentido vertical. Al nombrar matrices debemos seguir las reglas dadas para nombrar variables.

Los paréntesis con los dos valores enteros positivos son los que indican que una variable es un tipo de dato arreglo bidimensional.

Ejemplo 5.1: Declaración de la variable m de tipo matriz de enteros con cinco filas y cuatro columnas.

```
entero: m(5, 4)
```

Cuando se declara una matriz se reserva un espacio en memoria para almacenar valores; por tanto, está vacía y se le deben asignar valores para poder realizar cálculos. En la figura 5.1 se ilustra la forma común de representar matrices. Se puede ver claramente que la matriz m tiene cinco filas y cuatro columnas, y en cada posición se puede almacenar un valor de tipo entero. Como máximo en la matriz m se pueden almacenar 20 valores.

m	1	2	3	4
1				
2				
3				
4				
5				

Figura 5.1. Representación gráfica de la matriz creada en el ejemplo 5.1

5.4. REFERENCIA A LOS ELEMENTOS DE UNA MATRIZ

Para referirse a un componente o elemento particular de un arreglo bidimensional, ya sea para leerlo, mostrarlo o modificarlo, se escribe el nombre del arreglo del cual hace parte, seguido de la posición que ocupa el elemento entre paréntesis.

Ejemplo 5.2: Asigne el némero 23 a la primera posición de la matriz m creada en el ejemplo 5.1.

Solución: Para hacer la asignación debemos usar el nombre de la matriz, indicar la posición a la que vamos asignar el valor, el operador de asignación y el valor.

```
m(1, 1) = 23
```


Después de efectuar la anterior asignación el nuevo estado de la matriz m se muestra en la figura 5.2. Ahora, la matriz tiene un valor almacenado y las otras posiciones siguen vacías.

m	1	2	3	4
1	**23**			
2				
3				
4				
5				

Figura 5.2. Estado de la matriz **m** después de la asignación

Ejemplo 5.3: Diseñe un algoritmo para llenar una matriz de 5 filas y 5 columnas. A cada posición asígnele el resultado de sumar el índice de la fila más el índice de la columna que la identifica.

```
1.  algoritmo Ejemplo5_3
2.  variables
3.      entero: i, j, m(5, 5)
4.  inicio
5.      para i = 1 hasta 5
6.          para j = 1 hasta 5
7.              m(i, j) = i + j
8.          fin_para
9.      fin_para
10. fin
```

En la figura 5.3 se muestran los valores que toma la matriz m en el anterior algoritmo.

2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

Figura 5.3. Valores de la matriz **m** del algoritmo 5.3

Ejemplo 5.4: Diseñe un algoritmo que llene una matriz de cinco filas y cinco columnas con números enteros aleatorios entre uno y mil, y muestre el elemento mayor de la matriz y su posición.

Solución: Los números aleatorios los generaremos con la función interna aleatorio. Esta función recibe un entero x y devuelve un número entero aleatorio entre uno y x. En la línea 7, dentro del para interno se genera el número aleatorio y se asigna a la matriz. Una vez asignado comenzamos la búsqueda del elemento mayor.

```
1.  algoritmo MayorAleatorio
2.  variables
3.      entero: i, j, m(5, 5), mayor = 0, filaDelMayor = 0, columnaDelMayor =
    0
4.  inicio
5.      para i = 1 hasta 5
6.          para j = 1 hasta 5
7.              m(i, j) = aleatorio(1000)
8.              si mayor < m(i, j) entonces
9.                  mayor = m(i, j)
10.                 filaDelMayor = i
11.                 columnaDelMayor = j
12.             fin_si
13.         fin_para
14.     fin_para
15.     muestre('El elemento mayor es', mayor)
16.     muestre('El elemento mayor está en la fila', filaDelMayor, 'columna',
        columnaDelMayor)
17. fin
```

5.5. RECORRIDOS EN UNA MATRIZ

El acceso o lectura de cada uno de los elementos de una matriz se puede hacer por filas o por columnas dependiendo de la necesidad; en el recorrido por filas primero nos paramos en la primera fila y nos desplazamos por cada uno de los elementos de la fila; una vez llegamos al último elemento de la fila pasamos al primer elemento de la segunda fila, y así sucesivamente, hasta completar las filas de la matriz, como se muestra a continuación:

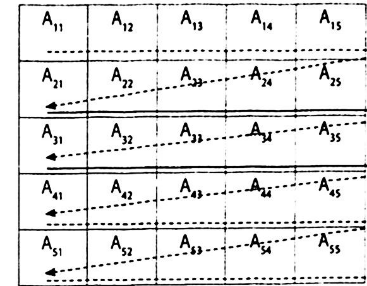


Figura 5.4. Recorrido de una matriz por filas

El siguiente segmento de seudocódigo muestra cómo se recorre una matriz por filas:

```
para i = 1 hasta filas
    para j = 1 hasta columnas
        lea(A(i, j))
    fin_para
fin_para
```

En el recorrido por columnas nos posicionamos en el primer elemento de la primera columna y recorremos en orden todos los elementos de la columna. Luego, nos posicionamos en el primer elemento de la siguiente columna y recorremos en orden todos los elementos de esta columna, y así sucesivamente hasta recorrer todas las columnas de la matriz. Este proceso se ilustra en la Figura 5.5.

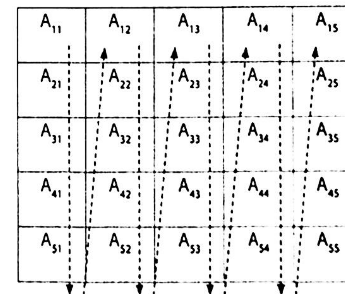


Figura 5.5. Recorrido de una matriz por columnas

El siguiente segmento de seudocódigo muestra cómo se recorre una matriz por columnas:

```
para j = 1 hasta columnas
    para i = 1 hasta filas
        lea(A(i, j))
    fin_para
fin_para
```

El siguiente ejemplo muestra la utilidad de cada uno de estos recorridos.

