NOTAS DE CLASE

FUNDAMENTOS DE PROGRAMACIÓN

{Con los lenguajes de programación Python, Java, PHP, C++, C#, Javascript, PSeInt y pseudocódigo}

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

NOTAS DE CLASE
FUNDAMENTOS DE PROGRAMACIÓN

{Con los lenguajes de programación Python, Java, PHP, C++, C#, Javascript, PSeint y pseudocódigo}

```
/**
* Versión 1.0
* Año: 2026
* Licencia software: GNU GPL
* Licencia doc: GNU Free Document License (GNU FDL)
*/

public class Author {
    String name = “Jaime E. Montoya M.”;
    String profession = “Ingeniero Informático”;
    String employment = “Docente y desarrollador”;
    String city = “Medellín - Antioquia - Colombia”;
    int year = 2026;
}
```

Tabla de contenido

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.

El texto nace de la necesidad de brindar a los estudiantes una guía puntual de las temáticas tratadas en clase, que sea de fácil acceso y lectura, y presentado de forma didáctica y concisa, para así concentrar los espacios del aula en la resolución de ejemplos y aclaración de conceptos.

La guía incluye, además de los conceptos teóricos, ejemplos, gráficas, desarrollos en clase, y al final de cada capítulo, unas preguntas y ejercicios que permiten 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. Las notas al pie de página contienen enlaces a lecturas complementarias.

Este documento incluye una serie de ejercicios resueltos de programación realizados en clase y propuestos por el autor que se anexan finalizando cada capítulo. Estos ejercicios incluyen algunos conceptos teóricos y prácticos adicionales, consejos, buenas prácticas y ejemplos explicados, divididos de acuerdo a la unidad temática estudiada.

Los ejemplos se ilustran con pseudocódigo, en el pseudo lenguaje PSeInt, y en algunos lenguajes de programación como Java, C/C++, PHP, Python, C# y Javascript, entre otros. También se presentan diferentes técnicas gráficas para diagramar algoritmos, entre ellas los *flujogramas* o *diagramas libres* y los *diagramas rectangulares* o de *Nassi-Schneiderman*.

El uso de los lenguajes y pseudo lenguajes de programación tienen como objetivo la práctica por parte del estudiante donde podrá poner a prueba sus conocimientos, así como analizar las diferencias entre la teoría (pseudocódigo) y la práctica (código), esto es, llevar un algoritmo y convertirlo pseudocódigo y luego en un programa.

Se recomienda al lector revisar los ejemplos y realizar los ejercicios propuestos para afianzar los conceptos mediante la práctica. Tanto en los ejemplos como en los ejercicios se encuentran conceptos adicionales a la teoría y se explican formas de enfrentar los problemas.

El texto se divide en cuatro partes, y puede ser usado en igual número de cursos de la línea del desarrollo de algoritmos, a saber: lógica de programación, programación orientada a objetos, estructuras de datos y análisis de algoritmos.

La primera parte, lógica de programación, trata los temas introductorios al desarrollo de software. Se hace una pequeña reseña histórica de las máquinas de cálculo, para entrar a

estudiar las bases de la programación, que incluyen entre otros aspectos, la definición de conceptos como variables y constantes, entrada y salida de información, operadores, comentarios, errores, palabras reservadas, etc. También se revisan las estructuras de control (condicionales y repetitivas) y la bifurcación de control. Por último, se aprovecha el paradigma de la programación estructurada que permite la modularidad para la creación de funciones y procedimientos. Esta técnica permite que el código de los programas sean más legible y fácil de entender y mantener.

La segunda parte del texto, programación orientada a objetos, estudia este paradigma casi que imprescindible en gran parte de las aplicaciones modernas, que se apoya en el paradigma de la programación estructurada, mejorando de gran manera la forma en que se desarrolla el software.

La tercera parte se ocupa del estudio de las estructuras de datos, comenzando por las estructuras fundamentales como las cadenas, los arreglos, los registros y los conjuntos, hasta llegar a los árboles y grafos, pasando por otras como las listas ligadas, las pilas y colas y estudiando la recursión, otra técnica algorítmica utilizada en ciertos tipos de problemas de naturaleza recursiva..

La cuarta parte del texto se enfoca en el análisis de algoritmos, que tiene como objetivo desarrollar en el estudiante un pensamiento lógico y estructurado a partir de conocimiento técnico, análisis de casos de estudio y desarrollo lógico de algoritmos que cumplan con requerimientos de eficiencia computacional.

El contenido se centra en el "análisis de algoritmos", término acuñado por Donald Knuth[1] que lo concibió como el análisis teórico de algoritmos para estimar su complejidad en sentido asintótico, es decir, estimar la función de complejidad para entradas arbitrariamente grandes. El análisis de algoritmos es una parte importante de la teoría de la complejidad computacional, que proporciona una estimación teórica de los recursos necesarios de un algoritmo para resolver un problema computacional específico. La mayoría de los algoritmos están diseñados para trabajar con entradas de longitud arbitraria. El análisis de algoritmos consiste en determinar la cantidad de recursos de tiempo y espacio necesarios para ejecutarlo, lo que se conoce como eficiencia computacional.

