



NOTAS DE CLASE LÓGICA DE PROGRAMACIÓN

[CON EL PSEUDO LENGUAJE PSEINT]

/* ***** Jaime E. Montoya M. ***** */

NOTAS DE CLASE

LÓGICA DE PROGRAMACIÓN

[CON EL PSEUDO LENGUAJE PSEINT]

```
// * Versión 2.0
// * Fecha: 2024, semestre 2
// * Licencia software: GNU GPL
// * Licencia doc: GNU Free Document License (GNU FDL)
```

Dimension Author[5]

Definir Author Como Caracter

```
Author[1] = "Jaime E. Montoya M."
Author[2] = "Ingeniero Informático";
Author[3] = "Docente y desarrollador"
Author[4] = "Medellín - Antioquia - Colombia"
Author[5] = "2024"
```

Tabla de contenido

<u>Introducción</u>	7
<u>Unidad 0. Preliminares</u>	8
<u>Breve historia de los computadores</u>	8
<u>Desde la antigüedad hasta el siglo XX</u>	8
<u>El Ábaco</u>	8
<u>La Pascalina</u>	9
<u>Stepped Reckoner o máquina de Leibniz</u>	10
<u>El telar de Jacquard</u>	11
<u>La máquina analítica de Babbage</u>	12
<u>Modelo simplificado de la máquina de Babbage</u>	13
<u>El álgebra booleana</u>	15
<u>Primera mitad del siglo XX</u>	15
<u>La máquina de Turing</u>	17
<u>El test de Turing</u>	18
<u>Arquitectura de Von Neumann</u>	19
<u>Las generaciones de computadores</u>	20
<u>Primera generación (1940 - 1955)</u>	20
<u>Segunda generación (1955 - 1965)</u>	22
<u>Tercera generación (1965 - 1975)</u>	22
<u>Cuarta generación (1975 - 1990)</u>	23
<u>Quinta generación (1990 - presente/futuro)</u>	23
<u>Preguntas</u>	24
<u>Ejercicios</u>	24
<u>Unidad 1. Conceptos básicos de programación</u>	25
<u>El computador</u>	25
<u>Periféricos</u>	25
<u>Periféricos de entrada</u>	26
<u>Periféricos de salida</u>	27
<u>Memoria auxiliar o secundaria</u>	27
<u>La Unidad Central de Procesamiento (CPU - Central Processing Unit)</u>	27
<u>Unidad Aritmético Lógica o Unidad de Cálculo (ALU - Arithmetic Logic Unit)</u>	27
<u>Unidad de Control (CU - Control Unit)</u>	27
<u>Registros internos</u>	28
<u>Buses</u>	28
<u>La unidad de memoria</u>	29
<u>Manejo de la memoria</u>	29
<u>Tipos de campos</u>	30
<u>Memoria RAM y ROM</u>	31
<u>Algoritmo</u>	31
<u>Algoritmos “cuantitativos”</u>	32

<u>Algoritmos “cuantitativos”: diagramas de flujo y pseudocódigo</u>	33
<u>Programa</u>	33
<u>El proceso de la programación</u>	34
<u>Los lenguajes de programación</u>	34
<u>Tipos de lenguajes</u>	34
<u>Lenguajes de máquina</u>	34
<u>Lenguajes de bajo nivel</u>	35
<u>Lenguajes de alto nivel</u>	35
<u>Editores de textos</u>	36
<u>Traductores de lenguaje</u>	37
<u>Intérpretes</u>	37
<u>Compiladores</u>	38
<u>Fases de la compilación</u>	38
<u>Datos y tipos de datos</u>	38
<u>Tipos de datos numéricos</u>	39
<u>Tipo de dato carácter/cadena</u>	39
<u>Tipo de dato lógico (booleano)</u>	40
<u>Constantes y Variables</u>	40
<u>Expresiones</u>	40
<u>Operadores básicos en matemáticas y computación</u>	41
<u>Operadores aritméticos</u>	41
<u>Operación de módulo y división entera</u>	42
<u>Operadores relationales o de comparación</u>	42
<u>Operadores lógicos (booleanos)</u>	43
<u>Tabla de verdad de los operadores lógicos</u>	43
<u>Operación de asignación</u>	43
<u>Salida de información</u>	44
<u>Entrada de información</u>	45
<u>Notación algorítmica</u>	45
<u>Declaración de variables y definición de constantes</u>	46
<u>Lenguajes de programación fuertemente tipados</u>	47
<u>Lenguajes de programación débilmente tipados</u>	47
<u>Comentarios</u>	48
<u>Errores</u>	49
<u>Errores de sintaxis</u>	49
<u>Errores de lógica/ejecución</u>	49
<u>Palabras reservadas</u>	49
<u>Notaciones comunes en programación</u>	50
<u>Notación camel case</u>	50
<u>Notación snake case</u>	51
<u>Notación húngara</u>	51
<u>Como interpretar la sintaxis de una instrucción</u>	52
<u>Problemas resueltos</u>	52
<u>Preguntas</u>	56

<u>Ejercicios</u>	57
<u>Unidad 2. Estructuras de control</u>	59
<u>Condicionales</u>	59
<u>Condicional simple</u>	59
<u>Condicional compuesto</u>	61
<u>Condicional anidado</u>	62
<u>Selector múltiple o estructura caso</u>	64
<u>Ciclos</u>	66
<u>Ciclo Mientras</u>	66
<u>Prueba de escritorio</u>	68
<u>Contadores y acumuladores</u>	69
<u>Registro centinela</u>	71
<u>Banderas o suiches</u>	72
<u>Ciclo Repetir ... Hasta</u>	73
<u>Ciclo Para</u>	78
<u>Ciclos anidados</u>	80
<u>Problemas resueltos</u>	81
<u>Preguntas</u>	86
<u>Ejercicios</u>	87
<u>Unidad 3. Subprogramas: procedimientos y funciones</u>	90
<u>Funciones incorporadas o predefinidas</u>	90
<u>Funciones matemáticas</u>	90
<u>Funciones para la manipulación de cadenas de caracteres</u>	92
<u>Funciones para la conversión de tipos de datos</u>	94
<u>Bifurcación de control</u>	95
<u>Interrumpir</u>	95
<u>Continuar</u>	96
<u>Salir</u>	96
<u>Retornar</u>	97
<u>Subprogramas o subalgoritmos</u>	97
<u>Procedimientos y Funciones</u>	99
<u>Funciones</u>	99
<u>Invocar (llamar) una función</u>	100
<u>Parámetros de un subprograma. Parámetros actuales y formales</u>	101
<u>Procedimientos</u>	102
<u>Invocar (llamar) un procedimiento</u>	103
<u>Ámbito de las variables. Variables globales y locales</u>	104
<u>Preguntas</u>	106
<u>Ejercicios</u>	106
<u>Unidad 4. Arreglos e Introducción a la Programación Orientada a Objetos (POO)</u>	109
<u>Estructuras de datos</u>	109
<u>Arreglos</u>	110
<u>Arreglos unidimensionales: Vectores</u>	110
<u>Representación en memoria</u>	111

<u>Declaración de vectores</u>	111
<u>Acceso a los elementos de un vector</u>	111
<u>Arreglos bidimensionales: Matrices o tablas</u>	114
<u>Representación en memoria</u>	115
<u>Declaración de matrices</u>	115
<u>Acceso a los elementos de una matriz</u>	116
<u>Arreglos multidimensionales</u>	117
<u>Representación en memoria</u>	118
<u>Declaración de arreglos multidimensionales</u>	118
<u>Acceso a los elementos de un arreglo multidimensional</u>	119
<u>Programación Orientada a Objetos (POO)</u>	121
<u>Clases y objetos</u>	122
<u>Características de la POO</u>	123
<u>Abstracción</u>	123
<u>Ocultación u ocultamiento</u>	123
<u>Modularidad</u>	124
<u>Encapsulamiento</u>	124
<u>Herencia</u>	124
<u>Reutilización</u>	124
<u>Polimorfismo</u>	124
<u>Modificadores de acceso</u>	124
<u>Público (Public)</u>	125
<u>Privado (Private)</u>	125
<u>Protegido (Protected)</u>	125
<u>Declarar una clase</u>	125
<u>Instanciar una clase</u>	126
<u>Eliminar una instancia de una clase. Recolector de basura</u>	127
<u>Preguntas</u>	131
<u>Ejercicios</u>	131
<u>Fuentes y referencias adicionales</u>	134
<u>Bibliografía del microcurrículo</u>	134
<u>Referencias adicionales</u>	134

Introducción

Este documento es un complemento a las clases presenciales y virtuales, y está basado en la bibliografía del curso, así como de otras fuentes adicionales que se indican a lo largo del texto, además de la experiencia del autor en su función docente en las áreas de programación y como desarrollador en distintas empresas del departamento. No se pretende reemplazar los textos guías con este manual, sino servir de ayuda didáctica y apoyo académico a los estudiantes.

La guía incluye, además de los conceptos teóricos, ejemplos, gráficas, desarrollos en clase, y al final de cada unidad, unas preguntas, ejercicios y una actividad que puede corresponder al entregable de seguimiento acordado desde el inicio de la asignatura. Se espera que estos ejercicios permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Al final de este manual, se indican fuentes y referencias adicionales que el estudiante puede consultar.

Este documento incluye una serie de ejercicios resueltos de programación realizados en clase y propuestos por el autor que se anexan finalizando cada unidad.

Estos ejercicios incluyen algunos conceptos teóricos y prácticos adicionales, consejos, buenas prácticas y ejemplos explicados, divididos de acuerdo a la unidad temática estudiada. Se espera que estos ejercicios permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Los ejemplos se ilustran con pseudocódigo realizado en el pseudo lenguaje PSeInt, y en algunos casos, se muestra el pseudocódigo puro, así como diferentes técnicas de diagramación, entre ellas, los *flujoogramas* o *diagramas libres* y los *diagramas rectangulares* o de *Nassi-Schneiderman*.

Además del énfasis algorítmico, también se muestran algunos de estos ejemplos resueltos llevados al lenguaje de programación Python, donde el estudiante podrá poner a prueba sus conocimientos, así como analizar las diferencias entre la teoría y la práctica, esto es, llevar un algoritmo y convertirlo en programa.

Unidad 0. Preliminares

Breve historia de los computadores

Podemos mirar la historia de las máquinas de cálculo y los computadores actuales desde los hombres primitivos hasta finales del siglo XIX en plena revolución industrial y el desarrollo dado en el siglo XX con el avance de la electrónica.

Desde la antigüedad hasta el siglo XX

El hombre desde la antigüedad ha usado las matemáticas para su beneficio, convirtiéndose en una herramienta fundamental para su progreso. Los hombres primitivos tuvieron un primer contacto con las matemáticas al realizar pequeños **conteos** de elementos que recolectaban o poseían, tales como frutas, ganado, número de integrantes de la comunidad, entre otros. Estos hombres representaban algunas cantidades que consideraban de importancia usando símbolos, pero aún no tenían una abstracción del número como tal. A ellos se les atribuye como los creadores de los primeros *sistemas numéricos no posicionales*.

El Ábaco

Se estima que hace unos cinco mil años, los chinos inventan el **Ábaco**, un dispositivo compuesto de unas barras paralelas que permiten mover unas fichas sobre ellas para realizar conteos, sumas, restas y otras operaciones aritméticas. Es considerada la primera máquina de cálculo de la historia.

La siguiente figura muestra un ábaco chino.¹

¹ Algunas referencias y la imagen fueron tomadas de: [Ábaco - Wikipedia, la enciclopedia libre](#)

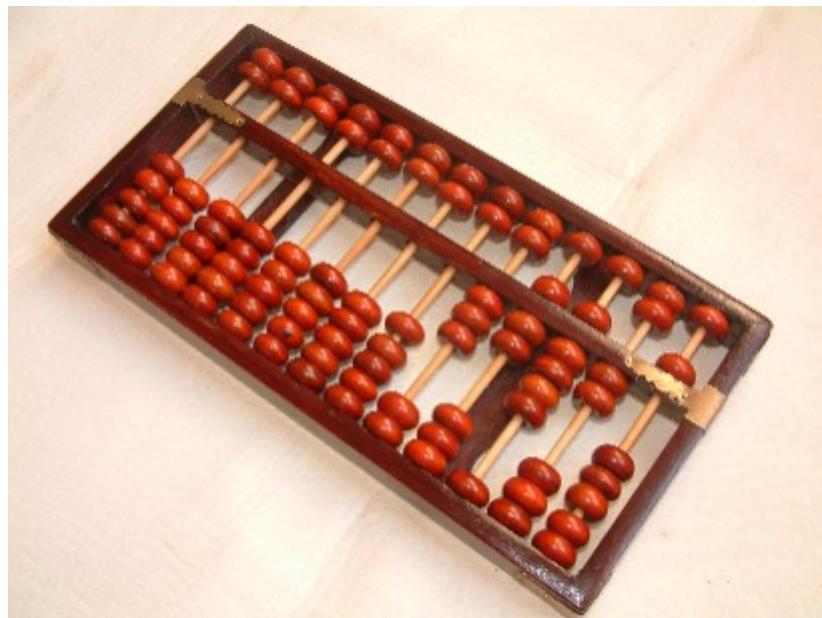


Figura 0.1. Ábaco chino

Pueblos como los babilonios, persas, chinos y griegos hicieron avances significativos en ciencias en general y matemáticas en particular, gracias a que contaban con sistemas numéricos adecuados que permitían este avance, caso contrario a lo sucedido con el pueblo romano, que aunque fue avanzado en lo cultural, carecía de un sistema numérico adecuado para realizar cálculos, lo que trajo consigo un estancamiento de la matemática allí.

Los algoritmos, base y fuente de la informática, se conocen desde la antigüedad; al matemático griego [Euclides](#) se le atribuye la creación de un algoritmo para encontrar el [Máximo Común Divisor \(MCD\)](#) de dos números enteros.

Con el advenimiento de la *Edad Media*, la ciencia se paralizó en una época conocida como “*El Oscurantismo*” y no hubo desarrollos científicos, sumiendo al mundo conocido en superstición y pseudociencia.

El final de esta etapa de la humanidad está en gran parte marcado por la rebeldía de los científicos al no poder callar acerca de sus observaciones y estudios, particularmente en el campo de la astronomía, lo que permite la entrada de una nueva época conocida com *El Renacimiento*, donde la ciencia vuelve a florecer, trayendo consigo la *Revolución Industrial*.

La Pascalina

En 1642 (año del fallecimiento de [Galileo Galilei](#) y del nacimiento de [Isaac Newton](#)), [Blaise Pascal](#) (1623 - 1662), un matemático, escritor y filósofo francés, inventa una máquina para realizar sumas y restas. A esta máquina la nombró inicialmente “**máquina de aritmética**”, luego “**rueda pascalina**” y por último la llamó **Pascalina**; tiene la particularidad de aplicar el concepto de **acumulador**, tan utilizado en programación y es considerada un antepasado de los computadores actuales. Esta calculadora mecánica funcionaba a base de ruedas

dentadas y engranajes. El científico computacional Niklaus Wirth creó el lenguaje de programación **Pascal**, muy empleado particularmente en ambientes académicos hasta los años 90's en honor a este importante matemático.

La siguiente figura muestra la cubierta de la pascalina y su mecanismo en una de las obras del mismo Pascal.²

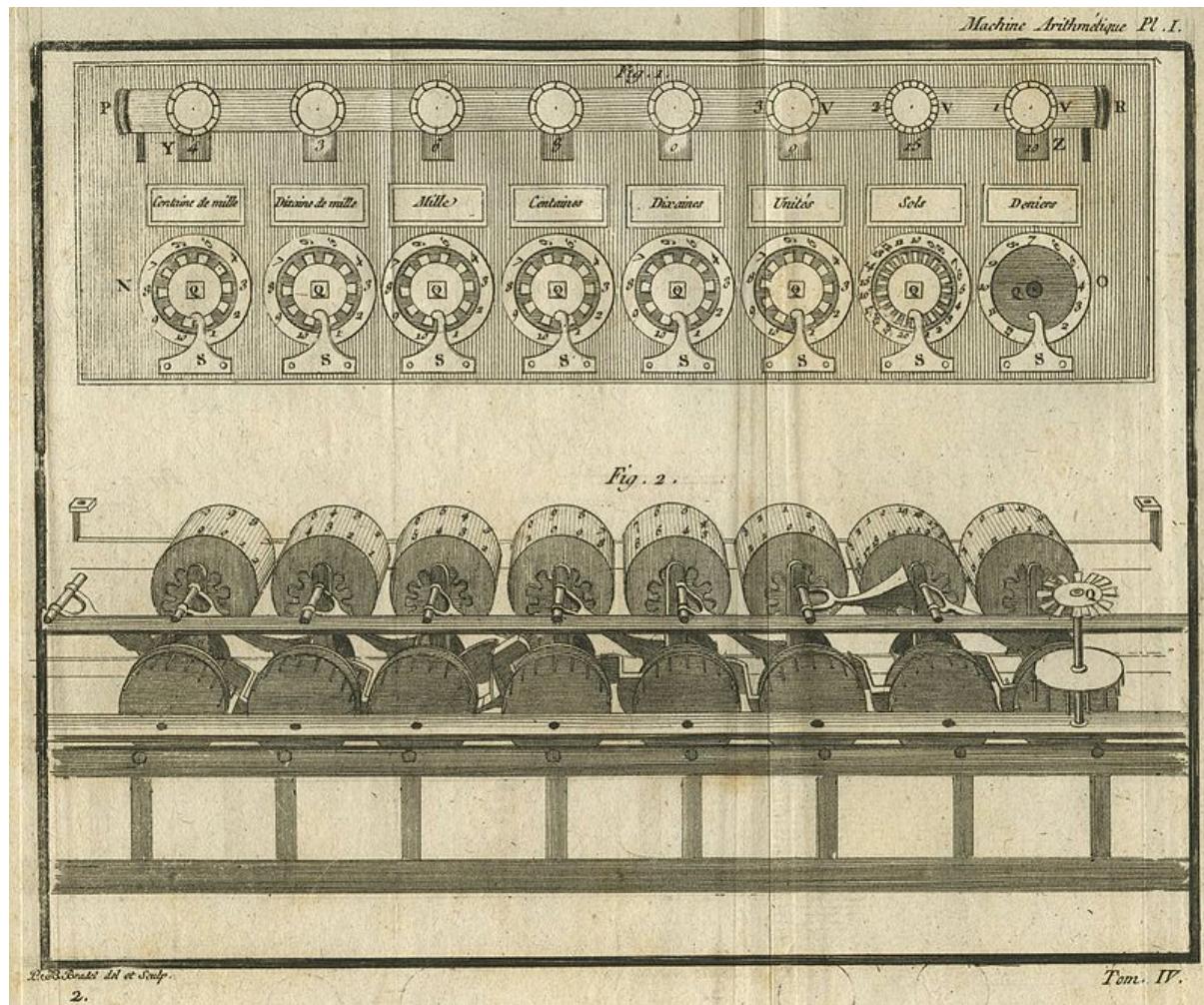


Figura 0.2. Planos originales de la pascalina

Stepped Reckoner o máquina de Leibniz

[Gottfried Wilhelm Leibniz](#) (1646 - 1716), un matemático y filósofo alemán, inventor junto con Newton del cálculo infinitesimal, aunque de forma independiente y casi al mismo tiempo, desarrolló en 1672 otra máquina que mejoraba la pascalina extendiendo las ideas de Pascal. Dicho artefacto permitía realizar las cuatro operaciones aritméticas básicas, además de calcular raíces cuadradas. Esta calculadora mecánica fue utilizada hasta la llegada de las calculadoras electrónicas en los años 70's y se conoce con el nombre de **Stepped Reckoner** o **máquina o rueda de Leibniz**.

² Algunas referencias y la imagen fueron tomadas de: [Pascalina - Wikipedia, la enciclopedia libre](#)

La siguiente figura muestra una réplica de la máquina de Leibniz.³

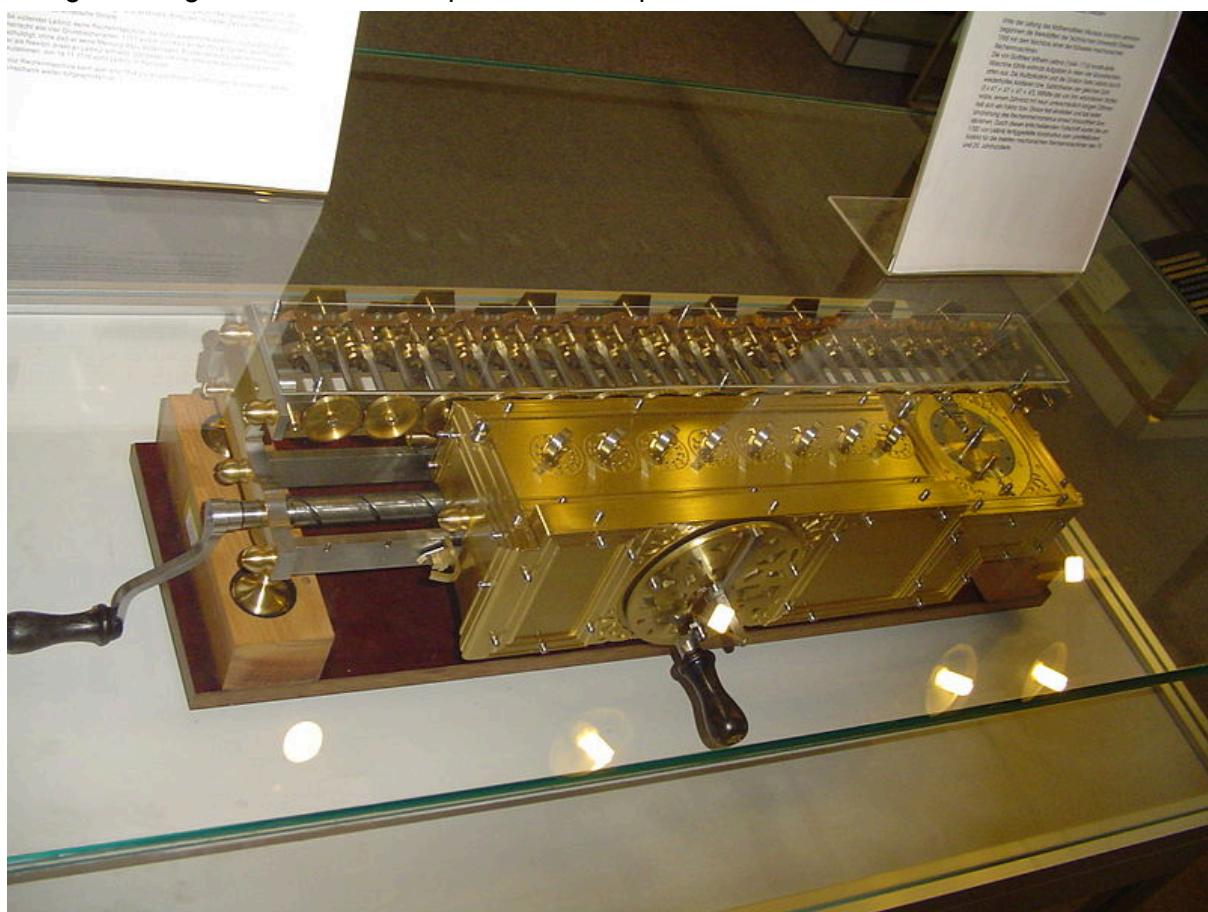


Figura 0.3. Réplica de la máquina de Leibniz

El telar de Jacquard

En 1804 Joseph Marie Jacquard, un comerciante francés, inventa un dispositivo que utilizaba *tarjetas perforadas* para tejer patrones sobre las telas, permitiendo que operarios sin muchos conocimientos pudieran elaborar diseños elaborados y complejos. Las tarjetas perforadas contienen una serie de marcas en un código binario y que son leídas por el dispositivo. Hasta solo hace unos cuantos años, el mismo principio de tarjeta perforada se utilizó en los computadores electrónicos.

La siguiente figura muestra una tarjeta perforada en un telar de Jacquard.⁴

³ Algunas referencias y la imagen fueron tomadas de: [Rueda de Leibniz - Wikipedia, la enciclopedia libre](#)

⁴ Algunas referencias y la imagen fueron tomadas de: [Telar de Jacquard - Wikipedia, la enciclopedia libre](#)

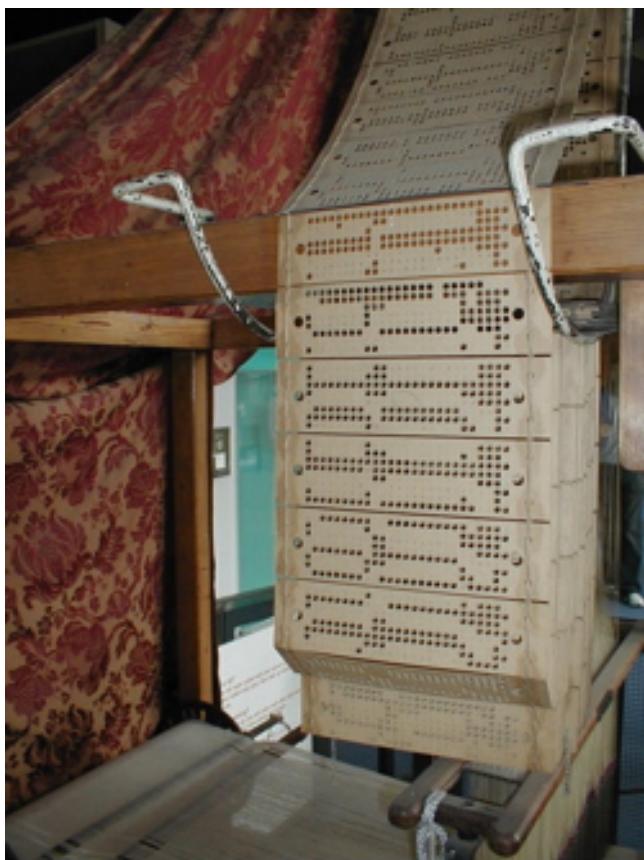


Figura 0.4. Tarjeta perforada en un telar de Jaquard

La máquina analítica de Babbage

En 1835 el matemático inglés [Charles Babbage](#) (1791 - 1871) inicia la construcción de la **máquina analítica**, que presenta en 1837 y que continúa mejorando hasta el año de su fallecimiento en 1871. Dicha máquina es la representación de un computador actual, y es por ello que a su creador, junto con Alan Turing, se les considera los padres de la informática. Las trabas políticas argumentando un posible mal uso de la máquina y las dificultades tecnológicas de la época, no permitieron que esta máquina pudiera terminarse de construir, pero sirvió de base para los computadores actuales especificando cómo debía ser su arquitectura.

La matemática inglesa [Ada Lovelace](#) (de cuyo nombre nace el lenguaje de programación orientado a objetos *Ada* en su honor), hija del poeta Lord Byron y de madre aristócrata fue muy cercana al trabajo de Babbage y redacta un artículo donde argumenta que la máquina analítica tenía más aplicaciones que el solo cálculo y escribe lo que se conoce como el primer algoritmo que puede ser procesado por una máquina, por lo que es considerada como la primera programadora de computadores de la historia. Dicho algoritmo permite calcular los números de Bernoulli.

Modelo simplificado de la máquina de Babbage

El modelo de la máquina analítica de Babbage consta de las siguientes partes:

Unidad de entrada

Compuesta por dispositivos que permiten introducir información, órdenes o datos a la máquina.

Unidad de salida

Compuesta por dispositivos que permiten mostrar la información proveniente de la máquina y con alguna utilidad para alguien.

Unidad de memoria

Dispositivo utilizado para recordar las instrucciones necesarias para que la máquina inicie, así como los datos que se le ingresen y con los que debe operar.

Unidad de cálculos

Tiene como fin efectuar las órdenes impuestas a la máquina y evaluar las preguntas que el programa debe resolver.

Unidad de control

Se encarga de coordinar y controlar la máquina, haciendo la tarea de administrador de ésta y permitiendo que funcione automáticamente sin intervención manual. Esta unidad no es como tal un dispositivo físico, a diferencia de las anteriores, sino que es un programa que contiene un conjunto de instrucciones para controlar los dispositivos y las órdenes dadas a la máquina por un hombre.

La siguiente figura muestra el modelo simplificado del *computador* propuesto por Babbage:

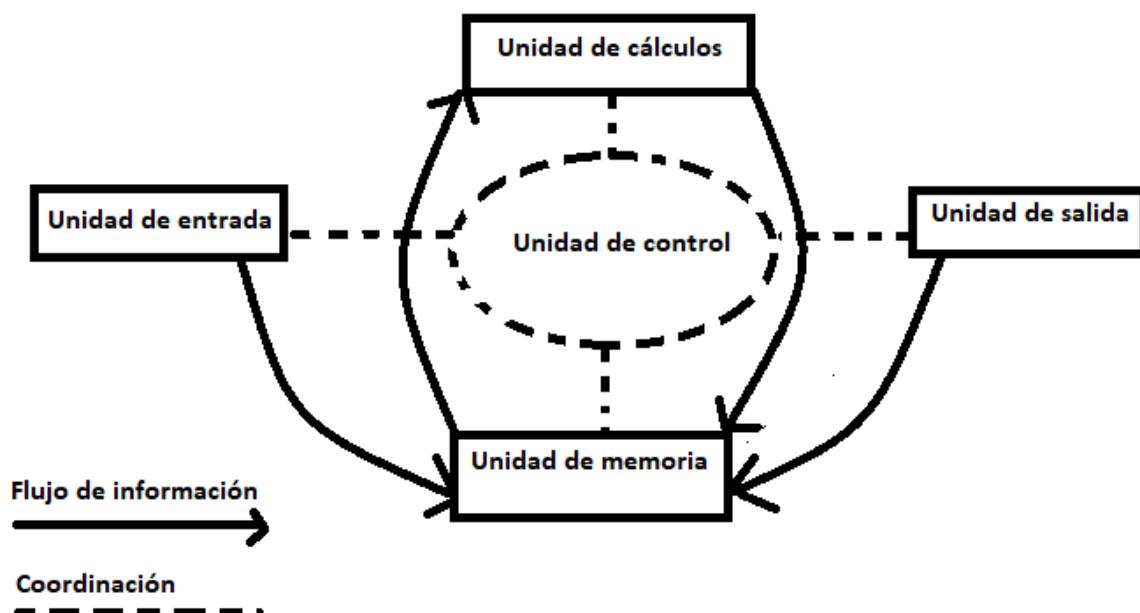


Figura 0.5. Modelo simplificado de un computador

El conjunto conformado por las unidades de cálculo, memoria y control es lo que se conoce actualmente como la **CPU (Central Processing Unit - Unidad de Procesamiento Central)**. En términos prácticos, podemos afirmar que la CPU es el computador.

La siguiente figura muestra la máquina analítica de Babbage.⁵

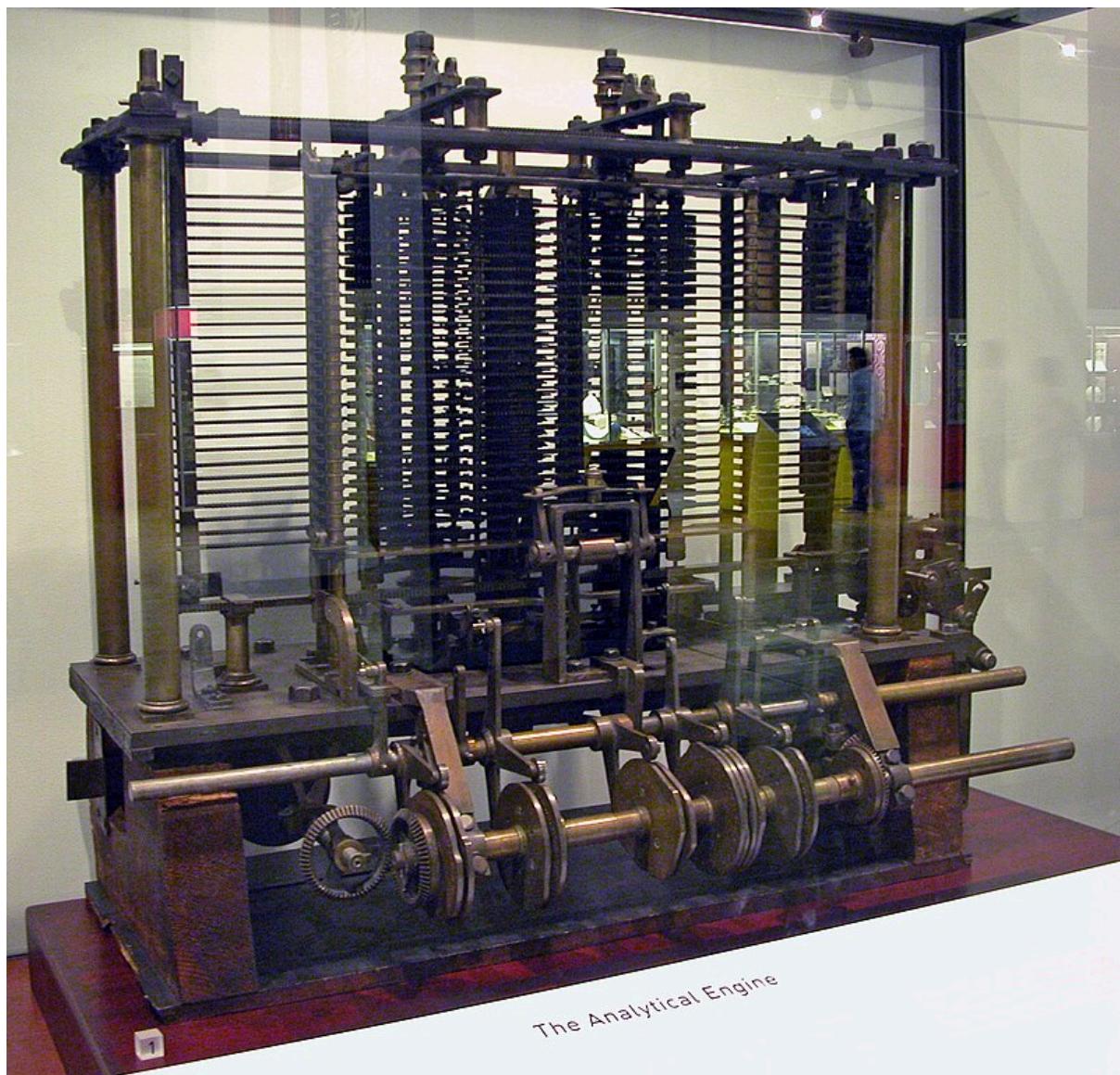


Figura 0.6. Máquina analítica de Babbage

⁵ Algunas referencias y la imagen fueron tomadas de: [Máquina analítica - Wikipedia, la enciclopedia libre](#)

El álgebra booleana

El matemático inglés [George Boole](#), inventor del álgebra que lleva su nombre, establece los fundamentos de la aritmética computacional moderna; es también considerado como uno de los fundadores del campo de las ciencias de la computación. En 1854 publicó “*An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*”, donde desarrolló un sistema de reglas para expresar, manipular y simplificar problemas lógicos y filosóficos cuyos argumentos admiten dos estados (verdadero o falso) por procedimientos matemáticos. Boole también es el creador y padre de los operadores lógicos simbólicos, y gracias a su trabajo, es posible operar simbólicamente para realizar diferentes operaciones lógicas. El álgebra de Boole es materia de estudio en distintos programas de ingeniería por sus múltiples aplicaciones a la informática, electrónica y mecánica.