Ejemplo 5.5: Una cadena de almacenes guarda en una matriz, de cinco filas y siete columnas, las ventas semanales en dólares realizadas en cada una de sus cinco sucursales. En la figura 5.1 se muestra la matriz. En este caso, las filas de la matriz representan las sucursales y las columnas los días de la semana. Así, en la sucursal 2 se recaudó US$457 por concepto de ventas el día martes.

340	345	567	458	145	265	897
540	457	125	104	100	745	895
830	910	860	450	782	566	489
120	457	411	555	888	568	471
130	140	150	160	125	128	489

Se requiere de un algoritmo que permita determinar:

a) El día de mayor venta

b) La sucursal más rentable

c) El promedio de venta diario de la cadena de almacenes

Solución: Para conocer el día con mayor venta debemos sumar los valores de cada columna de manera independiente y buscar la mayor de estas sumas. Por tanto, debemos recorrer la matriz por columnas. Para hallar la sucursal más rentable debemos sumar cada fila de manera independiente y buscar la mayor de estas sumas. Para calcular el promedio de venta diario debemos sumar todos los elementos de la matriz y dividir por siete.


```
1.  algoritmo CadenaDeAlamacenes
2.  variables
3.    entero: i, j, venta, ventas(5, 7), sumatoria = 0, mayorVentaDiaria = 0
4.    entero: diaMayorVenta, sucursalMasRentable, mayorVentaSucursal = 0
5.    real: promedioDiario
6.  inicio
7.      para i = 1 hasta 5
8.          para j = 1 hasta 7
9.              muestre('Ingrese venta en la fila', i, 'columna', j, ':')
10. lea(venta)
11.             ventas(i, j) = venta
12.             sumatoria = sumatoria + ventas(i, j)
13.         fin_para
14.     fin_para
15.     promedioDiario = sumatoria / 7
16.     para j = 1 hasta 7
17.         sumatoria = 0
18.         para i = 1 hasta 5
19.             sumatoria = sumatoria + ventas(i, j)
20.         fin_para
21.         si sumatoria > mayorVentaDiaria entonces
22.             mayorVentaDiaria = sumatoria
23.             diaMayorVenta = j
24.         fin_si
25.     fin_para
26.     muestre('El dia de mayor venta fue el', diaMayorVenta)
27.     para i = 1 hasta 5
28.         sumatoria = 0
29.         para j = 1 hasta 7
30.             sumatoria = sumatoria + ventas(i, j)
31.         fin_para
32.         si sumatoria > mayorVentaSucursal entonces
33.             mayorVentaSucursal = sumatoria
34.             sucursalMasRentable = i
35.         fin_si
36.     fin_para
37.     muestre('La sucursal más rentable es la', sucursalMasRentable)
38.     muestre('El promedio diario de ventas es', promedioDiario)
39. fin
```

En los ciclos anidados entre las líneas 7 y 14 solicitamos que se ingresen las ventas y las almacenamos en la matriz *ventas*, y vamos sumando todos los valores ingresados para calcular el promedio de venta diario de la cadena de almacenes. Una vez sumados todos los valores calculamos el promedio en la línea 15.

En los ciclos anidados entre las líneas 16 y 25 recorremos la matriz por columnas para hallar el día de mayor venta para la cadena de almacenes. El ciclo externo genera los índices de las columnas y cada vez que se cambia de columna la variable *sumatoria* se reinicia en cero para que almacene la suma de los valores de cada columna independiente de las demás columnas. Una vez termina el ciclo interno buscamos en el condicional de la línea 21 el valor mayor de los que se asigna a la variable *sumatoria*. Para hallar la sucursal seguimos una lógica similar pero recorriendo la matriz por filas.

ACTIVIDAD EVALUATIVA

Resuelva los siguientes ejercicios:

1. Cree una matriz de cinco filas y cinco columnas (5x5) de datos enteros. Determine el promedio de las filas y de las columnas de la matriz de manera independiente.
2. Cree dos matrices de enteros con cinco filas y cinco columnas, sume las matrices y muestre el resultado de la suma.
3. Cree una matriz de cinco filas y cinco columnas de datos enteros; determine la suma y el promedio de los valores que están en la diagonal principal de la matriz.
4. Cree una matriz de cinco filas y cinco columnas de datos enteros; almacene en un vector los valores que están en la diagonal principal de la matriz.
5. Diseñe un programa que permita llenar una matriz de 8x8, con números del 1 al 64, siguiendo el movimiento del caballo de ajedrez empezando con uno desde cualquier posición.
6. Diseñe un algoritmo que permita crear una matriz con elementos diferentes; si el usuario ingresa un valor que ya está en la matriz el algoritmo le indica que ese elemento ya está.
7. Diseñe un algoritmo que permita determinar elemento que más se repite en una matriz de orden MxN.