Complementariamente, se pretende que el estudiante reconozca los problemas según su planteamiento y solución, y conozca las técnicas de diseño de algoritmos que pueden mejorar la eficiencia computacional, o si se entrega una solución con un grado de eficacia aceptable.

La última parte del texto comprende las referencias bibliográficas con una lista de textos impresos y páginas web y un apéndice con aplicaciones de problemas matemáticos.

[1] Donald Ervin Knuth es un reconocido experto en ciencias de la computación estadounidense y matemático, famoso por su fructífera investigación dentro del análisis de algoritmos y compiladores. Es Profesor Emérito de la Universidad de Stanford. Donald Knuth - Wikipedia, la enciclopedia libre

Recursos

Adicional a la bibliografía disponible al final del texto y en las notas al pie de página que contienen referencias a otras lecturas, para este curso se debe contar con un equipo de cómputo que soporte los lenguajes de programación actuales, entre ellos Java y Python, con el fin de realizar las prácticas. Se asume que el lector ya está familiarizado con éstos y otros lenguajes de programación si va a estudiar de la segunda parte del texto en adelante. El sistema operativo es a gusto del estudiante, puede trabajar bajo Windows, Linux, Mac u otro. En la primera parte del texto se ilustran diversos ejemplos haciendo uso del pseudo lenguaje PSeInt.

Además de disponer de conexión a Internet, debe contar también con un IDE o editor (se sugiere Visual Studio Code, pero no es obligatorio) para la edición de programas en distintos lenguajes.

Con una cuenta de Google se puede hacer uso de Colab, una herramienta para la edición y ejecución de partes de código Python.

El sitio web Programiz ofrece una interfaz para el desarrollo de código con la opción de poder seleccionar entre varios lenguajes de programación, tales como Java, Python, C, C++, C#, PHP, R y Javascript, entre otros.

Lista de abreviaturas

ASCII: American Standard Code for Information Interchange

BD: Base de Datos

CF: Contador de Frecuencias

DyV: Divide y Vencerás

ED: Estructura(s) de Datos

IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)

LSL: Lista Simplemente Ligada

LSLC: Lista Simplemente Ligada Circular

LDL: Lista Doblemente Ligada

LDLC: Lista Doblemente Ligada Circular

MCD: Máximo Común Divisor

NS: Nassi-Schneiderman (diagrama)

PHP: PHP Hypertext Preprocessor

POO: Programación Orientada a Objetos

SI: Sistema de Información

SO: Sistema Operativo

VSC: Visual Studio Code

Prefijos y unidades de almacenamiento

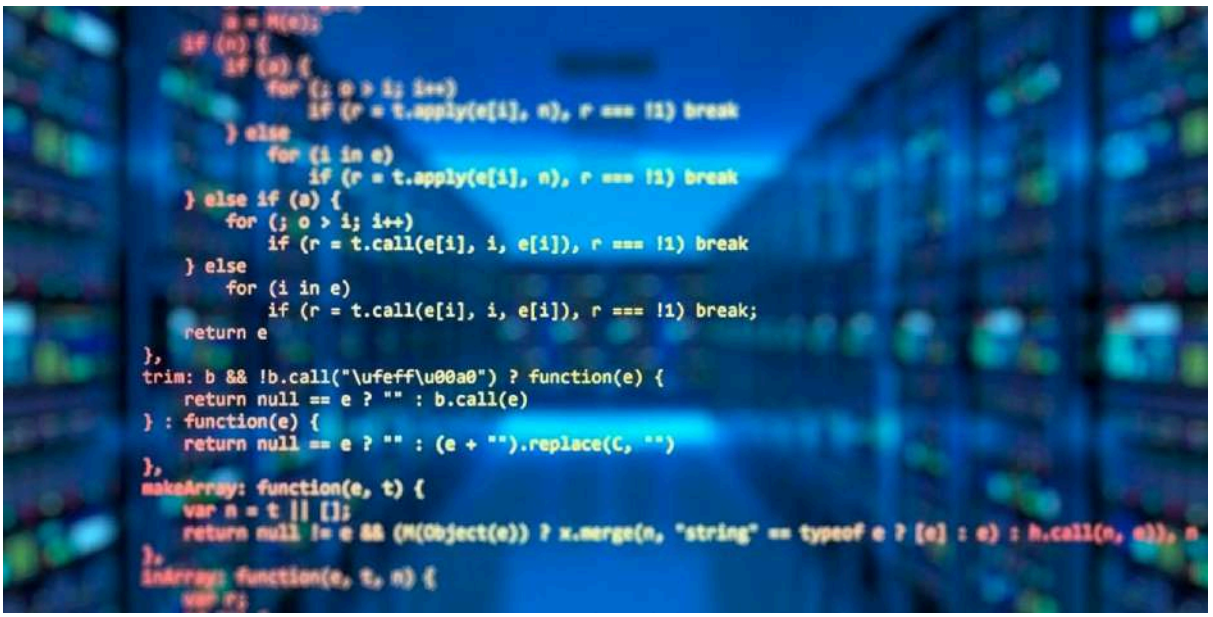
Prefijos

Prefijo	Nombre/Significado	Equivalencia
f	Femto	10^{-15}
p	Pico	10^{-12}
n	Nano	10^{-9}
μ	Micro	10^{-6}
m	Mili	10^{-3}
c	Centi	10^{-2}
d	Deci	10^{-1}
-	Uno/unidad	10^{0}
da	Deca	10^{1}
h	Hecto	10^{2}
K	Kilo	10^{3}
M	Mega	10^{6}
G	Giga	10^{9}
T	Tera	10^{12}
P	Peta	10^{15}