Primera mitad del siglo XX

En la década de 1920 fue enunciado el [*Entscheidungsproblem*](#) (*problema de decisión*)⁶ por el matemático alemán [David Hilbert](#) donde planteó el asunto sobre la *decibilidad* de las matemáticas, esto es, si hay un método definido que pueda aplicarse a cualquier lógica de predicados y que pueda concluir si se trata de un teorema.

El matemático alemán [Kurt Gödel](#) demuestra la *completitud* del cálculo de predicados en 1930 y para el año siguiente cierra las cuestiones referentes a la completitud y de la consistencia de ciertas teorías matemáticas, dando así solución a los planteado por Hilbert. Gödel enuncia y demuestra los siguientes teoremas:

Primer teorema de incompletitud de Gödel: cualquier teoría que contenga la aritmética de Peano convenientemente axiomatizada, y sea consistente, es incompleta.

Segundo teorema de incompletitud de Gödel: es imposible dar una demostración de la consistencia de la teoría ‘dentro’ de la teoría.

El segundo teorema nos dice que la matemática (y la ciencia en general) no se demuestra así misma y que es necesario partir de supuestos (axiomas, postulados) para su construcción y desarrollo; a su vez, está muy relacionado con los planteamientos que un poco después propusieron Church y Turing simultáneamente, pero de forma independiente, acerca del *Entscheidungsproblem*. Gödel más adelante comentaría (1963) que las aportaciones de Turing permitían “una definición precisa e indudablemente adecuada de la noción general de sistema formal”⁷.

En 1936 el matemático inglés [Alan Turing](#), considerado como uno de los padres de la informática moderna junto con Babbage, mencionó por primera vez el concepto de

⁶ Puede leer acerca del *Entscheidungsproblem* en: [Entscheidungsproblem - Wikipedia, la enciclopedia libre](#)

⁷ Acerca de algunas similitudes entre las teorías propuestas por Gödel y Turing, puede remitirse al siguiente artículo: [Algunos vínculos entre los teoremas de Gödel y Turing](#)

“[máquina de Turing](#)” en su trabajo “*On computable numbers, with an application to the Entscheidungsproblem*” publicado por la Sociedad Matemática de Londres. En dicho trabajo, Turing resuelve el problema del *Entscheidungsproblem* un poco después que [Alonzo Church](#), pero de forma independiente.

Church, que se basó en el trabajo de Stephen Kleene, demostró que no existe un algoritmo definido por funciones recursivas que decida para dos expresiones del [cálculo lambda](#) si son equivalentes o no, con lo que daba una respuesta negativa a dicho problema.

Turing redujo este problema al [problema de la parada](#) para las **máquinas de Turing**. Dicho problema es el siguiente:

“Dada una Máquina de Turing M y una palabra ω , determinar si M terminará en un número finito de pasos cuando es ejecutada usando ω como dato de entrada”.

En su famoso artículo, Turing demuestra que dicho problema es *indecidible*, es decir, *no computable o no recursivo*, en el sentido de que ninguna máquina de Turing lo puede resolver.

En términos prácticos, Turing encontró que hay problemas que no pueden ser resueltos algorítmicamente y que no hay una “**máquina Universal de Turing**” que resuelva cualquier problema.

Otro aspecto relevante en la práctica de problemas irresolubles es el siguiente:

La definición de algoritmo nos dice que es un “*conjunto de pasos finito y ordenado para llegar a un resultado*”; al ejecutar un programa, siempre buscamos que éste detenga su flujo en algún momento para que nos entregue algún resultado, pero es posible que éste permanezca en una ejecución indefinida sin entregar resultado alguno. Este último caso de una ejecución “infinita” generalmente se presenta porque el programa entró en un ciclo (bucle) infinito que termina trabando o bloqueando el programa, y en ocasiones afectando el funcionamiento del mismo sistema operativo.

Lo que se pretende en informática con el problema de la parada, es resolver si existe el programa **P**, tal como se indica a continuación:

“Existe un programa **P**, tal que, dado un programa cualquiera **q** y unos datos de entrada **x**, muestre como salida **1** si **q** con entrada **x** termina en un número finito de pasos o muestre como salida **0** si **q** con **x** entra a un bucle infinito”

Este chequeo permite controlar si los datos de entrada producirán un número infinito de pasos y así controlarlo, algo muy importante y que permite concluir que en la informática no hay o no pueden haber **ciclos infinitos**.

La prueba de Turing ha tenido más influencia que la de Church. Ambos trabajos se vieron influidos por trabajos anteriores de Kurt Gödel sobre el teorema de incompletitud, especialmente por el método de asignar números a las fórmulas lógicas para poder reducir la lógica a la aritmética. Dichas investigaciones se conocen en el argot científico como la “[Tesis de Church - Turing](#)”.

La máquina de Turing

Es un dispositivo teórico matemático que puede procesar símbolos (por simplicidad, binarios, ceros y unos, con los cuales podemos representar cualquier número, carácter o símbolo del alfabeto) sobre una **cinta** de longitud infinita que sigue determinadas reglas y que se considera la memoria de la máquina con capacidad ilimitada.

Una de las operaciones fundamentales de la máquina es que permite desplazar la cinta hacia atrás (izquierda) o hacia adelante (derecha), y está marcada con cuadrados donde se puede **imprimir** un símbolo (como un 1, por ejemplo) o donde simplemente no hay nada (un blanco -0-). En cualquier momento la máquina puede **leer** un símbolo (**símbolo leído**) y puede modificarlo; dicho símbolo determina el comportamiento de la máquina, diciéndole que debe hacer; los demás símbolos ubicados en otros lugares de la cinta, no afectan el comportamiento de la máquina. Dicha lectura y escritura se efectúa mediante una **cabeza** o **cabezal** de lecto/escritura ubicada en un punto fijo por donde la cinta se desplaza, y dado que lo hace hacia atrás o hacia adelante, todos los símbolos tienen la oportunidad de ser leídos, incluso varias veces.

La máquina también tiene un **registro de estado** que almacena el estado (alguno de los finitos) y un estado especial que permite determinar cuándo inicia a operar la máquina.

La máquina cuenta con una **tabla finita de instrucciones** (llamada también **tabla de acción** o **función de transición**) que contiene las instrucciones de cómo debe operar la ésta siguiendo una secuencia ordenada de pasos, esto es, ejecutar un algoritmo.

La siguiente figura muestra la poderosa idea en un simple diseño de una máquina de Turing:

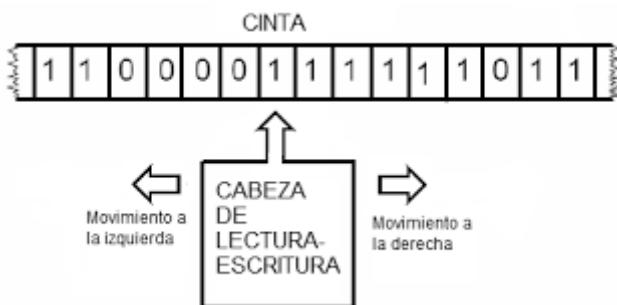


Figura 0.7. Máquina de Turing⁸

⁸ Figura tomada de: [Redalyc.DESARROLLO DE UN ENTORNO DE SIMULACIÓN PARA AUTÓMATAS DETERMINISTAS](#)

Una máquina de Turing que es capaz de simular cualquier otra máquina de Turing, es llamada una **Máquina Universal de Turing (UTM)** por sus siglas en inglés).

La tesis de Church-Turing mencionada más arriba, establece que las máquinas de Turing son un método eficaz, y si se quiere informal, en lógica y matemáticas, que proporciona una definición clara y concisa de algoritmo o 'procedimiento mecánico'.

El test de Turing⁹

Es una prueba para medir la capacidad de una máquina de simular el comportamiento "inteligente" de un ser humano. La idea propuesta por Turing en 1950 consistía en que una persona usando el lenguaje natural pudiera tener una conversación con una máquina diseñada para responder a las preguntas que aquella persona le realizará y de forma similar a los humanos. La persona y la máquina se separan, incluyendo un tercero que puede ver la conversación pero no intervenir y que sabe que uno de los interlocutores es una máquina. La interfaz de comunicación es irrelevante, por lo que se asume una pantalla y un teclado para comunicarse a través de mensajes de texto. Si luego de un tiempo (que Turing estimaba de 5 minutos de conversación o 70% del tiempo total) el tercero no lograba distinguir quién era la máquina, entonces se concluía que ésta había pasado la prueba.

Turing propuso esta prueba en su ensayo *Computing Machinery and Intelligence*. Éste inicia con estas palabras: "Propongo que se considere la siguiente pregunta, '¿Pueden pensar las máquinas?'. Turing replantea luego la pregunta con otra: "¿Existirán computadoras digitales imaginables que tengan un buen desempeño en el juego de imitación?". Turing estaba convencido que esta pregunta sí era posible de responder y en buena parte de dicho trabajo argumenta en contra de las objeciones a la idea de que "las máquinas pueden pensar".

Esta prueba ha sido muy influyente, así como ampliamente criticada, y aunque el tema relacionado con la capacidad de las máquinas de "pensar" estuvo en cierta forma quieto durante varios años, se ha retomado con gran fuerza, llevando el concepto de la **Inteligencia artificial (IA)** a un lugar importante en la filosofía moderna y en el desarrollo de software actual.

Por este trabajo, Turing también es considerado uno de los padres de la IA junto a John McCarthy, Marvin Minsky, Nat Rochester y Claude Shannon quienes bautizaron esta disciplina¹⁰.

⁹ Puede leer más al respecto en el siguiente artículo: [Prueba de Turing - Wikipedia, la enciclopedia libre](#)

¹⁰ Puede leer más al respecto en el siguiente artículo: [Los padres de la Inteligencia Artificial no son del siglo XXI](#)

Arquitectura de Von Neumann

También conocida como **arquitectura Princeton**, es una arquitectura de computadores basada en la descrita en 1945 por el matemático y físico [John von Neumann](#) junto a otros científicos, en el primer borrador de un informe sobre el *EDVAC*. Allí describe una arquitectura de diseño para un computador digital electrónico con partes que constan de una unidad de procesamiento que contiene una unidad aritmético lógica y registros del procesador, una unidad de control que contiene un registro de instrucciones y un contador de programa, una memoria para almacenar tanto datos como instrucciones, almacenamiento masivo externo, y mecanismos de entrada y salida. El concepto ha evolucionado para convertirse en un computador de programa almacenado en el cual no pueden darse simultáneamente una búsqueda de instrucciones y una operación de datos, ya que comparten un bus en común, lo que se conoce como el *cuello de botella Von Neumann*, y muchas veces limita el rendimiento del sistema¹¹.

El diseño de una arquitectura von Neumann es más simple que la **arquitectura Harvard** más moderna, que también es un sistema de programa almacenado, pero tiene un conjunto dedicado de direcciones y buses de datos para leer datos desde memoria y escribir datos en la misma, y otro conjunto de direcciones y buses de datos para ir a buscar instrucciones.

Un ordenador digital de programa almacenado es aquel que mantiene sus instrucciones de programa, así como sus datos, en una memoria de acceso aleatorio (RAM) de lectura-escritura. Los computadoras de programa almacenado representaron un avance sobre los ordenadores controlados por programas de la década de 1940, como la Colossus y la ENIAC, que se programaron mediante el establecimiento de conmutadores y la inserción de cables de interconexión para enrutar datos y para controlar señales entre varias unidades funcionales. En la gran mayoría de las computadoras modernas, se utiliza la misma memoria tanto para datos como para instrucciones de programa, y la distinción entre von Neumann vs. Harvard se aplica a la arquitectura de memoria caché, pero no a la memoria principal.

En la siguiente figura podemos observar un diagrama de la arquitectura von Neumann, la cual no dista de lo propuesto por Babbage con su máquina analítica.

¹¹ N. del A.: El programa almacenado en los computadores actuales se encuentra en la **ROM** (*Read Only Memory* - Memoria de Solo Lectura) que cuenta con las instrucciones para que la máquina inicie, así como la **RAM** (*Random Access Memory* - Memoria de Acceso Aleatorio) donde puede almacenar datos temporalmente y realizar operaciones.

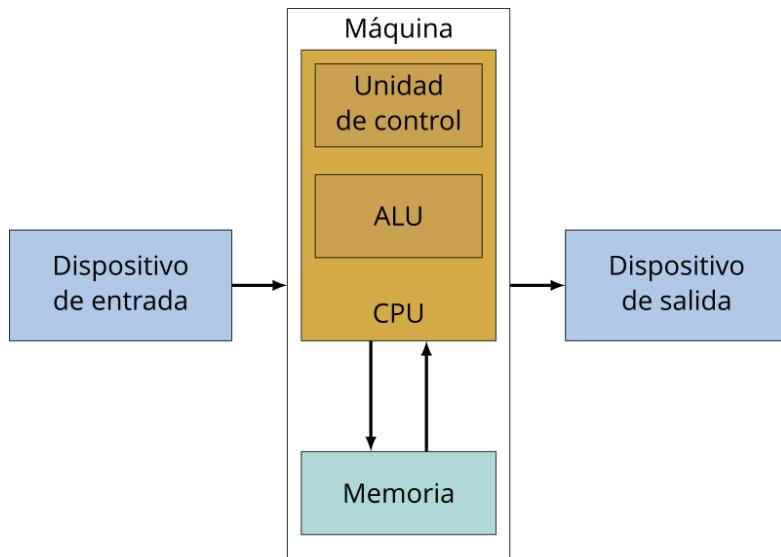


Figura 0.8. Diagrama de la arquitectura von Neumann.

Las generaciones de computadores

Las llamadas generaciones de computadores son períodos marcados por grandes desarrollos a nivel informático, tanto lo referente a hardware como software, que comienza desde 1940 hasta el presente/futuro. En general, se habla de cinco generaciones¹².

Primera generación (1940 - 1955)

El computador [Z3](#), creado por el ingeniero alemán [Konrad Zuse](#) en 1941, fue la primera máquina programable y completamente automática, características que debe cumplir un computador, por lo que es considerada por muchos como el primer computador de la historia, a pesar de algunas controversias al respecto. Su funcionamiento era electromecánico, estaba construido con 2300 relés, tenía una frecuencia de reloj de ~5Hz, y una longitud de palabra de 22 bits. Los cálculos eran realizados con aritmética en coma flotante puramente binaria. En el *Deutsches Museum* se encuentra una réplica de esta máquina, ya que la original fue destruida en un bombardeo en Berlín en 1943. En 1998 [Raúl Rojas](#), un profesor y científico informático mexicano alemán, demostró que el Z3 es Turing completo, esto es, cumple con las características de una Máquina Universal de Turing.

¹² N. del A. Las fechas pueden variar entre diferentes fuentes de consulta, pero en general, los períodos que presentan esas diferencias son solo de unos pocos años.

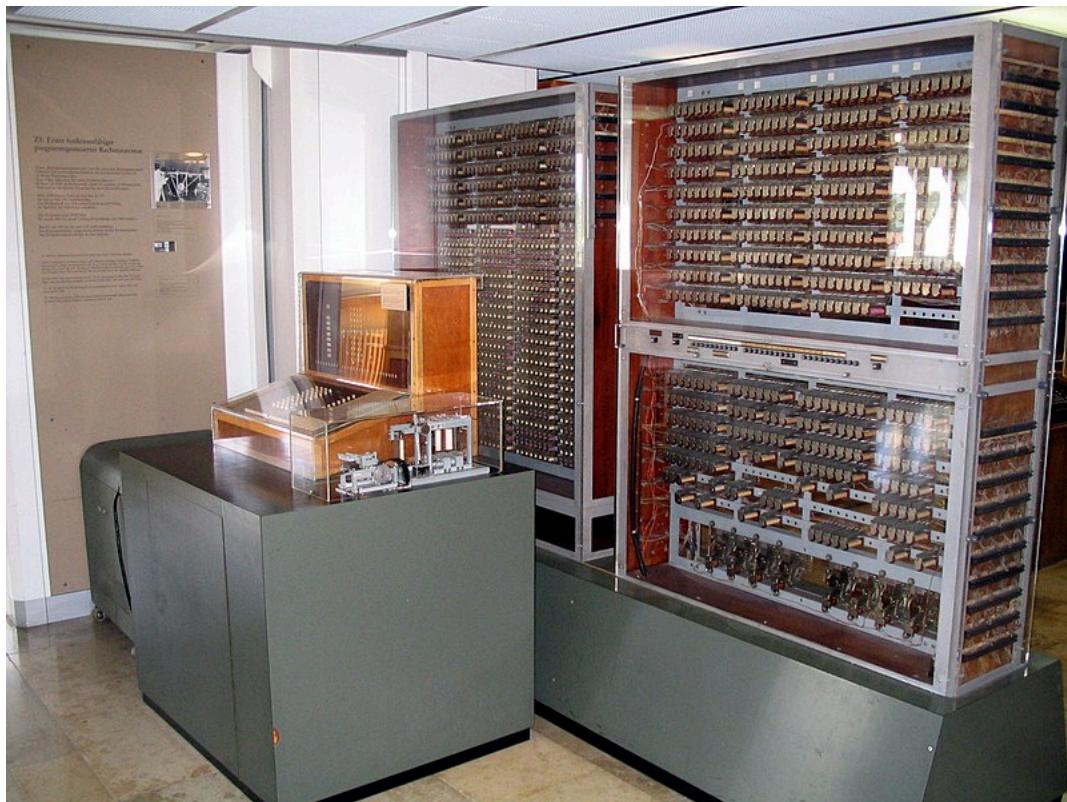


Figura 0.9. Fotografía de la réplica del Z3 (tomada de Wikipedia).

Se destaca también esta generación por la aparición de los primeros computadores digitales electrónicos, entre ellos el [EDVAC](#) y el [ENIAC](#), ambos desarrollados por [J. Presper Eckert](#) y [John William Mauchly](#), a quienes se les unió luego von Neumann en la construcción del EDVAC introduciendo la arquitectura que éste desarrolló. La programación de estas primeras máquinas era en lenguaje de máquina (1, 0), consumían grandes cantidades de calor, ocupaban grandes espacios con gran cantidad de cable y funcionaban con válvulas de vacío que se conectaban en grandes tarjetas perforadas electrónicas ancladas a la pared; las altas temperaturas hacían que estas válvulas se tuvieran que cambiar constantemente debido a que se dañaban. Su operación era compleja, por lo que requería de personal experto.

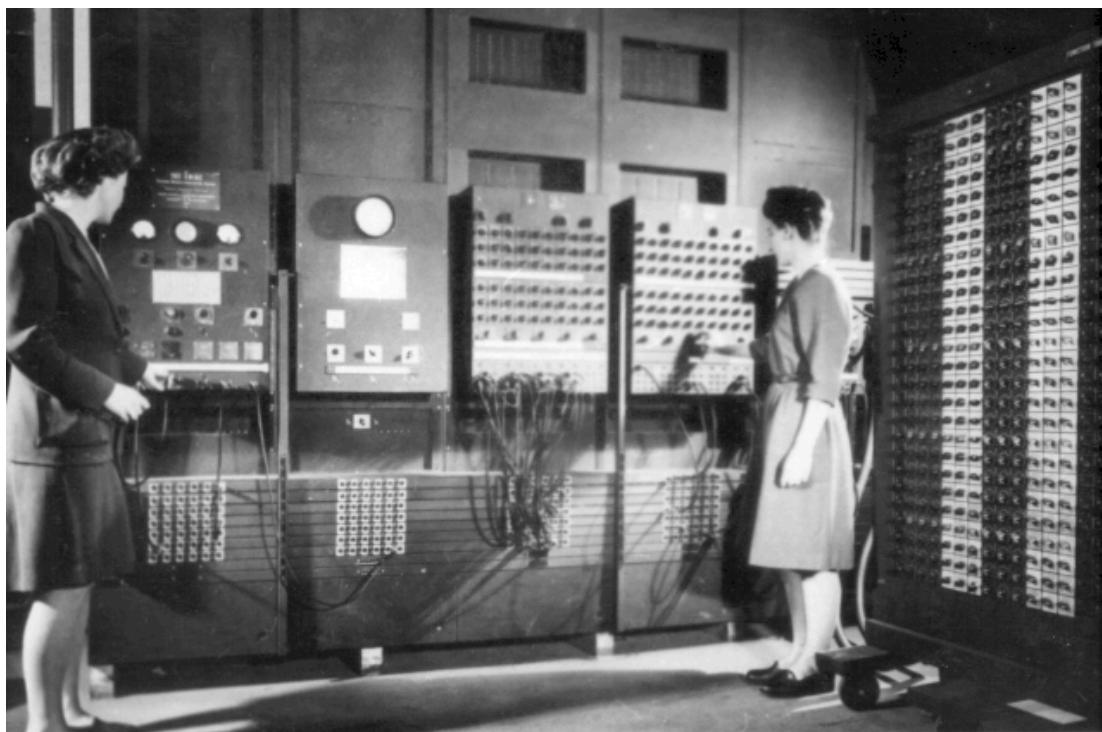


Figura 0.10. Fotografía del ENIAC (tomada de Wikipedia).

Segunda generación (1955 - 1965)

Se caracteriza por grandes avances, entre ellos, la invención del [transistor](#) que permitió la miniaturización de los componentes electrónicos. Aparecen los primeros lenguajes de programación de alto nivel COBOL (Common Business Oriented Language) implementado en empresas y Fortran (Formula Translation) para el uso académico y científico. Dichos lenguajes existen aún hoy en día, aunque su uso ya no es del alto número de programadores de otrora. Estos lenguajes reemplazaron la dura tarea de programar en lenguaje de máquina, a utilizar palabras en el idioma inglés, significando un paso fundamental en la evolución de la informática.

Tercera generación (1965 - 1975)

Continúan las revoluciones trayendo consigo el desarrollo del **circuito integrado (chip)**, el cual permite almacenar en un mismo componente, cientos de transistores, llevando la electrónica a una miniaturización sin precedentes. Surgen nuevos lenguajes de programación, entre los que se cuentan algunos como Basic, Pascal (creado por el profesor y científico informático [Niklaus Wirth](#)), C (creado por el científico de la computación [Dennis Ritchie](#)). También se realizan las primera conexiones entre computadores, estableciendo así la creación de redes de área local (LAN - Local Area Network); y para 1969 aparece el proyecto [ARPANET](#) desarrollado por el departamento de defensa de Estados Unidos (EU), que marca el inicio de la red de redes [Internet](#). Aparece el dispositivo de entrada "ratón" o "mouse", pero a falta de interfaces gráficas, no se hace popular.

Cuarta generación (1975 - 1990)

Las tecnologías existentes mejoran significativamente. Se fundan las compañías *Apple* y *Microsoft*, que años más tarde se convertirían en “imperios” de la informática. Aparece el paradigma de la [POO \(Programación Orientada a Objetos\)](#), así como nuevos lenguajes, entre ellos Ada, un lenguaje orientado a objetos desarrollado por el departamento de defensa de EU y cuyo nombre fue dado en honor a Ada Lovelace. La empresa Apple desarrolla el sistema Windows para trabajar con interfaces gráficas, permitiendo que el dispositivo *mouse* se pueda utilizar y popularizar; sin embargo, es la empresa Microsoft la que logra masificar el uso de las interfaces gráficas con entorno Windows.

Quinta generación (1990 - presente/futuro)

Continúan las mejoras sobre las tecnologías existentes. Aparecen nuevos paradigmas de programación, particularmente impulsados por la aparición de la [World Wide Web \(WWW\)](#)¹³ y los dispositivos móviles. Resurge el interés por la IA luego de muchos años, logrando grandes avances importantes en sus distintas líneas de investigación gracias a cambios de enfoque y perspectiva de los algoritmos para la IA; algunos de los usos actuales están relacionados con aplicaciones del test de Turing con máquinas que llevan conversaciones con humanos y responden preguntas, la robótica, los sistemas expertos y el reconocimiento digital de imágenes entre otras. La seguridad informática y la inteligencia de negocios son líneas de investigación permanentes, y aunque el tema de seguridad no es nuevo, si se han refinado bastante las técnicas de protección; la inteligencia de negocios es una línea de suma importancia dado los grandes volúmenes de información de las bases de datos actuales.

A nivel de hardware se superan inconvenientes en cuanto al procesamiento y almacenamiento, tanto temporal como permanente, que se creían muy difíciles de resolver, sin embargo, las máquinas actuales aún guardan muchas limitaciones que hacen que ciertos problemas sean irresolubles.

Existe un nuevo modelo computacional alternativo al clásico conocido como [computación cuántica](#) donde en vez de tener dos estados representados en un bit, se pueden tener hasta diez representados en **cúbits (qbits o bit cuántico)**. En la computación clásica un bit sólo puede tener uno de dos estados: 1 o 0, mientras que un cúbit puede tener los dos estados al mismo tiempo. Este nuevo paradigma implica la reformulación de los algoritmos utilizados y trae además otras implicaciones a nivel del diseño de máquinas debido al [principio de incertidumbre de Heisenberg](#). En la actualidad hay computadores cuánticos en centros de investigación universitarios y de grandes empresas donde ya se han logrado crear y ejecutar algoritmos cuánticos para realizar operaciones aritméticas, entre otras aplicaciones.

¹³ N. del A. La WWW es considerada uno de los tres inventos más importantes del siglo XX, el cual cambió de forma radical la forma en que los humanos interactuamos

Preguntas

1. (F o V) La arquitectura de Von Neumann de los computadores actuales fue propuesta por el científico Alan Turing

Ejercicios

Unidad 1. Conceptos básicos de programación

El computador

La historia de las matemáticas y de las máquinas de cálculo, nos lleva a conceptualizar la palabra “**computador**” como un dispositivo capaz de ejecutar operaciones a grandes velocidades agilizando el procesamiento de grandes volúmenes de información. Dicho dispositivo debe ser programable y completamente automático, donde no requiera la intervención humana para realizar sus funciones¹⁴.

Esta máquina opera a través de **instrucciones**, las cuales han sido diseñadas para que ésta las entienda. Un conjunto de instrucciones que permiten la resolución de un problema, se suele llamar **programa**. En otras palabras, **un programa es un algoritmo llevado a un computador**. Los pasos finitos del algoritmo son las instrucciones del programa, que se ejecutan según el orden indicado por aquel.

La siguiente imagen muestra dos computadores personales (PC - *Personal Computer*) en distintas épocas.



Figura 1.1. Computadores personales (PC) en distintas épocas¹⁵

Periféricos

Según la segunda definición de la RAE, un periférico es un “aparato auxiliar e independiente conectado a la unidad central de una computadora u otro dispositivo electrónico”, definición

¹⁴ Vale aclarar que la máquina requiere elementos para que esta trabaje y que debe suministrar una persona, tal como insertar un medio de almacenamiento para guardar datos o poner papel sobre la impresora, entre otros casos.

¹⁵ Imagen tomada de: [Computador antiguo y moderno - rompecabezas en línea](#)

ajustada a la jerga informática. Los periféricos no forman parte del núcleo central de un computador, pero son esenciales para la realización de las tareas, ya que son los encargados de comunicar a éste con el exterior. Los periféricos se conectan al computador a través de puertas de enlace conocidas como **puertos**. Los puertos pueden ser físicos, desde donde conectamos dispositivos, o lógicos, los cuales permiten “conectar” programas o acceder a servicios.

Puertos físicos

Las nuevas tecnologías han traído nuevos puertos que permiten mejores conexiones al computador, algunos ya son parte de la historia y solo quedan en equipos antiguos. Existen también adaptadores para lograr conexiones cuando se tienen puertos diferentes.

- PS/2 (anteriormente muy usado para ratón y teclado)
- Ethernet RJ-45
- USB
- Paralelo
- MIDI
- COM
- Serial
- VGA
- Audio Entrada/Salida Jacks de 3.5 mm

Puertos lógicos

Dependen de la instalación de ciertas aplicaciones y tienen un número asociado por defecto que puede cambiarse usando los programas de configuración de la aplicación respectiva. Algunos son:

- Servidor web: puerto 80
- SMTP: puerto 25
- FTP: puerto 21
- NameServer: puerto
- MySQL: puerto 3306
- PostgreSQL: puerto 5432
- Printer: 515

Periféricos de entrada

Utilizados para ingresar información al computador en forma de datos y órdenes. Se considera al *teclado* como el dispositivo estándar de entrada; otros son: ratón (*mouse*), lápiz óptico, pantallas táctiles, escáner, lector de códigos (de barras, QR), micrófono, etc.

Periféricos de salida

Permiten entregar información en forma de datos, los cuales pueden ser de tipo texto, audiovisual u otro. El dispositivo estándar de salida es la *pantalla*; otros son: impresora, parlantes, *joystick* (al vibrar), etc.

Memoria auxiliar o secundaria

Otros tipos de periféricos son los medios externos de almacenamiento permanente. Éstos tampoco se requieren para el funcionamiento del núcleo del computador, pero son esenciales para conservar la información en el tiempo. La memoria principal (RAM) es una memoria de trabajo, de tipo volátil y temporal, con la cual no podemos almacenar información. Algunos medios de almacenamiento externo son (algunos en desuso): discos duros, memorias USB y SD, discos flexibles, CD/DVD, cintas magnéticas, etc.

La Unidad Central de Procesamiento (CPU - *Central Processing Unit*)

La CPU (Central Processing Unit por sus siglas en inglés) es un componente del hardware de un computador que controla el funcionamiento de éste interpretando las instrucciones de un programa informático mediante operaciones básicas aritméticas, lógicas y externas procedentes de las unidades de entrada/salida. Su evolución ha sido notable desde su creación, aumentando su eficiencia, capacidad de procesamiento, disminuyendo costos en su fabricación y tecnología que permite ahorrar energía; actualmente una CPU ocupa un pequeño espacio físico en un componente conocido como **microprocesador**. Es considerada coloquialmente como el “cerebro” del computador, en el sentido que coordina las actividades y componentes de la máquina y realiza las operaciones necesarias sin intervención humana. Los componentes que conforman una CPU son:

Unidad Aritmético Lógica o Unidad de Cálculo (ALU - *Arithmetic Logic Unit*)

Encargada de realizar las operaciones aritméticas y lógicas, incluyendo los cálculos aritméticos en punto flotante (cifras decimales). El dispositivo de hardware dentro de la CPU encargado de hacer esta parte, es el *coprocesador matemático*.

Unidad de Control (CU - *Control Unit*)

Dirige el tráfico de información entre los registros de la CPU y conecta con la ALU las instrucciones extraídas de la memoria. En un procesador común que ejecuta nativamente instrucciones de una arquitectura x86, la unidad de control realiza las tareas de leer, decodificar, controlar la ejecución y almacenar los resultados.

Registros internos

No accesibles (de instrucción, de bus de datos y bus de dirección) y accesibles de uso específico (contador programa, puntero de pila, acumulador, banderas, etc.) o de uso general. Los registros internos están en una parte de la memoria con alta velocidad de recuperación que permite controlar y almacenar las instrucciones en ejecución.

Buses

El bus o canal es un sistema digital por dónde se transfieren datos entre los componentes de un computador. Se compone de cables o pistas en un circuito impreso, dispositivos como resistores y condensadores, además de circuitos integrados. Hay básicamente dos tipos de transferencia en los buses:

- Serie: el bus solamente es capaz de transferir los datos bit a bit, es decir, el bus tiene un único cable que transmite la información.
- Paralelo: el bus permite transferir varios bits simultáneamente, por ejemplo 8 bits.

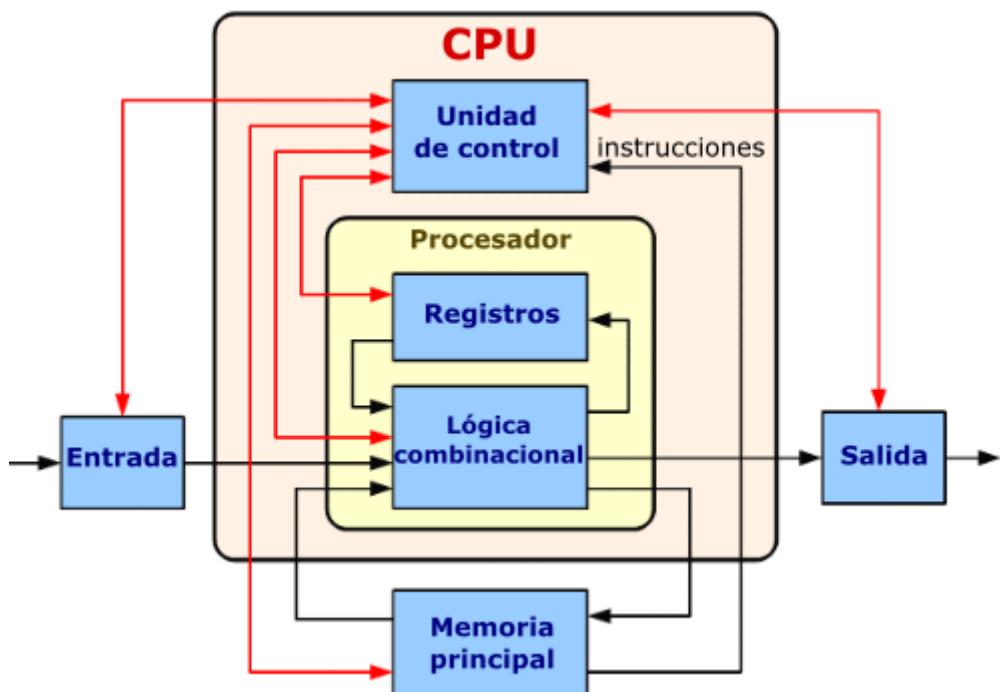


Figura 1.2. Funcionamiento de la CPU¹⁶

¹⁶ Imagen tomada de: [Diagrama del funcionamiento de la CPU](#)

La unidad de memoria

Es un dispositivo encargado de recordar órdenes que indica un agente externo por medio de un dispositivo de entrada, así como la información dividida en datos dados a la máquina y/o que ésta genera a través de algún proceso determinado; además almacena las instrucciones que contienen las operaciones que debe realizar la máquina para que pueda trabajar sin intervención humana.