8. Dada una matriz de cuatro filas y siete columnas, en la cual se almacena el valor de las ventas durante cuatro semanas de un almacén, diseñe un algoritmo que solucione cada uno de los siguientes literales:
   a) Mostrar el día del mes con más ventas
   b) Mostrar la semana con más ventas
   c) Calcular el promedio de ventas por semana
   d) Calcular el promedio de ventas por día de la semana
   e) Calcular el promedio de ventas del mes
9. Se tienen los resultados de las últimas elecciones a gobernador en el estado X, el cual está conformado por 30 municipios. En dichas elecciones hubo 17 candidatos. Diseñe un algoritmo que solucione cada uno de los siguientes literales:
   a) Muestre el municipio donde más se votó.
   b) Muestre el candidato que más votos recibió.
   c) Indique cuál candidato es el ganador. Las elecciones las gana quien reciba más del 50 % de los votos. Si ningún candidato recibió más del 50 % de los votos, debe mostrar los dos candidatos más votados, que serán los que pasen a la segunda vuelta de las elecciones.
10. Los resultados de un torneo de fútbol de 16 equipos se encuentran almacenados en una matriz de 16 filas por 16 columnas. Por filas se tienen los goles que un equipo anotó a los demás, y por columnas se tienen los goles que un equipo recibió de los demás. Por ejemplo, para un torneo de cuatro equipos se tiene la siguiente matriz:

	1	2	3	4
1		0	4	0
2	2		1	2
3	3	2		0
4	0	1	1	

En la diagonal principal de la matriz no hay elementos. ¿Por qué?

Se puede decir que los marcadores del equipo 1 fueron:

- Perdió 0 – 2 con el equipo 2.
- Ganó 4 – 3 al equipo 3.
- Empató 0 – 0 con el equipo 4.

Anotó cuatro goles y recibió cinco goles en el torneo.

Diseñe un algoritmo que solucione cada uno de los siguientes literales:

a) Muestre los marcadores de los partidos jugados por el equipo n, indicando con quién jugó y si ganó, perdió o empató cada partido.

b) Muestre los goles a favor y los goles en contra del equipo n.

c) Muestre el equipo con mayor número de partidos ganados (suponga que un solo equipo cumple con esta característica).

d) Muestre el equipo que más goles anotó.

e) Muestre el equipo que más goles recibió.

f) Muestre el equipo con mejor gol diferencia.

g) Muestre el promedio de goles anotados por equipo.

Capítulo 6

Subalgoritmos

6.1. PRESENTACIÓN

Los problemas complejos pueden resolverse más fácilmente dividiéndolos en subproblemas. Para cada subproblema se plantea un subalgoritmo o módulo que lo solucione. Así, la solución del problema complejo se construye a partir de la invocación ordenada de subalgoritmos desde el algoritmo principal (ver figura 6.1).

Cada subalgoritmo se diseña para solucionar un único problema y se puede utilizar tantas veces cuantas sea necesario, sin necesidad de diseñarlo nuevamente. Esta forma de proceder facilita el diseño y la evaluación de algoritmos porque el algoritmo principal es más corto, es más fácil la búsqueda de errores, el mantenimiento y la comprensión del algoritmo, y permite reutilizar subalgoritmos.

Los lenguajes de programación de alto nivel están diseñados pensando en esta forma de trabajo y se les da el nombre de subprogramas.

Al trabajar con subalgoritmos se debe tener en cuenta que:

a) Cada subalgoritmo se define o especifica de manera independiente, por fuera de cualquier algoritmo o subalgoritmo.

b) Se pueden invocar (utilizar) en un algoritmo principal o en otro subalgoritmo.

c) Los datos de entrada se le deben suministrar en el lugar donde se invoque.

d) La ejecución del algoritmo principal se pausa mientras se ejecutan las instrucciones del subalgoritmo invocado.

En la figura 6.1 mostramos la forma en que cooperan distintos subalgoritmos con un algoritmo principal para solucionar un problema. Cada subalgoritmo es una unidad lógica independiente que colabora con las

demás para resolver un problema. La comunicación entre ellos se logra por medio de sus datos de entrada y de salida.

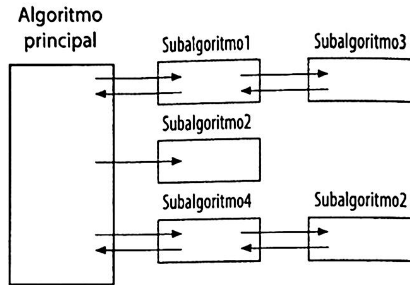


Figura 6.1. Invocación de subalgoritmos

6.2. DEFINICIÓN

Un subalgoritmo es una secuencia de declaraciones e instrucciones que realiza una tarea específica.

Los subalgoritmos se dividen en funciones y procedimientos. La diferencia básica entre los dos es que las funciones siempre deben devolver un valor, y los procedimientos pueden devolver más de un valor o no devolver.

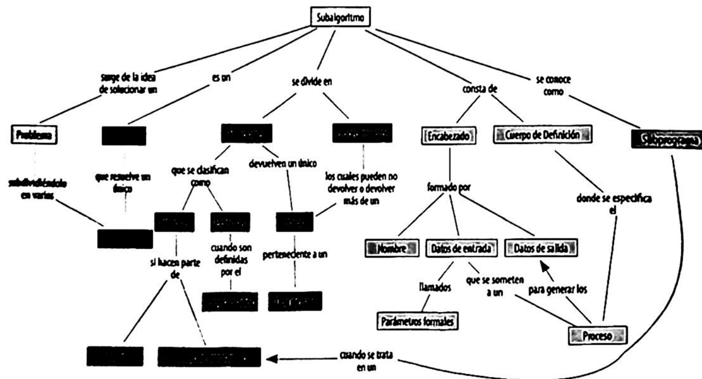


Figura 6.2. Mapa conceptual subalgoritmo

Los subalgoritmos se componen de dos partes principales: encabezado y cuerpo de definición. En el encabezado se especifica su nombre y los datos de entrada y de salida. En el cuerpo de definición se declaran variables y se describe la secuencia de pasos que soluciona un problema.

6.3. PARÁMETROS

Son los datos que utilizan los subalgoritmos para comunicarse con el algoritmo principal o el subalgoritmo desde donde se llaman o invocan. Los parámetros se listan después del nombre del subalgoritmo entre paréntesis en el encabezado del subalgoritmo. Por cada parámetro se debe indicar su nombre y su tipo de dato. Se reconocen dos tipos de parámetros: según su ubicación y según su función.