Unidades de almacenamiento

Unidad	Símbolo	Equivalencia
Bit	b	0, 1
Byte	B	2^{3}b = 8b
Kilobyte	KB	1024B = 2^{10}B = 8192b
Megabyte	MB	1024KB = 1048576B

Gigabyte	GB	1024MB
Terabyte	TB	1024GB
Peta	PB	1024TB



PRIMERA PARTE. LÓGICA DE PROGRAMACIÓN

Capítulo 1. 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.[2]

[2] Algunas referencias y la imagen fueron tomadas de: Ábaco - Wikipedia, la enciclopedia libre

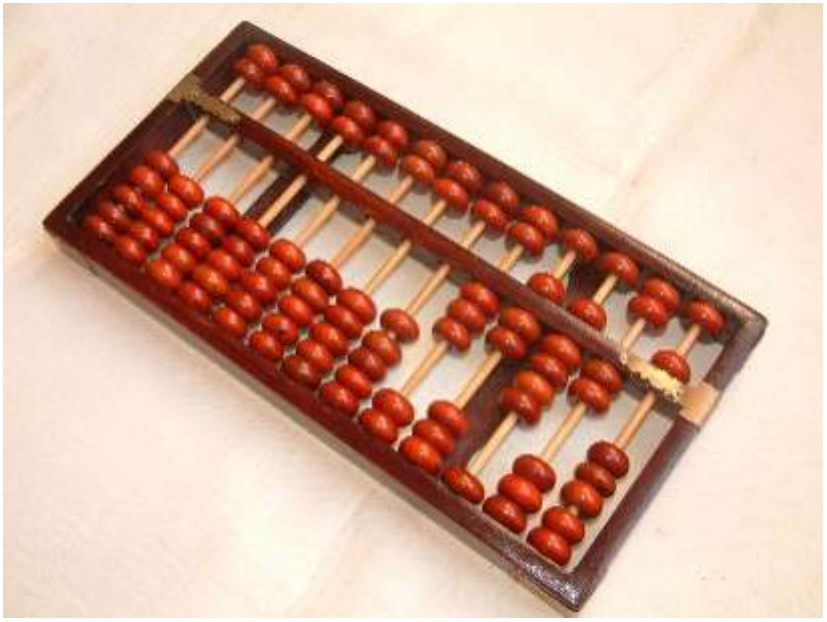


Figura 0.1. Ábaco chino

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.[3]