Esta unidad está integrada en la actualidad por millones de circuitos electrónicos invisibles al ojo humano, donde hablamos de unidades del orden 10^{-9}m o nanómetros ($1\text{nm} = 10^{-9}\text{m}$) que requiere intervención tecnológica con láser y otras técnicas para la manipulación de los componentes. Cada uno de estos elementos tiene la propiedad física de ser **biestable**, esto es, posee uno de dos estados que puede tener en cualquier momento (pero no ambos a la vez). Las convenciones para llamar estos estados binarios puede variar de acuerdo al autor; algunos de los que han sido usados son:

- Encendido - apagado
- On - off
- Si - no
- Yes - no
- Verdadero - falso
- True - false
- 1 - 0

Los valores 1 y 0 al ser numéricos, permiten construir toda una aritmética binaria, base matemática de los sistemas digitales y particularmente del computador, y que a su vez abarca cualquier nombre que quiera darse a cada estado.

Manejo de la memoria

La memoria está conformada por una gran cantidad de elementos que pueden manipularse uno a uno; sin embargo, esto no es algo práctico y resulta mejor agrupar varios de estos elementos para manipularlos, sin importar cuantos conformen dicha agrupación. Dichos grupos se conocen como **campos** y se distinguen por un *nombre único* que los identifica. El *tamaño* de cada campo está dado por el número de elementos biestables que lo conforman. Un campo por tanto almacena información de una forma más organizada.

La siguiente figura muestra una representación de la memoria y de grupos de elementos biestables (campos) marcados con colores.

1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0	1	1
0	0	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	0	0	0
1	0	1	1	1	0	0	0	1	1	0	1	0	1	1	0	0	1	0	0

1	1	0	0	1	0	1	1	0	0	0	1	1	1					

Figura 1.3. Representación de la memoria y de grupos de elementos biestables o campos

Tipos de campos

Un campo puede ser **constante** o **variable**. Un campo variable puede cambiar su contenido durante el tiempo de ejecución del programa; por otro lado, un campo constante siempre mantendrá su valor de forma rígida mientras el programa se esté ejecutando; si se trata de cambiar el valor de un campo constante en tiempo de ejecución, se generará un error. Una constante también puede estar representada por un valor fijo y no necesariamente con un campo, como por ejemplo el literal de cadena “lógica de programación” (encerrado entre comillas), o el número 215, o el valor booleano Falso, entre otros.

Recordemos que un campo se identifica de manera única en la memoria utilizada por un programa con un nombre. Al hacer referencia a este nombre, estamos en realidad accediendo a su información o contenido, ya sea para leerla o modificarla.

Al nombrar un campo, se deben seguir unas cuantas reglas que se establecen en la lógica de programación, así como en la mayoría de lenguajes de programación:

- Puede contener caracteres alfanuméricos, pero debe iniciar con una letra
- No puede contener caracteres especiales, a excepción del guión bajo (_ underline), con el cual también puede iniciar el nombre del campo (esto evita confusiones con las operaciones que realiza el computador)
- Muchos lenguajes de programación son sensibles a los caracteres, lo que significa que distinguen entre mayúsculas y minúsculas, por lo que un campo con el nombre A es diferente de otro con el nombre a.¹⁷

Ejemplos

Los siguientes son nombres válidos para campos:

a, x, z2, Placa, EDAD, nom, nombre_apellido, salarioEmpleado, _num3, NotaAsignatura

Los siguientes son nombres para campos no válidos:

5B, nombre-persona, num 1, z*, miedad}, mamá

¹⁷ En este curso asumiremos los nombres de campos sensibles a los caracteres

Memoria RAM y ROM

La **RAM (Random Access Memory)** o Memoria de Acceso Aleatorio es la parte de la memoria principal del computador necesaria para que los distintos programas trabajen (corran) allí, por lo que también es llamada memoria de trabajo; es de acceso aleatorio y de tipo volátil, lo que quiere decir que la unidad de control (CU) accede a cualquier lugar libre de ella cuando requiere memoria y que cuando el programa deja de ejecutarse o se acaba el suministro de energía, lo almacenado en ella se pierde.

Por otro lado, la **ROM (Read Only Memory)** o Memoria de Solo Lectura es la parte de la memoria principal donde se guardan las instrucciones que debe ejecutar la CPU cuando ésta recibe órdenes provenientes de periféricos u otros programas. En ella se encuentra el programa que contiene las instrucciones de cómo debe operar el computador. Viene configurada de fábrica y solo deja la modificación de algunos parámetros mediante el programa de *sistema de arranque (boot system)* de la máquina, de ahí que se denomine de “solo lectura”.

Algoritmo

Aunque es un término bastante empleado tanto en matemáticas como en las ciencias computacionales, es común encontrar variantes en la definición. Sin embargo, el consenso general en ciencias, permite definir un **algoritmo** como *un conjunto de pasos finitos y ordenados que buscan la solución de un problema*. Este nombre al parecer tuvo influencia en el matemático persa Al-Juarismi, que en latín antiguo se conocía como *Algorithmi*.

En la antigüedad hubo desarrollos de procesos algorítmicos para resolver problemas, entre ellos se encuentra uno de los más famosos conocido como *Algoritmo de Euclides* para hallar el *Máximo Común Divisor (MCD)* de dos enteros.

Un algoritmo se puede escribir siguiendo una serie de reglas sintácticas que permiten crear un **pseudocódigo** basado en él y que puede llevarse luego a un computador, en otras palabras, se puede escribir de una forma muy parecida a como un computador entendería cada paso del algoritmo.

Un algoritmo puede describirse gráficamente así:



Figura 1.4. Representación gráfica del proceso algorítmico

Algoritmos “cualitativos”

Son muchos los procesos/problemas que pueden ser solucionados por algoritmos, si éstos pueden ser descritos según la definición que aplica para éstos. Un algoritmo “cualitativo” es un tipo de solución informal y busca describir una solución al problema, sin un acercamiento a una solución por computador. Un algoritmo cualitativo dice en general **qué** hacer, pero no cómo implementarlo en una máquina. Veamos algunos ejemplos de la vida cotidiana.

Ejemplo 1.1

Una persona pide un producto en una tienda. Si se encuentra lo paga y espera la devuelta. En caso contrario, se retira del local.

Solución

Algoritmo:

1. Inicio
2. Hacer el pedido del producto al encargado de la tienda
3. Si el producto está disponible, entonces pagarla y esperar la devuelta; en caso contrario, retirarse de la tienda
4. Fin

Ejemplo 1.2

Escriba un algoritmo que describa cómo ponerse los zapatos luego de bañarse.

Solución

Algoritmo:

1. Inicio
2. Elegir calzado a usar
3. Secar los pies y usar un talco
4. Ponerse las medias
5. Ponerse los zapatos
6. Si los zapatos tienen cordones, entonces sujetarlos
7. Fin

Ejemplo 1.3

Escriba un algoritmo para determinar si un número es primo.

Solución

Un **número primo** es un número entero positivo que es divisible por sí mismo y por la unidad. Si al dividir el número sucesivamente desde 2 hasta la mitad de éste no encontramos ningún divisor exacto, podemos concluir que el número es primo.

Algoritmo:

```
1. Inicio
2. Escoger el número a determinar si es primo
3. Si el número es un entero positivo (número >= 0) entonces
4. Asigne 2 a divisor
5. Si número / divisor es división exacta entonces
6. El número no es primo y voy al paso 14
7. Si no cumple 5., entonces
8. Incremento en 1 a divisor
9. Si divisor <= número / 2 entonces
10. Vuelvo al paso 5.
11. En caso contrario (divisor > número / 2)
12. El número es primo y voy al paso 14
13. Si el paso 3 no se cumple, entonces la entrada no es válida y voy
    al paso 14
14. Fin
```

Algoritmos “cuantitativos”: diagramas de flujo y pseudocódigo

Son soluciones algorítmicas presentadas en **diagramas de flujo**, **diagramas rectangulares** o **pseudocódigo**, una forma *estructurada del algoritmo* donde ya hay un acercamiento para llevar la solución a una forma que comprenda una máquina (computador). En este tipo de algoritmos vamos a centrar los esfuerzos, ya que siguen unas reglas sintácticas con cierta flexibilidad que permiten una fácil traducción posterior a la creación de programas en los lenguajes. Un “algoritmo cuantitativo” nos dice **cómo** podemos codificarlo (o seudo codificarlo) convirtiendo los pasos en instrucciones que comprende el procesador. En la sección de “Problemas resueltos” se presentan diferentes situaciones solucionadas aplicando distintas técnicas de diagramación de algoritmos.

Programa

A partir del pseudocódigo de un algoritmo, podemos construir un *programa* para llevarlo a un computador. Un **programa** es por tanto un conjunto de **instrucciones** finitas dispuestas en orden para solucionar un problema y que las entiende un computador para ser ejecutadas. Esta definición coincide con la de algoritmo, donde los pasos de éste equivalen a las instrucciones del programa. En otras palabras, un programa es un *formato especial* que representa la solución de un algoritmo y que comprende un procesador.

Las operaciones que conducen a expresar un algoritmo en forma de programa, se conoce como **programación** y los que escriben dichos programas se conocen como **programadores**.¹⁸

¹⁸ N. del A. En la actualidad a dicha actividad se le suele llamar “desarrollo de software” y a los programadores se les conoce comúnmente como “desarrolladores de software”

El proceso de la programación

Consiste en proponer una solución para computador (programa) luego de tener un problema y haber llegado a una solución algorítmica. Una vez creado el programa, éste es **ejecutado** para dar solución al problema propuesto.

Gráficamente, podemos verlo así:

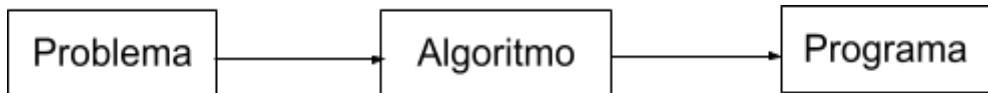


Figura 1.5. Representación gráfica del proceso de la programación

Los lenguajes de programación

Un **lenguaje de programación** es un programa para crear programas. Como tal, debe ser capaz de *interpretar* el algoritmo, lo que significa que comprenda cada paso de éste en forma de instrucciones y realice las operaciones correspondientes sin intervención humana. Un programa es en últimas un formato del algoritmo escrito de tal forma que sea entendido por una máquina (procesador) y que debe ser escrito en un lenguaje de programación en particular, proceso que también se conoce como **codificación**, y al texto generado, **código de programación**.

Tipos de lenguajes

Cada lenguaje de programación tiene un **propósito**, esto es, tiene un fin para el que fue creado, lo cual significa que no todo problema algorítmico puede tratar de resolverse en cualquier lenguaje. Algunos lenguajes son considerados de propósito general, lo que en teoría significa que puede resolverse cualquier algoritmo en estos lenguajes, otros son para el desarrollo Web, para el desarrollo móvil, para el “backend” o el “frontend”, para bases de datos, para cálculos matemáticos, para el diseño gráfico, para los componentes físicos, etc. Podemos clasificar los lenguajes en tres grandes grupos:

Lenguajes de máquina

Utilizan el lenguaje binario (1s y 0s) para cada instrucción y operación, por lo que suele conocerse también como **código** o **lenguaje máquina**. Este código depende de la máquina (fabricante del hardware). Estos lenguajes cargan los programas directamente sin requerir “traductores”, y al ser operaciones en código nativo, la velocidad y rendimiento es mayor, pero tienen las desventajas de la complejidad, coste en tiempo, dificultad para detectar errores. Una mala programación en lenguaje máquina de un componente puede generar daños costosos o incluso dejarlo inutilizable.

Lenguajes de bajo nivel

El Lenguaje Ensamblador (*Assembly Language*), así como los lenguajes PLC (*Programmable Logic Controller*) hacen parte de los lenguajes que componen este grupo de lenguajes de bajo nivel. Son un poco más fáciles de usar que los lenguajes de máquina, pero son dependientes de la máquina. El lenguaje Ensamblador tiene múltiples aplicaciones para la programación de dispositivos físicos, pero depende de la fabricación de estos, ya que las especificaciones de cada fabricante varía entre componentes del mismo tipo. PLC es muy utilizado en entornos industriales para desarrollos mecánicos, mecatrónicos y de otras aplicaciones donde es necesario utilizar un lenguaje binario para programar algunos tipos de máquinas. Estos lenguajes no se ejecutan directamente, requieren ser **traducidos** al lenguaje de máquina (o simplemente, lenguaje máquina). El programa escrito en lenguaje ensamblador se conoce como **código (programa) fuente** y el traducido como **código (programa) objeto**. El traductor de código está presente en la mayoría de computadores y es un programa que se conoce como *Ensamblador (Assembler)*.

Lenguajes de alto nivel

Son los más utilizados por los programadores en general y están diseñados para que el programador se pueda “entender” más fácilmente con la máquina, ya que permiten el ingreso de las instrucciones usando palabras con caracteres del idioma inglés¹⁹, incluyendo los caracteres especiales de dicha lengua y del alfabeto latino en general. Esto los hace más atractivos y su campo de aplicación abarca prácticamente todas las áreas, además que en general, son independientes de la máquina (no dependen de las características del hardware), lo cual hace a los programas escritos *portables* a otras plataformas. Algunos permiten interactuar con lenguajes de bajo nivel y de máquina, lo cual facilita el trabajo en dichos entornos de trabajo.

Con respecto a los otros tipos de lenguajes, los de alto nivel tienen algunas desventajas, como por ejemplo un mayor consumo de memoria, mayor tiempo de ejecución y subutilización de los recursos del hardware, entre otras.

Desde la invención de los primeros computadores electrónicos, han sido muchos los lenguajes que se han creado, muchos han desaparecido, surgiendo nuevas de estas herramientas según las necesidades creadas con los desarrollos actuales tecnológicos. Veamos algunos de los más destacados a nivel histórico y relevantes en el nuevo milenio.

1. COBOL (Common Business Oriented Language)
2. Fortran (Formula Translation)
3. Pascal
4. Delphi
5. Ada

¹⁹ Esto se debe a que la mayoría de lenguajes de programación han sido creados en Estados Unidos, país de habla inglesa.

6. SmallTalk
7. DBase
8. Foxpro
9. Visual Foxpro
10. Basic
11. Visual Basic
12. Visual Basic for Applications (VBA)
13. Visual Basic Script (VBS)
14. Visual Basic .NET (VB .NET)
15. Matlab
16. Mathematica
17. Java
18. Javascript
19. Kotlin
20. Go
21. PHP
22. Perl
23. Python
24. Ruby
25. C#
26. C/C++
27. R
28. Prolog
29. LabVIEW
30. PLC

Editores de textos

El código de programación, o simplemente código, debe ser escrito utilizando programas del tipo **editores de texto**, los cuales producen texto plano, esto es, sin formato, y no **procesadores de texto** que generan formatos que no son leídos en los lenguajes de programación.

Algunos editores son productos de software libre y otros no, y no todos son gratuitos, por lo que tendrá que pagar por el uso de algunos de ellos. Decidir con cual editor trabajar es una cuestión de necesidades y gusto que solo se define al experimentar con varios de éstos. Existen muchos editores para los distintos lenguajes de programación, algunos creados por las mismas empresas que desarrollan los lenguajes de programación. Veamos algunos clasificados de acuerdo a sus capacidades:

- IDE's (*Integrated Development Environment* - Entorno de Desarrollo Integrado): son entornos avanzados que ofrecen grandes características en el desarrollo de programas, permitiendo el uso de interfaces gráficas para construir aplicaciones y hacer uso de las funcionalidades de *frameworks* (marcos de trabajo) generalmente incluidos para el desarrollo. Por sus grandes capacidades, tienen la desventaja de ocupar gran volumen en disco, así como del consumo de recursos de máquina.

Algunos IDE's reconocidos en el mundo del desarrollo de software se indican a continuación, algunos de ellos con la capacidad de soportar múltiples lenguajes de programación: NetBeans, Visual Studio (para .NET Framework), IntelliJ Idea, Eclipse

- Editores avanzados: No tienen las capacidades de los IDE's, pero tienen la ventaja de ser más ligeros, con una buena cantidad de funcionalidades nativas y permiten la personalización a través de extensiones (*plugins*) de terceros que los hacen tener muchas de las funcionalidades de los IDE's, esto claro está, a costa de algo de un poco del rendimiento del programa. Algunos de estos son: Visual Studio Code, Sublime Text, Atom (descontinuado desde el 2022), PHP Designer.
- Editores semi avanzados: Ofrecen algunas funcionalidades que ayudan en la edición del código, como tabulación, marcado de sintaxis, entre otras. Entre ellos tenemos Programmer's Notepad, Notepad++, Textpad, Brackets, Crimson, Geany.
- Editores tipo WYSIWYG (*What You See Is What You Get* - Lo Que Ves Es Lo Que Obtienes): Ofrecen características avanzadas particularmente para el desarrollo de páginas y aplicativos web, los cuales permiten crear desarrollo gráficos que generan el código automáticamente y el área de trabajo muestra cómo serán los resultados finales. Son particularmente usados para la creación de páginas Web HTML por personas familiarizadas con el diseño gráfico, más no con la programación; algunos son: Dreamweaver, Expression Web (descontinuado), FrontPage (descontinuado), OpenOffice.org.
- Editores sencillos: Ofrecen muy pocas opciones de edición de código o ninguna, algunos de éstos son: gedit de Linux, Bloc de notas de Windows, Edit del D.O.S.

Traductores de lenguaje

Son programas incorporados en los lenguajes de programación y que se encargan de traducir el código fuente a código máquina. Hay dos tipos de traductores:

- Intérpretes
- Compiladores

Intérpretes

Leen línea a línea del programa y lo ejecutan directamente si no detectan errores. Algunos lenguajes interpretados, son:

- PHP
- Python
- Javascript
- Perl
- Ruby

Compiladores

Son programas que verifican los errores sintácticos y luego transforman el *código fuente* en *código objeto* que es ejecutado posteriormente por la máquina. Algunos lenguajes compilados, son:

- C/C++
- Java
- COBOL
- Ada
- Go

Fases de la compilación

- Problema (enunciado, planteamiento)
- Algoritmo (pseudocódigo, diagramas)
- Codificación (programa fuente)
- Compilación (verificación de errores, traducción a código máquina, programa objeto)
- Ejecución del programa objeto (ejecutable)

Datos y tipos de datos

Ya vimos como en la memoria se pueden agrupar varios elementos biestables para formar campos para hacer más fácil y manejable el tratamiento de la información. También anotamos que al hacer referencia al nombre de un campo, hacemos mención a su contenido. El contenido (información) de un campo es conocido como **dato**.

Los lenguajes de programación permiten realizar abstracciones para no preocuparnos por los estados de cada elemento biestable o por las cadenas binarias que forman el dato del campo, pudiendo ignorar los detalles de implementación interna. Así, podemos referirnos al campo **nombre** cuyo contenido es “**Ana Gil**” o al campo **número** cuyo contenido es **10**.

Cada dato contiene un *tipo* de información específica, que puede ser numérica, alfabética u otra y que obedece a la forma como haya sido definido el campo en el programa. A un campo por tanto se le asocia un **tipo de dato**, el cual le indica al programa qué tipo de información puede aceptar éste. Un tipo de dato puede ser **simple (primitivo)** o **compuesto (estructurado)**. Un campo definido como tipo de dato simple solo permite almacenar un valor, mientras que en un campo compuesto se pueden almacenar varios valores y su definición es en función de los tipos de datos simples; éstos se tratarán más adelante.

Los **tipos de datos primitivos** en los lenguajes de programación, son:

1. Numéricos

- a. Enteros (*integer*)
- b. Reales (*real*)
- 2. Carácter, Cadena (*char, string*)
- 3. Lógicos (*boolean*)

Los lenguajes de programación modernos admiten una gran cantidad de tipos de datos primitivos para almacenar datos, entre ellos se destacan los tipos:

- Fecha (*date*)
- Hora (*time*)
- FechaHora (*datetime*)
- Entero largo (*long int*)
- Entero corto (*smallint*)
- Entero muy corto (*tinyint*)
- Real en punto flotante precisión simple (*float*)
- Real en punto flotante precisión doble (*double*)

Las fechas se suelen tratar como cadenas de caracteres cuando el tipo de dato no está disponible en el lenguaje y las horas como números enteros, aunque también pueden ser tratadas como un dato compuesto por varios números/cadenas.

Tipos de datos numéricos

Representa el conjunto de valores numéricos, que pueden ser enteros o reales (en punto flotante). Los números enteros son un subconjunto de los reales, son números que no tienen parte decimal y pueden ser negativos, positivos o cero: -9, 5, 4, 0, 500, -1005.

Los números reales a nivel computacional son en realidad aproximaciones, ya que no podemos escribir infinitas cifras decimales por las limitaciones tecnológicas. Un número real o en punto flotante está compuesto de una parte entera y posiblemente una parte decimal (mantisa): 1, -5.6, 9.66666, -0.0001, 350, 41.2. Para acortar la escritura de ciertos números muy grandes o muy pequeños, se puede utilizar la **notación científica**: $3E10 = 3 \times 10^{10}$, $5.4E-5 = 5.4 \times 10^{-5}$. Vale aclarar que el carácter utilizado en esta notación para separar las cifras decimales es el **punto** (.) y no la **coma** (,).

Tipo de dato carácter/cadena

Es el conjunto finito de todos los caracteres disponibles en el computador:

- Caracteres alfabéticos = {a, b, c, ..., z, A, B, C, ..., Z }
- Caracteres numéricos = {0, 1, ..., 9}
- Caracteres especiales = {+, -, *, /, (,), @, ^, \$, #,...}

Los datos tipo carácter están delimitados entre *comillas dobles* o *simples* (apóstrofos) y el número de caracteres que contenga determina su **longitud**: “Hola”, ‘Pedro Zapata’, “Hoy es martes”. Estas expresiones encerradas entre comillas se conocen como **literales de cadena**.

Una **cadena de caracteres** es una secuencia compuesta por caracteres del código **ASCII** (*American Standard Code for Information Interchange*) y que están disponibles en todas las distribuciones comerciales en los distintos idiomas en que están los teclados.

Tipo de dato lógico (booleano)

Permite almacenar un valor binario. Algunos lenguajes antiguos no incluían este tipo de datos y hacían el tratamiento con campos de tipo entero, con los valores 1 y 0. Las constantes lógicas que usaremos en lógica de programación, son:

- **Verdadero (true)**
- **Falso (false)**

Constantes y Variables

Como vimos anteriormente, un campo puede ser variable o constante. De ahora en adelante, al referirnos a un campo, asumimos de acuerdo al contexto, que es sinónimo de constante o variable para referirnos a esos espacios de la memoria donde podemos guardar un dato; así, podemos definir alternativamente como una **constante o variable** como *un espacio de la memoria asociado a un tipo de dato que le asignamos un nombre único y donde almacenamos un dato*.

Expresiones

Son combinaciones de constantes, variables, operadores, paréntesis y funciones especiales formadas con el objetivo de solucionar algo:

- $ab^2 - 4 / 3 + c$
- $5 ^ 3 / 10 + 2 * 6 - 4$

Una expresión no necesariamente tiene que ser aritmética como la anterior, también hay expresiones lógicas y de cadenas:

- $4 > (a + 5b)$

- “Luis” + “Arango” (una *suma* de cadenas se conoce como **concatenación**, una operación que permite unirlas; para el ejemplo, dicha concatenación forma la cadena “LuisArango”)

Operadores básicos en matemáticas y computación

Un *operador* es un símbolo usado en matemáticas para representar una operación a realizar, la cual puede ser unaria (con un *operando*) o binaria (con dos *operandos*). En la aritmética y en el álgebra se cuenta, entre otros, con varios operadores elementales; cada operador tiene una *prioridad* asignada, lo cual significa que los de mayor prioridad, se ejecutarán primero. Se dividen en tres grupos, de los cuales se muestra su representación matemática, así como algorítmica.

Operadores aritméticos

Utilizados para realizar las operaciones aritméticas básicas y otros cálculos (operaciones) matemáticos; tenemos los siguientes:

Nombre Operador	Símbolo matemático y algorítmico (computacional)	Prioridad
Negación aritmética unaria	–	Alta
Potencia	x^y \wedge $**$	Alta - media
Raíz cuadrada	\sqrt{x} $ /$ $raiz2(x)$ $sqrt(x)$	
Multiplicación	\times $*$ $.$ OO	Media
División	\div $/$ $\frac{a}{b}$ $a div$	
Módulo	$\%$ mod	
División entera	\backslash div $//$	
Suma	+	Baja
Resta	–	

Notas

- La negación aritmética es una operación *unaria* que consiste en negar el símbolo del número (operando). Ejemplo: – (+ 2), – (8), – (– 5), – 94
- La prioridad se refiere al orden en que los operadores se efectúan en una expresión aritmética: los de mayor prioridad se efectúan primero.
- Las operaciones encerradas entre paréntesis se efectúan primero, por lo que tienen mayor prioridad. Los paréntesis modifican la prioridad de los operadores en una expresión.
- Si hay dos operadores de igual prioridad, se ejecuta primero el que se encuentre más a la izquierda, esto es, se sigue el orden de izquierda a derecha.
- $raiz2(x)$ $sqrt(x)$, son en realidad funciones, más no operadores. Sin embargo, cualquier raíz puede ser calculada usando el operador de potencia aprovechando las propiedades del álgebra para los exponentes fraccionarios: $\sqrt[n]{a^m} = a^{\frac{m}{n}}$

Operación de módulo y división entera

Es una división entera que devuelve el residuo de ésta. Se representa con el símbolo % o la palabra **mod**, entre otros usados. La división entera se encarga de devolver solo la parte entera de dividir dos números enteros.

Ejemplo 1.4

- a. $7 \% 5 = 2; 7 // 5 = 1$
- b. $17 \% 2 = 1; 17 \text{ div } 2 = 8$
- c. $48 \text{ mod } 4 = 0; 48 \setminus 4 = 12$
- d. $57 \% 6 = 3; 57 // 6 = 9$
- e. $49 \text{ mod } 5 = 4; 49 // 5 = 9$
- f. $9 \text{ mod } 20 = 9; 9 \setminus 20 = 0$
- g. $55 \% 11 = 0; 55 // 11 = 5$

Operadores relacionales o de comparación

Son operadores binarios utilizados para comparar expresiones. El resultado de una comparación entre dos expresiones es un valor lógico (booleano), devolviendo o un verdadero (V, 1) o un falso (F, 0); estos son:

Nombre Operador	Símbolo	Prioridad
Igual	= ==	Alta
Diferente	≠ <> !=	
Mayor que	>	Media
Menor que	<	
Mayor o igual que	≥ ≥=	Baja
Menor o igual que	≤ ≤=	

Ejemplo 1.5

Resultados devueltos al realizar operaciones con los operadores relacionales.

- a. $8 ≠ 9 \rightarrow (V)$
- b. $9 ≥ 9 \rightarrow (V)$
- c. $7 ≠ 14 ÷ 2 \rightarrow (F)$
- d. $9 × 2 ≤ 50 ÷ 10 \rightarrow (F)$
- e. $-8 = 8 \rightarrow (F)$

Operadores lógicos (booleanos)

Permiten conectar (unir) expresiones de comparación y realizar operaciones lógicas. El valor devuelto (verdadero o falso) depende del conectivo lógico utilizado, según las leyes del álgebra proposicional y booleana; estos son los más utilizados en algoritmia y en los lenguajes de programación:

Nombre Operador	Símbolo						Prioridad
Negación lógica unaria	\neg	\sim	$\bar{}$!	<i>no</i>	<i>not</i>	Alta
Conjunción	\wedge	$\&\&$	\bullet	<i>y</i>	<i>and</i>		\downarrow
Disyunción	\vee	\parallel	$+$	<i>o</i>	<i>or</i>		(mayor a
Disyunción exclusiva	$\underline{\vee}$	\oplus	W	<i>'o bien'</i>	<i>xor</i>	<i>eor</i>	menor)

Ejemplo 1.6

Resultados devueltos al realizar operaciones con los operadores booleanos.

- a. Verdadero Y Verdadero \rightarrow (Verdadero)
- b. Falso Y Verdadero \rightarrow (Falso)
- c. No Verdadero \rightarrow (Falso)
- d. Verdadero O Falso \rightarrow (Verdadero)
- e. Falso O Falso \rightarrow (Falso)

Tabla de verdad de los operadores lógicos²⁰

A continuación se presenta una tabla de verdad resumida con los operadores lógicos usados en la mayoría de lenguajes de programación para dos expresiones de comparación E_1 y E_2 .

E_1	E_2	$No E_1$	$E_1 Y E_2$	$E_1 O E_2$	$E_1 O Bien E_2$
v	v	f	v	v	f
v	f	f	f	v	v
f	v	v	f	v	v
f	f	v	f	f	f

Operación de asignación

Esta operación consiste en darle un valor a un campo. Si el campo es constante, dicha asignación se realiza antes de poner en marcha el programa, ya que el valor de éste no

²⁰ En los cursos de *Lógica Matemática* o *Matemáticas Discretas* se estudian estos y otros operadores donde se amplía más el tema.

podrá cambiarse en tiempo de ejecución. También se conoce como *sentencia* o *instrucción de asignación*.

El operador utilizado en lógica de programación es una *flecha* apuntando hacia la izquierda: (\leftarrow) También se suele utilizar el símbolo *igual* ($=$). Los lenguajes de programación utilizan en general el operador igual para la asignación. La forma de su uso se indica a continuación:

Sintaxis

En lógica podemos usar estas formas:

```
constante ← valofijo  
constante = valofijo  
variable ← expresión  
variable = expresión
```

Ejemplo 1.7

1. $x \leftarrow 3$
2. $nombre \leftarrow "Diana María"$
3. $b = Verdadero$
4. $z \leftarrow 5 * x / 4$
5. $mensaje = 'Bienvenido a la programación'$

Nota

El operador de asignación es un operador **asimétrico** (a diferencia de los operadores binarios aritméticos, lógicos o relacionales que son *simétricos*), ya que la máquina primero debe evaluar la expresión a la derecha del operador de asignación y luego tomar el resultado para asignarlo a la variable que se encuentra a la izquierda de éste.

Salida de información

La salida estándar utiliza dos formas claves: las instrucciones **Imprimir (print)** o **Escribir (write)**, las cuales permiten mostrar información por pantalla. Estas sentencias especifican la salida estándar.

Sintaxis

```
Imprimir expresión  
Escribir expresión
```

Ejemplo 1.8

Mostrar dos mensajes por pantalla. Los literales de cadenas se encierran entre comillas dobles o apóstrofos.

Pseudocódigo:

```
Inicio  
Imprimir "Hoy es martes"  
Escribir "Hola mundo"  
Fin
```

Entrada de información

La entrada estándar permite el ingreso de datos usando el teclado: la instrucción **Leer** (**read**) permite la captura de datos por parte del usuario.

Sintaxis

```
Leer lista_de_variables
```

Si se especifican varias variables en la lectura (lista de variables), se deben separar por comas.

Ejemplo 1.9

Ingresar el nombre y tres números por teclado y mostrarlas por pantalla.

Pseudocódigo:

```
Inicio  
Leer nombre  
Leer a, b, c  
Imprimir "Nombre ingresado: ", nombre  
Imprimir a, b, c  
Fin
```

Notación algorítmica

Al trabajar con computadores y lenguajes de programación, algunos símbolos matemáticos son difíciles de obtener desde los caracteres estándar del teclado. Es por ello que las expresiones matemáticas deben ser reescritas cuando las llevamos a un lenguaje de programación utilizando para ello la notación algorítmica típica de la informática.

Para ello, nos basaremos en los operadores vistos anteriormente y su prioridad, así como en las propiedades del álgebra para los números reales y observando cuáles de estos operadores pueden ser usados en un lenguaje determinado. En lógica de programación, el

tema de los operadores puede flexibilizarse, pero siempre manteniendo la notación algorítmica. Veamos algunos ejemplos.

Ejemplo 1.10

Escribir en notación algorítmica las siguientes expresiones matemáticas

1. $ab + 3ac^3$
2. $\sqrt[3]{b^2} + \frac{a}{3}$
3. $\frac{a-2b+3c}{\sqrt{2}}$
4. $a \geq 0 \wedge b \neq (4 + 2ab^3) \vee [\neg(a + 2 < b) \wedge (-9 = c)]$

Solución

1. $a * b + 3 * a * c * 3$
2. $b \wedge (2/3) + a / 3$
3. Veamos varias formas de escribir esta expresión
 - a. $(a - 2 * b + 3 * c) / 2 \wedge (1/2)$
 - b. $(a - 2 * b + 3 * c) / 2 \wedge 0.5$
 - c. $(a - 2 * b + 3 * c) / \text{raizc}(2); \text{ donde raizc}() \text{ es una función}$
4. Veamos cómo escribir esta expresión que incluye todos los operadores. Para la conjunción podemos usar: **y**, **and**, ó **&&**, que son admitidos en lógica de programación y algunos lenguajes; análogamente para la disyunción podemos usar: **o**, **or** ó **||**. Por último, podemos usar para la negación: **no**, **not** ó **!**. Recordemos que los operadores lógicos trabajan como conectivos.

$$a \geq 0 \&\& b \neq (4 + 2 * a * (b \wedge 3)) \mid\mid (! (a + 2 < b) \&\& (-9 = c))$$

Nota

Observe que por la prioridad de los operadores, no es necesario usar paréntesis en algunas expresiones, a no ser que se quiera modificar ésta.