6.3.1. *Parámetros según su ubicación*

Se dividen en parámetros formales y parámetros actuales. Parámetros formales es el nombre dado a los parámetros cuando se especifican en la definición del subalgoritmo. Los parámetros actuales son los valores pasados al subalgoritmo cuando se llama o invoca. Los parámetros actuales también son conocidos como argumentos. Por tanto, para invocar correctamente un subalgoritmo, los argumentos deben corresponder con los parámetros formales, en cantidad, orden y tipo de dato.

6.3.2. *Parámetros según su función*

Estos pueden ser de entrada para funciones y procedimientos, y de salida únicamente para procedimientos. Dichos parámetros se diferencian porque los de entrada están antecedidos por la letra **E** (Entrada) y los de salida por la letra **S** (Salida).

6.4. FUNCIONES

Matemáticamente, una función es una operación que toma uno o más valores, llamados argumentos, y produce un valor denominado resultado.

Tomemos como ejemplo la función

Esta función tiene dos argumentos: *x* y *y*. Si x = 2 y y = 1 el valor de la función es 5. Veamos,

Las funciones en programación operan de una manera similar: reciben argumentos, realizan operaciones sobre los argumentos y devuelven un resultado.

Los lenguajes de programación tienen funciones incorporadas (internas o intrínsecas) y permiten definir o crear nuevas funciones definidas por el programador (funciones externas). En el seudocódigo empleado en el texto se han incorporado algunas funciones matemáticas básicas. Estas se encuentran listadas en la tabla 2.1. Veamos ahora cómo definir funciones utilizando seudocódigo.

6.4.1. *Definición de funciones externas*

La definición de funciones se hará de forma independiente, es decir, por fuera de cualquier algoritmo o subalgoritmo. Para tal propósito emplearemos las palabras reservadas **funcion** y **devolver**, y seguiremos la siguiente sintaxis.

```
<TipoDeResultado> funcion <NombreFuncion>(<ParametrosFormales>)
[Declaración de variables]
inicio
    <Instrucciones>
    devolver <Expresión>
fin
```

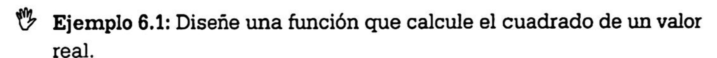
6.4.2. *Elementos del encabezado de las funciones:*

- <TipoDeResultado>: Indica el tipo del valor devuelto por la función. Puede ser cualquiera de los tipos de datos simples o compuestos o estudiados.
- Funcion: Indica que se está definiendo una función.
- <NombreFuncion>: Nombre asociado con la función, debe ser un nombre de identificador válido seguido de paréntesis. El nombre debe seguir las mismas reglas para nombrar variables.
- <ParametrosFormales>: Conjunto de declaración de variables sobre las cuales operará la función para producir el resultado. Estas variables son locales a la función donde se declaran. Constituyen los datos de entrada de la función. Los valores mínimos que se requieren para solucionar el problema.

6.4.3. *Elementos del cuerpo de las funciones*

- [Declaración de variables locales]: Se especifica el nombre de las variables y su tipo en caso que se requieran para la solución del problema. Las variables declaradas dentro de la función pertenecen exclusivamente a esta.
- <Instrucciones>: Instrucciones que constituyen la definición de la función. En esa parte se especifican las operaciones que se harán con los datos de entrada para generar el resultado o salida.
- devolver: Se utiliza para regresar el resultado de la función. Cuando se ejecuta esta instrucción termina inmediatamente la función, y el control del programa se transfiere al llamador de la función.
- <Expresion>: Indica el valor devuelto por la función. El tipo de valor devuelto siempre debe coincidir con el tipo dato que devuelve la función y que se especificó en el encabezado.

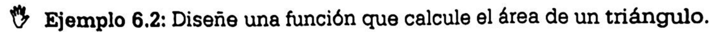
Observación: Procure no utilizar el procedimiento *lea* dentro de los subalgoritmos. Los datos de entrada se deben proporcionar por medio de los parámetros.



Ejemplo 6.1: Diseñe una función que calcule el cuadrado de un valor real.

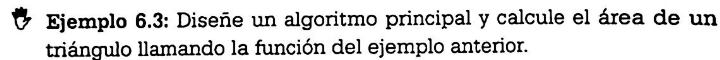
Solución: La función debe recibir el valor que se desea elevar al cuadrado; para tal fin especificamos que la función tiene un parámetro formal llamado *número* de tipo real. El tipo de dato de retorno es real porque se retornará el cuadrado de un número que es real. La función la llamamos *calcularCuadrado*.

```
1.  real función calcularCuadrado(real: numero)
2.  variables
3.      real: c
4.  inicio
5.      c = numero * numero
6.      devolver c
7.  fin
```


Ejemplo 6.2: Diseñe una función que calcule el área de un triángulo.

Solución: Para calcular el área de un triángulo necesitamos conocer la base y la altura. Por tanto, la función debe tener dos parámetros formales.

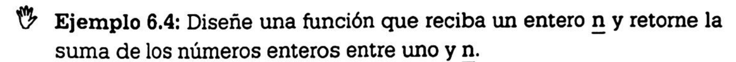
```
1.  real funcion calcularAreaTriangulo(real: base, altura)
2.  variables
3.      real: area
4.  inicio
5.      area = base * altura / 2
6.      devolver area
7.  fin
```


Ejemplo 6.3: Diseñe un algoritmo principal y calcule el área de un triángulo llamando la función del ejemplo anterior.