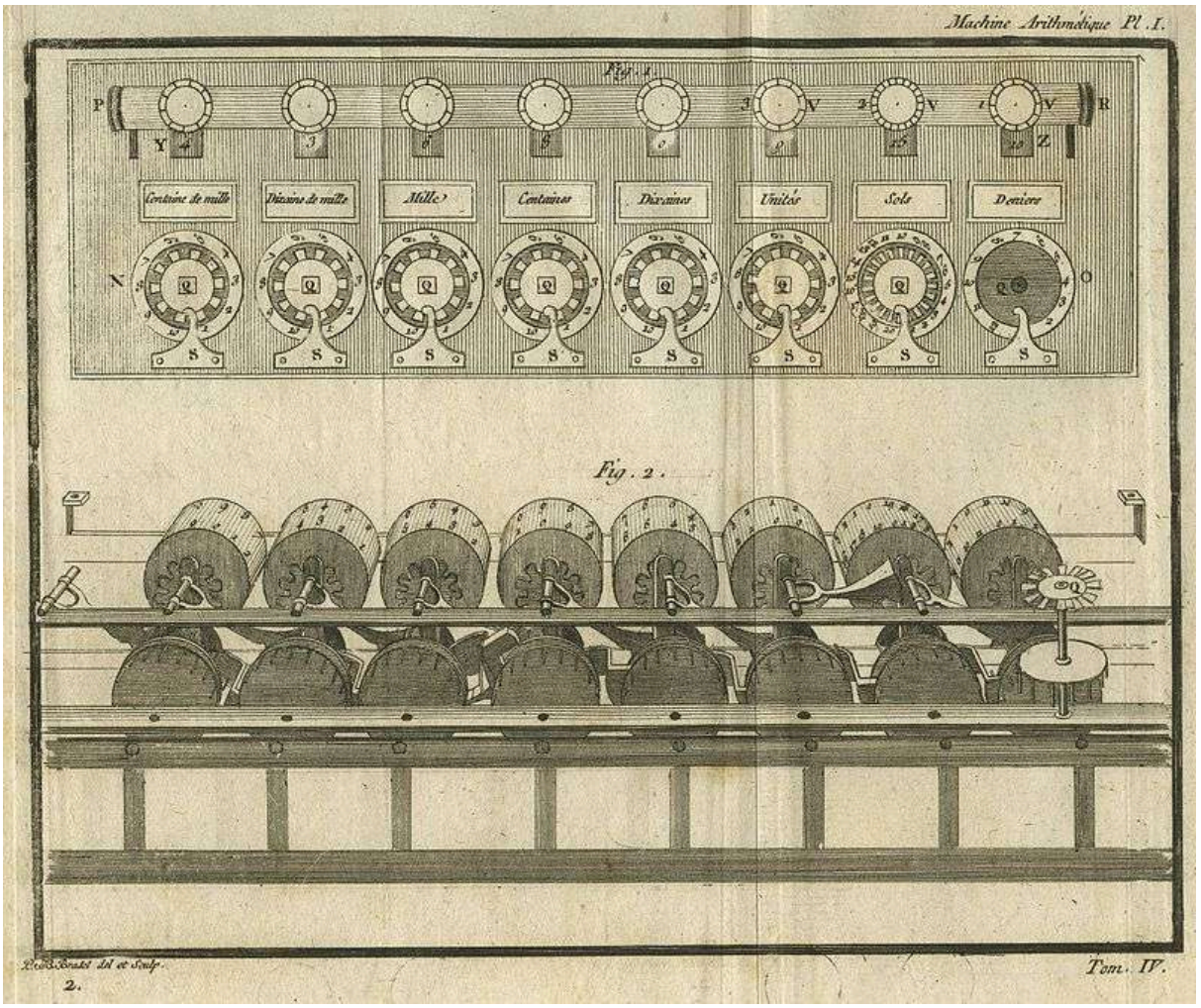


Figura 0.2. Planos originales de la pascalina

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**.

[3] Algunas referencias y la imagen fueron tomadas de: Pascalina - Wikipedia, la enciclopedia libre

La siguiente figura muestra una réplica de la máquina de Leibniz.[4]

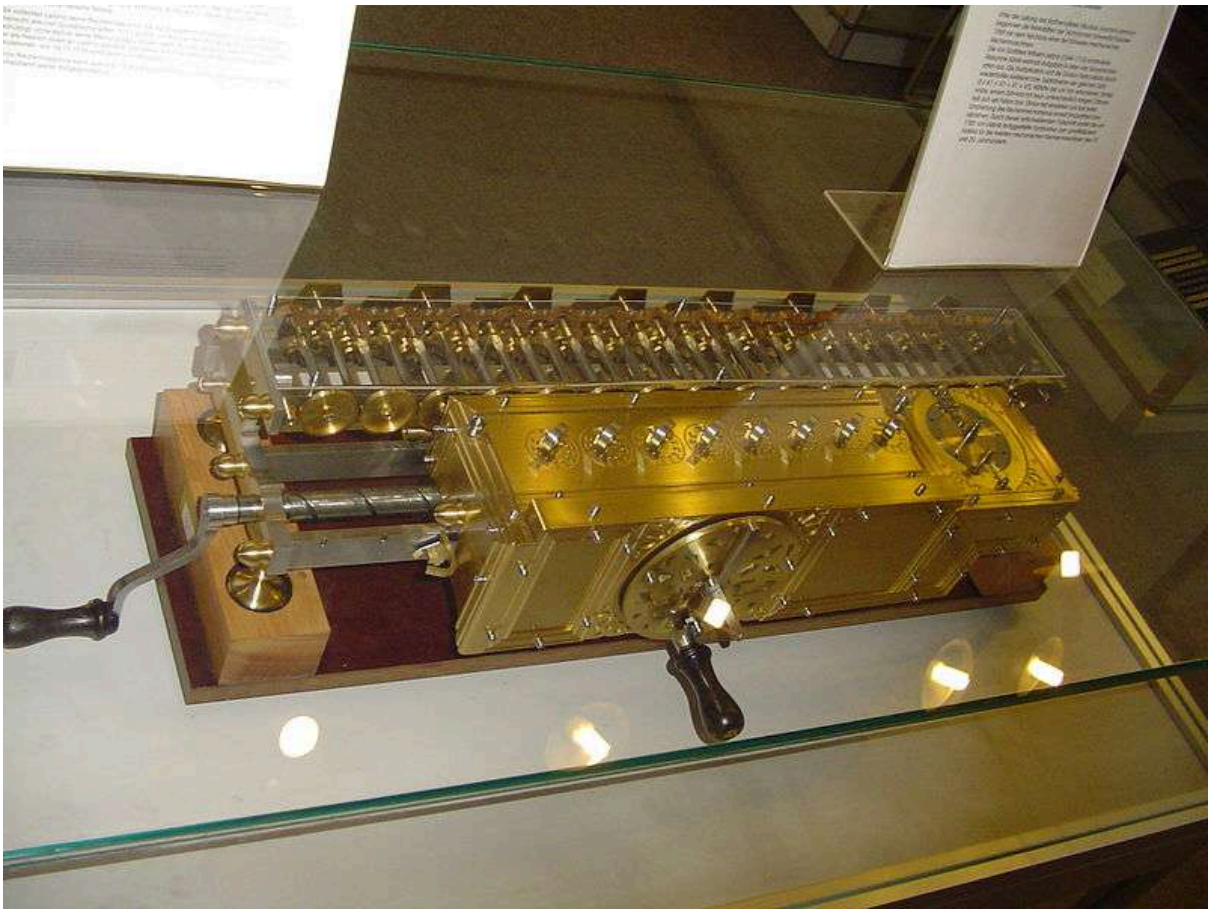


Figura 0.3. Réplica de la máquina de Leibniz

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.[5]