Declaración de variables y definición de constantes

En un algoritmo (y programa) es común (y muchas veces obligatorio) indicar el tipo de dato de cada variable que se va a utilizar, con lo cual el lenguaje de programación sabe con exactitud qué tipo de información puede aceptar. Generalmente, esto se hace en las primeras líneas del programa y se conoce como **declaración de variables y definición de constantes**.

Ya vimos que los lenguajes de programación se clasifican en *interpretados* y *compilados*; además también pueden ser **fuertemente tipados** o **débilmente tipados**.

Lenguajes de programación fuertemente tipados

Exigen estrictamente declarar todas las variables especificando el tipo de dato de cada una de éstas; de no seguir esta regla, generan errores de ejecución. Algunos de estos lenguajes exigen incluso indicar el tipo de dato de las constantes, aunque la mayoría opta por asumir de forma implícita el tipo de dato de acuerdo al tipo de dato del valor asignado, con lo cual que definida la constante. Algunos lenguajes de este tipo son C/C++, Java, C#, Python y Ada.

Lenguajes de programación débilmente tipados

Son flexibles en cuanto a la declaración de variables y la especificación de sus correspondientes tipos de datos. En algunos lenguajes se tiene la posibilidad de declarar las variables de forma opcional sin especificar el tipo de dato, el cual se asigna a la variable de manera implícita con el primer valor que se le asigne a éstas. Esto permite que las variables luego puedan tomar valores de otros tipos de datos sin generar errores de ejecución; sin embargo, esto puede llevar a confusiones y malas interpretaciones si no se tiene el cuidado pertinente. Algunos lenguajes de este tipo son PHP, Javascript y Visual Basic.

Si el lenguaje permite la declaración de variables, así sea débilmente tipado, es una buena técnica hacerlo a la hora de escribir programas; esto ayudará a mantener un mayor orden, una mejor estructura y facilitará futuros mantenimientos y migraciones.

Sintaxis

Declaración de variables

Tipo_de_Dato: *lista_de_variables*

Donde:

Tipo_de_Dato: cualquiera de los tipos de datos primitivos.

lista_de_variables: representa una o varias variables separadas por comas.

Sintaxis

Definición de constantes

Constante *dato_constante* <- *valor*
Constante *dato_constante* = *valor*

Ejemplo 1.11

Ilustración del uso de la declaración de variables y constantes y los tipos de datos.

Pseudocódigo:

```
Inicio
Constante pi <- 3.141592
Cadena: nombre //También puede indicarse: Carácter: nombre
Enteros: a, b, c
nombre <- "Pepe"
Leer a, b, c
Imprimir nombre
Imprimir a, b, c
Fin
```

Nota

Al usar declaraciones de variables, decimos que estamos en **modo estricto**, en caso contrario, en **modo flexible**.

Comentarios

Una buena técnica y práctica de programación a la hora de escribir código, es usar comentarios. Un **comentario** es una instrucción que es *ignorada* por el compilador o intérprete en la ejecución, son algo así como *sentencias invisibles* para éstos. Su utilidad radica en que nos permiten **documentar** el código que estamos creando, haciendo que sea más ordenado y entendible, y facilitando así que otros programadores puedan retomar los desarrollos y estudiarlos y/o modificarlos de algún modo para realizar mantenimiento sobre ellos.

Los comentarios también son útiles en la etapa de *desarrollo* (antes del lanzamiento final -*producción*-), porque permiten tener varias versiones de una posible solución y realizar pruebas con cada una de forma independiente, comentando y descomentando según el caso, para evitar la engorrosa tarea de borrar, copiar, cortar y pegar texto para múltiples pruebas.

Cada lenguaje define su sintaxis propia para especificar los comentarios, por lo cual se deberá tener a la mano la documentación para saber aplicarlos según el que estemos utilizando. La forma para especificar comentarios que utilizaremos en algoritmia, será usando dos símbolos de **barra oblicua (slash) //**, y esto significará que todo lo que continúe hacia la derecha de dichos símbolos serán ignorados en la ejecución. En el ejemplo anterior ya ilustró el uso de comentarios, como puede observarse en la tercera línea (instrucción).

Sintaxis

```
// Texto del comentario
```

Ejemplo 1.12

Uso de comentarios.

Pseudocódigo:

```
Inicio
// A continuación se leerán y mostrarán las variables a, b y c
Leer a, b, c
Imprimir a, b, c
Fin
```

Errores

Es muy común que cuando escribimos código, se presenten errores y no obtengamos los resultados esperados. Los errores pueden darse por varias razones, y los analizamos de la siguiente manera cuando el programa es puesto bajo el análisis del compilador o intérprete.

Errores de sintaxis

Se dan por mala escritura de las sentencias del programa, lo cual se revela al digitar incorrectamente una palabra reservada, no seguir de manera indicada la sintaxis (forma) de una instrucción, no cerrar correctamente los paréntesis en una expresión aritmética o las comillas en una cadena de caracteres, escribir dos operadores binarios consecutivos, usar caracteres especiales para nombrar variable o no finalizar instrucciones que lo requieren, entre otros ejemplos, causan errores de sintaxis.

Errores de lógica/ejecución

Se generan por no cumplir alguna regla impuesta por el lenguaje, como por ejemplo tratar de cambiar el valor de una constante en tiempo de ejecución, asignar un dato a una variable que no corresponde al tipo de dato de ésta o especificar instrucciones donde el lenguaje lo tiene prohibido.

Los errores de ejecución también pueden darse por planteamientos incorrectos del programador en la lógica del problema, que pueden dar como resultados bucles “infinitos” que causan bloqueos del programa y/o máquina, salidas inesperadas, cálculos erróneos, entre otros casos.

Palabras reservadas

A medida que hemos ido avanzando, encontramos que hay ciertas palabras que se utilizan siempre o en distintos casos en la **estructura** del algoritmo. Dichas palabras se consideran

reservadas y no pueden utilizarse para el nombre de variables o constantes. Las palabras reservadas en algoritmia que utilizaremos, son las siguientes, sin caracteres especiales y teniendo en cuenta la sensibilidad de los caracteres:

Inicio	Si	Repetir	EnCasoDe	Continuar
Fin	SiNo	Hasta	Segun	Interrumpir
Entero(s)	Entonces	De	FinSegun	Salir
Real(es)	FinSi	HastaQue	Caso	Valor (Val)
Cadena	Y	Hacer	FinEnCasoDe	Referencia (Ref)
Logico	O	FinMientras	FinCaso	FinFuncion
Constante	No	FinPara	Funcion	FinProcedimiento
Imprimir	OBien	FinRepita	Procedimiento	Algoritmo
Escribir	Para	Desde	Retornar	FinAlgoritmo
Leer	Mientras	FinDesde	Devolver	Arreglo
Clase	Privado	Publico	Metodo	FinMetodo
constructor	FinClase	Protegido	Nuevo	destructor
Nulo	minusculas	mayusculas	longitud	subCadena
concatenar	SubAlgoritmo	FinSubAlgoritmo		

Notaciones comunes en programación

En la escritura en general es común encontrarnos con formas particulares de representar palabras o frases. Esto ha sido llevado y aplicado en la informática con excelentes resultados, ya que ha permitido mejorar la semántica del código de programación, así como mejorar los estilos y buenas prácticas; se aplica para los nombres de variables, funciones, procedimientos y objetos, entre otros. Veamos algunas de las más utilizadas.

Notación *camel case*²¹

La notación **camel case** o **camelCase**, (*Letra de caja de camello*) es una forma de escritura inicialmente adoptada en lengua inglesa y más tarde extendida a otros idiomas y usada

²¹ Puede consultar más sobre esta notación en: [Camel case - Wikipedia, la enciclopedia libre](https://es.wikipedia.org/wiki/Camel_case)

particularmente en la escritura de códigos de programación; es utilizada para escribir frases cortas compuestas de unas cuantas palabras todas pegadas y en donde la letra inicial de cada una de éstas se escribe en mayúscula y las demás letras en minúscula. Su nombre obedece a la similitud con la joroba de un camello.

Hay dos formas de *camel case*:

- *UpperCamelCase*: cada palabra en mayúscula inicial; por ejemplo: ClaseLógicaProgramación.
- *lowerCamelCase*: o simplemente *camelCase*, es similar a la anterior, pero con la diferencia que la primera letra está en minúscula; por ejemplo: claseLógicaProgramación.

Notación *snake case*²²

Es un estilo de escritura donde cada espacio se reemplaza por un guión bajo o carácter de subrayado (*snake_case*). Aunque se indica que esta notación debe iniciar cada palabra en minúscula, en la práctica se combinan escrituras de acuerdo a las necesidades; por ejemplo:

- Clase_Lógica_Programación
- clase_lógica_programación
- CLASE_LÓGICA_PROGRAMACIÓN

Notación húngara²³

Empleada en el campo de la programación, utiliza prefijos en los nombres de las variables, dando una descripción de ellas. Fue desarrollada por el ingeniero informático [Charles Simonyi](#), nacido en Hungría y de allí su nombre. Por ejemplo, podemos utilizar esta notación para describir el tipo de objeto en una interfaz gráfica o el tipo de dato de una variable: numEdad, cadNombre, lstLista, txt_cuadroTexto, btn_boton_mostrar, log_suiche. Observe cómo en esta notación se combinan las anteriores, tales como el *camelCase* o *snake_case*, con el fin de proveer una mejor escritura de las variables y facilitar así su lectura.

Notas

- Las notaciones *camelCase* y *snake_case*, basan su importancia en la facilidad que brinda a la hora de leer varias palabras que se encuentran unidas sin espacios, algo muy común al nombrar variables en un programa.
- Las empresas generalmente adoptan estándares propios para facilitar el desarrollo de sus tareas, por tanto, es posible que se encuentre con combinaciones de estos tipos de notaciones empleadas en proyectos.

²² Puede consultar más sobre esta notación en: [Snake case - Wikipedia, la enciclopedia libre](#)

²³ Puede consultar más sobre esta notación en: [Convenciones de codificación de Windows - Win32 apps | Microsoft Learn](#) y [Notación húngara - Wikipedia, la enciclopedia libre](#)

Como interpretar la sintaxis de una instrucción

En los textos de informática, particularmente de programación, encontramos repetidamente referirse a la sintaxis de una u otra instrucción. La **sintaxis** se refiere a las reglas (sintácticas) establecidas para combinar los distintos símbolos en un lenguaje de programación. Al encontrar la descripción de la sintaxis de una sentencia, nos encontraremos con algunos estilos tipográficos, así como algunos caracteres, los cuales tienen un significado a la hora de interpretar ésta. Veamos cómo se compone una sintaxis en una instrucción:

- Palabras en **negrita**: son palabras reservadas del lenguaje de programación
- Palabras en *cursiva*: son datos, expresiones o sentencias que se piden
- Barra vertical |: se pide elegir entre uno u otro
- Datos entre corchetes []: sentencias opcionales
- Puntos suspensivos: continúa de la misma forma que anteceden

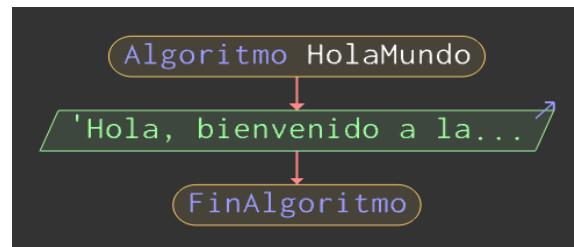
Problemas resueltos

Los siguientes ejercicios de programación resueltos incluyen los temas del manejo de variables, operadores, asignaciones, entrada y salida de información.

Ejercicio 1.1

Mostrar un mensaje de bienvenida al usuario por pantalla. Este es el famoso programa “Hola Mundo” que ilustra la salida estándar por pantalla y permite dar los primeros pasos en programación.

Diagrama libre²⁴:



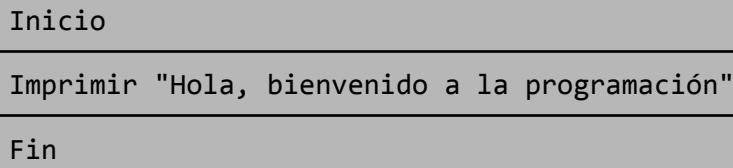
Pseudocódigo:

Inicio

²⁴ Varias de las imágenes utilizadas en el texto para ilustrar diagramas libres y de Nassi-Schneiderman son tomadas del pseudo lenguaje PSeInt, herramienta empleada por los estudiantes para ayudar en el aprendizaje de la lógica de programación.

```
Imprimir "Hola, bienvenido a la programación"
Fin
```

Diagrama Nassi-Schneiderman:



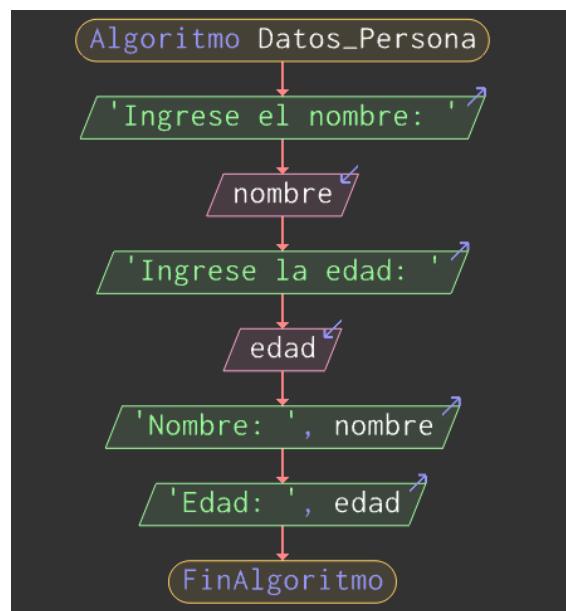
Pseudo programa:

```
Algoritmo HolaMundo
    Imprimir "Hola, bienvenido a la programación"
FinAlgoritmo
```

Ejercicio 1.2

Ingresar el nombre y edad de una persona por teclado y luego mostrarlos por pantalla. Este programa ilustra la entrada estándar por teclado.

Diagrama libre:

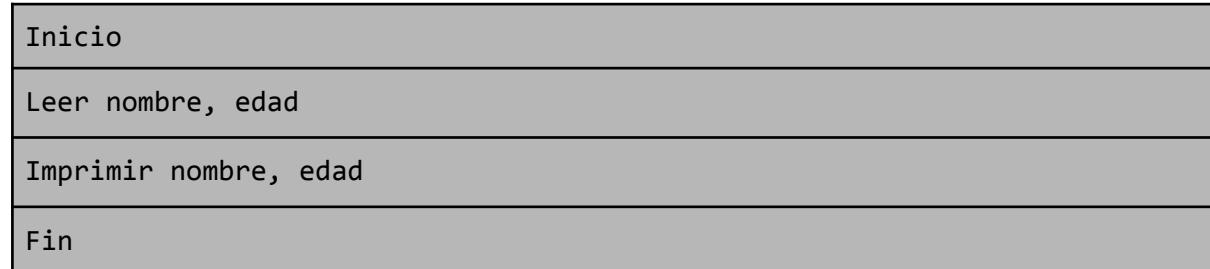


Pseudocódigo:

```
Inicio
    Leer nombre, edad
```

```
Imprimir nombre, edad  
Fin
```

Diagrama Nassi-Schneiderman:



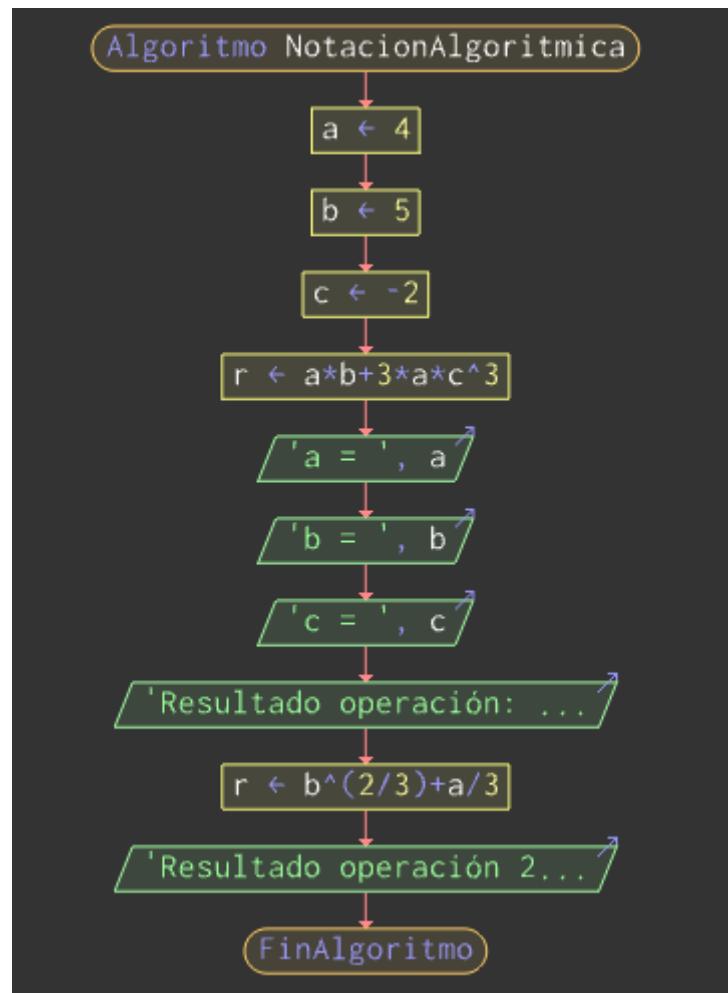
Pseudo programa:

```
Algoritmo Datos_Persona
    Imprimir "Ingrese el nombre: "
    Leer nombre
    Imprimir "Ingrese la edad: "
    Leer edad
    Escribir "Nombre: ", nombre
    Escribir "Edad: ", edad
FinAlgoritmo
```

Ejercicio 1.3

Escribir el pseudocódigo para convertir la expresión aritmética a notación algorítmica y hallar su valor numérico: $ab + 3ac^3$, si $a = 4$, $b = 5$, $c = -2$ (ejemplo 1.9 a).

Diagrama libre:

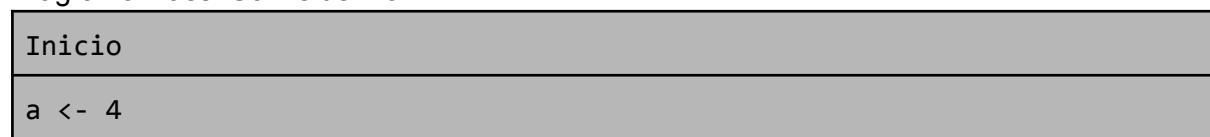


Pseudocódigo:

```

Inicio
a <- 4
b <- 5
c <- -2
r <- a * b + 3 * a * c ^ 3
Imprimir "a = ", a
Imprimir "b = ", b
Imprimir "c = ", c
Imprimir "Resultado operación: ", r
r = b ^ (2 / 3) + a / 3
Imprimir "Resultado operación 2: ", r
Fin
    
```

Diagrama Nassi-Schneiderman:



```
b <- 5  
c <- -2  
r <- a * b + 3 * a * c ^ 3  
Imprimir "a = ", a  
Imprimir "b = ", b  
Imprimir "c = ", c  
Imprimir "Resultado operación: ", r  
r = b ^ (2 / 3) + a / 3  
Imprimir "Resultado operación 2: ", r  
Fin
```

Pseudo programa:

```
Algoritmo NotacionAlgoritmica  
    a <- 4  
    b <- 5  
    c <- -2  
    r <- a * b + 3 * a * c ^ 3  
    Imprimir "a = ", a  
    Imprimir "b = ", b  
    Imprimir "c = ", c  
    Imprimir "Resultado operación: ", r  
    r = b ^ (2 / 3) + a / 3  
    Imprimir "Resultado operación 2: ", r  
FinAlgoritmo
```

Preguntas

1. Describa los operadores más comunes utilizados en matemáticas y computación
2. ¿Qué es un operador matemático?
3. ¿Qué es un número primo?
4. ¿Qué devuelve una operación de comparación?
5. ¿Qué devuelve un conectivo lógico?
6. ¿Cuáles son los operadores fundamentales?
7. ¿Cuáles son los valores de verdad de las constantes lógicas?
8. ¿A qué hace referencia la prioridad y cómo puede alterarse?

9. ¿Todos los números primos son impares?
10. ¿Qué se entiende por “leer”? ¿Quién “lee”, cómo lo hace?
11. ¿Qué significa “salida de datos”, cómo se representa algorítmicamente?
12. ¿Qué sucede si se trata de hacer una división por cero al ejecutar un programa?
13. ¿Cuáles son las notaciones más usadas en programación, por qué son útiles?
14. A nivel algorítmico ¿cuáles operadores de asignación se utilizan? ¿Qué significa dicha operación?
15. Si $p = \text{Verdadero}$, $q = \text{Verdadero}$ y $r = \text{Falso}$, entonces el resultado de la expresión: $p \text{ Y } \text{No}(q \text{ O } (r \text{ Y } \text{No}(p)))$ ¿qué da como resultado?
16. ($F \text{ o } V$) La expresión: $\text{Falso} \text{ O } \text{No}(\text{No}("z" > "m")) \text{ Y } (2 * -8 < 0)$ da como resultado verdadero
17. La expresión: $2 + (3 + 5 * (8 - 3))$ da como resultado 30
18. La expresión: $2 - 3 = -1$ OBien $3 <> 12 / 4$ da como resultado verdadero

Ejercicios

1. Realice los siguientes cálculos para encontrar el valor numérico de cada expresión si $a = 3$, $b = 4$, $c = -1$, $d = -5$, $e = 2$ y reescriba cada expresión del punto en notación algorítmica
 - a. $ab^2 + 3c - \sqrt{b}$
 - b. $5e - 2bcd + 4(d^3 - 2a + c) + a \% e$
 - c. $(\frac{b}{e} + \frac{e}{c}) - 6(\frac{c^2}{a})$
 - d. $\sqrt[3]{d^6} + \frac{a+b+c}{e} + e - b \% 2$
 - e. $\sqrt{ab - a} + 5d \div c - a^4 be$
2. Teniendo en cuenta los resultados encontrados en el punto 1), determine el valor lógico de las siguientes comparaciones (las letras corresponden a resultados encontrados en los literales del punto 1), no a los valores numéricos dados allí)
 - a. $a > b$
 - b. $ab = cd / e$
 - c. $d ^ 2 \leq 5ae - b/2$
 - d. $d <> 2ea$
 - e. $3/c \geq 6be + 4a$
3. Diseñe algoritmos (cualitativos) para resolver las siguientes situaciones que se plantean:
 - a. Ir a cine
 - b. Ir al estadio
 - c. Preparar un huevo revuelto
 - d. Alistarse para dormir
 - e. Preparar un café
 - f. Lavar los trastes de la cocina
 - g. Buscar el número telefónico de un compañero
 - h. Cambiar una llanta chuzada (elija el tipo de vehículo)

- i. Pagar una cuenta (servicios, crédito, etc.)
 - j. Matricularse en la universidad
4. Ingrese dos valores numéricos en dos variables e intercambie sus valores, esto es, si las variables son a y b , el valor de a debe quedar en b y el de b en a . Muestra las variables antes y después del intercambio
 5. Intercambie los valores de tres variables así: el valor de la primera debe quedar en la segunda, el valor de la segunda variable en la tercera, y el valor de la tercera variable en la primera
 6. Ingrese un número y un porcentaje por teclado. Calcule a cuánto equivale dicho porcentaje
 7. Elabore un algoritmo para realizar conversiones de temperaturas dadas en grados centígrados (Celsius) a grados Fahrenheit $F = \frac{9}{5}C + 32$
 8. Elabore un algoritmo para realizar conversiones de temperaturas dadas en grados Fahrenheit a grados Celsius
 9. Calcular el salario básico (sb), total de deducciones y salario neto (sn) de un empleado. De éste se conoce su nombre, salario básico hora (sbh) y número de horas trabajadas (nht). También se sabe que las deducciones equivalen a un 8% del salario básico por concepto de salud y pensión
 10. Usando las constantes lógicas, realice operaciones con los conectivos lógicos y muestre qué resultados producen
 11. Dada la velocidad promedio de un vehículo y su tiempo de desplazamiento, encuentre la distancia recorrida $d = vt$
 12. Encuentre la fuerza para una masa y aceleración dadas $F = ma$
 13. Dados los lados de un rectángulo, calcule su perímetro y su área
 14. Dado un lado de un cuadrado, calcule su perímetro y su área
 15. Dados los lados de un triángulo, calcule su perímetro
 16. Dada la base b y la altura h de un triángulo, calcule su área $A = \frac{bh}{2}$
 17. Dados los lados de un rectángulo, calcule su perímetro y determine si es un cuadrado
 18. Se tienen los catetos de un triángulo rectángulo. Calcule su hipotenusa $h^2 = c_1^2 + c_2^2$ (teorema de Pitágoras)
 19. Dado el radio de una circunferencia, calcule su perímetro y área $p = 2\pi r$, $A = \pi r^2$
 20. Calcule el volumen y área (superficie) de una esfera si se conoce su radio $V = \frac{4}{3}\pi r^3$, $A = 4\pi r^2$
 21. Dado el radio r y altura h de un cilindro, calcule su volumen $V = \pi r^2 h$
 22. Ingrese el año de nacimiento de una persona y calcule su edad aproximada
 23. Calcule el área de un triángulo en función de sus lados:

$$A = \sqrt{p(p - a)(p - b)(p - c)}$$
, donde $p = \frac{a+b+c}{2}$ es el semiperímetro
 24. Diseñe un programa que permita calcular el valor en pesos de una cantidad en dólares. Debe especificar la tasa de cambio

Unidad 2. Estructuras de control

En los lenguajes de programación, las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con ellas se puede:

- De acuerdo a una condición (comparación), ejecutar un grupo u otro de sentencias
- Ejecutar un grupo de sentencias un número determinado de veces
- Interrumpir la ejecución normal del programa

Todas las estructuras de control tienen un único punto de entrada y un único punto de salida. Éstas se pueden clasificar en: decisión, iteración y de control avanzadas.

Condicionales

Se utilizan para tomar decisiones a partir de valores booleanos obtenidos de la comparación de expresiones lógicas. Veamos cómo se implementan algorítmicamente mediante la instrucción **Si**.

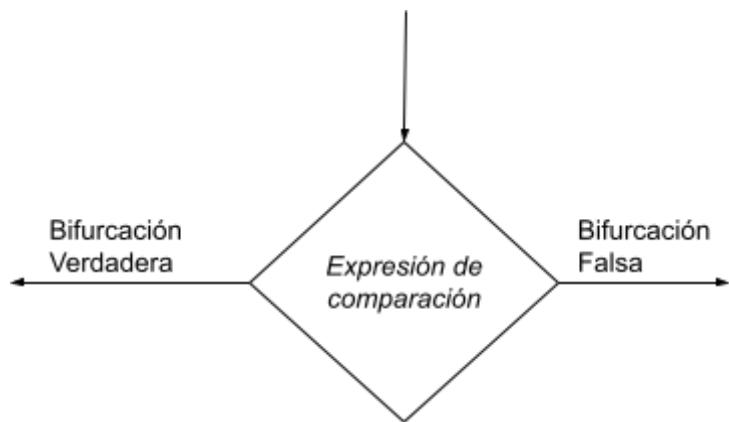
Condicional simple

Evalúa una determinada condición o expresión de comparación, en caso de ser verdadera, se ejecuta un bloque de instrucciones. Si dicha condición no se cumple, esto es, es falsa, entonces ninguna de las instrucciones es ejecutada.

Sintaxis

```
Si expresión_de_comparación [Entonces]
    Bloque de instrucciones si expresión_de_comparación es verdadera
FinSi
```

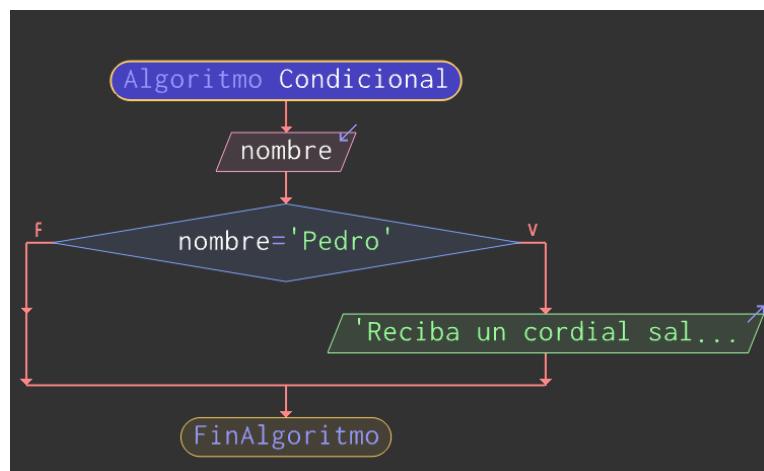
Gráficamente, el condicional se ve así (flujograma):



Ejemplo 2.1

Leer el nombre de una persona. Enviar un mensaje de saludo si el nombre ingresado es “Pedro”.

Diagrama libre:



Pseudocódigo:

```

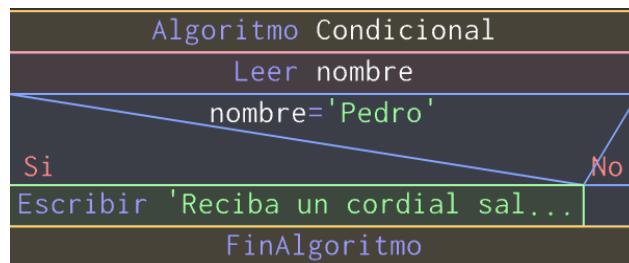
Inicio
Carácter: nombre
Ler nombre
Si nombre = "Pedro" Entonces
    Imprimir "Reciba un cordial saludo, señor ", nombre
FinSi
Fin
  
```

Nota

Observe el uso de la sangría en las instrucciones que se indican si la expresión de comparación se cumple en el caso del pseudocódigo. Esa es una práctica muy generalizada que permite conservar las **reglas de estilos** definidas en los diferentes

lenguajes, ya que permite mayor orden y mejor lectura del código. La sangría estándar es de cuatro (4) espacios en blanco.

Diagrama Nassi-Schneiderman:



Condicional compuesto

Evalúa una determinada condición o expresión de comparación, en caso de ser verdadera, se ejecuta un bloque de instrucciones. Si dicha condición no se cumple, esto es, es falsa, entonces ninguna de las instrucciones es ejecutada a no ser que se especifique la cláusula **SiNo**, y decimos que se trata de un *condicional compuesto*. Sin embargo, la sentencia SiNo es opcional y su uso depende de las necesidades del programador.

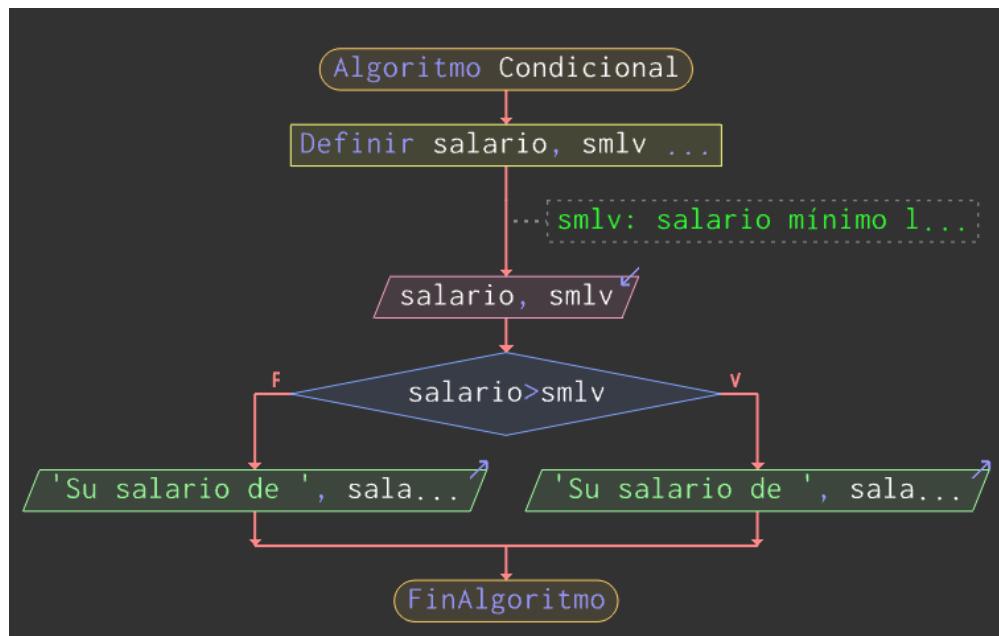
Sintaxis

```
Si expresión_de_comparación [Entonces]
    Bloque de instrucciones si expresión_de_comparación es verdadera
[SiNo]
    Bloque de sentencias si expresión_de_comparación es falsa]
FinSi
```

Ejemplo 2.2

Leer el salario de un trabajador. Determinar si gana el salario mínimo o más de éste. El salario mínimo debe ingresarse para el año actual.

Diagrama libre:

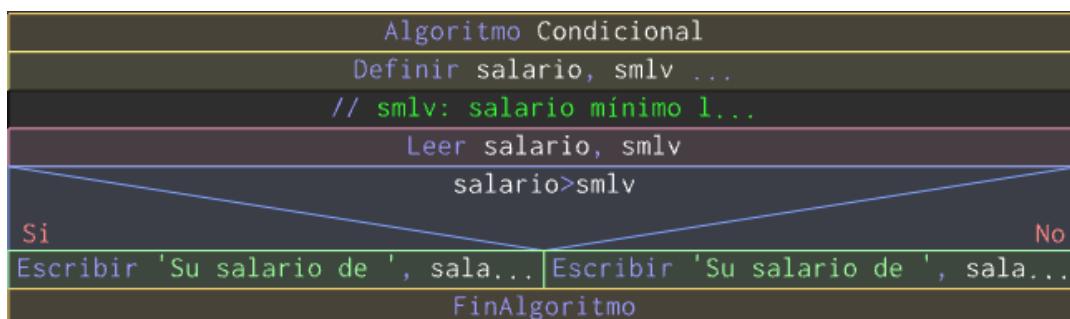


Pseudocódigo:

```

Inicio
Reales: salario, smlv //smlv: salario mínimo legal vigente
Leer salario, smlv
Si salario > smlv Entonces
    Imprimir "Su salario de ", salario, " supera el smlv actual de ", smlv
SiNo
    Imprimir "Su salario de ", salario, " es igual o inferior a ", smlv
FinSi
Fin
    
```

Diagrama Nassi-Schneiderman:



Condicional anidado

Es un condicional que se encuentra dentro de otro condicional. Es muy común encontrar condicionales anidados en programas, debido a las soluciones que se deben proponer en distintos problemas.