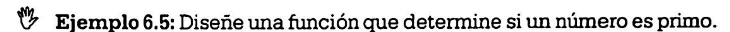
```
1.  algoritmo LlamarFuncion
2.  variables
3.      real: b, h, a
4.  inicio
5.      muestre('Ingrese la base del triángulo:')
6.      lea(b)
7.      muestre('Ingrese la altura del triángulo:')
8.      lea(h)
9.      a = calcularAreaTriangulo(b, h)
10.     muestre('El área es:', a)
11. fin
```

En las líneas 6 y 8 se permite el ingreso de valores para b (base) y h (altura), respectivamente. Supongamos que un usuario asignó el 4 a la variable b y 5 a la variable h. Para calcular el área invocamos la función *calcularAreaTriangulo* en la línea 9 pasándole los valores almacenados en las variables b y h. El valor del argumento b se almacena en el primer parámetro formal de la función (base) y el valor del segundo argumento h se almacena en el segundo parámetro formal de la función (altura). Cuando se invoca la función correctamente se ejecutan las instrucciones que la componen y el algoritmo principal

detiene su ejecución hasta que la función devuelva un valor. En el cuerpo de definición de la función *calcularAreaTriangulo* se calcula el área, se asigna el resultado a la variable *área* y se devuelve el valor almacenado en esta variable. Cuando devuelve el valor termina la ejecución de la función y continúan ejecutándose las instrucciones del algoritmo principal. Por esto, el valor devuelto se asigna a la variable a del algoritmo principal y se muestra el resultado.


Ejemplo 6.4: Diseñe una función que reciba un entero n y retorne la suma de los números enteros entre uno y n.

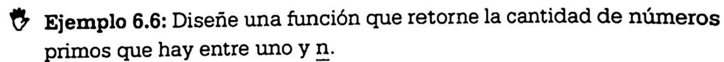
```
1.  entero funcion sumarEnterosConsecutivos(entero: n)
2.  variables
3.      entero: i, sumatoria
4.  inicio
5.      sumatoria = 0
6.      para i = 1 hasta n
7.          sumatoria = sumatoria + i
8.      fin_para
9.      devolver sumatoria
10. fin
```


Ejemplo 6.5: Diseñe una función que determine si un número es primo.

Solución: Este problema ya lo hemos solucionado en dos oportunidades. Aquí, simplemente adaptaremos la solución propuesta en el ejemplo 3.19 para definir la función. La función recibe un número entero y devuelve un valor lógico. Si la función devuelve verdadero significa que el número es primo.

```
1.  logico función esPrimo(entero: n)
2.  variables
3.      entero: i
4.      real: r
5.      logico: primo = verdadero
6.  inicio
7.      si n > 1 entonces
8.          i = 2
```

```
9.          r = techo(raiz2(n))
10.         mientras primo y i <= r hacer
11.             si n mod i == 0 entonces
12.                 primo = falso
13.             si_no
14.                 i = i + 1
15.             fin_si
16.         fin_mientras
17.     si_no
18.         primo = falso
19.     fin_si
20.     devolver primo
21. fin
```

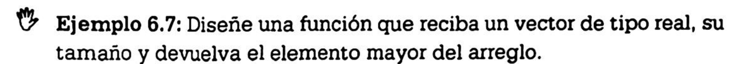


Ejemplo 6.6: Diseñe una función que retorne la cantidad de números primos que hay entre uno y n.

Solución: El diseño de esta función se simplifica porque utilizaremos la función que definimos en el ejemplo anterior para verificar si un número es primo. La función *contarPrimos* recibe un número entero n y genera los números enteros desde dos hasta n mediante las iteraciones de un ciclo. En cada iteración se le pasa a la función *esPrimo* la variable controladora del **para** y verificamos el valor que retorna. Si devuelve verdadero incrementamos la variable *contador* en una unidad.

```
1.  entero funcion contarPrimos(entero: n)
2.  variables
3.      entero: contador, i
4.  inicio
5.      contador = 0
6.      para i = 2 hasta n
7.          si esPrimo(i) == verdadero entonces
8.              contador = contador + 1
9.          fin_si
10.     fin_para
11.     devolver contador
12. fin
```

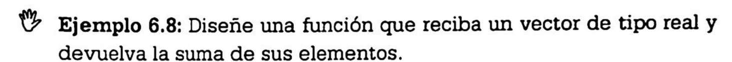

Observación: Cuando un subalgoritmo recibe arreglos como parámetros también debemos pasarle el tamaño del arreglo.



Ejemplo 6.7: Diseñe una función que reciba un vector de tipo real, su tamaño y devuelva el elemento mayor del arreglo.

Solución:

```
1.  real función elementoMayor(real: datos(), entero: n)
2.  variables
3.      entero: i
4.      real: mayor
5.  inicio
6.      mayor = datos(1)
7.      para i = 2 hasta n
8.          si datos(i) > mayor entonces
9.              mayor = datos(i)
10.         fin_si
11.     fin_para
12.     devolver mayor
13. fin
```



Ejemplo 6.8: Diseñe una función que reciba un vector de tipo real y devuelva la suma de sus elementos.