[4] Algunas referencias y la imagen fueron tomadas de: Rueda de Leibniz - Wikipedia, la enciclopedia libre

[5] 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 análitica 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:

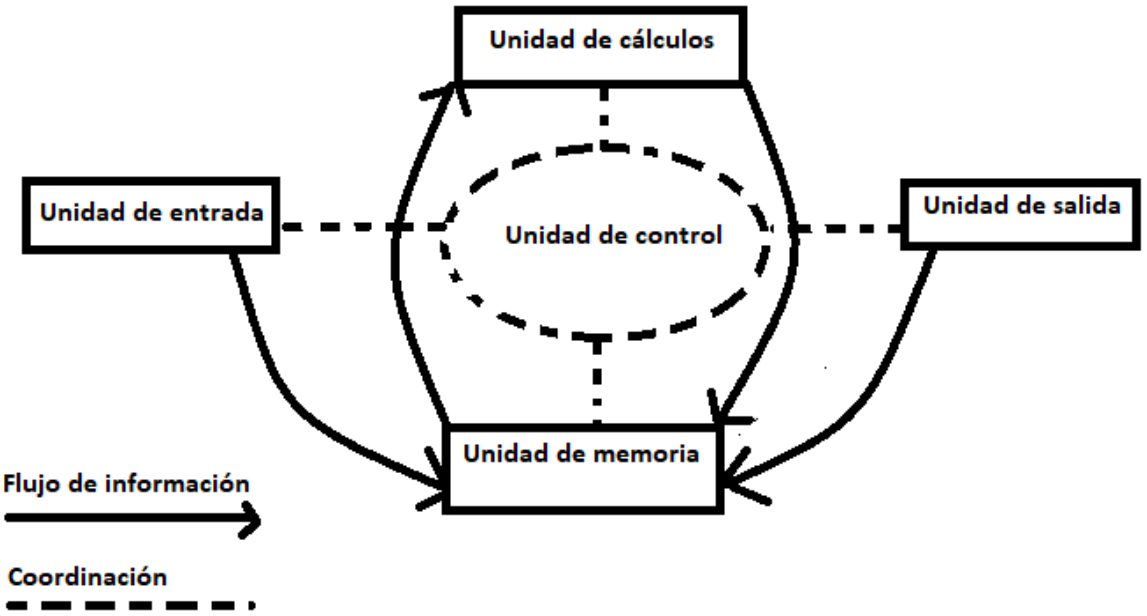


Figura 0.5. Modelo simplificado de un computador

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.[6]