Sintaxis

```

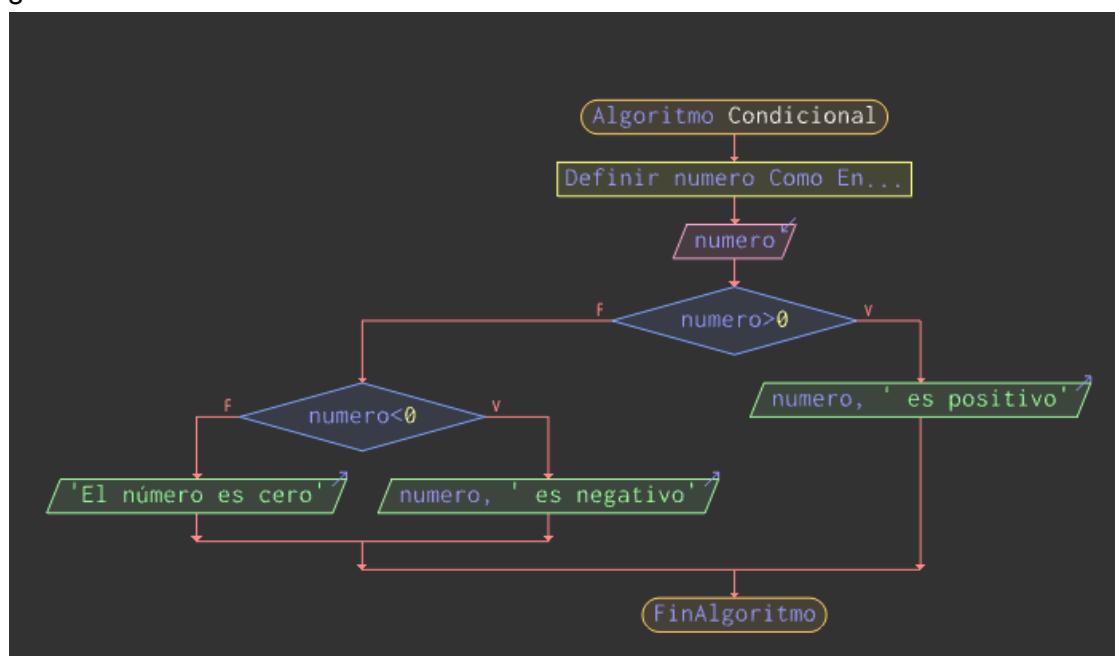
Si expresión_de_comparación1 [Entonces]
    Si expresión_de_comparación2 Entonces
        Instrucciones si expresión_de_comparación 1 y 2 son verdaderas
    FinSi
[SiNo
    Bloque de sentencias si expresión_de_comparación1 es falsa]
FinSi

```

Ejemplo 2.3

Ler un número. Determinar si es positivo, negativo o cero.

Diagrama libre:



Pseudocódigo:

```

Inicio
    Enteros: numero
    Leer numero
    Si numero > 0 Entonces
        Imprimir numero, " es positivo"
    SiNo

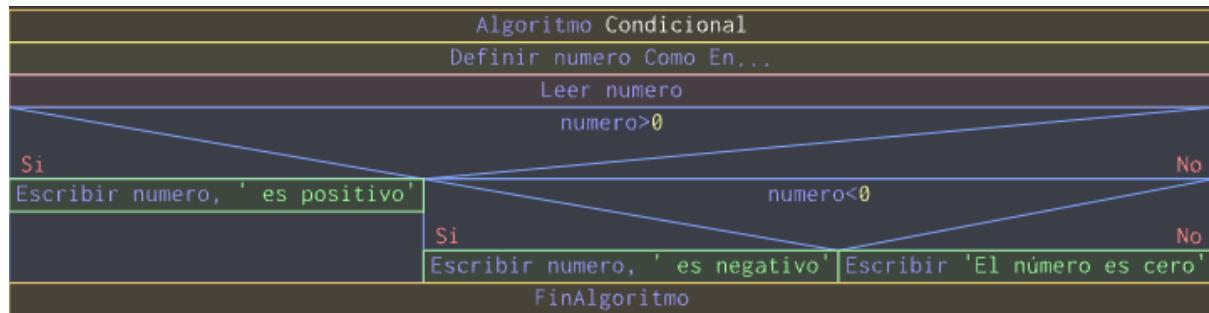
```

```

Si numero < 0 Entonces
    Imprimir numero, " es negativo"
SiNo
    Imprimir "El número es cero"
FinSi
FinSi
Fin

```

Diagrama Nassi-Schneiderman:



Selector múltiple o estructura caso

Permite la ejecución de un bloque de instrucciones en función del valor que tome una expresión. Es utilizada en lógica de programación como alternativa al condicional cuando se tienen más de dos salidas lógicas. También se conoce como la estructura “Según”. Veamos dos alternativas para usar esta estructura de control en lógica de programación.

Sintaxis

1.

```

EnCasoDe expresión [Hacer]
    Caso valor_1:
        Instrucciones si valor_1 coincide con expresión
    Caso valor_2:
        Instrucciones si valor_2 coincide con expresión
    ...
    Caso valor_n:
        Instrucciones si valor_n coincide con expresión
    [EnOtroCaso:
        Instrucciones si ningún valor coincide con expresión]
FinCaso

```

2.

```

Según expresión [Hacer]
    Caso valor_1:
        Instrucciones si valor_1 coincide con expresión

```

```

Caso valor_2:
    Instrucciones si valor_2 coincide con expresión
...
Caso valor_n:
    Instrucciones si valor_n coincide con expresión
[DeOtroModo:
    Instrucciones si ningún valor coincide con expresión]
FinSegún

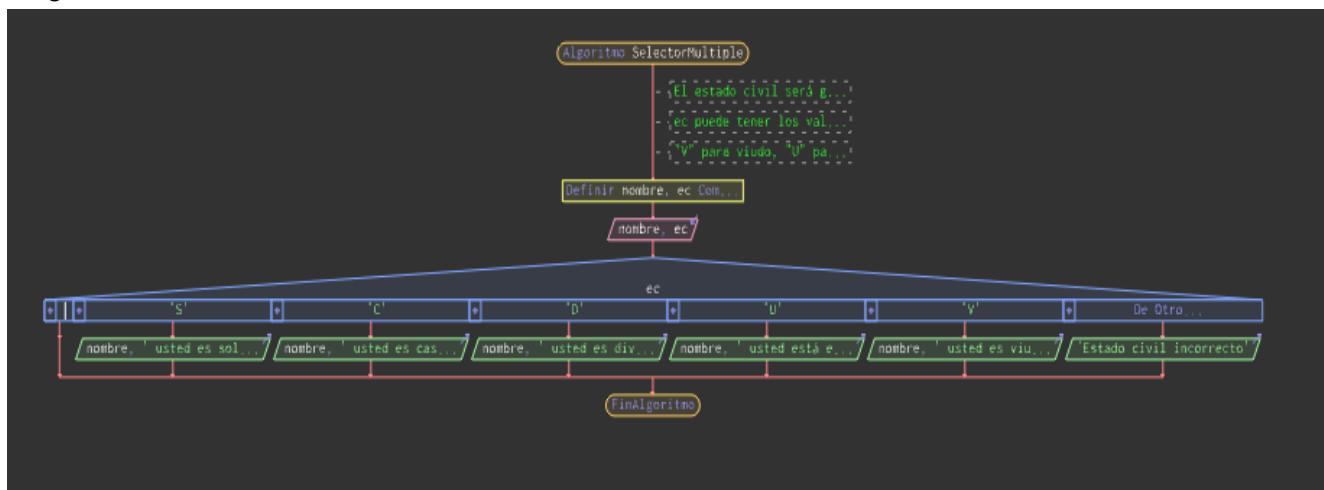
```

Donde *expresión* es el resultado de alguna operación o variable, generalmente de tipo entero o carácter.

Ejemplo 2.3

Leer el nombre y estado civil de una persona. Indique cuál es el estado civil de esta persona.

Diagrama libre:



Pseudocódigo:

```

Inicio
//El estado civil será guardado en la variable de tipo carácter ec
//ec puede tener los valores: "C" para casado, "S" para soltero
//,"V" para viudo, "U" para unión libre y "D" para divorciado
Carácter: nombre, ec
Leer nombre, ec
EnCasoDe ec Hacer
    Caso "S":
        Imprimir nombre, " usted es soltero(a)"
    Caso "C":
        Imprimir nombre, " usted es casado(a)"
    Caso "D":
        Imprimir nombre, " usted es divorciado(a)"
    Caso "U":
        Imprimir nombre, " usted está en una relación..."
    Caso "V":
        Imprimir nombre, " usted es viudo(a)"
    Caso "De Otro...":
        Imprimir nombre, " Estado civil incorrecto"
    FinCaso
FinAlgoritmo

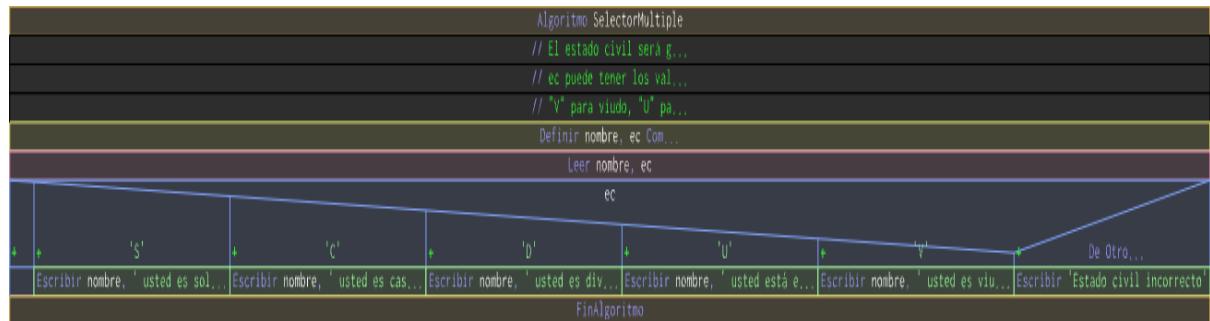
```

```

Caso "U":
    Imprimir nombre, " usted está en unión libre"
Caso "V":
    Imprimir nombre, " usted es viudo(a)"
EnOtroCaso:
    Imprimir "Estado civil incorrecto"
FinCaso
Fin

```

Diagrama Nassi-Schneiderman:



Nota

Observe cómo esta estructura simplifica el uso del condicional que para ciertos casos requiere realizar varios, añadiéndole complejidad al algoritmo y su ejecución.

Ciclos

Los ciclos o bucles son estructuras de control que repiten un grupo de instrucciones mientras se cumpla una condición o expresión de comparación, o incluso mientras ésta no se cumpla. En lógica de programación se cuenta con tres tipos de ciclos, a saber: **Para**, **Mientras** y **Repetir**, los cuales también se encuentran presentes en los distintos lenguajes de programación, a excepción del tercero que no todos los lenguajes cuentan con él. Veamos cada uno de ellos e ilustremos su uso con ejemplos.

Ciclo Mientras

Permite la repetición de un bloque de instrucciones un determinado número de veces de acuerdo a una condición: si ésta es verdadera, las instrucciones del ciclo se repiten, en caso contrario finaliza la ejecución de éste y continúa con la siguiente instrucción después del ciclo. Es posible que las sentencias del bucle no se lleguen a ejecutar nunca, ya que antes de proceder a interpretar la primera instrucción se evalúa la condición, y si ésta resulta ser falsa, no entrará en las instrucciones del bloque.

Sintaxis

```
Mientras expresión_de_comparación Hacer
    Instrucciones si expresión_de_comparación es verdadera
FinMientras
```

Ejemplo 2.4

Mostrar los números del 1 al 10.

Este problema puede resolverse así:

Pseudocódigo:

```
Inicio
Imprimir "1, 2, 3, 4, 5, 6, 7, 8, 9, 10"
Fin
```

O también de esta forma

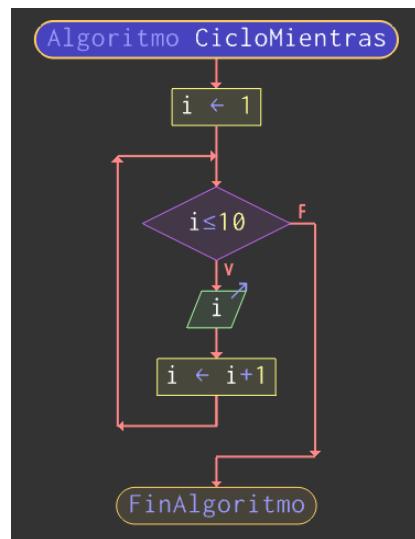
Pseudocódigo:

```
Inicio
Imprimir 1
Imprimir 2
Imprimir 3
Imprimir 4
Imprimir 5
Imprimir 6
Imprimir 7
Imprimir 8
Imprimir 9
Imprimir 10
Fin
```

¿Pero, qué sucede si en vez de mostrar 10 números, se deben mostrar 100, 1000, 10000 etc.?

Como podemos ver, para grandes cantidades de datos no es práctica de ninguna manera las soluciones mostradas arriba. Sin embargo, la segunda solución nos muestra un *patrón* que se *repite* un número determinado de veces y que podemos aprovechar para llevarla a una estructura *repetitiva (cíclica)* que nos permita simplificar la escritura de esta solución.

Diagrama libre:

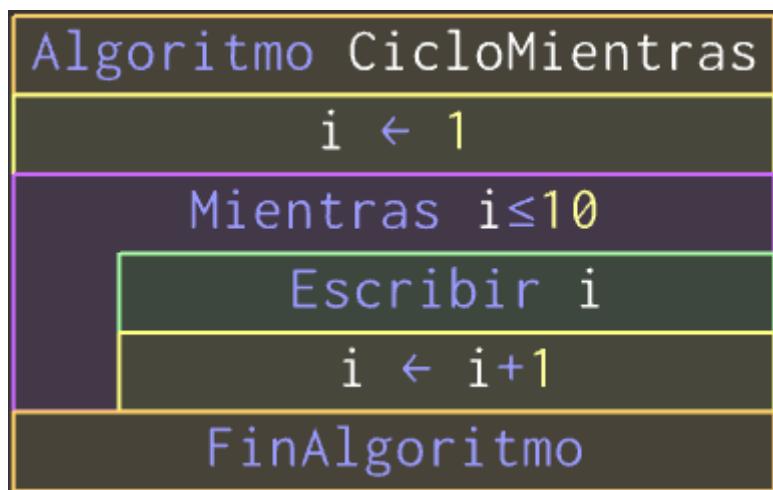


Pseudocódigo:

```

Inicio
Enteros: i
i = 1
Mientras i <= 10 Hacer
    Imprimir i
    i = i + 1
FinMientras
Fin
    
```

Diagrama Nassi-Schneiderman:



Observe que esta última solución es la más óptima, ya que fácilmente podemos cambiar el límite de la condición para valores grandes; por otro lado, reduce significativamente el

número de líneas de código, hace más elegante la solución, y para problemas que veremos más adelante, es el tipo de solución que debemos adoptar de forma obligatoria.

Observe también cómo es necesario cambiar el valor de la variable *i* en cada pasada del ciclo de tal forma que en algún momento deje de cumplirse la condición del ciclo y consiguiendo así que este finalice y no entre en un bucle infinito.

Veamos cómo se representa la estructura de un ciclo con el diagrama Nassi-Schneiderman, para este con el ciclo *Mientras*.

Prueba de escritorio

Es un seguimiento que se realiza a las variables de un programa mostrando cómo se comportan a medida que el programa “se ejecuta”, recibiendo entradas y siendo parte de operaciones y salidas. La prueba de escritorio es una prueba manual de cómo el algoritmo ejecuta cada paso, y se convierte en una justificación y argumentación del correcto funcionamiento de éste. Es una herramienta muy importante y útil para comprobar si un programa está construido correctamente o no.

Ejemplo 2.5

Realizar la prueba de escritorio del ejemplo 2.4 para un límite de datos de 5.

Solución

Construimos una tabla y en ella ubicamos en la primera fila las variables a medida que vayan apareciendo en el programa, ocupando cada una, una columna. Comenzamos a leer las siguientes líneas del programa, teniendo en cuenta que éstos se leen de arriba hacia abajo, además de las estructuras de control como condicionales o ciclos, las cuales siguen las reglas ya especificadas. Las distintas operaciones se van registrando sobre las variables involucradas a medida que se ejecuta cada instrucción (línea).

Prueba de escritorio:

i	Salida
1	1
2	2
3	3
4	4
5	5
6	

Contadores y acumuladores

Ejemplo 2.6

Sumar los números enteros de 1 a n (n es un número entero ingresado por teclado). Realizar la prueba de escritorio para n = 5.

Solución

En este programa, la variable i es un **contador**, mientras que la variable suma hace las veces de un **acumulador**.

Pseudocódigo:

```
Inicio
    Enteros: i, n, suma
    Leer n
    i = 1
    suma = 0
    Mientras i <= n Hacer
        suma = suma + i
        i = i + 1
    FinMientras
    Imprimir "Suma de 1 a ", n, ":" , suma
    Fin
```

Prueba de escritorio:

i	n	suma	Salida
1	5	0	Suma de 1 a 5: 15
2		1	
3		3	
4		6	
5		10	
6		15	

Nota (Anécdota)

Cuando el matemático Gauss²⁵ se encontraba en la escuela, el profesor le asignó esta tarea a él y a sus demás compañeros de grupo para que sumaran los números del 1 al 100 porque andaban muy inquietos y éste quería que guardaran silencio, así podría estar más tranquilo. Sin embargo, a los pocos minutos el joven Gauss se levantó y le dijo al profesor que ya había logrado resolver la tarea. Sorprendido el profesor revisa la actividad y puede ver como este niño plantea una fórmula para sumar los número de 1 a n, basándose en las diferencias que hay entre el primer y último número, segundo y penúltimo, tercer y antepenúltimo, etc.:

$$100 + 1 = 101$$

$$99 + 2 = 101$$

$$98 + 3 = 101$$

...

$$3 + 98 = 101$$

$$2 + 99 = 101$$

$$1 + 100 = 101$$

La fórmula que planteó Gauss para la suma de los número de 1 a n, fue la siguiente:

$$S = n * (n + 1) / 2$$

Donde n es el número entero hasta donde se desea sumar.

La solución planteada por Gauss se puede codificar fácilmente y representa una solución más efectiva y eficiente que la anterior que requiere ejecutar n veces el código para lograr el resultado, mientras que ésta solo una vez. Gauss es considerado uno de los tres matemáticos más grandes de la historia de la humanidad junto a Newton y Arquímedes.

Pseudocódigo (solución usando la fórmula de Gauss):

```
Inicio
    Eneros: n, suma
    Leer n
    suma = n * (n + 1) / 2
    Imprimir suma
Fin
```

Prueba de escritorio:

n	suma	Salida
5	15	15

²⁵ Puede leer más acerca de este gran personaje en [Carl Friedrich Gauss - Wikipedia, la enciclopedia libre](#)

Registro centinela

Ejemplo 2.7

Leer el nombre y salario de un grupo de empleados. Encontrar el total de personas, el promedio de salarios, el mayor salario y a quién pertenece éste. El último registro que se lee es un nombre con los caracteres “***”.

Solución

En este programa, la variable nombre toma un valor al final de “***” para dar por terminada la ejecución del ciclo; dicho valor hace parte del **“registro centinela”** de la lista dada. El valor de dicho centinela es arbitrario y puede depender de diversos factores.

Pseudocódigo:

```
Inicio
    Enteros: totalPersonas
    Reales: salario, mayorSalario, sumaSalario, promedioSalario
    Carácter: nombre, nombreMayor
    totalPersonas = 0
    sumaSalario = 0
    mayorSalario = 0
    Leer nombre
    Mientras nombre <> “***” Hacer
        Leer salario
        totalPersonas = totalPersonas + 1
        sumaSalario = sumaSalario + salario
        Si salario > mayorSalario Entonces
            mayorSalario = salario
            nombreMayor = nombre
        FinSi
        Leer nombre
    FinMientras
    Si totalPersonas > 0 Entonces
        promedioSalario = sumaSalario / totalPersonas
        Imprimir totalPersonas, promedioSalario, mayorSalario, nombreMayor
    SiNo
        Imprimir “No se procesó información”
    FinSi
Fin
```

Banderas o suiches

Ejemplo 2.8

Leer el código y la nota de un grupo de n estudiantes. Encontrar el promedio de notas e informar si al menos un estudiante obtuvo una nota de 5.0.

Solución

En este programa, la variable sw hace las veces de la **bandera** o **suiche**, que inicia “apagada” y que se “prende” en cualquier momento y permanece de esta forma hasta finalizar la ejecución del programa si los datos que se van suministrando cumplen ciertas condiciones.

Pseudocódigo:

```
Inicio
    Esteros: n, i
    Reales: nota, suma, promedio
    Lógicos: sw
    Carácter: codigo
    i = 1
    sw = Falso //Supuesto: nadie sacó 5.0
    Leer n
    Mientras i <= n Hacer
        Leer codigo, nota
        suma = suma + nota
        Si nota = 5 Entonces
            sw = Verdadero
        FinSi
        i = i + 1
    FinMientras
    promedio = suma / n
    Si sw Entonces
        Imprimir "Al menos un estudiante obtuvo una nota de 5.0"
    SiNo
        Imprimir "No hubo estudiantes que obtuvieran una nota de 5.0"
    FinSi
Fin
```

Ciclo Repetir ... Hasta

Es similar al ciclo while, con la diferencia de que la condición se evalúa al final del ciclo, garantizando que las instrucciones dentro de él se ejecutarán por lo menos una vez; y para que las sentencias se repitan, dicha expresión de comparación debe ser falsa, esto es, el ciclo deja de ejecutarse cuando la condición pasa a ser verdadera. Es ideal para implementar menús y validaciones, entre otras aplicaciones.

Sintaxis

Repetir

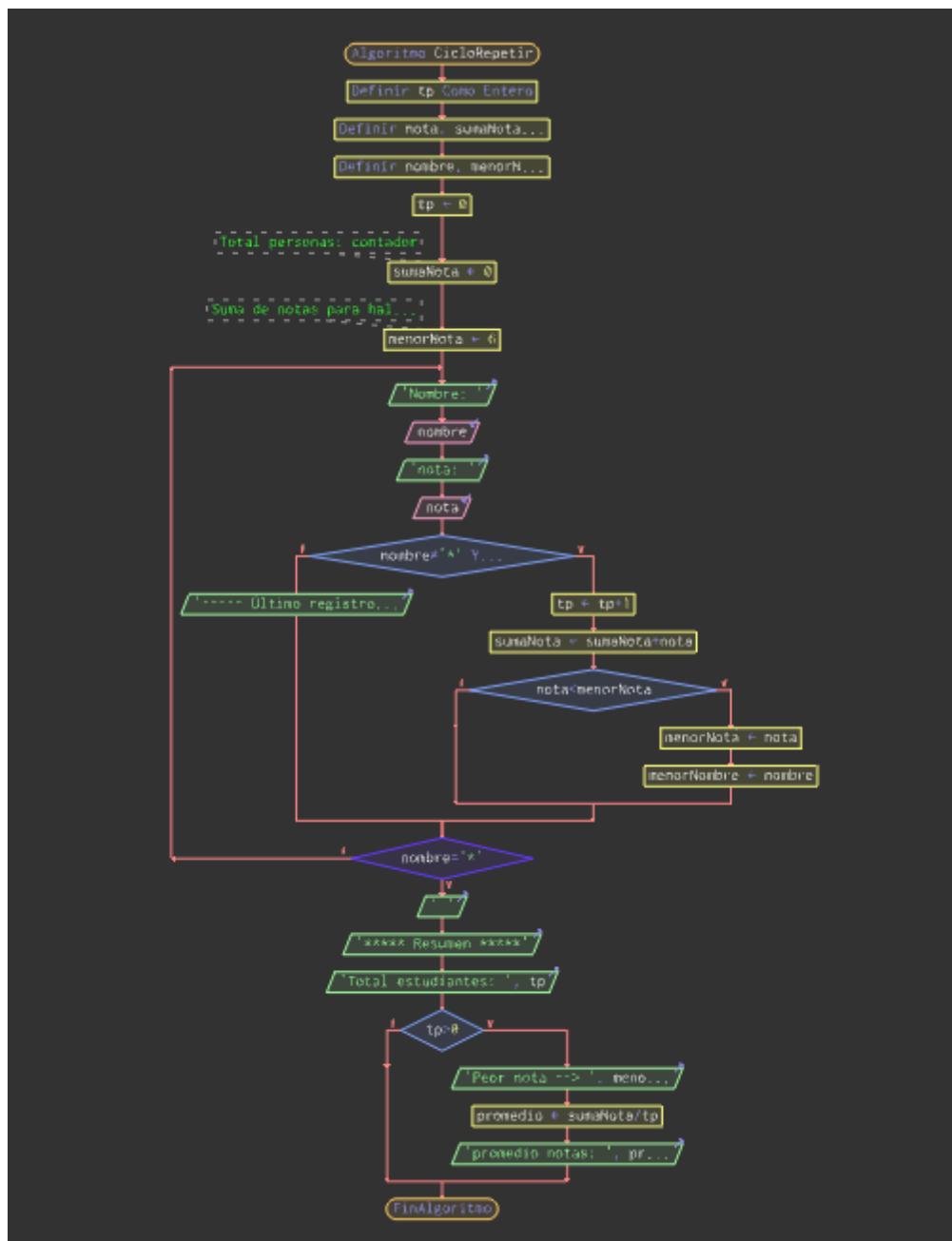
Instrucciones si expresión_de_comparación es falsa

Hasta [Que] expresión_de_comparación

Ejemplo 2.9

Leer el nombre y la nota de un grupo de estudiantes. Determinar el promedio validando que las notas estén entre 0.0 y 5.0; muestre además quien obtuvo la peor nota y cuál fue ésta. La lectura de datos finaliza cuando ingresen un código igual a “*”.

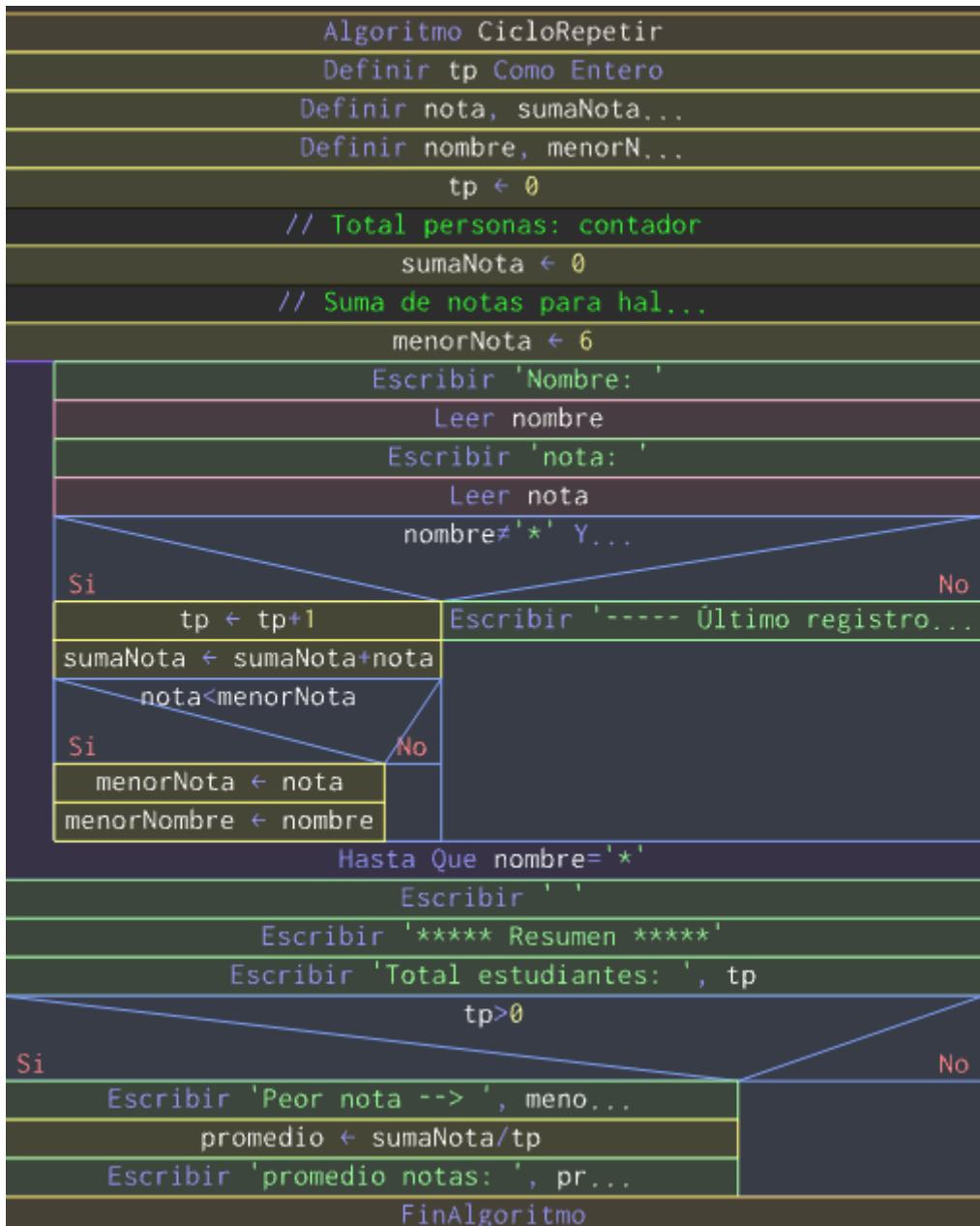
Diagrama libre:



Pseudocódigo (PSelInt):

```
Inicio
Definir tp Como Entero
Definir nota, sumaNota, menorNota, promedio Como Real
Definir nombre, menorNombre Como Caracter
tp = 0 //Total personas: contador
sumaNota = 0 //Suma de notas para hallar promedio
menorNota = 6
Repetir
    Imprimir "Nombre: " Sin Saltar
    Leer nombre
    Imprimir "nota: " Sin Saltar
    Leer nota
    Si nombre <> '*' Y nota >= 0 Y nota <= 5
        tp = tp + 1
        sumaNota = sumaNota + nota
        Si nota < menorNota Entonces
            menorNota = nota
            menorNombre = nombre
        FinSi
    SiNo
        Imprimir "----- Último registro no procesado -----"
    FinSi
Hasta Que nombre = '*'
Imprimir " "
Imprimir "***** Resumen *****"
Imprimir "Total estudiantes: ", tp
Si tp > 0
    Imprimir "Peor nota --> ", menorNombre, ": ", menorNota
    promedio = sumaNota / tp
    Imprimir "promedio notas: ", promedio
FinSi
Fin
```

Diagrama Nassi-Schneiderman:



Ejemplo 2.10

Ler el código y la nota de un grupo de estudiantes. Determinar el total de estudiantes, cuántos ganaron y perdieron y el porcentaje que éstos representan. Se debe crear un menú con las siguientes las siguientes opciones para gestionar los datos antes de entregar los resultados finales: 1) Lectura de datos; 2) Total de personas ingresadas hasta el momento; 3) Total ganadores, perdedores y porcentajes; 4) Salir.

Pseudocódigo (PSelInt):

```

Inicio
Definir totPersona, totGanador, totPerdedor Como Entero

```

```
Definir nota, ptjeGanador, ptjePerdedor Como Real
Definir codigo, opc Como Caracter
totPersona = 0 //Total personas
totGanador = 0 //Total ganadores
Repetir
    Imprimir "Menú de opciones"
    Imprimir "1. Leer datos"
    Imprimir "2. Total datos"
    Imprimir "3. Total ganadores/perdedores y porcentajes"
    Imprimir "4. Salir"
    Imprimir "Ingrese su opción: " Sin Saltar
    Leer opc //Opción que ingresa el usuario
    Segun opc Hacer
        Caso "1":
            Imprimir "Código: " Sin Saltar
            leer codigo
            Imprimir "nota: " Sin Saltar
            leer nota
            totPersona = totPersona + 1
            Si nota >= 3 Entonces
                totGanador = totGanador + 1
            FinSi
        Caso "2":
            Imprimir "Total personas: ", totPersona
        Caso "3":
            totPerdedor = totPersona - totGanador //Total perdedores
            Imprimir "Total personas: ", totPersona
            Imprimir "Total ganadores: ", totGanador
            Imprimir "Total perdedores: ", totPerdedor
            Si totPersona > 0
                ptjeGanador = totGanador * 100 / totPersona //Porc ganador
                ptjePerdedor = 100 - ptjeGanador
                Imprimir "Porcentaje ganadores: ", ptjeGanador, "%"
                Imprimir "Porcentaje perdedores: ", ptjePerdedor, "%"
            FinSi
        Caso "4":
            Imprimir "***** Programa finalizado *****"
    De Otro Modo:
        Imprimir "Opción no válida"
    FinSegun
Hasta Que opc = "4"
Fin
```

Ciclo Para

Tiene como fin repetir un bloque de instrucciones mientras cumpla una condición preestablecida. Se deben indicar tres parámetros: La condición que determina si se debe seguir ejecutando o no el bucle (*expresión de comparación*), una condición que vaya haciendo cambiar algún parámetro que varíe el cumplimiento de la condición anterior (*actualización o incremento -decremento-* de la variable controladora del ciclo), y por supuesto, una expresión que determine cuál es la situación de partida en el cumplimiento de dicha condición (*inicialización*). Este ciclo sólo puede usarse si se conoce el número de iteraciones a llevar a cabo.

Veamos varias formas en que puede escribirse un ciclo *Para*.