Solución:

```
1.  real función sumarElementosVector(real: datos(), entero: n)
2.  variables
3.      entero: i
4.      real: sumatoria = 0
5.  inicio
6.      para i = 1 hasta n
7.          sumatoria = sumatoria + datos(i)
8.      fin_para
9.      devolver sumatoria
10. fin
```

Ejemplo 6.9: Diseñe una función que reciba dos vectores de tipo real del mismo tamaño los sume y devuelva el resultado. La suma se debe realizar entre las posiciones correspondientes.

```
1.  real( ) función sumarVetores(real: vector1(), vector2(), entero: n)
2.  variables
3.      entero: i
4.      real: vectorSuma(n)
5.  inicio
6.      para i = 1 hasta n
7.          vectorSuma(i) = vector1(I) + vector2(I)
8.      fin_para
9.      devolver vectorSuma
10. fin
```

Ejemplo 6.10: Diseñe una función que reciba una matriz y devuelva la suma de sus elementos.

```
1.  real función sumarElementosMatriz(real: datos(), entero: filas, columnas)
2.  variables
3.      entero: i, j
4.      real: sumatoria = 0
5.  inicio
6.      para i = 1 hasta filas
7.          para i = j hasta columnas
8.              sumatoria = sumatoria + datos(i, j)
9.          fin_para
10.     fin_para
11.     devolver sumatoria
12. fin
```


6.5. PROCEDIMIENTOS

Los procedimientos, al igual que las funciones, realizan una tarea específica. Las funciones retornan un único valor, pero en ocasiones se requieren subalgoritmos que retornen varios valores en lugar de uno solo; en estas situaciones las funciones pierden utilidad y se requiere disponer de otro tipo de subalgoritmo. Los procedimientos pueden retornar más de un valor o no retornar ninguno y tienen la siguiente sintaxis,

```
procedimiento <nombre>(<ParametrosFormales>)
[Declaración de variables]
inicio
    <Instrucciones>
fin
```

En la lista de parámetros formales se debe indicar cuáles parámetros son de entrada (E) y cuáles son de salida (S).

Ejemplo 6.11: Diseñe un procedimiento que calcule el área y la circunferencia de un círculo y los devuelve mediante los parámetros.

Solución:

```
procedimiento circulo(E real: radio; S real: area, circunferencia)
inicio
    area = 3.141592654 * radio * radio
    circunferencia = 2 * 3.141592654 * radio
fin
```

La letra E en los parámetros indica que *radio* es un parámetro de entrada y la letra S indica que *area* y *circunferencia* son parámetros de salida.

Ejemplo 6.11: Diseñe un algoritmo principal donde se ingresa el radio de un círculo, y calcule su área y su circunferencia llamando el procedimiento círculo diseñado en el ejemplo anterior.

```
1.  algoritmo LlamarProcedimientoCirculo
2.  variables
3.      real: r, a, c
4.  inicio
5.      muestre('Entre el radio: ')
6.      lea(r)
7.      circulo(r, a, c)
8.      muestre('El área es ', a)
9.      muestre('La circunferencia es ', c)
10. fin
```

En la línea 7 del algoritmo 6.11 se invoca el procedimiento *círculo* para calcular el área y la circunferencia de un círculo cuyo radio se ingresa en la línea 6 y se asigna a la variable r. Como el procedimiento tiene tres parámetros, uno de entrada y dos de salida, cuando lo llamemos debemos pasarle tres argumentos. Recuerde que los parámetros formales deben corresponder con los parámetros actuales o argumentos en cantidad, orden y tipo. Así, el valor almacenado en r se copia en el parámetro formal de entrada radio y los valores de las variables a y c almacenaran los valores que devuelva el procedimiento mediante los parámetros formales de salida. Una vez llamado el procedimiento se ejecutan las instrucciones presentes en su cuerpo y termina su ejecución cuando llegue a la palabra reservada **fin**. Cuando el procedimiento termina, los valores que se hayan asignado a los parámetros de salida se copian en los argumentos correspondientes. Es decir, si los valores de los parámetros formales de salida se modificaron durante la ejecución del procedimiento este cambio se reflejará en los argumentos correspondientes que se pasaron desde el algoritmo principal. Por tanto, en a y en c están los valores pedidos y se muestran en las líneas 8 y 9. Esto no sucede con la variable r porque el parámetro formal radio es de entrada.

Ejemplo 6.12: Diseñe un procedimiento que muestre los números primos entre 1 y n.

```
1.  procedimiento mostrarPrimos(E entero: n)
2.  variables
3.      entero: i
4.  inicio
5.      para i = 2 hasta n
6.          si esPrimo(i) entonces
7.              muestre(i)
8.          fin_si
9.      fin_para
10. fin
```

Ejemplo 6.13: La junta directiva del Área Metropolitana del Valle de Aburrá, 10 ciudades, ha instalado radares meteorológicos con el fin realizar mediciones de la lluvia en tiempo real en esta zona; la precipitación pluvial se mide en milímetros. Las precipitaciones durante los 30 días del mes de abril tuvieron las siguientes mediciones:

		Ciudades									
		1	2	3	4	5	6	7	8	9	10
Días	1	4.6	5.0	3.2	1.3	1.7	0.7	0.0	1.1	0.4	2.8
	2	3.0	5.1	3.3	2.3	1.7	0.8	0.0	2.0	0.5	2.9
	3	0.0	4.2	3.4	1.5	2.7	0.9	1.0	1.4	0.6	2.1
	4	0.0	3.3	3.5	5.3	1.9	1.0	0.0	3.0	0.7	2.7
	5	0.0	5.4	3.6	1.2	1.7	0.9	2.1	2.7	0.8	2.5
	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.
	29	1,8	0.8	0.0	2.0	0.5	1.2	1.7	1.3	2.1	5.4
	30	0.4	1.2	0.0	1.1	0.4	0.0	5.4	5.3	3.5	1.3

Diseñe un subalgoritmo para cada uno de los siguientes literales:

a) Busque el día o los días de mayor precipitación. Por ejemplo, 5.4 es la mayor precipitación y se presentó el día 5 en la ciudad 2; el día 29 en la ciudad 10 y el día 30 en la ciudad 7.

b) Muestre las precipitaciones promedio de cada ciudad.

c) Devolver un vector que contenga las precipitaciones menores a la precipitación promedio del mes.

d) Muestre la ciudad con menor precipitación en cada uno de los días.