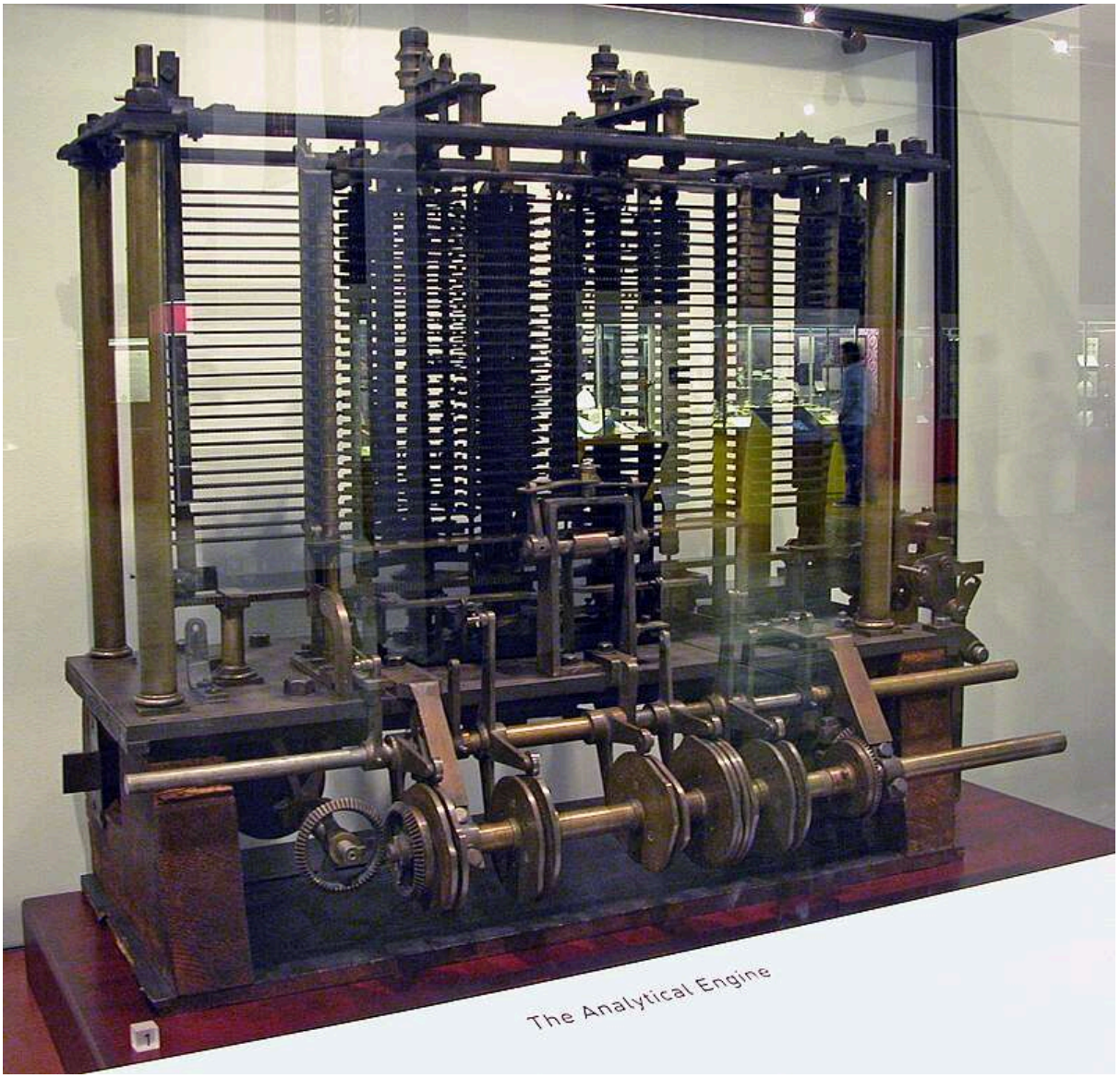


Figura 0.6. Mäquina analítica de Babbage

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*

[6] Algunas referencias y la imagen fueron tomadas de: Máquina analítica - Wikipedia, la enciclopedia libre

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)*[7] 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"[8].

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 "***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.

[7] Puede leer acerca del *Entscheidungsproblem* en: Entscheidungsproblem - Wikipedia, la enciclopedia libre

[8] 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

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 ***t*abla 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:

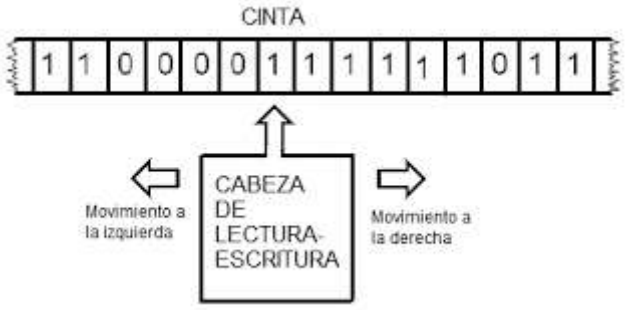


Figura 0.7. Máquina de Turing[9]

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'.

[9] Figura tomada de: Redalyc.DESARROLLO DE UN ENTORNO DE SIMULACIÓN PARA AUTÓMATAS DETERMINISTAS

El test de Turing[10]

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 realizara 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 disciplina[11].

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,

[10] Puede leer más al respecto en el siguiente artículo: Prueba de Turing - Wikipedia, la enciclopedia libre
[11] Puede leer más al respecto en el siguiente artículo: Los padres de la Inteligencia Artificial no son del siglo XXI

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 sistema[12].

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.

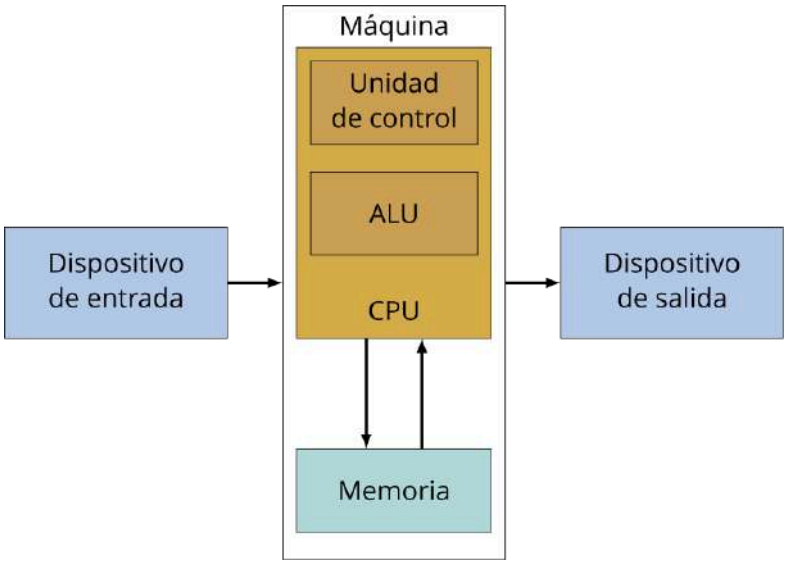


Figura 0.8. Diagrama de la arquitectura von Neumann.

[12] 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.

Las generaciones de computadores

Las llamadas generaciones de computadores son periodos 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 generaciones[13].

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.