Sintaxis

1.

```
Para var_num <- val_ini Hasta val_fin [[Con] Paso incremento [Hacer]]  
    Instrucciones del ciclo  
FinPara
```

2.

```
Para var_num = val_ini Hasta val_fin [incremento [Hacer]]  
    Instrucciones del ciclo  
FinPara
```

3.

```
Para var_num = val_ini, val_fin[, incremento]  
    Instrucciones del ciclo  
FinPara
```

Donde:

- *var_num*: es una variable de tipo numérico, entera o real, también conocida como variable controladora del ciclo.
- *val_ini*: valor inicial que toma la variable controladora antes de iniciar el ciclo
- *val_fin*: valor final que toma la variable controladora antes de finalizar el ciclo; es el valor límite de la variable numérica
- *incremento*: es la forma como cambia la variable controladora a manera de contador en cada nueva iteración. El incremento puede ser negativo (decremento)

Notas

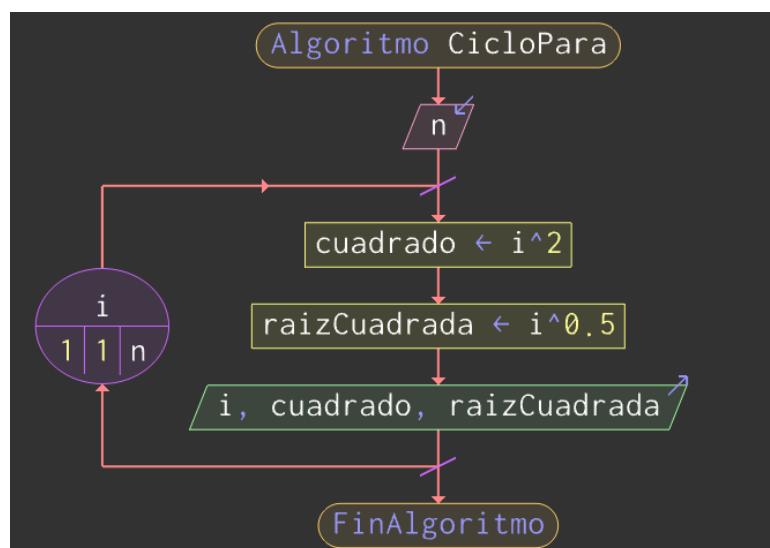
- Observe que el ciclo *Para* está compuesto de tres parámetros; la tercera sintaxis sólo simplifica la escritura.
- Si se omite el tercer parámetro, se asume por defecto que el incremento es de a uno (1).

- El segundo parámetro representa la condición de finalización, la cual está dada en función del valor especificado en el tercer parámetro.

Ejemplo 2.11

Mostrar el cuadrado y la raíz cuadrada de los primeros n números naturales.

Diagrama libre:

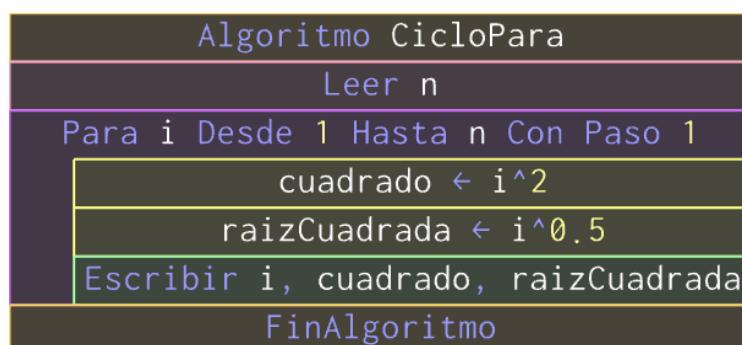


Pseudocódigo:

```

Inicio
Enteros: n, i
Reales: cuadrado, raizCuadrada
Leer n
Para i = 1 Hasta n Paso 1
    cuadrado = i ^ 2
    raizCuadrada = i ^ 0.5
    Imprimir i, cuadrado, raizCuadrada
FinPara
Fin
    
```

Diagrama Nassi-Schneiderman:



Ejemplo 2.12

Calcular el factorial de N. El factorial de un número se define para números enteros positivos como: $N! = N * (N - 1) * (N - 2) * \dots * 3 * 2 * 1$, por ejemplo $5! = 5 * 4 * 3 * 2 * 1 = 120$. Por definición $0! = 1$ y $1!=1$

Pseudocódigo:

```
Inicio
    Enteros: N, i, fact
    Leer N
    fact = 1
    Para i = N, 1, -1
        fact = fact * i
    FinPara
    Imprimir N, “! = ”, fact
    Fin
```

Nota

- Podemos ver como el ciclo **Mientras** es el más fundamental de todos, ya que todo proceso iterativo puede realizarse con éste, mientras que los otros dos tipos de ciclos son casos particulares que pueden ser resueltos por el primero escribiendo unas cuantas líneas de código más.
- Es de especial cuidado las condiciones que se planteen en los ciclos, ya que pueden generar bucles infinitos, lo cual en informática es un error en la lógica del planteamiento de la solución de algún problema y esto ocasionará que el programa se bloquee y deba cancelarse usando “fuerza bruta”.

Ciclos anidados

Son ciclos que se encuentran dentro de otro ciclo o bucle. En ocasiones es necesario plantear soluciones con ciclos anidados en programas, pero debe tenerse cuidado de que en realidad se requieran, ya que cada ciclo anidado añade gran complejidad al algoritmo y a su vez aumentando en gran medida su tiempo de ejecución. Al anidar ciclos, se pueden tener combinaciones de los distintos tipos: un ciclo *Mientras* dentro de un *Para* o viceversa, etc.

Ejemplo 2.13

Una empresa con cinco (3) sucursales desea conocer cuántos empleados participarán de una rifa. La actividad es informal, por lo que las personas que están recolectando la información no tienen claridad del número de empleados por sucursal ni de toda la empresa. Ellos, aprovecharán esta encuesta para saber el total de empleados de la empresa y por sucursal.

Pseudocódigo:

```
Algoritmo CiclosAnidados
    Definir cpe, cps, cp_si, cp_no Como Entero
    Definir respuesta, nombre Como Caracter
    cpe = 0 //Contador personas empresa
    cp_si = 0 //Contador personas participan
    cp_no = 0 //Contador personas no participan
    Para i = 1 Hasta 3 Con Paso 1
        cps = 0 //Contador personas sucursal
        Imprimir "--- Sucursal: ", i, " ---"
        Imprimir("Ingrese su nombre (* para terminar): ") Sin Saltar
        Leer nombre
        Mientras nombre <> '*' Hacer
            cps = cps + 1
            Imprimir("¿Participa en la rifa? (s/n): ") Sin Saltar
            Leer respuesta
            Si respuesta == 's'
                cp_si = cp_si + 1
            SiNo
                cp_no = cp_no + 1
            FinSi
            Imprimir("Ingrese su nombre (* para terminar): ") Sin Saltar
            Leer nombre
        FinMientras
        Imprimir "Total empleados sucursal: ", i, ":", cps
        cpe = cpe + cps
    FinPara
    Imprimir "Total empleados empresa: ", cpe
    Imprimir "Total empleados que participan: ", cp_si
    Imprimir "Total empleados que no participan: ", cp_no
FinAlgoritmo
```

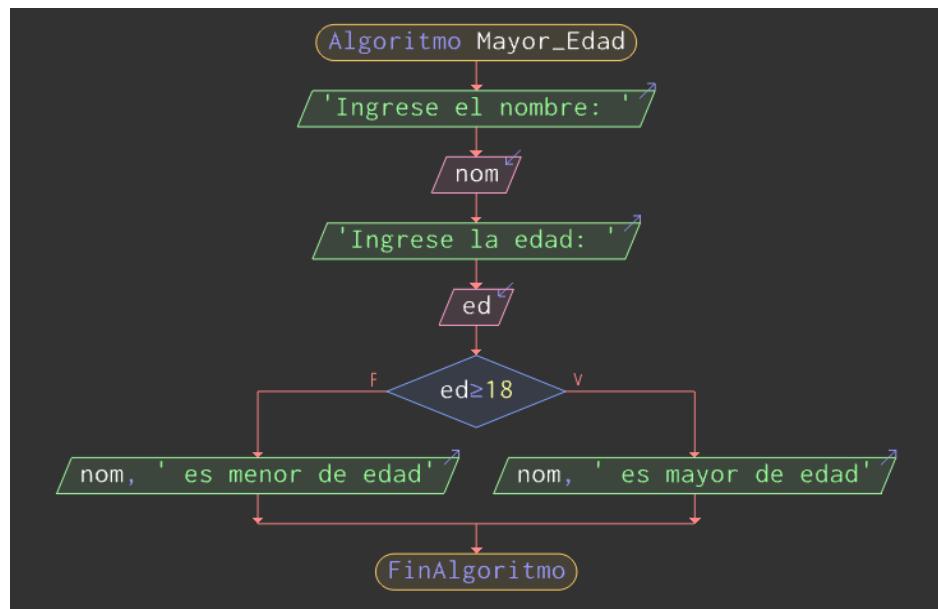
Problemas resueltos

Los siguientes ejercicios de programación resueltos incluyen los temas del manejo de variables, operadores, asignaciones, entrada y salida de información y estructuras de control (ciclos, condicionales, selector múltiple)

Ejercicio 2.1

Ingresar el nombre y edad de una persona y determinar si es mayor o menor de edad.

Diagrama libre:

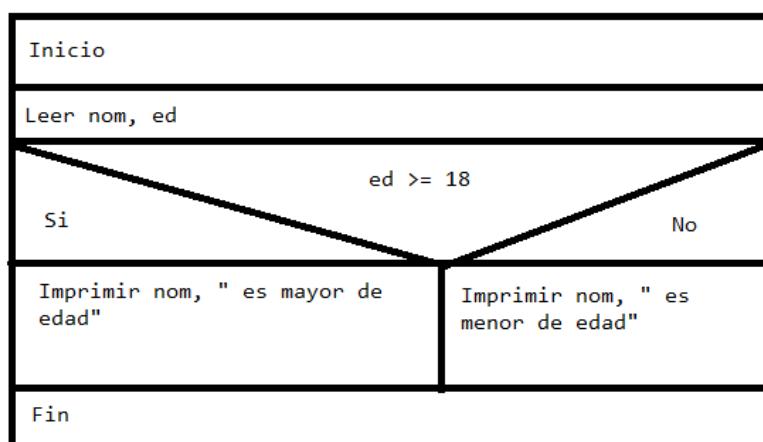


Pseudocódigo:

```

Inicio
Leer nom, ed
Si ed >= 18 Entonces
    Escribir nom, " es mayor de edad"
SiNo
    Escribir nom, " es menor de edad"
FinSi
Fin
    
```

Diagrama Nassi-Schneiderman:



Pseudo programa:

```

Algoritmo Mayor_Edad
    Imprimir "Ingrese el nombre: "
    
```

```

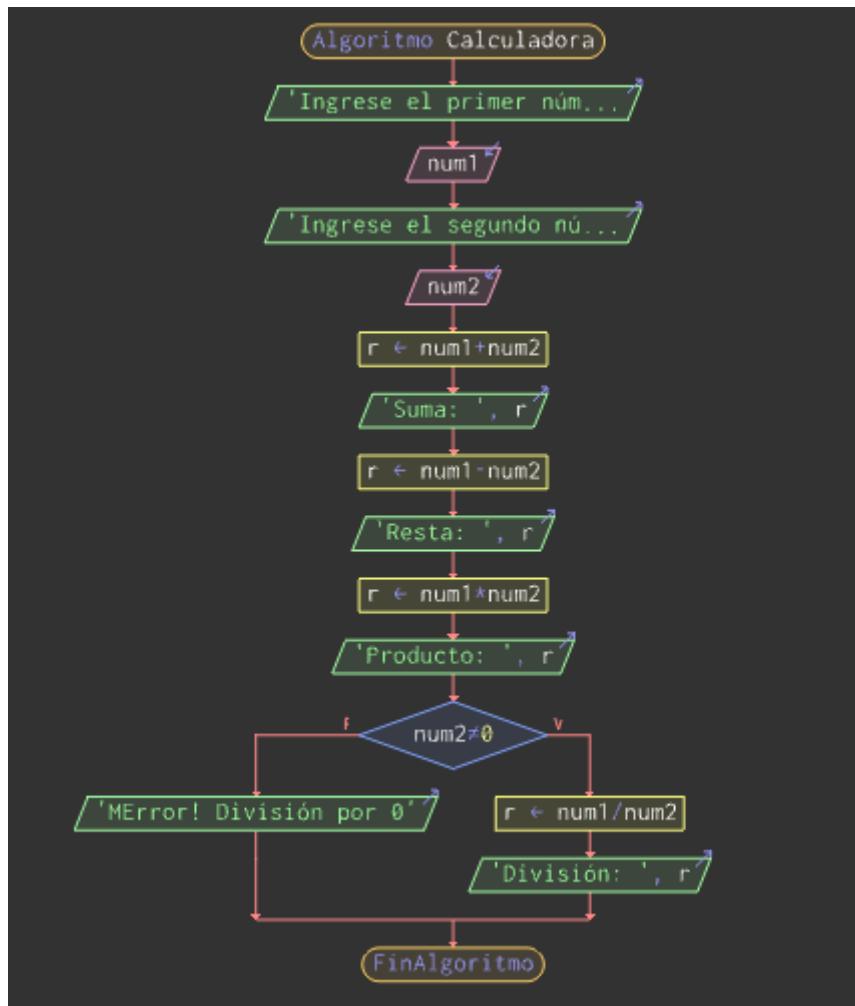
Leer nom
Imprimir "Ingrese la edad: "
Leer ed
Si ed >= 18 Entonces
    Escribir nom, " es mayor de edad"
SiNo
    Escribir nom, " es menor de edad"
Fin Si
FinAlgoritmo

```

Ejercicio 2.2

Crear una calculadora sencilla para realizar las operaciones aritméticas básicas a partir de dos números ingresados por teclado.

Diagrama libre:



Pseudocódigo:

Inicio

```
// Se leen los números num1 y num2
// Las operaciones se almacenan en la variable r (resultado)
Leer num1, num2
r = num1 + num2
Imprimir "Suma: ", r
r = num1 - num2
Imprimir "Resta: ", r
r = num1 * num2
Imprimir "Producto: ", r
Si num2 <> 0 Entonces
    r = num1 / num2
    Imprimir "División: ", r
    r = num1 % num2
    Imprimir "Módulo: ", r
SiNo
    Imprimir "¡Error! División por 0"
FinSi
Si num1 = 0 Y num2 = 0 Entonces
    Imprimir "No se puede efectuar la potencia"
SiNo
    r = num1 ^ num2
    Imprimir "Potencia: ", r
FinSi
Fin
```

Pseudo programa:

Algoritmo Calculadora

```
// Se leen los números num1 y num2
// Las operaciones se almacenan en la variable r (resultado)
Escribir "Ingrese el primer número: "
Leer num1
Escribir "Ingrese el segundo número: "
Leer num2
r = num1 + num2
Imprimir "Suma: ", r
r = num1 - num2
Imprimir "Resta: ", r
r = num1 * num2
Imprimir "Producto: ", r
Si num2 <> 0 Entonces
    r = num1 / num2
    Imprimir "División: ", r
    r = num1 % num2
    Imprimir "Módulo: ", r
SiNo
    Imprimir "¡Error! División por 0"
```

```

FinSi
Si num1 = 0 Y num2 = 0 Entonces
    Imprimir "No se puede efectuar la potencia"
SiNo
    r = num1 ^ num2
    Imprimir "Potencia: ", r
FinSi
FinAlgoritmo

```

Ejercicio 2.3

Lea tres números e imprímalos en orden ascendente. Resuélvalo sin utilizar conectivos lógicos.

Solución

Sean a, b y c los números. Los casos que se pueden dar para la salida en orden ascendente, esto es, las posibles combinaciones, está dada por la fórmula $n!$, donde n es el número de variables que intervienen. Así, tenemos las siguientes posibilidades para a, b y c:

$n! = 3! = 3 * 2 * 1 = 6$ combinaciones posibles

a, b, c
a, c, b
b, a, c
b, c, a
c, a, b
c, b, a

Veamos el comportamiento del algoritmo para los siguientes casos:

a	b	c
5	7	9
5	9	7
5	3	1
5	3	4
5	7	1
5	3	7

Pseudocódigo:

```
Inicio
    Eneros: a, b, c
    Leer a, b, c
    Si a < b Entonces
        Si b < c Entonces
            Imprimir a, b, c
        SiNo
            Si c < a Entonces
                Imprimir c, a, b
            SiNo
                Imprimir a, c, b
            FinSi
        FinSi
    SiNo
        Si c < b Entonces
            Imprimir c, b, a
        SiNo
            Si c < a Entonces
                Imprimir b, c, a
            SiNo
                Imprimir b, a, c
            FinSi
        FinSi
    FinSi
Fin
```

Preguntas

1. ¿Qué entiende por condicional?
2. ¿Qué es un bucle?
3. Plantee procesos cíclicos de la vida cotidiana en palabras, es decir, elabore algoritmos cualitativos que involucren ciclos
4. ¿Qué es un contador y un acumulador, para qué sirven, cómo trabajan?
5. Explique que es un registro centinela
6. ¿Qué es una bandera o suiche?
7. ¿Qué es una estructura “caso”, cómo puede reemplazarse en caso de no contar con ella?
8. ¿Qué es una prueba de escritorio, cuál es su importancia, qué pasos se siguen para realizarla?
9. ¿Qué significa “anidar”? Muestre casos de estructuras que se puedan anidar y cómo se pueden dar esas combinaciones
10. ¿Cuáles son los tipos de ciclos y cuáles son sus diferencias?
11. ¿Qué se entiende cuándo se dice “leer n datos”?

12. ¿Todo ciclo Para puede ser hecho por un ciclo Mientras? ¿Se cumple al contrario?

Ejercicios

1. Leer el nombre y edad de una persona. Si es un adulto mayor, mostrar un mensaje de felicitación por su día
2. Ingrese el documento de identificación y el salario mensual de un trabajador. Si gana el mínimo, calcule un bono del 15% y súmelo al salario
3. Lea una hora e informe si es de día o de noche
4. Ingrese el nombre y edad de una persona para clasificarla en niño, joven, adulto, adulto mayor
5. Dado un número, determine si es par o impar
6. Ingrese un número y determine si es positivo, negativo o cero
7. Se requiere un programa para apoyar las labores de ventas de una tienda. El programa debe permitir ingresar un producto, su precio y la cantidad a llevar. Cada producto tiene un IVA del 19%. El programa debe recibir la también el pago del cliente una vez haya calculado el precio total y entregar devuelta si es del caso. Si el pago no es válido, la venta se debe cancelar
8. Calcular el valor de una llamada con base en los siguientes criterios: si la llamada es nacional, tiene un recargo del 5%; si es internacional, tiene un recargo del 10%; si es local, no tiene recargo. Los datos de entrada son el tipo de llamada, el valor del minuto y el número de minutos
9. Lea tres números e imprímalos en orden descendente. Resuélvalo utilizando conectivos lógicos y sin ellos
10. Ingrese un número y determine a qué día equivale
11. Ingrese un número y determine a qué mes equivale
12. Ingresar un año y determinar si es bisiesto. Un año es bisiesto si es divisible entre 4, salvo que sea año secular -último de cada siglo, terminado en 00-, en cuyo caso también ha de ser divisible entre 400
13. Determine las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$, cuya fórmula general de solución es: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- 14.
15. Determine si un número es par o impar sin utilizar la operación de módulo (*Un par se define como $2k$ y un impar como $2k + 1$, $k \in \mathbb{Z}$*)
16. Leer el nombre y estado civil de M personas. Encuentre cuántas son solteras, cuántas son casadas, cuántas son divorciadas, cuántas son separadas y cuántas viudas. Determine el porcentaje que representan las personas solteras y casadas
17. A un grupo de hombres y mujeres les realizan una prueba de conocimientos. Se desea conocer cuántas mujeres y cuántos hombres presentaron la prueba y el total de personas que se procesaron
18. En una fábrica contratan personal que cumpla con los siguientes requisitos para laborar medio tiempo: Mayor de 18 años y menor de 50, estado civil soltero y que actualmente se encuentre estudiando. A la fábrica se presentaron M aspirantes. Encuentre cuántos cumplen con los requisitos que se piden y qué porcentaje sobre el total de personas representan.

19. Calcular la suma de los números pares e impares del 1 al 100 y comparar cual de las dos sumas es mayor
20. Leer n números desde el teclado y determinar cuántos son positivos, cuántos negativos y cuántos son cero
21. Calcule la suma de los primeros m términos de la serie Armónica²⁶ $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{m}$
22. Leer el nombre y la nota de un grupo de T estudiantes. Encuentre quien sacó la mejor nota y de cuanto fue ésta
23. Leer el nombre y salario de un grupo de trabajadores. Muestre cuál es el empleado de más bajo salario y a cuánto equivale éste
24. Hacer un algoritmo que imprima las tablas de multiplicar
25. Implementar un reloj mostrando las horas, minutos y segundos, así como el día y fecha actual
26. Una empresa con N empleados desea realizar una estadística de su nómina. Para ello ingresa el nombre, número de horas trabajadas (nht), salario básico hora (sbh) y sexo del empleado (a). Se desean conocer los siguientes datos: Salario básico (sb), valor de la retención en salud (4%) y en pensión (3.5%) sobre el básico, total retención, salario neto (sn), empleado con mejor salario y a cuánto equivale su monto, empleado con más bajo salario y a cuánto equivale su monto, promedio de salarios, monto total de salarios pagados en la empresa, porcentaje que representan tanto los hombres como las mujeres en la empresa, porcentaje que representan las personas que ganan más de un millón de pesos, total de salarios por debajo de \$500000
27. Leer un número y determinar si es o no, un número primo. Un número es primo cuando es entero y cuando solo tiene dos (2) divisores exactos: la unidad y el mismo número
28. Hallar el factorial de un número N. El factorial de un número se define para números enteros positivos como: $N! = N * (N - 1) * (N - 2) * \dots * 3 * 2 * 1$, por ejemplo $5! = 5 * 4 * 3 * 2 * 1 = 120$. Por definición $0! = 1$ y $1! = 1$
29. Leer N números. Halle el promedio de los números pares y el promedio de los impares
30. Una empresa tiene 5 sucursales y en cada sucursal hay M empleados. De cada empleado se conoce su cédula y salario. Encuentre: El salario promedio de cada sucursal y de toda la empresa. El empleado que gana mayor salario en cada sucursal y en toda la empresa. El empleado que gana menor salario en cada sucursal y en toda la empresa. El monto pagado por concepto de salarios en cada sucursal y en toda la empresa
31. Leer el nombre y nota final de N estudiantes de un curso y encontrar los siguientes datos: Mejor y peor nota. Promedio general de notas. Cuántos ganan y cuantos pierden y el porcentaje que representan. Cuántos obtuvieron nota de 5. Determinar si algún estudiante es de nombre "Pedro José"
32. Imprimir los términos de la serie de Fibonacci menores o iguales a 10000. La serie de Fibonacci es: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

²⁶ Se llama así porque la longitud de onda de los sucesivos armónicos de una cuerda que vibra es proporcional a la longitud de onda del modo de oscilación fundamental a través de los factores de proporcionalidad dados por los correspondientes términos de la serie: 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7... El primer término representa por tanto al modo fundamental. ([Serie armónica \(matemática\)](#) - [Wikipedia, la enciclopedia libre](#))

33. Leer una cantidad indeterminada de números e informar si se ingresó ordenadamente
34. Determinar los números perfectos entre 1 y 10000. Un número es perfecto si al sumar sus divisores, excepto el mismo número, da como resultado dicho número. Por ejemplo: 6 es perfecto, pues sus divisores (excepto el 6) son: 1, 2 y 3, los cuales suman 6
35. Se tiene el nombre, número de horas diurnas (nhd), número de horas nocturnas (nhn), número de horas festivas (nhf) y salario básico hora de un grupo de empleados. Calcular el salario básico teniendo en cuenta que la hora nocturna tiene un recargo del 35% y la hora festiva un recargo del 75%. Terminar la lectura de datos cuando se ingrese el nombre "****". Informar cuántos empleados se ingresaron y el promedio de salarios básicos que devengan éstos.
36. Convertir un número arábigo entre 1 y 10000 a un número romano.
37. Cree un juego de azar similar al tragamonedas que muestra tres o cuatro imágenes usando caracteres en lugar de imágenes. Determine los premios a entregar según acierte el usuario y dependiendo de la imagen (carácter)
38. Realizar el cálculo de una devuelta. Debe contar con denominaciones de monedas y billetes para especificar cuántos de cada uno debe entregar. Por ejemplo si la devuelta son \$450, podría devolver 2 monedas de \$200 y una de \$50 (implementarlo en el ejercicio de la factura)
39. Mostrar el equivalente de un número decimal en binario, octal y hexadecimal
40. Ingrese un número en binario, octal o hexadecimal y conviértalo a decimal
41. Muestre y calcule las series de las funciones trigonométricas seno y coseno. Realice la aproximación mínimo con 100 datos y luego compare con las funciones correspondientes del lenguaje:

$$\text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots;$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

Unidad 3. Subprogramas: procedimientos y funciones

Funciones incorporadas o predefinidas

Los lenguajes de programación disponen de una serie de funciones para realizar tareas diversas y que tienen como objetivo, además de solucionar ciertos problemas, facilitar a los programadores el tema de construir una funcionalidad. Es así como se dispone de funciones matemáticas, para el tratamiento de cadenas de caracteres, para el manejo de fechas y para otros muchos casos.

Funciones matemáticas

Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la resta o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando funciones incorporadas. Dichas funciones están optimizadas y de no contar con ellas, se haría necesario construirlas para obtener los cálculos deseados, como el coseno trigonométrico, una raíz cuadrada, etc. En algoritmia se pueden definir una gran cantidad de funciones, análogamente a las existentes en los distintos lenguajes de programación y que se asemejan tanto en número como en nombre. Las más comunes se muestran a continuación.

Función	Descripción	Ejemplo
<code>abs(x)</code>	valor absoluto de x	<code>abs(-8) //devuelve 8</code>
<code>trunc(x)</code>	valor truncado de x	<code>trunc(2.6) //devuelve 2</code>
<code>redondear(x)</code>	valor redondeado de x	<code>redondear(2.6) //devuelve 3</code>
<code>rc(x), raiz(x)</code>	raíz cuadrada de x	<code>raiz(9) //devuelve 3</code>
<code>sen(x)</code>	seno trigonométrico en radianes	<code>sen(0) //devuelve 0</code>
<code>cos(x)</code>	coseno trigonométrico en radianes	<code>cos(0) //devuelve 1</code>
<code>tan(x)</code>	tangente trigonométrica en radianes	<code>tan(0) //devuelve 0</code>
<code>asen(x)</code>	arcoseno de x	
<code>acos(x)</code>	arcocoseno de x	
<code>atan(x)</code>	arcotangente de x	

$\ln(x)$	logaritmo natural de x	$\ln(1)$ //devuelve 0
$\exp(x)$	exponencial de x	$\exp(1)$ //devuelve 2.718281... (este es el número E)
$\text{aleatorio}(x, y)$	número aleatorio (al azar) entre x e y	$\text{aleatorio}(1, 20)$
$\pi()$ (como función), PI (como constante)	número pi	$\pi()$ //devuelve 3.14159... PI //devuelve 3.14159...

Ejemplo 3.1

Ilustración de funciones matemáticas “incorporadas” que pueden utilizarse en lógica de programación. Se muestra también el uso en PSeInt donde puede verse algunas pequeñas diferencias, dadas las convenciones que se adopten en algoritmia, así como la sintaxis misma del pseudo intérprete; es tarea del programador revisar en cada lenguaje que trabaje la implementación de estas funciones.

Pseudocódigo:

```

Inicio
Reales xx, yy, zz
Imprimir "Número 1:"
Leer xx
Imprimir "Número 2:"
Leer yy
z = abs(xx)
Imprimir "Valor absoluto de ", xx, ": ", z
z = raiz(z)
Imprimir "Raíz cuadrada (raiz) de ", abs(xx), ": ", z
z = rc(abs(xx))
Imprimir "Raíz cuadrada (rc) de ", abs(xx), ": ", z
z = aleatorio(trunc(xx), trunc(yy))
Imprimir "Aleatorio entre ", trunc(xx), " y ", trunc(yy), ": ", z
z = trunc(yy)
Imprimir "Truncar ", yy, ": ", z
z = redondear(yy)
Imprimir "Redondear ", yy, ": ", z
z = sen(xx)
Imprimir "Seno ", xx, ": ", z
z = cos(xx)
Imprimir "Coseno ", xx, ": ", z
z = tan(xx)
Imprimir "Tangente ", xx, ": ", z
z = exp(xx)
Imprimir "Exponencial ", xx, ": ", z
z = ln(xx)

```

```
Imprimir "Ln ", xx, ": ", z
Imprimir "Pi = ", pi()
Fin
```

Pseudo programa PSeInt:

```
Algoritmo FuncionesMatematicas
    Definir xx, yy, zz Como Real
    Imprimir "Número 1:" Sin Saltar
    Leer xx
    Imprimir "Número 2:" Sin Saltar
    Leer yy
    z = abs(xx)
    Imprimir "Valor absoluto de ", xx, ": ", z
    z = raiz(z)
    Imprimir "Raíz cuadrada (raiz) de ", abs(xx), ": ", z
    z = rc(abs(xx))
    Imprimir "Raíz cuadrada (rc) de ", abs(xx), ": ", z
    z = Aleatorio(trunc(xx), trunc(yy))
    Imprimir "Aleatorio entre ", trunc(xx), " y ", trunc(yy), ": ", z
    z = azar(trunc(yy))
    Imprimir "Aleatorio entre 0 y ", trunc(yy), ": ", z
    z = trunc(yy)
    Imprimir "Truncar ", yy, ": ", z
    z = redon(yy)
    Imprimir "Redondear ", yy, ": ", z
    z = sen(xx)
    Imprimir "Seno ", xx, ": ", z
    z = cos(xx)
    Imprimir "Coseno ", xx, ": ", z
    z = tan(xx)
    Imprimir "Tangente ", xx, ": ", z
    z = exp(xx)
    Imprimir "Exponencial ", xx, ": ", z
    z = ln(xx)
    Imprimir "Ln ", xx, ": ", z
    Imprimir "Pi = ", Pi
FinAlgoritmo
```

Funciones para la manipulación de cadenas de caracteres

Así como existen funciones para el tratamiento matemático, también se dispone de una serie de funciones para la manipulación de cadenas de caracteres. Cada lenguaje de programación ofrece un conjunto de funciones para tal fin, ya que la manipulación de texto es algo común y recurrente en las aplicaciones, y que en los inicios de éstos fue un asunto

que traía bastantes dificultades a los programadores; estas funcionalidades facilitan la labor del desarrollador permitiendo la correcta manipulación de las cadenas de texto.

Nota

Cada carácter de una cadena tiene una posición asociada dentro de ella. Muchos lenguajes acostumbran a tomar la primera posición, esto es, el lugar donde está el primer carácter, como cero, pero también se puede tomar desde uno.

Función	Descripción	Ejemplo
longitud(cadena)	longitud de la cadena	longitud("hola") //devuelve 4
mayusculas(cadena)	convierte cadena mayúsculas a	mayusculas("abc") //devuelve "ABC"
minusculas(cadena)	convierte cadena minúsculas a	minusculas("ABC") //devuelve "abc"
subCadena(cadena, inicio, num_caract)	extrae una cadena de otra comenzando desde <i>inicio</i> , extrayendo <i>num_caract</i> caracteres	subCadena("Hoy es martes", 5, 2) //devuelve "es" (comienza en 1)
concatenar(cad1, cad2, ,,, cadN)	concatena (une) cad1 con cad2, ..., cadN	concatenar("Pedro, ", " ", "Gil") // devuelve "Pedro Gil" (hace lo mismo que "Pedro" + " " + "Gil")

Ejemplo 3.2

Uso de las funciones para la manipulación de cadenas de caracteres. También se presenta el pseudo programa en PSeInt y la forma cómo esta herramienta implementa las funciones.

Pseudocódigo:

```

Inicio
    Caracter texto, z
    Imprimir "Ingrese un texto: "
    Leer texto
    Imprimir "Texto ingresado: ", texto
    Imprimir "Cantidad de caracteres: ", longitud(texto)
    Imprimir "Cadena en mayúscula: ", mayusculas(texto)
    Imprimir "Cadena en mayúscula: ", minusculas(texto)
    z = subCadena(texto, 1, 3)
    Imprimir "Cadena extraída: ", z
    Imprimir "Concatenar cadenas: ", concatenar(texto, z)
Fin

```

Pseudo programa PSeInt:

```

Algoritmo FuncionesCadenas
    Definir texto, z Como Caracter
    Imprimir "Ingrese un texto: " Sin Saltar
    Leer texto
    Imprimir "Texto ingresado: ", texto
    Imprimir "Cantidad de caracteres: ", Longitud(texto)
    Imprimir "Cadena en mayúscula: ", Mayusculas(texto)
    Imprimir "Cadena en mayúscula: ", Minusculas(texto)
    z = Subcadena(texto, 1, 3)
    Imprimir "Cadena extraída: ", z
    Imprimir "Concatenar cadenas", Concatenar(texto, z) //solo 2 cadenas
FinAlgoritmo

```

Funciones para la conversión de tipos de datos

Otro grupo importante de funciones predefinidas en los lenguajes son las que permiten la conversión de un tipo de dato a otro, operación también conocida como **casteo** de variables. El valor devuelto depende de las reglas establecidas en el lenguaje, así como los parámetros que recibe cada función. En nuestro caso, seguiremos las reglas indicadas como se especifica a continuación; para los casos de PSeInt y lenguajes de programación es probable que haya algunas diferencias.