Solución:

a) Busque el día o días de mayor precipitación.

```
procedimiento mostrarDiasMayorPrecipitacion(E real: precipitaciones(),
      entero: días, ciudades)
variables
      entero: i, j
      real: mayorPrecipitacion
inicio
mayorPrecipitacion = precipitaciones(1, 1)
      para i = 1 hasta días %Primero buscamos la precipitación mayor
          para j = 1 hasta ciudades
              si precipitaciones (i, j) > mayorPrecipitacion entonces
                  mayorPrecipitacion = precipitaciones(i, j)
              fin_si
          fin_para
      fin_para
      muestre('La mayor precipitación fue ', mayorPrecipitacion, ' y se presentó:
      ')
      para i = 1 hasta días %Buscamos las precipitaciones que son iguales
      a la mayor
          para j = 1 hasta ciudades
              si precipitaciones(i, j) == mayorPrecipitacion entonces
                  muestre('El dia ', i, ' en la ciudad ', j)
              fin_si
          fin_para
      fin_para
fin
```


b) Muestre las precipitaciones promedio de cada ciudad.

```
procedimiento mostrarPrecipitacionPromedioCiudad(E real: precipitaciones(),
      entero: días, ciudades)
variables
      entero: i, j
      real: sumatoria, promedio
inicio
      para j = 1 hasta ciudades %Recorremos la matriz por columnas
      (ciudades)
          sumatoria = 0
          para i = 1 hasta días
              sumatoria = sumatoria + precipitaciones(i, j)
          fin_para
          promedio = sumatoria / días
          muestre('La precipitación promedio en la ciudad ', j, ' fue ', promedio)
      fin_para
fin
```

c) Devolver un vector que contenga las precipitaciones menores a la precipitación promedio del mes.

```
real() funcion precipitacionMenorPromedioMes(real: precipitaciones(),
      entero: días, ciudades)
variables
      entero: i, j, k
      real: sumatoria, promedio, menor(1)
inicio
sumatoria = 0
      para i = 1 hasta días
          para j = 1 hasta ciudades
              sumatoria = sumatoria + precipitaciones(i, j)
          fin_para
      fin_para
      promedio = sumatoria / (días * ciudades)
k = 0
      para i = 1 hasta días
          para j = 1 hasta ciudades
```

```
                si precipitaciones(i, j) < promedio entonces
                    k = k + 1
                fin_si
            fin_para
        fin_para
    menor(k)
    k = 0
    para i = 1 hasta días
            para j = 1 hasta ciudades
                si precipitaciones(i, j) < promedio entonces
                    k = k + 1
    menor(k) = precipitaciones(i, j)
                fin_si
            fin_para
        fin_para
        devolver menor
    fin
```

d) Muestre la ciudad con menor precipitación en cada uno de los días

```
procedimiento mostrarMenorPrecipitacion(E real: precipitaciones(), entero:
    días, ciudades)
variables
    entero: i, j, ciudadMenorPrecipitacion
    real: menorPrecipitacion
inicio
    para i = 1 hasta días
        menorPrecipitacion = precipitaciones (i, 1)
        ciudadMenorPrecipitacion = 1
     para j = 2 hasta ciudades
            si precipitaciones (i, j) < menorPrecipitacion entonces
            menorPrecipitacion = precipitaciones (i, j)
            ciudadMenorPrecipitacion = j
        fin_si
     fin_para
muestre('La menor precipitación del día ', i, ' se presentó en la ciudad ',
ciudadMenorPrecipitacion)
fin_para
fin
```


ACTIVIDAD EVALUATIVA

1. Diseñe una función que reciba la base y la altura de un triángulo y devuelva el área.
2. Diseñe una función reciba que la base y la altura de un rectángulo y devuelva el perímetro.
3. Diseñe una función que reciba el radio de un círculo y devuelva el área de un círculo.
4. Diseñe una función que calcule y devuelva el perímetro de un círculo.
5. Diseñe una función que calcule y devuelva el área de un trapecio.
6. Diseñe una función que calcule y devuelva el volumen de un cubo.
7. Diseñe una función que calcule y devuelva el volumen de una esfera.
8. Diseñe una función que calcule y devuelva el área superficial de una esfera.
9. Diseñe una función que calcule y devuelva el volumen de una pirámide.
10. Diseñe una función que reciba dos reales y devuelva el mayor de ellos.
11. Diseñe una función que reciba tres reales y devuelva el mayor de ellos.
12. Diseñe una función que reciba dos reales y devuelva el menor de ellos.
13. Diseñe una función que reciba un ángulo en grados y devuelva el ángulo en radianes.
14. Diseñe una función que reciba un ángulo en radianes y devuelva el ángulo en grados.
15. Diseñe una función que devuelva un valor lógico que indique si un año es bisiesto. Un año es bisiesto si es múltiplo del cuatro (2012), excepto los de 100 que no son bisiestos, salvo que, a su vez, también sean múltiplos de 400 (1800 no es bisiesto, 2000 sí).
16. Diseñe una función que reciba un entero n y devuelva su factorial.
17. Diseñe una función que devuelva el valor absoluto de un número.
18. Diseñe una función que reciba dos números enteros positivos a y b y devuelva el residuo de dividir a entre b. No use el operador **mod**.
19. Diseñe una función que reciba dos números enteros a y b y devuelva el número de veces que esta exactamente b en a. No use el operador **div**.