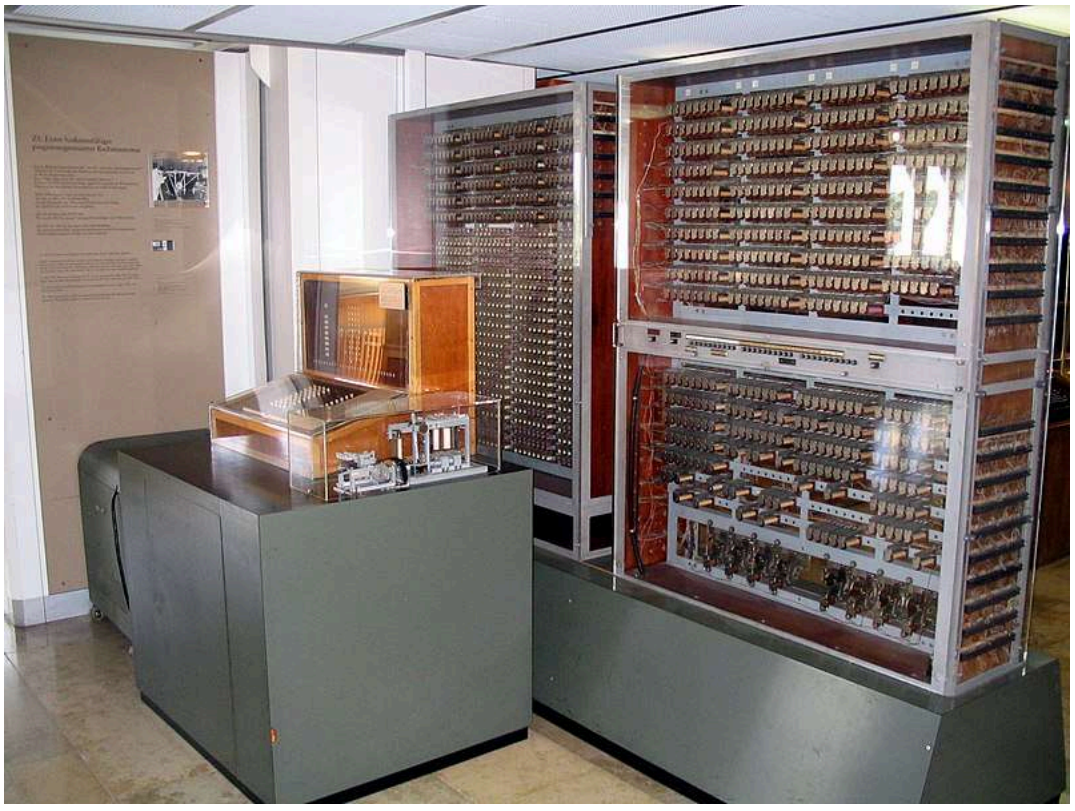


Figura 0.9. Fotografía de la réplica del Z3 (tomada de Wikipedia).

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

[13] 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.

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.

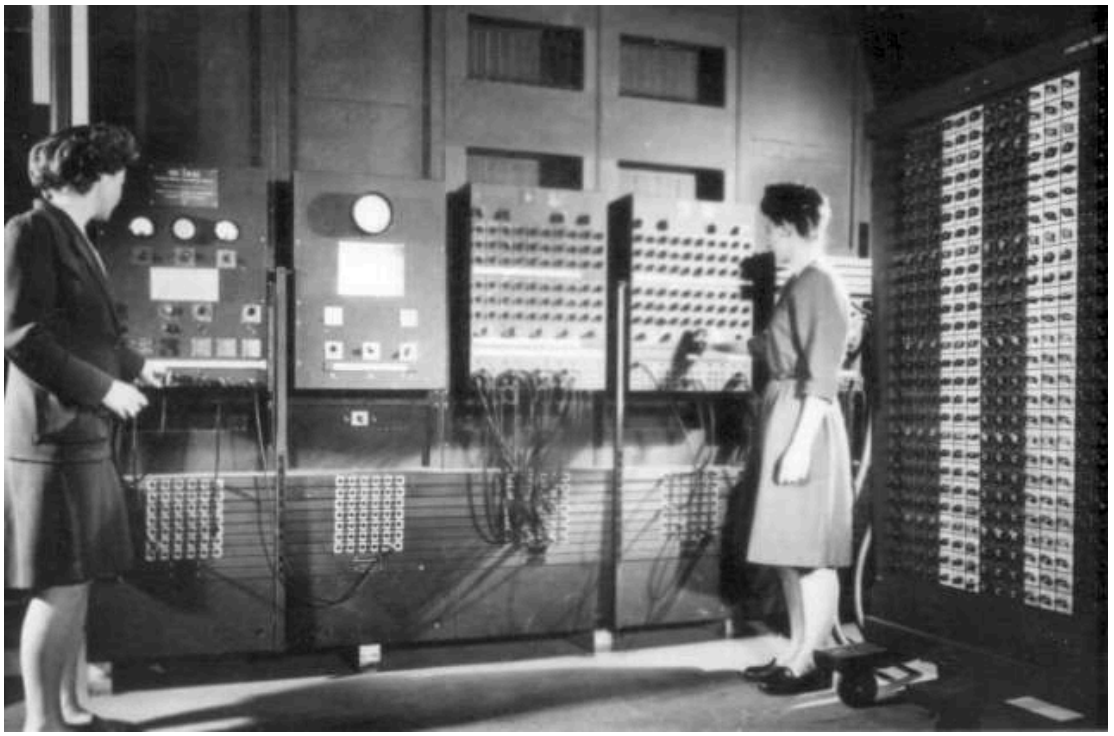


Figura 0.10. Fotografía del ENIAC (tomada de Wikipedia).