Función	Descripción	Ejemplo
numero(cadena)	longitud de la cadena	numero("15.5") //devuelve 15.5 numero("hola") //devuelve 0
cadena(numero)	convierte cadena a mayúsculas	cadena(2.6) //devuelve "2.6" cadena("pc") //devuelve "pc"
logico(arg)	convierte un argumento tipo cadena o número a booleano	logico(1) //devuelve Verdadero logico("1") //devuelve Verdadero logico(0) //devuelve Falso logico("0") //devuelve Falso logico(55) //devuelve Verdadero

Ejemplo 3.3

Pseudo programa PSeInt:

```

Inicio
Imprimir "Número a cadena: " + cadena(85)
Imprimir "Cadena a número a : ", numero("85")
Fin

```

Pseudo programa PSeInt:

```
Algoritmo sin_titulo
    Imprimir "Número a cadena: " + ConvertirATexto(85)
    Imprimir "Cadena a número a : ", ConvertirANumero("85")
FinAlgoritmo
```

Bifurcación de control

Son sentencias que interrumpen el flujo normal de un programa; aunque en teoría no son necesarias (a excepción de Retornar), diversos lenguajes tienen a disposición varias de ellas para ciertos objetivos. Otras de las instrucciones para bifurcación de control usadas en algunos lenguajes de programación, a saber: *continue* (Continuar), *break* (Interrumpir), *exit* (Salir), *return* (Retornar).

Interrumpir

Permite interrumpir, romper o finalizar la estructura de control donde se encuentre pasando a la instrucción que le sigue. En particular, si la instrucción se encuentra en un ciclo, lo finaliza sin importar la condición de control y continúa con la instrucción siguiente al ciclo.

Ejemplo 3.4

Leer el nombre de un grupo de personas y contar cuantas se registraron; el último nombre en ingresar es un registro centinela con nombre igual a “*”.

El ciclo contendrá una condición que en teoría genera un bucle infinito, pero dentro de éste se encontrará una lectura de datos que llevará en algún momento a un registro centinela permitiendo ejecutar la sentencia break que da por terminado el ciclo.

Pseudocódigo:

```
Inicio
    Enteros: c
    Carácter: nombre
    c = 0
    Mientras Verdadero Hacer
        Imprimir "Ingrese un el nombre (* para terminar): "
        Leer nombre
        Si nombre = "*" Entonces
            Interrumpir
        FinSi
        c = c + 1
    FinMientras
```

```
Imprimir "Total personas: ", c  
Fin
```

Continuar

Utilizada en los ciclos, hace que salte el resto de instrucciones de éste desde donde se encuentra la instrucción, pasando a la siguiente iteración.

Ejemplo 3.5

Imprimir cada carácter de un texto ingresado por teclado, excepto si éste es una “a”.

Podemos usar las funciones de cadenas vistas anteriormente para solucionar este problema.

Pseudocódigo:

```
Inicio  
Caracter cadena, letra  
Enteros: i  
Imprimir "Ingrese un texto: "  
Para i = 1, longitud(cadena), 1  
    letra = subCadena(cadena, i, 1)  
    Si letra = "a" Entonces  
        Continuar  
    FinSi  
    Imprimir letra  
FinPara  
Fin
```

Nota

La diferencia de *Continuar* con *Interrumpir*, es que esta última sentencia rompe el ciclo y sale de éste ignorando las instrucciones que estuvieran pendientes de ejecutar; mientras que la primera también ignora las instrucciones que le siguen, pero dirige el flujo del programa a una nueva iteración del ciclo.

Salir

Esta sentencia de bifurcación de control interrumpe el programa que se esté ejecutando, cancelándolo totalmente e ignorando las instrucciones que se encuentren después de ella.

Ejemplo 3.6

Leer el código de un grupo de empleados que inician labores y contar cuantas llegaron a trabajar; una vez ingresen todos, se ingresa el registro centinela con código igual a “*”.

El ciclo contendrá una condición que en teoría genera un bucle infinito, pero dentro de éste se encontrará una lectura de datos que llevará en algún momento a un registro centinela permitiendo ejecutar la función **exit()** que da por terminado el programa ignorando el código que continúe luego de ella.

Pseudocódigo:

```
Inicio
    Eneros: c
    Carácter: codigo
    c = 0
    Mientras Verdadero Hacer
        Imprimir "Ingrese código (* para terminar): "
        Leer codigo
        Si codigo = "*" Entonces
            Salir
        FinSi
        c = c + 1
    //Observe que la siguiente línea no se ejecuta, diferente a Interrumpir
    //que continúa el flujo del programa luego del ciclo
    Imprimir "Total empleados: ", c
Fin
```

Retornar

Devuelve el control del programa a una *función* o porción de programa que *llama*; a su vez puede devolver un valor a dicha función. Esta instrucción al igual que las anteriores, interrumpe el flujo normal de ejecución, devolviendo el control a otra parte del programa e ignorando las instrucciones que pudieran continuar. En la siguiente subsección se ilustra el uso de esta sentencia muy utilizada en funciones.

Subprogramas o subalgoritmos

Hasta el momento hemos desarrollado algoritmos en un mismo componente de código que comienza con la instrucción *Inicio* y finaliza con la sentencia *Fin*; a dicho código se le conoce comúnmente como **programa principal**.

Hemos visto que si algunas partes del código se requieren ejecutar en varias partes, la única alternativa es duplicar el código que hace la respectiva tarea, lo que hace que la solución sea ineficiente desde varias perspectivas. Por una lado, el duplicar código, aumenta el tamaño de los archivos, por tanto la carga en los servidores, transferencias, discos, etc. Si realizamos una modificación sobre la funcionalidad, debemos buscar las

partes del código donde se encuentra duplicada y realizar lo mismo, lo cual puede traer problemas de olvidos involuntarios y que no se actualicen todas las partes, o que la copia se realice incompleta, entre otros aspectos, haciendo que el mantenimiento sea una tarea complicada y poco confiable. Las aplicaciones de la vida real son desarrollos que involucran grandes cantidades de horas de trabajo y código asociado; a medida que se requieren funcionalidades, el programa va requiriendo de nuevas variables, y al tener todo un código en un mismo componente, su revisión y manejo de los datos se hace más complejo, pudiendo llevar a errores en cálculos, operaciones, asignaciones, etc. La escalabilidad de los programas es otro problema al que se ve enfrentado un desarrollo de software de este tipo, ya que al tener un mantenimiento complejo, poca fiabilidad en las actualizaciones y cambios, a medida que el código crezca, el problema va a aumentar hasta hacerse inmanejable.

Una solución que propone la algoritmia a los problemas mencionados, es mediante el paradigma de la **programación modular**, el cual consiste en *dividir*²⁷ un programa en partes más pequeñas llamadas **módulos (subprogramas)**. Este paradigma introduce conceptos como la *legibilidad* y *manejabilidad* del software, esto es, programas más legibles y manejables para los programadores, lo que a su vez permite mejorar la eficiencia de los algoritmos. Cada parte en que se divide el programa se encarga de una tarea específica que puede ser utilizada por otras partes del programa.

Un **subprograma** o **subalgoritmo** es un programa “más pequeño” que se encarga de una tarea específica y que está en el contexto de un programa principal. Esto significa que, aunque un subprograma es también es un programa, tiene varias características que los diferencian:

- Generalmente dependen de un programa principal, aunque también pueden hacer parte de librerías y ser utilizados por diversos “programas principales”. Sin embargo, esto no quita su dependencia para ser utilizado.
- No se ejecutan por sí solos, deben ser invocados o llamados desde un programa principal que lo inicia.
- Generalmente se escriben antes del programa principal, aunque también pueden estar en archivos independientes.
- Tiene un encabezado propio indicando que es un subprograma y puede tener parámetros que ingresan o sacan información de éste.
- Pueden tener un único valor de retorno.
- Cuando el subprograma termina su ejecución, el flujo del programa regresa al punto desde donde se realizó la llamada a éste o a la instrucción siguiente.

Los subprogramas permiten escribir programas más legibles y manejables, permitiendo que su mantenimiento sea más fácil y que los programas sean escalables; permiten además la creación de *librerías*, que son bibliotecas enteras con distintas funcionalidades que pueden ser usadas por distintos programas. Este paradigma también busca hacer que los programas sean “más simples”, de tal forma que se puedan escribir componentes

²⁷ Es común que distintos autores citen la conocida frase “divide y vencerás” en la explicación del tema de subprogramas dadas las ventajas que trae la programación modular en la creación de aplicaciones.

independientes que se encarguen solo de realizar lo que deben hacer y reduciendo cada funcionalidad a la más mínima expresión posible.

A continuación, vamos a tratar los dos tipos de subalgoritmos que pueden utilizarse en programación.

Procedimientos y Funciones

Básicamente, los subprogramas se clasifican en dos tipos: **procedimientos y funciones**. Aunque su estructura y funcionamiento es muy similar, tienen algunas características que los diferencian. Aunque es de anotar que algunos lenguajes solo implementan el concepto de *función*, sin embargo, esto no entra en contradicción con el paradigma de la programación modular, ya que éstos a su vez ofrecen métodos de simular una *función* como un *procedimiento*. Veamos más en detalle cada tipo de subprograma.

Funciones

Matemáticamente, se define una función como una relación especial entre dos conjuntos, en la que a cada elemento del primer conjunto le corresponde solo un elemento del segundo conjunto. Una función se encarga de transformar mediante una regla, un valor en otro. Hay muchas funciones usadas en matemáticas para realizar distintas operaciones, por ejemplo: $\tan(x)$, $\sen(x)$, $\cos(x)$, $\abs(x)$, etc., que además se encuentran disponibles no solo en calculadoras científicas, sino también en los lenguajes de programación. Es así como $\tan(30^\circ) = 0.5$, $\sen(-30^\circ) = 0.5$, $\cos(60^\circ) = 0.5$, $\abs(-9.6) = 9.6$, etc. En estas funciones podemos ver un **argumento (parámetro)** que recibe la función y que luego lo transforma, según la regla como opere, en otro valor. Por ejemplo, la función *valor absoluto* (*abs*) tiene como fin, a grandes rasgos, convertir un número a positivo, es por ello que al recibir el argumento -9.6, la función devuelve 9.6.

Un subprograma tipo función, también conocida como **función definida por el usuario**, de manera análoga a las funciones matemáticas, reciben unos argumentos (aunque puede recibir cero parámetros) y devuelve un único valor, el cual es hallado de alguna forma según la tarea (regla) que realice el algoritmo.

Teniendo en cuenta lo comentado anteriormente, podemos declarar el prototipo de una función como se muestra a continuación.

Sintaxis

```
Funcion nombre_funcion([parámetros formales])[: tipo_funcion]
    Cuerpo de La función
    Retornar valor_retorno
FinFuncion
```

Donde:

- nombre_función: es el identificador único de la función, cuyo nombre sigue las mismas reglas que para el nombre de variables
- parámetros formales: lista de *argumentos de entrada* para la función y que hacen parte de la definición de ésta. Cada parámetro es separado por comas y especificado con su respectivo tipo de dato
- tipo_funcion: opcional; indica tipo del valor devuelto por la función
- cuerpo de la función: instrucciones válidas algorítmicas que requiera ejecutar la función
- valor_retorno: variable o expresión a retornar por la función. Su tipo de dato debe coincidir con el tipo de la función

Nota: Instrucción Retornar

Como vimos arriba, la sentencia **Retornar** es una instrucción de *bifurcación de control* presente en todos los lenguajes, muy usada en subprogramas, particularmente en funciones, y que se encarga de devolver el control del flujo del programa al punto donde se hizo la llamada a la función. La sentencia Retornar no necesariamente se ubica al final de la función, puede estar en otros puntos que se consideren apropiados. Una vez la sentencia se ejecute, el control retorna al programa que realizó la llamada y el código que haya por debajo de dicha sentencia no se ejecutará.

Invocar (llamar) una función

Una función devuelve un único valor, por tanto puede ser llamada asignándola a una variable o incluyéndola en una operación o salida de datos.

La invocación de una función se realiza generalmente por fuera de ella, ya sea desde un “programa principal” u otro subprograma. Si dicho llamado se hace desde la misma función, se dice que es una **función recursiva**. Diversos casos matemáticos se dan como procesos recursivos que pueden ser representados algorítmicamente y llevados a un lenguaje de programación; sin embargo, debe tenerse especial cuidado en la implementación de funciones recursivas, ya que implica una carga mayor en la *pila* de la memoria que puede agotarla, además que conlleva mayor complejidad de desarrollo dada la simplicidad de la solución que ofrece. En este texto no se aborda la *recursividad*, tema que se desarrolla en Estructuras de Datos, donde se tratan problemas que implican obligatoriamente el uso de esta técnica, como en el caso de los algoritmos para *árboles*.

Sintaxis

```
variable = nombre_funcion([parámetros actuales])
```

Ejemplo 3.7

Crear una función para calcular el módulo de una división entera. La función recibe dos parámetros y devuelve el residuo de la división entre estos.

Pseudocódigo:

```
// Definición del prototipo de la función módulo()
// para devolver el residuo de una división entera
// con parámetros formales x, y
Funcion modulo(Enteros x, Enteros y): Enteros
    Enteros: z
    z <- x % y
    Retornar z
FinFuncion

// Programa principal
Inicio
Enteros: a, b, c
Leer a, b
Si y <> 0 Entonces
    //Invocación de función con parámetros actuales a, b
    c <- modulo(a, b)
    Imprimir "Módulo: ", c
SiNo
    Imprimir "No se puede dividir por 0"
FinSi
Fin
```

Parámetros de un subprograma. Parámetros actuales y formales

Cuando hablamos de argumentos o parámetros en un subprograma, debemos tener en cuenta las siguientes cuestiones:

- Al invocar un subprograma, hablamos de sus argumentos como **parámetros actuales**, mientras que en el prototipo de la función los llamamos **parámetros formales**.
- Los parámetros actuales y formales de una función deben coincidir en tipo, número, pero no necesariamente en nombre
- Por definición, los parámetros de una función son de **entrada** o pasados por **valor**.
- En un procedimiento los parámetros pueden ser pasados por *valor* (**entrada**) o por **referencia** o **dirección**, lo cual significa que también pueden ser de **salida** o **entrada/salida**, esto es, el subprograma puede cambiar los valores originales de los argumentos
- Un **parámetro opcional o por defecto**, es aquel que puede o no, usarse desde la invocación, esto es, especificarse de manera opcional, por lo que en el prototipo del subprograma debe inicializarse con un valor por defecto; se recomienda que estos argumentos estén al final de la lista de parámetros

Ejemplo 3.8

Crear un programa que reciba dos edades y dos salarios. Se pide calcular el promedio de edades y salarios. Para esto, debe crear una función que calcule los promedios.

Este programa aprovecha una función que evita duplicar código para realizar una operación común: calcular el promedio de dos valores numéricos.

Pseudocódigo:

```
// Definición del prototipo de la función promedio()
// para devolver el promedio de dos valores numéricos
Funcion promedio(Reales x, Reales y): Reales
    Reales: prom
    prom = (x + y) / 2
    Retornar prom
FinFuncion

// Programa principal
Inicio
Reales: ed1, ed2, sal1, sal2, prom
Leer ed1, ed2, sal1, sal2
prom = promedio(ed1, ed2)
Imprimir "Promedio edad: ", prom
prom = promedio(sal1, sal2)
Imprimir "Promedio salario: ", prom
Fin
```

Procedimientos

En muchos casos necesitamos escribir subprogramas que no devuelvan ningún valor, o que por el contrario, puedan devolver muchos. Para este tipo de situaciones, las funciones se quedan cortas y no permiten dar una solución al problema. Para ello, existen los **procedimientos** (aunque no presentes en todos los lenguajes), otra clase de subprograma que podría verse como el tipo general, siendo las funciones el caso especial de subprogramas, esto es, una función es un tipo especial de procedimiento en el sentido que toda función puede ser escrita con un procedimiento, pero al contrario no es posible. Como se mencionó antes, la estructura de las funciones y procedimientos es muy similar, pero estos últimos no tendrán una sentencia *Retornar*.

Sintaxis

```
Procedimiento nombre_procedimiento([parámetros formales])
    Cuerpo del procedimiento
FinProcedimiento
```

Ejemplo 3.9

Parámetros formales por valor y referencia en un prototipo de procedimiento

Pseudocódigo:

```
//Procedimiento Proc1 con 4 parámetros
Procedimiento Proc1(Caracter dato, Enteros a, Enteros Val b, Reales Ref x)
    //Parámetros dato, a, b pasados por valor (de entrada)
    //Si no se indica Val, por defecto el parámetro es por valor
    //Parámetro x pasado por referencia (entrada/salida)
    Imprimir dato, a, b
    a = a + b //valor de a cambia solo localmente
    Imprimir a
    //x puede tener un valor de entrada y el subprograma cambia su valor
    //El programa que llama puede usar el valor de x
    x = a * b / 5
    Imprimir x
FinProcedimiento

//Programa principal que llama al procedimiento Proc1()
Inicio
Caracter: nombre
Enteros: num1, num2
Reales: numero
Leer nombre, num1, num2
Imprimir nombre, num1, num2
Proc1(nombre, num1, num2, numero)
Imprimir num1, numero
Fin
```

Invocar (llamar) un procedimiento

Estrictamente hablando, un procedimiento no devuelve valores, aunque puede modificar los parámetros si éstos son pasados por referencia o dirección, lo que nos lleva a decir de manera informal que un procedimiento puede devolver cero, uno o más valores. Como no sabemos si el procedimiento devuelve valores o no, no lo podemos invocar como una función, pero la forma es un poco similar, solo que no lo asignamos a una variable o la incluimos dentro de una expresión.

Sintaxis

nombre_procedimiento([parámetros actuales])

Ámbito de las variables. Variables globales y locales

Decimos que las variables que están declaradas en un subprograma, incluyendo sus parámetros, tienen un **ámbito local**; esto quiere decir que dichas variables sólo existen dentro del subprograma y mientras éste se esté ejecutando, una vez termine de correr, estas variables desaparecerán de la memoria. Esto significa que podemos tener nombres de variables con el mismo nombre definidas en subprogramas diferentes sin que entren en conflicto, ya que los lenguajes serán capaces de reconocer el contexto del nombre del campo que se esté utilizando. Las variables definidas en los subprogramas, incluyendo los argumentos formales, son **variables locales** y solo pueden ser usadas dentro de éstos.

Una **variable global** es aquella que se define dentro del programa principal y que por tanto podrá ser accedida desde cualquier parte o subprograma perteneciente al programa. Decimos por tanto que estas variables tienen un **ámbito global**.

Notas

- En el paradigma de la Programación Modular es común hablar del ámbito de las variables, en particular de las variables globales, sin embargo, esto pierde peso al carecer de sentido en el paradigma de la Programación Orientada a Objetos (POO) con respecto al ámbito global, quién aprovecha la mayoría de conceptos de la programación modular, añadiendo avances significativos a la teoría algorítmica.
- Un parámetro pasado por valor puede ser modificado dentro de un subprograma, pero el cambio no se verá reflejado a nivel global
- Se debe tener especial cuidado con las variables globales, ya que pueden ser modificadas en cualquier parte del programa incluyendo los subprogramas y se pueden dar cambios accidentales
- Un nombre de variable local igual al de una variable global, puede traer resultados inesperados. A medida que los programas crecen, también el número de variables, por lo que hay que tener cuidado en el uso de las variables globales en los lenguajes que las permiten
- Las variables globales se mantienen durante toda la ejecución del programa, por lo que estarán consumiendo recursos, otro aspecto a tener presente a la hora de definir variables de ámbito global

Ejemplo 3.10

Crear subprogramas para:

- Sumar dos enteros
- Duplicar dos números en procedimientos con argumentos pasados por valor y referencia
- Imprimir información

Pseudocódigo:

```
// Función sumaNumeros para sumar dos números enteros
Funcion sumaNumeros(Enteros x, Enteros z)
    Enteros: sum
```

```

        sum <- x + z
        Retornar sum
FinFuncion

// Procedimiento imprimirInfo para mostrar un dato. No devuelve valores
Procedimiento imprimirInfo(Caracter dato)
    Imprimir dato
FinProcedimiento

// Procedimiento duplicar; recibe un parámetro por valor
// y otro por referencia
Procedimiento duplicar(Enteros num1, Enteros Ref num2)
    num1 <- num1 * 2
    num2 <- num2 * 2
    Imprimir "Número 1 duplicado: ", num1
    Imprimir "Número 2 duplicado: ", num2
FinProcedimiento

// Programa principal desde donde se llaman los subprogramas
Inicio
Enteros: n1, n2, s
Imprimir "Ingrese número 1: "
Leer n1
Imprimir "Ingrese número 2: "
Leer n2
imprimirInfo("Número 1: " + ConvertirATexto(n1))
imprimirInfo("Número 2: " + ConvertirATexto(n2))
s <- sumaNumeros(n1, n2)
imprimirInfo("Resultado suma: " + ConvertirATexto(s))
duplicar(n1, n2)
imprimirInfo("Número 1: " + ConvertirATexto(n1))
imprimirInfo("Número 2: " + ConvertirATexto(n2))
Fin

```

Pseudo Programa PSeInt:

```

// Función sumaNumeros para sumar dos números enteros
Funcion sum <- sumaNumeros(Enteros x, Enteros z)
    sum <- x + z
Fin Funcion

// Procedimiento imprimirInfo para mostrar un dato. No devuelve valores
SubAlgoritmo imprimirInfo(Carácter dato)
    Imprimir dato
FinSubAlgoritmo

```

```
// Procedimiento duplicar; recibe un parámetro por valor y otro por
referencia
SubAlgoritmo duplicar(Enteros num1 Por Valor, Enteros num2 Por Referencia)
    num1 <- num1 * 2
    num2 <- num2 * 2
    Imprimir "Número 1 duplicado: ", num1
    Imprimir "Número 2 duplicado: ", num2
FinSubAlgoritmo

// Programa principal desde donde se llaman los subprogramas
Algoritmo ProgramaPrincipal
    Definir a, b Como Caracter
    Definir n1, n2, s Como Entero
    Imprimir "Ingrese a: " Sin Saltar
    Leer a
    si a no es numero Entonces
        a <- "0"
    FinSi
    n1 <- ConvertirANumero(a)
    Imprimir "Ingrese b: " Sin Saltar
    Leer b
    si b no es numero Entonces
        b <- "0"
    FinSi
    n2 <- ConvertirANumero(b)
    s <- sumaNumeros(n1, n2)
    imprimirInfo("Resultado suma: " + ConvertirATexto(s))
    duplicar(n1, n2)
    imprimirInfo("Valor de a: " + ConvertirATexto(n1))
    imprimirInfo("Valor de b: " + ConvertirATexto(n2))
FinAlgoritmo
```

Preguntas

Ejercicios

1. Cree un procedimiento para intercambiar el valor de dos variables. Las variables deben ser ingresadas desde un procedimiento. Mostrar los valores antes y después del intercambio en otro procedimiento.
2. Se tiene un grupo de estudiantes del TdeA, de los cuales se lee la nota del parcial y del final, así como el código de éstos. El último código que se lee es un registro

centinela con código igual a “**”. Cree un subprograma para hallar el promedio de notas obtenido en el parcial, en el final y general.

3. Ingrese un número entero por teclado. Cree una función que determine si dicho número pertenece a la serie de Fibonacci
4. Cree un programa que solicite cuatro datos numéricos para armar una dato para fechas: día de la semana entre 1 y 7 (el domingo es el día 1, el lunes el 2, etc.), día del mes entre 1 y 31, número del mes entre 1 y 12 (el 1 es el mes enero, el 2 febrero, etc.) y el año. Cree una función que devuelva una fecha en un formato similar a este: Jueves, 26 de Octubre de 2023, si los datos son: 5, 26, 10, 2023
5. Dado un valor real entre 0 y 2π , cree dos funciones para calcular el seno y el coseno trigonométricos en radianes usando la serie de Taylor para 100 términos:

$$\text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots; \cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

6. Calcule el valor de π (pi) usando la **serie de Leibniz**²⁸ para un n grande: $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots + 1/n$
7. Crear un programa para conversión de coordenadas. Debe tener un menú para especificar el tipo de conversión y los datos a ingresar, así como un procedimiento que convierta coordenadas polares (r , θ) a coordenadas cartesianas: $x = r\cos(\theta)$, $y = rsen(\theta)$, y otro procedimiento que convierta de coordenadas cartesianas a polares: $r = \sqrt{x^2 + y^2}$; $\theta = \tan^{-1} \frac{y}{x}$
8. Leer el nombre y salario básico de n empleados. Cree una función que calcule el aporte a salud (4%) y a pensión (4%) y un procedimiento para mostrar el salario neto
9. Dados dos números enteros positivos; cree una función para multiplicarlos como una suma sucesiva. Ejemplo: $2 * 3 = 2 + 2 + 2 = 3 + 3$.
10. Dados dos números enteros positivos; cree una función para elevar el primero al segundo como una multiplicación sucesiva. Ejemplo: $2 ^ 3 = 2 * 2 * 2$.
11. Se tiene un grupo de empleados en una empresa, de los cuales se lee el salario y el código. Cree un subprograma para incrementar el salario de cada empleado en un 10%. El último código que se lee es un registro centinela con código igual a “**”.
12. Ingrese un texto por teclado. Cree un subprograma para determinar cuántas vocales se encuentran en éste
13. Ingrese un texto y una letra. Cree un subprograma para encontrar cuántas veces está dicha letra en el texto
14. Ingrese un texto. Cree un subprograma que indique cuántas palabras conforman el texto
15. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena separada cada carácter por un espacio en blanco. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “h o y e s j u e v e s”
16. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena sin espacios en blanco. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “hoyesjueves”
17. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena con los caracteres intercambiados entre minúscula y mayúscula. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “HoY Es jUeVeS”

²⁸ Nombrada así en honor a Gottfried Wilhelm Leibniz, filósofo y matemático alemán (1646 - 1716). Ver más en [Serie de Leibniz - Wikipedia, la enciclopedia libre](#)

18. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena en letra capital. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “Hoy Es Jueves”
19. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena en orden inverso. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “seveuj se yoh”
20. Ingrese el nombre de una persona. Utilice una función para validar que no ingresen números
21. Dadas dos cadenas de caracteres, use un subalgoritmo que realice lo siguiente: si la primera cadena tiene un número impar de caracteres, concatene la segunda cadena al inicio de la primera; en caso contrario, concaténela al final.
22. Dada una cantidad numérica, use funciones para convertirla a letras
23. Dado un número entero positivo, use funciones para convertirlo a número romano
24. Dado un número, utilice subprogramas para determinar si es un número capicúa (aquelllos que se leen igual de ambos sentidos: ejemplo: 12321)
25. Dada una frase, cree subprogramas para determinar si es un palíndromo o no (aquellas que se leen igual de ambos sentidos: ejemplo: amad a la dama)
26. Dado un valor real x , cree una función para calcular su exponencial usando la serie de Taylor para 100 términos: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$
27. La función lineal está expresada por $y = mx + b$, con m , b números reales. Cree un programa usando funciones para generar una tabla de valores con x especificada por el usuario y/o generada por el sistema. Grafique esta función por pantalla.

Unidad 4. Arreglos e Introducción a la Programación Orientada a Objetos (POO)

Estructuras de datos

Son colecciones de datos que pueden ser categorizadas por su organización y las operaciones que se pueden hacer sobre ellas. Podemos decir que una **estructura de datos** es un **tipo de dato complejo** capaz de almacenar más de un dato al tiempo, a diferencia de los tipos datos simples o primitivos, los cuales soportan un solo dato a la vez.

Ya vimos los tipos de datos primitivos: numéricos (real, entero), lógico y carácter. En los **tipos estructurados**, tenemos varios tipos en dos categorías:

Estáticas

- Arreglos: vectores, matrices, arreglos multidimensionales
- Registros
- Archivos
- Conjuntos
- Cadenas
- Pilas
- Colas

Dinámicas

- Pilas
- Colas
- Listas ligadas
- Árboles y grafos

Un tipo de dato simple o primitivo significa que no está compuesto de otros tipos o estructuras de datos, mientras que un tipo de dato compuesto están basados en los tipos de datos primitivos, esto es, se forman a partir de los datos simples.

Una **estructura de datos estática** es aquella que reserva un espacio fijo de memoria al declararse; sin embargo, los lenguajes modernos permiten un manejo dinámico de la memoria para los arreglos y otras estructuras que se han considerado estáticas.

Una **estructura de datos dinámica** no reserva un espacio determinado, sino que se basa en solicitar memoria a medida que lo requiera; para ello se apoya en el uso de **punteros**, un tipo especial de tipo de dato presente en algunos lenguajes con los cuales se puede acceder a las direcciones de memoria.

En este curso sólo se abordarán las estructuras de datos tipo arreglos y los registros se verán desde el enfoque de la Programación Orientada a Objetos (POO). Las demás serán tratadas en el curso de Estructuras de Datos, donde se retoma nuevamente el tema de arreglos.

Arreglos

Un **arreglo (array)** es un conjunto finito y ordenado de elementos homogéneos, esto significa que cada elemento puede ser identificado y que la información es del mismo tipo, aunque algunos de los nuevos lenguajes permiten tener arreglos con información heterogénea.

En los lenguajes de programación existen **estructuras de datos** especiales que nos sirven para guardar información más compleja que simples variables. Una estructura típica en todos los lenguajes son los **arreglos**, y que es generalmente la primera estructura de datos en materia de estudio. Existen varios tipos de arreglos: **unidimensionales** (vectores), **bidimensionales** (matrices) y **n-dimensionales** ($n > 2$). Sin embargo, su implementación depende del lenguaje, pues aunque teóricamente ya hay un amplio desarrollo al respecto, no todas estas “máquinas” implementan un uso extendido en su manejo.

Arreglos unidimensionales: Vectores

Los **vectores** permiten almacenar varios valores del mismo tipo (*información homogénea*, aunque los lenguajes modernos también permiten tener vectores que guardan *datos heterogéneos*) utilizando un mismo **nombre de variable** e identificando cada elemento con un **índice** que representa la **posición** de éste en el *vector*, el cual va desde **uno** hasta el **total de elementos -tamaño-** (algunos lenguajes toman la primera posición como cero y la última como el total de elementos del arreglo menos uno).

Matemáticamente, un vector se representa con sus elementos separados por comas, ya sea entre corchetes o entre paréntesis:

```
vec1 = [2, 3, 4, -5, 0]
```

```
vec2 = (b, a, d, z)
```

Un vector está compuesto de una serie de espacios consecutivos de memoria a los que se accede por medio de un nombre y un índice entero y que puede representarse gráficamente.

Representación en memoria

A nivel informático, los arreglos de una, dos y tres dimensiones se pueden representar gráficamente. Un vector se representa como una secuencia de cajones consecutivos, cada uno asociado a un índice y al nombre del arreglo.

vec1 (n = 5)

2	3	4	-5	0
---	---	---	----	---

vec2 (n = 4)

b
a
d
z

Figura 4.1. Representación gráfica de arreglos unidimensionales: vector fila y vector columna, respectivamente

Declaración de vectores

Los vectores son arreglos de una dimensión, por tanto usamos un valor para especificar el tamaño de éste, es decir, la cantidad de espacios de memoria que contendrá. Para esto indicamos el tipo de dato seguido del nombre del vector, y seguido de éste y entre corchetes, el tamaño que tendrá, esto es, la longitud o número de posiciones.

Sintaxis

tipo_de_dato: *nombre_vector*[*tamaño*]

Donde *tamaño* es un valor entero que define la longitud del vector. Debe tenerse en cuenta las limitantes de cada lenguaje de programación a la hora de asignar dicho valor.

Acceso a los elementos de un vector

Los arreglos se acceden elemento por elemento, esto significa que debemos especificar el nombre de éste y las dimensiones respectivas. Para el caso de un vector, indicamos el nombre del vector y seguido de éste y entre corchetes, la posición (índice) a la que queremos acceder, ya sea para guardar un dato allí, mostrarlo o usarlo en una operación.

Sintaxis

nombre_vector[*posición*]

Dónde *posición* es un número entero entre 1 y el total de elementos del vector.

Ejemplo 4.1

Crear un vector de 5 posiciones. Agregar dos elementos en las dos primeras posiciones y luego sume estos valores y guárdalos en la tercera posición. Imprima estas posiciones del vector.

Pseudocódigo:

```
Inicio
    Enteros: vector[5]
    Leer vector[1]
    vector[2] = 4
    vector[3] = vector[1] + vector[2]
    Imprimir vector[1], vector[2], vector[3]
Fin
```

Ejemplo 4.2

Crear un vector de *t* elementos de máximo de 30 posiciones con datos aleatorios entre 1 y 50, mostrar sus datos, hallar la suma y el mayor de éstos.

Para hallar el mayor o el menor en un vector, partimos del supuesto que el primer elemento es el que cumple la condición y a partir del segundo comenzamos a comparar.

Se presenta la solución en pseudocódigo y PSeInt.