20. Diseñe una función que reciba un entero n y devuelva un valor lógico que indique si n es par o no.

21. Diseñe una función que reciba dos enteros y devuelva el máximo común divisor. Utilice el algoritmo de Euclides.

22. Diseñe una función que reciba como parámetro un entero n y devuelva un valor lógico que indique si n es primo o no.

23. Diseñe una función que reciba un entero n y muestre por pantalla los n primeros números primos.

24. Diseñe una función que reciba un entero n y retorne la cantidad de primos que hay entre uno y n.

25. Diseñe una función que reciba un entero positivo n y devuelva el resultado de calcular la siguiente sumatoria:

26. Diseñe una función que reciba un entero positivo n y devuelva el resultado de calcular la siguiente sumatoria:

27. Diseñe una función que reciba un real x y un entero positivo n y devuelva el resultado de calcular la siguiente sumatoria:

28. Diseñe una función que reciba un real x y un entero positivo n y devuelva el resultado de calcular la siguiente sumatoria:

Cuestionario

Las preguntas 1 y 2 se responden de acuerdo con los resultados de la ejecución del siguiente algoritmo:

```
algoritmo prueba
variables
      entero: i, vector(5)
inicio
      vector(1) = 3
      vector(2) = 5
      para i = 3 hasta 5
            vector(i) = vector(i - 1) + vector(i - 2)
      fin_para
fin
```


1. Después de la ejecución del anterior algoritmo, ¿cuál es el estado del vector?
   a) 3 5 8 13 21
   b) 3 5 8 11 14
   c) 3 5 9 14 21
   d) 3 5 6 7 8
   e) 3 8 6 7 5

2. El número de veces que se ejecuta la instrucción vector(i) = vector(i - 1) + vector(i - 2) es:
   a) 5
   b) 2
   c) 3
   d) 1
   e) 4

Las preguntas 3 y 4 se responden de acuerdo con los resultados de la ejecución del siguiente algoritmo:

```
algoritmo prueba2
variables
      entero: i, a = 20, vector(10)
inicio
      para i = 1 hasta 10
         a = a - 2
         vector(i) = a - 2 * i
      fin_para
fin
```

3. El valor de la cuarta posición del vector es:
   a) 6
   b) 5
   c) 10
   d) 4
   e) 8

4. Cuando el valor de la variable i es 8 el valor de la variable a es:
   a) 2
   b) 6
   c) 4
   d) 8
   e) 10

Las preguntas 5 y 6 se responden de acuerdo con los resultados de la ejecución del siguiente algoritmo:

```
algoritmo prueba3
variables
      entero: i, j, m(6, 4)
inicio
      para i = 1 hasta 6
            para j = 1 hasta 4
                  m(i, j) = (i * 3 + j * 2) mod 7
            fin_para
      fin_para
fin
```

5. Respecto a los valores de la matriz m se puede afirmar que:
   a) Pueden ser negativos
   b) Son inferiores a siete
   c) Son aleatorios
   d) Son mayores que cinco
   e) Están entre cinco y siete

6. El valor de la matriz m en la posición (5, 3) es:
   a) 2
   b) 20
   c) 1
   d) 0
   e) 21

7. Indique la instrucción que debe ir en la línea punteada del algoritmo prueba4 para obtener una matriz con los siguientes valores:

8	6	4
2	0	-2
-4	-6	-8

   a) a = a - 2
   b) a = m(i, j) - 2
   c) a = 2 - m(i, j)
   d) a = a * 2 - 1
   e) a = j * 2

```
algoritmo prueba4
variables
      entero: i, j, a, m(3, 3)
inicio
      a = 8
      para i = 1 hasta 3
            para j = 1 hasta 3
                  m(i, j) = a
                  ......................
            fin_para
      fin_para
fin
```

8. Se quiere almacenar los valores mayores de cada fila de la matriz m un vector v. Indique la instrucción que debe ir en la línea punteada del algoritmo prueba5.
   a) v(j) = mayor
   b) m(i, j) = mayor
   c) mayor = j
   d) v(i) = mayor
   e) mayor = i

```
algoritmo prueba5
variables
      entero: i, j, m(5, 5), v(5), mayor = 0
inicio
      para i = 1 hasta 5
            para j = 1 hasta 5
                  m(i, j) = aleatorio(1000)
                  si m(i, j) > mayor entonces
                        mayor = m(i, j)
                  fin_si
            fin_para
            ............................................
            mayor = 0
      fin_para
fin
```

Bibliografía

Becerra Santamaría C. A. (1995), Turbo Pascal. Programación orientada a objetos, Bogotá: César Barrera.

Becerra Santamaría C. A. (1998), Algoritmos: Conceptos básicos, Bogotá: Kimpres.

Bohm C. & Jacopini G. (1996), 'Flow diagrams, Turing machines, and languages with only two formation rules', Communications of the ACM 9(5), 336-371.

Deitel H. M. & Deitel P. J. (1995), Cómo programar en C/C++, México: Prentice Hall, (2 ed.)

Forsythe A. & Otros (1979), Lenguajes de diagramas de flujo, Limusa.

Heileman G. L. (1997), Estructura de datos, Algoritmos y programación orientada a objetos, Madrid: McGraw Hill.

Joyanes Aguilar L. (1990), Problemas de metodología a la programación, México: McGraw Hill.

Joyanes Aguilar L. (1999), Fundamentos de programación, México: McGraw Hill.

Joyanes Aguilar L. (2008). Fundamentos de Programación. 4.ª edición. México DF: McGraw-Hill

Joyanes Aguilar L. (2013). Fundamentos generales de Programación. México DF: McGraw-Hill.

Joyanes Aguilar L. (2014). Programación en C/C++, Java y UML. 4.ª edición. México DF: McGraw-Hill.

Koffman E. B. (1985), Pascal. Introducción al lenguaje y resolución de problemas con programación estructurada, Wilmington (Estados Unidos): Fondo Educativo Interamericano.

Orrego Villa G. A. (2003), Diseño de algoritmos, Medellín: Escuela de Ingeniería de Antioquia.

Oviedo, E. (2002). Lógica de Programación. Bogotá: Ecoe Ediciones.

Ríos F. (1995), Soluciones secuenciales. Ciencia y Tecnología, Medellín: Universidad de Antioquia.