Pseudocódigo:

```
Funcion mayorDatoVector(Enteros: vec, Enteros: n): Enteros
    Enteros: i, mayor
    mayor = vec[1] // Supuesto: el mayor dato está en la posición 1
    Para i = 2 Hasta n Con Paso 1 Hacer
        Si vec[i] > mayor Entonces
            mayor = vec[i]
        FinSi
    FinPara
    Retornar mayor
FinFuncion

Funcion sumaVector(vec, n)
    Enteros: i, s
    s = 0
    Para i = 1, n, 1
        s = s + vec[i]
```

```
FinPara
    Retornar s
FinFuncion

Procedimiento llenarVector(vec, n)
    Para i = 1 Hasta n Hacer
        vec[i] = Aleatorio(1, 50)
    FinPara
FinProcedimiento

Procedimiento mostrarVector(vec, n)
    Enteros: i
    Para i = 1 Hasta n Hacer
        Imprimir vec[i]
    FinPara
FinProcedimiento

Inicio
Enteros: V[30], t, i
Leer t
llenarVector(V, t)
mostrarVector(V, t)
Imprimir "Suma vector: ", sumaVector(V, t)
Imprimir "Mayor dato vector: ", mayorDatoVector(V, t)
Fin
```

Pseudo Programa PSeInt:

```
Funcion mayor = mayorDatoVector(vec, n)
    Definir i, mayor Como Entero
    mayor = vec[1] // Supuesto: el mayor dato está en la posición 1
    Para i = 2 Hasta n Con Paso 1 Hacer
        Si vec[i] > mayor Entonces
            mayor = vec[i]
        FinSi
    FinPara
Fin Funcion

Funcion s = sumaVector(vec, n)
    Definir i, s Como Entero
    s = 0
    Para i = 1 Hasta n Con Paso 1 Hacer
        s = s + vec[i]
```

```
FinPara
Fin Funcion

SubAlgoritmo llenarVector(vec, n)
    Para i = 1 Hasta n Hacer
        vec[i] = Aleatorio(1, 50)
    FinPara
FinSubAlgoritmo

SubAlgoritmo mostrarVector(vec, n)
    Definir i Como Entero
    Para i = 1 Hasta n Hacer
        Imprimir vec[i], "    " Sin Saltar
    FinPara
FinSubAlgoritmo

Algoritmo Vectores
    Dimension V[30]
    Definir V, t, i Como Entero
    Imprimir "Total elementos vector: " Sin Saltar
    Leer t
    llenarVector(V, t)
    mostrarVector(V, t)
    Imprimir ""
    Imprimir "Suma vector: ", sumaVector(V, t)
    Imprimir "Mayor dato vector: ", mayorDatoVector(V, t)
FinAlgoritmo
```

Arreglos bidimensionales: Matrices o tablas

Una matriz es una colección de elementos dispuestos en filas (horizontales) y columnas (verticales), cada una etiquetada con un número entero que indica su número, lo cual significa que para hacer referencia a un elemento, debemos especificar dos índices. Tanto los vectores como las matrices son materia de estudio muy utilizados en matemáticas en distintas líneas como el Álgebra Lineal y en otras áreas de las ciencias naturales y aplicadas.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Figura 4.2. Representación gráfica de un arreglo bidimensional: matriz $A_{m \times n}$.²⁹

Representación en memoria

La representación en memoria de una matriz es a manera de tabla. La siguiente figura muestra un ejemplo de representación de matriz a nivel informático.

mat _{5x3}		
2	1	0
99	10	-1
6	-8	11
22	16	2
65	-31	47

Figura 4.3. Representación gráfica de un arreglo bidimensional en memoria

Nota

Observe que un vector es una matriz de $1 \times n$ ó $n \times 1$ elementos, es por ello que también se habla de **vector fila** o **vector columna** para referirse a estos tipos especiales de matrices.

Declaración de matrices

Las matrices son arreglos de dos dimensiones compuestos por filas y columnas, por tanto usamos dos valores para especificar el total de filas y de columnas, respectivamente, separados por comas.

Sintaxis

```
tipo_de_dato: nombre_matriz[total_filas, total_columnas]
```

²⁹ Imagen tomada de: [Matriz \(matemática\) - Wikipedia, la enciclopedia libre](#)

Nota

Observe que el tamaño de una matriz es igual al número de filas multiplicado por el número de columnas.

Acceso a los elementos de una matriz

Al tener dos dimensiones, utilizamos dos índices (posiciones) para acceder a un elemento de una matriz. El primer índice hace referencia a la fila, y el segundo a la columna.

Sintaxis

```
nombre_matriz[número_fila, número_columna]
```

Ejemplo 4.3

Crear una matriz de orden 3x3. Agregar algunos elementos y realizar algunas operaciones.

Pseudocódigo:

```
Inicio
    Enteros: matriz[3, 3]
    Leer matriz[1, 1]
    matriz[1, 3] = 4
    matriz[2, 3] = matriz[1, 1] + matriz[1, 3]
    Imprimir matriz[1, 1], matriz[1, 3], matriz[2, 3]
```

Ejemplo 4.4

Crear una matriz con tamaño máximo de 30 filas y 30 columnas con datos aleatorios entre 1 y 50 y mostrarla en forma de tabla.

Se presenta la solución en pseudocódigo y PSeInt.

Pseudocódigo:

```
Procedimiento mostrarMatriz(mat, m, n)
    Entero: i, j
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Imprimir mat[i, j]
        FinPara
    FinPara
FinProcedimiento
```

```
Inicio
    Enteros: M[30, 30], i, j
    Para i = 1 Hasta 3 Hacer
        Para j = 1 Hasta 3 Hacer
            M[i, j] = Aleatorio(1, 50)
        FinPara
    FinPara

    mostrarMatriz(M, 3, 3)
Fin
```

Pseudo Programa PSeInt:

```
SubAlgoritmo mostrarMatriz(mat, m, n)
    Definir i, j Como Entero
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Imprimir ConvertirATexto(mat[i, j]) + "    " Sin Saltar
        FinPara
        Imprimir ""
    FinPara
FinSubAlgoritmo
```

Algoritmo Matrices

```
    Dimension M[30, 30]
    Definir M Como Entero
    Definir i, j Como Entero
    Para i = 1 Hasta 3 Hacer
        Para j = 1 Hasta 3 Hacer
            M[i, j] = Aleatorio(1, 50)
        FinPara
    FinPara

    mostrarMatriz(M, 3, 3)
FinAlgoritmo
```

Arreglos multidimensionales

Son arreglos de más de dos dimensiones. Hasta tres dimensiones, pueden ser representados gráficamente (un cubo o caja); más allá de ahí, es imposible, pero se pueden implementar tanto matemática como algorítmicamente. Sin embargo, no todos los lenguajes implementan arreglos multidimensionales; dentro de los que permiten crear este tipo de arreglos se encuentran C/C++, PHP y Java.

Representación en memoria

Arreglos de más de tres dimensiones no pueden representarse gráficamente; un arreglo tridimensional se representa como un cubo o caja, como se muestra en las siguientes figuras.

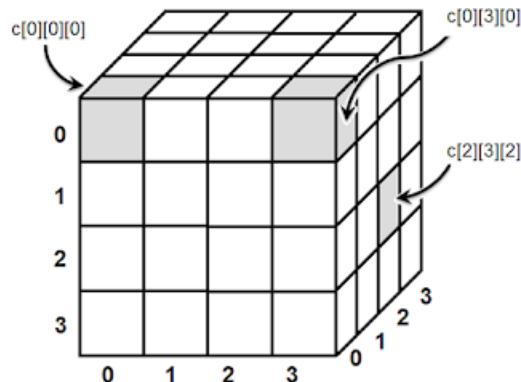


Figura 4.4. Representación gráfica de un arreglo tridimensional en forma de cubo. Aquí la primera posición para cada dimensión es la cero (0)³⁰

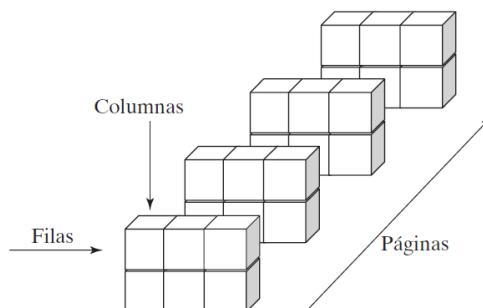


Figura 4.5. Representación gráfica de un arreglo tridimensional³¹

Declaración de arreglos multidimensionales

Para declarar arreglos multidimensionales, simplemente separamos por comas tantas dimensiones como necesitemos, teniendo presente las limitaciones de los lenguajes de programación.

Sintaxis

```
tipo_de_dato: nombre_arreglo[dimension1, dimension2, ..., dimensionN]
```

³⁰ Imagen tomada de [Arreglos de n dimensiones](#).

³¹ Imagen tomada de [Tipos de Arreglos en Matlab - \[octubre, 2023\]](#)

Acceso a los elementos de un arreglo multidimensional

Especificamos un índice por cada dimensión del arreglo separados por comas.

Sintaxis

```
nombre_arreglo[indice1, indice2, ..., indiceN]
```

Nota

A medida que se aumentan las dimensiones de un arreglo, aumenta tanto la complejidad de los algoritmos, como el consumo de recursos de la máquina, por lo que se debe tener precaución al usar arreglos de varias dimensiones. Incluso un problema resuelto mediante vectores y que podría ser solucionado sin ellos, consumirá más recursos de máquina que otro que no los use.

A nivel matemático y algorítmico se puede teorizar a un nivel arbitrario finito de dimensiones, sin embargo, los lenguajes de programación imponen restricciones en cuanto a la cantidad que pueden usarse en éstos, por lo que es necesario revisar la documentación de cada uno antes de intentar realizar algún desarrollo.

Veamos una aplicación del uso de arreglos tridimensionales (tres dimensiones).

Ejemplo 4.5

Una empresa lleva registro de la producción de 10 artículos que elaboran 5 empleados en la semana. Se requiere crear un arreglo que almacene la cantidad de unidades que cada empleado produce al día para sacar diversos reportes.

Necesitamos crear un arreglo de tres dimensiones de orden $5 \times 10 \times 7$. La primera dimensión (*filas*) representa los empleados, y el índice representa su código; la segunda dimensión (*columnas*) representa los productos que igualmente se identifican con un código asociado al índice; y la tercera dimensión (*profundidad, páginas*) representa los días de la semana. Así, si el arreglo se llama *produccion*:

```
produccion[2, 7, 3] = 250
```

Significa que el empleado de código 2 produjo el martes 250 unidades del artículo con código 7.

Tres vectores de tamaños iguales a cada dimensión, almacenan el nombre de los empleados, los artículos y los días respectivamente.

Crear subprogramas que resuelvan además:

- Dado un empleado y un artículo, sume las unidades producidas
- Encuentre el día en que el empleado obtiene mejor rendimiento de un artículo dado
- Promedio de producción de un día determinado

Pseudocódigo:

```
Procedimiento mostrarArreglo(arreglo, m, n, p)
    Entero: i, j, k
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Para k = 1 Hasta p Hacer
                Imprimir arreglo[i, j, k]
            FinPara
        FinPara
    FinPara
FinProcedimiento
```

```
Procedimiento llenarArreglo(arreglo, m, n, p)
    Entero: i, j, k
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Para k = 1 Hasta p Hacer
                arreglo[i, j, k] = Aleatorio(1, 100)
                // Otra opción: Leer arreglo[i, j, k]
            FinPara
        FinPara
    FinPara
FinProcedimiento
```

```
Funcion sumarProduccionDiasEmpleado(prod, codEmp, codArt, dias): Entero
    Entero suma, i
    suma = 0
    Para i = 1, dias, 1
        suma = suma + prod[codEmp, codArt, i]
    FinPara
    Retornar suma
FinFuncion
```

```
Funcion mejorDiaEmpleado(prod, codEmp, codArt, dias): Entero
    Entero mayor, i
    mayor = prod[codEmp, codArt, 1] //Supuesto: domingo mejor producción
    Para i = 2, dias, 1
        Si prod[codEmp, codArt, i] > mayor Entonces
            mayor = prod[codEmp, codArt, i]
        FinSi
    FinPara
    Retornar mayor
FinFuncion
```

```

Funcion promedioDia(prod, m, n, dia): Real
    Entero: i, j, suma
    Real: promedio
    suma = 0
    Para i = 1, m, 1
        Para j = 1, n, 1
            suma = suma + prod[i, j, dia]
        FinPara
    FinPara
    promedio = suma / (m * n)
    Retornar promedio
FinFuncion

Inicio
Entero: produccion[5, 10, 7], s
llenarArreglo(produccion, 5, 10, 7)
mostrarArreglo(produccion, 5, 10, 7)
// Empleado 3, artículo 2, para sumar todos los días de la semana
s = sumarProduccionDiasEmpleado(produccion, 3, 2, 7)
Imprimir "Producción empleado 3 artículo 2: ", s, " unidades"
Imprimir "Promedio de unidades por día: ", promedioDia(prod, 5, 10, 4)
Fin

```

Programación Orientada a Objetos (POO)

La **POO** es otro paradigma de programación que surge esencialmente por las limitaciones de otras técnicas de programación; aprovecha la mayoría de conceptos del paradigma de la programación modular para mejorar las técnicas de desarrollo de software, en particular para la construcción de proyectos grandes.

A diferencia de la programación modular que hace énfasis en los algoritmos, la POO se enfoca en los datos, permitiendo modelar un *mundo basado en objetos*, teniendo como referencia los objetos del mundo real: personas, computadores, casas, vehículos, etc; así como objetos no tangibles: música, deporte, educación, etc., esto es, para la POO cualquier cosa es un objeto, y por tanto puede tener una representación algorítmica en cualquier lenguaje de programación que soporte el paradigma, sea hipotético o real.

Clases y objetos

Un **objeto** es un elemento de una **clase**, conocido como **instancia** de la clase. Por ejemplo, podemos tener animales como gatos, aves, peces, etc., los cuales podemos ver como objetos de una clase *animal*; esto porque todos los animales comparten características comunes de todos los seres vivos.

Una **clase** puede verse como una **plantilla** o un **tipo estructurado de datos** que permite crear objetos de dicho tipo. A manera de analogía podemos decir que una variable es a un tipo de dato como un objeto es a una clase.

Una clase define en una misma unidad (componente, artefacto) **propiedades** o **atributos** (variables miembro, campos) que describen a cualquier objeto de la clase, y **métodos** (subprogramas) que manipulan estos atributos (datos de la clase); los métodos representan **acciones** que se pueden realizar sobre el objeto y son conocidos como la **interfaz del objeto**, ya que es a través de dichos métodos que se puede acceder a los atributos de éste.

Por ejemplo, podemos tener una clase llamada *Animales* con los atributos *subgrupo*, *reproducción*, *hábitat*, etc. A partir de esta clase (plantilla), podemos definir (crear objetos) o crear un canino, felino, ave, pez o cualquier tipo de animal, ya que todos comparten características comunes. También podemos definir acciones (métodos) para cambiar los atributos de la clase u operar con ellos; por ejemplo, podemos tener métodos para determinar si el animal es terrestre o acuático, determinar el tipo de alimentación, cambiar su edad, etc.

Otros ejemplos de clases y objetos:

Clase: Vehículos. Objetos de tipo (clase) vehículos: carro, bicicleta, avión, moto, barco, etc.

Clase Figura geométrica: Objetos: círculo, triángulo, paralelogramo, rombo, etc.

Clase Propiedad raíz: Objetos: edificio, finca, casa unifamiliar, casa prefabricada, etc.

Nota

En ocasiones, y de acuerdo al contexto, es posible que las palabras *clase* y *objeto* se tomen como sinónimos.

El **estado interno** de un objeto es determinado por los valores de aquellas **variables privadas** que solo pueden ser accedidas por otros métodos de la clase.

Para comunicarse con un objeto, es necesario enviarle un **mensaje** a éste, lo que consiste en llamar a uno de sus métodos, posiblemente con parámetros, para que ejecute alguna tarea sobre sus atributos.

Una clase puede disponer de un método **constructor**, el cuál permite realizar una precarga del objeto antes de iniciar la aplicación, esto es, los atributos adquieren un valor por defecto. En el método constructor podemos inicializar las propiedades del objeto a conveniencia y recibir parámetros para crear objetos con una configuración por defecto.

Gráficamente puede usarse una representación que envuelva en un mismo componente tanto los atributos como los métodos del objeto. Por ejemplo, la clase Vehículo podríamos representarla como se muestra en la siguiente figura; sin embargo, es probable que en los textos también se muestren representaciones gráficas diferentes, pero equivalentes a la ilustrada aquí.

Observe de la gráfica, que con esta clase podemos crear distintos *tipos* de vehículos. Los métodos al representar acciones, son descritos generalmente mediante verbos.

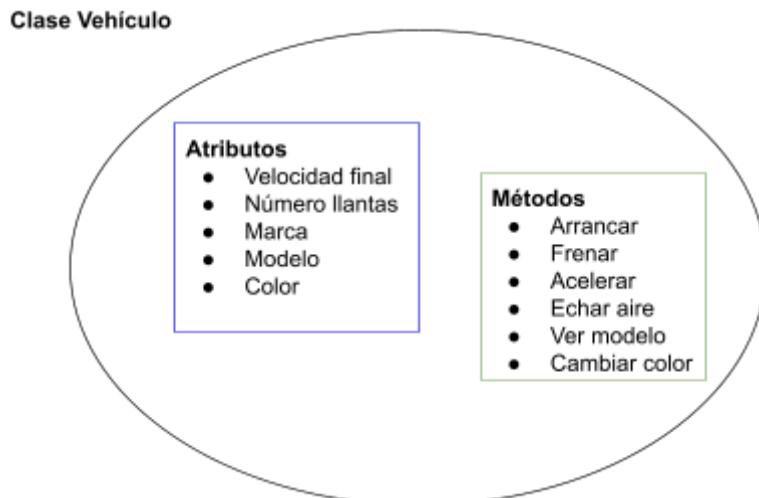


Imagen: representación de una clase para crear objetos de tipo Vehículo

Características de la POO

La orientación a objetos existe en un lenguaje de programación, esto es, se dice que un lenguaje de programación es orientado a objetos, si cumple con las siguientes características.

Abstracción

Cada objeto tiene la capacidad de realizar un trabajo específico, informar y cambiar su estado interno y comunicarse con otros objetos sin revelar cómo se implementan estas características. El proceso de abstracción permite seleccionar características relevantes de un conjunto para identificar comportamientos comunes y así definir nuevos tipos de entidades a partir de éstas.

Ocultación u ocultamiento

Cada objeto está aislado de su entorno exterior y solo puede ser accedido parcialmente a través de su interfaz definida por los métodos de la clase. Esto protege a la clase de posibles modificaciones de accesos no autorizados. La ocultación se logra gracias a los modificadores de acceso, que se describen más adelante.

Modularidad

Es una propiedad que indica que toda aplicación debe dividirse en partes más pequeñas y que sean tan independientes entre sí como sea posible. Dichas partes son llamadas **módulos**.

Encapsulamiento

El concepto que define a un objeto, es precisamente la característica que permite **encapsular** su información, esto es, tener o agrupar en una misma unidad (entidad, artefacto, componente) a los atributos (variables) que describen al objeto y a los métodos (funciones) que acceden a dichos atributos.

Herencia

La **herencia** es una de las características principales de la POO, ya que permite la reutilización de código y crear jerarquías entre clases. De la herencia nace el concepto de **súper clase**, la cual es una definición abstracta de una entidad y de la cual se pueden construir otros objetos. Por ejemplo entidades como estudiante, cliente, empleado, etc., son todos personas, por lo cual se puede tener una *súper clase* Persona con los atributos que poseen todas las personas, y luego crear clases **heredadas** para estudiantes, clientes, etc., con sus propios atributos y métodos, pero heredando todas las propiedades y funcionalidades de la **clase padre**.

Reutilización

Polimorfismo

Modificadores de acceso

Son palabras clave que se utilizan para controlar el acceso a los atributos y métodos de una clase, con lo cual se especifica quién puede acceder al objeto desde el exterior. Con dichos modificadores, es posible ocultar el estado de un objeto al exterior.

Público (Public)

Este es el modificador menos restrictivo. Los atributos y métodos declarados como públicos pueden ser accesibles desde cualquier parte de la aplicación.

Privado (Private)

Los atributos y métodos declarados sólo son accesibles dentro de la propia clase en la que se definen. No pueden accederse desde fuera de la clase, ni siquiera desde sus clases derivadas, la única forma es a través de otros métodos que sean públicos.

Protegido (Protected)

Las propiedades y métodos protegidos permiten el acceso desde la propia clase y también desde sus clases derivadas (heredadas). Sin embargo, no se puede acceder a los elementos desde fuera de la jerarquía de herencia. Es similar al modificador privado con algo más de alcance.

Declarar una clase

Hay una serie de nuevos términos que entran en juego al declarar una clase que permiten especificar cómo se podrá acceder a ésta y como podrá interactuar con otros objetos.

Sintaxis

```
Clase NombreClase
    Privado|Publico|Protegido tipo_dato propiedad1
    Privado|Publico|Protegido tipo_dato propiedad2
    ...
    Privado|Publico|Protegido tipo_dato propiedadN

    [Publico Metodo constructor([argumentos])
        //Carga por defecto para las instancias que se crean
        FinMetodo]

    [Publico Metodo destructor([argumentos])
        //Acciones a ejecutar al eliminar la instancia
        FinMetodo]

    Publico Metodo asignarPropiedad1(tipo_dato: valor)
        propiedad1 = valor
        FinMetodo
```

```
Publico Metodo asignarPropiedad2(tipo_dato: valor)
    propiedad2 = valor
FinMetodo

...
Publico Metodo asignarPropiedadN(tipo_dato: valor)
    propiedadN = valor
FinMetodo

Publico Metodo obtenerPropiedad1(): [tipo_dato]
    Retornar propiedad1
FinMetodo

Publico Metodo obtenerPropiedad2(): [tipo_dato]
    Retornar propiedad2
FinMetodo

...
Publico Metodo obtenerPropiedadN(): [tipo_dato]
    Retornar propiedadN
FinMetodo

Publico|Privado|Protegido Metodo otroMetodo1([argumentos]):[tipo_dato]
    // Cuerpo del método
    [Retornar valor]
FinMetodo

...
Publico|Privado|Protegido Metodo otroMetodoN([argumentos]):[tipo_dato]
    // Cuerpo del método
    [Retornar valor]
FinMetodo
FinClase
```

Instanciar una clase

Instanciar una clase consiste en declarar un **objeto** del tipo de dicha clase.

Sintaxis

```
NombreClase nombreObjeto = Nuevo NombreClase([argumentos])
```

Por ejemplo, para instanciar la clase Vehículo, haríamos lo siguiente:

```
// Instanciar la clase Vehículo  
Vehículo veh = Nuevo Vehículo()
```

Donde **veh** es el objeto de tipo **Vehículo**.

Eliminar una instancia de una clase. Recolector de basura

Esta operación consiste en eliminar (destruir) el objeto y las referencias a éste, ahorrando con esto recursos de la máquina y posibles errores debido a las referencias a la memoria que aún quedan presentes. Los lenguajes modernos eliminan automáticamente los objetos que dejan de ser usados y son enviados a **recolectores de basura (garbage collection)**, por lo que esta operación es opcional y haciendo que el programador no se tenga que preocupar por la gestión de la memoria.

Un **recolector de basura (garbage collector)** es un mecanismo implícito de gestión de la memoria implementado en la mayoría de lenguajes de programación orientados a objetos, ya sean puros o híbridos. Sin que se requiera eliminar de forma explícita un objeto de la memoria, el recolector de basura se encarga de “limpiar la basura”, o en otras palabras. borrar de la memoria aquellos objetos en desuso dentro del programa.

Algunos lenguajes que disponen del recolector de basura son: Python, PHP, Java, Javascript, Ruby, C# y Visual Basic .NET, entre otros. En C++ (la “versión” de C orientada a objetos) no existe este mecanismo, por lo que los objetos deben ser destruidos de forma explícita.

En caso de querer eliminar el objeto de forma explícita o en aquellos lenguajes que no disponen de un recolector de basura, existe una sentencia que permite borrarlos (destruirlos) de la memoria. A continuación se muestra la forma general en algoritmia.

Sintaxis

```
nombreObjeto = Nulo
```

Por ejemplo, para el objeto **veh** de tipo **Vehículo**, haríamos lo siguiente:

```
// Eliminar el objeto veh  
veh = Nulo
```

Donde **veh** es el objeto de tipo **Vehículo**.

Ejemplo 4.6

Crear una clase (super clase) llamada *Persona* con las propiedades *nombre* y *edad*. La clase debe contener los métodos para agregar datos, devolverlos y otro que determine si la persona es mayor de edad. Realizar además lo siguiente:

- Crear dos objetos de tipo Persona e ingresar la información respectiva
- Mostrar los datos de las personas
- Determinar cuál de éstas personas es mayor
- A partir de la clase Persona, crear otra clase para gestionar empleados llamada *Empleado* con los atributos *salario mínimo* y *salario*. Además de los métodos de asignación y devolución, crear otro para determinar si un empleado gana el salario mínimo. El constructor de la clase debe permitir cargar opcionalmente el salario mínimo.

Pseudocódigo:

```
// Super clase Persona
Clase Persona
    // Atributos de clase
    Privado Caracter nombre
    Privado Entero edad

    Publico Metodo constructor()
        //Carga por defecto
    FinMetodo

    Publico Metodo destructor()
        //Acciones al destruir el objeto
    FinMetodo

    Publico Metodo asignarNombre(Caracter: nom): Ninguno
        nombre = nom
    FinMetodo

    Publico Metodo obtenerNombre(): Caracter
        Retornar nombre
    FinMetodo

    Publico Metodo asignarEdad(Enteros: ed): Ninguno
        edad = ed
    FinMetodo

    Publico Metodo obtenerEdad(): Entero
        Retornar edad
    FinMetodo

    Publico Metodo mayorEdad(): Logico
        Si edad >= 18 Entonces
            Retornar Verdadero
        SiNo
```

```
    Retornar Falso
FinSi
FinMetodo
FinClase

// Clase derivada o heredada: la clase Empleado hereda de la clase Persona
Clase Empleado HeredaDe Persona
    Privado Real salarioMinimo
    Privado Real salario

    Publico Metodo constructor(salMin = 0): Ninguno
        salarioMinimo = salMin
    FinMetodo

    Publico Metodo destructor(): Ninguno
        //Acciones al destruir el objeto
    FinMetodo

    Publico Metodo asignarSalarioMinimo(Real: salMin): Ninguno
        salarioMinimo = salMin
    FinMetodo

    Publico Metodo obtenerSalarioMinimo(): Real
        Retornar salarioMinimo
    FinMetodo

    Publico Metodo asignarSalario(Real: sal): Ninguno
        salario = sal
    FinMetodo

    Publico Metodo obtenerSalario(): Real
        Retornar salario
    FinMetodo

    Publico Metodo ganaSalarioMinimo(): Logico
        Si salarioMinimo = salario Entonces
            Retornar Verdadero
        SiNo
            Retornar Falso
        FinSi
    FinMetodo
FinClase

Inicio
Entero: edad
```

```
Caracter: nombre
Real: salario

Persona per1 = Nuevo Persona()
Persona per2 = Nuevo Persona()
Empleado emp = Nuevo Empleado(1000000)

Leer nombre, edad
per1.asignarNombre(nombre)
per1.asignarEdad(edad)

Leer nombre, edad
per2.asignarNombre(nombre)
per2.asignarEdad(edad)

Si per1.mayorEdad() Entonces
    Imprimir per1.obtenerNombre(), " es mayor de edad"
SiNo
    Imprimir per1.obtenerNombre(), " es menor de edad"
FinSi
Si per2.mayorEdad() Entonces
    Imprimir per2.obtenerNombre(), " es mayor de edad"
SiNo
    Imprimir per2.obtenerNombre(), " es menor de edad"
FinSi
Si per1.obtenerEdad() > per2.obtenerEdad() Entonces
    Imprimir per1.obtenerNombre(), " es mayor que ", per2.obtenerNombre()
SiNo
    Imprimir per2.obtenerNombre(), " es mayor que ", per1.obtenerNombre()
FinSi

Leer nombre, edad, salario
emp.asignarNombre(nombre)
emp.asignarEdad(edad)
emp.asignarEdad(salario)
Imprimir "Nombre empleado: ", emp.obtenerNombre()
Imprimir "Edad empleado: ", emp.obtenerEdad()
Imprimir "Salario empleado: ", emp.obtenerSalario()
Si emp.obtenerSalarioMinimo() > 0 Entonces
    Imprimir "Salario mínimo actual: ", emp.obtenerSalarioMinimo()
    Imprimir "Salario empleado actual: ", emp.obtenerSalario()
    Si emp.ganaSalarioMinimo() Entonces
        Imprimir "Salario igual al mínimo"
    SiNo
        Imprimir "Salario diferente al mínimo"
    FinSi
SiNo
```

```
    Imprimir "No ha indicado el salario mínimo"  
FinSi  
Fin
```

Preguntas

Ejercicios

Leer N números y guardar en una vector los números positivos. Crear subprogramas para mostrar el arreglo y mostrar el mayor número almacenado.

1. Crear el vector V con n valores numéricos aleatorios entre 1 y 50. Usando subprogramas, realizar las siguientes operaciones sobre él:
 - a. Mostrar el arreglo
 - b. $\sum_{i=1}^n V_i$ (sumatoria)
 - c. $\prod_{i=1}^n V_i$ (productoria)
 - d. \bar{V} (media o promedio)
 - e. Encontrar el mayor dato
 - f. Encontrar el menor dato
 - g. Buscar un elemento
 - h. Encontrar la moda (elemento que más se repite)
 - i. Eliminar un elemento
 - j. Insertar un elemento dada una posición
 - k. Ordenar el vector
2. La universidad está procesando la información por grupos de sus estudiantes en arreglos, para lo cual requiere un programa para llevar registro de su documento de identidad, el nombre y la nota. La solución debe implementar programación modular para los siguientes requerimientos utilizando un menú de opciones:
 - a. Pedir los datos de un estudiante y agregarlos a las listas. Debe validar que no se ingrese el mismo documento de identidad más de una vez y que la nota esté entre 0 y 5
 - b. Listar todos los estudiantes en forma tabulada y añadir una columna adicional que muestre si ganó o perdió la asignatura
 - c. Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
 - d. Mostrar el promedio de notas del grupo
 - e. Mostrar la mayor nota del grupo y a quien pertenece ésta
 - f. Mostrar la menor nota del grupo y a quien pertenece ésta
 - g. Permitir la eliminación de estudiantes

- h. Permitir actualizar la nota de un estudiante
 - i. Informar si hay un estudiante de nombre Pedro Zapata o Ana Zapata
- 3. La universidad está procesando la información de sus estudiantes en arreglos, para lo cual requiere un programa para llevar registro de su documento de identidad, el nombre y la nota de cinco asignaturas. La solución debe implementar programación modular para los siguientes requerimientos utilizando un menú de opciones:
 - a. Pedir los datos de un estudiante y agregarlos a las listas. Debe validar que no se ingrese el mismo documento de identidad más de una vez y que las notas estén entre 0 y 5
 - b. Listar todos los estudiantes en forma tabulada junto con su promedio de notas
 - c. Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
 - d. Mostrar el promedio general de notas
 - e. Mostrar la mayor nota de un estudiante cualquiera
 - f. Mostrar la menor nota de un estudiante cualquiera
 - g. Mostrar la mayor nota de todos los estudiantes y a quien pertenece ésta
 - h. Mostrar la menor nota de todos los estudiantes y a quien pertenece ésta
 - i. Permitir la eliminación de estudiantes
 - j. Permitir actualizar la nota de un estudiante
- 4. Una tienda procesa la siguiente información de sus productos: nombre, código, precio y cantidad. La tienda maneja un dato general para controlar el *stock* mínimo de cada producto, esto es, la cantidad mínima de productos para que se lance una alerta y se deba pedir mercancía. La tienda requiere una aplicación modular que mediante un menú de opciones, permita realizar las siguientes operaciones sobre su información que se guarda en arreglos unidimensionales:
 - a. Agregar productos de forma ordenada por código, validando que éstos no se repitan
 - b. Mostrar el producto más costoso
 - c. Mostrar el producto más barato
 - d. Listar el inventario de forma tabulada y añadir una columna adicional que muestre un IVA del 19% para todos los artículos
 - e. Permitir buscar un producto por código
 - f. Listar los productos cuya cantidad sea inferior al *stock* mínimo y alertar en caso de encontrarlos
 - g. Permitir actualizar cualquier dato del producto, excepto el código, validando además que la cantidad no sea negativa y el precio mayor a cero
 - h. Permitir eliminar productos
 - i. Informar si hay un producto que tenga una existencia de 300 o de 400 unidades
- 5. Crear una matriz M con $m \times n$ valores numéricos aleatorios entre 1 y 50. Usando subprogramas, realizar las siguientes operaciones sobre ella:
 - a. Mostrar el arreglo
 - b. $\sum_{i=1}^n M_{i,j}$ (sumatoria)
 - c. $\prod_{i=1}^n M_{i,j}$ (productoria)

- d. \bar{M} (media o promedio)
 - e. Encontrar el mayor dato
 - f. Encontrar el mayor dato de una fila o columna dada
 - g. Encontrar el menor dato
 - h. Encontrar el menor dato de una fila o columna dada
 - i. Buscar un elemento
 - j. Eliminar una fila o columna
 - k. Insertar una fila o columna dada una fila y columna de referencia
 - l. Ordenar una fila o columna dada
6. Un almacén registra las ventas de la semana de n productos que manejan en una matriz. Cada fila de la matriz representa un producto que está codificado de acuerdo al número de ésta, así, la primera fila significa que el producto tiene código 1, la fila 2 es para el producto con código 2 y así sucesivamente; cada columna representa un día de la semana: la primera equivale al domingo, la segunda al lunes, etc. Usando programación modular y mediante un menú de opciones, se pide:
- a. Ingresar las ventas de totales de cada producto por día
 - b. Mostrar la matriz completa en forma de tabla
 - c. Buscar un producto y mostrar las ventas de la semana
 - d. Dado un día, mostrar las ventas de cada producto en éste
 - e. Mostrar el total de ventas por día
 - f. Mostrar el promedio de ventas por día
 - g. Mostrar el total de ventas por producto
 - h. Mostrar el promedio de ventas por producto
 - i. Mostrar el total de ventas de la semana
 - j. Mostrar el promedio de ventas de la semana
 - k. Mostrar la mejor venta del día
 - l. Mostrar el mejor día en ventas de un producto
 - m. Mostrar la peor venta del día
 - n. Mostrar el peor día en ventas de un producto
 - o. Mostrar la mejor venta e indicar a qué producto corresponde y en qué día se realizó
 - p. Mostrar la peor venta e indicar a qué producto corresponde y en qué día se realizó
 - q. Permitir actualizar los valores de las ventas

Fuentes y referencias adicionales

Bibliografía del microcurrículo

VILLALOBOS J., CASALLAS R. Fundamentos de programación: Aprendizaje activo basado en casos. Bogotá, Pearson - Prentice Hall. 2006.

JOYANES AGUILAR, LUIS. Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos. McGraw-Hill/Interamericana de España. Madrid, 2004.

BOTERO R., CASTRO C., MAYA J., TABORDA G. y VALENCIA M.. Lógica y Programación orientada a objetos: un enfoque basado en problemas. CITIA, proyecto SISMOO, Tecnológico de Antioquia. Medellín, 2009.

Referencias adicionales

JOYANES AGUILAR, LUIS. Fundamentos de Programación, Algoritmos y Estructuras de Datos. Segunda edición. McGraw-Hill/Interamericana de España. Madrid, 1996.

RÍOS, FABIÁN. Soluciones secuenciales. Universidad de Antioquia. Medellín, 1995.

CASTILLO SUAZO, ROMMEL. Programación en: PSInt. Original para LPP. Implementado por: CARO, ALEJANDRO. Documento PDF.

En Internet (recuperado: 08/10/2023)

[Manual PSInt - Programación en](#)