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)***[14] 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

[14] 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

crear y ejecutar algoritmos cuánticos para realizar operaciones aritméticas, entre otras aplicaciones.

Preguntas

1.

Ejercicios

Capítulo 2. 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 funciones[15].

Esta máquina opera a través de **instrucciones**, las cuales han sido 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 épocas[16]

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 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

[15] 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.
[16] Imagen tomada de: Computador antiguo y moderno - rompecabezas en línea

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.

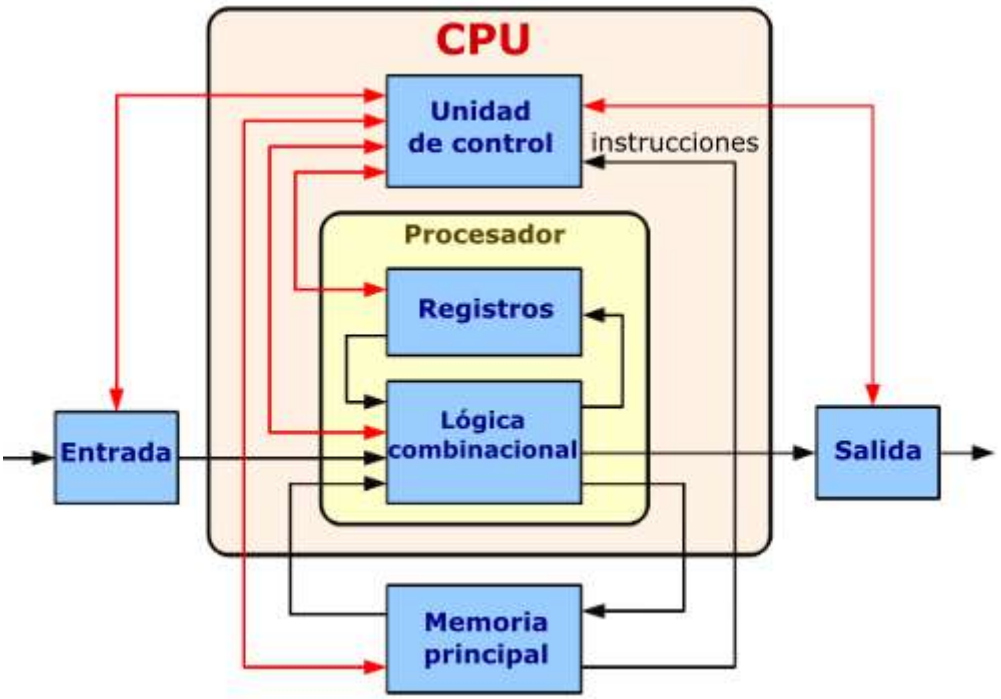


Figura 1.2. Funcionamiento de la CPU[17]

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 (1nm = 10^{-9}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

[17] Imagen tomada de: Diagrama del funcionamiento de la CPU

- 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	0	1	1
0	0	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	0	0
1	0	1	1	1	0	0	0	1	1	0	1	0	1	1	0	0	1	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*.[18]

Ejemplo
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

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*

[18] En este curso asumiremos los nombres de campos sensibles a los caracteres

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:

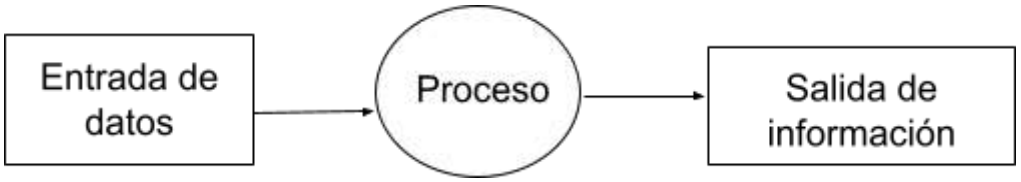


Figura 1.4. Representación gráfica del proceso algorítmico

Algoritmos “cualitativos”

Son muchos los procesos/problemas que pueden ser solucionados por algoritmos, sí 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 pagarlo 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 los sujeto
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**.[19]

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:

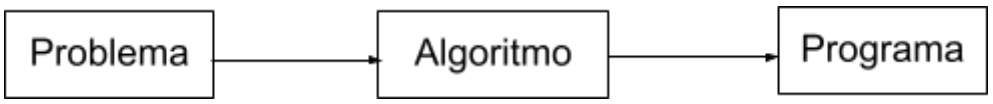


Figura 1.5. Representación gráfica del proceso de la programación

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

[19] 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*"

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 inglés[20], 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.

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

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
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
- Python
- 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 **`numero`** 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` = 3×10^{10}, `5.4E-5` = 5.4×10^{-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.

Tabla de caracteres ASCII[21]

ASCII (acrónimo inglés de *American Standard Code for Information Interchange* — Código Estándar Estadounidense para el Intercambio de Información—), es un código de caracteres basado en el alfabeto latino. Fue creado en 1963 por el Comité Estadounidense de Estándares (ASA, conocido desde 1969 como el Instituto Estadounidense de Estándares Nacionales, o ANSI) como una evolución de los conjuntos de códigos utilizados en telegrafía.

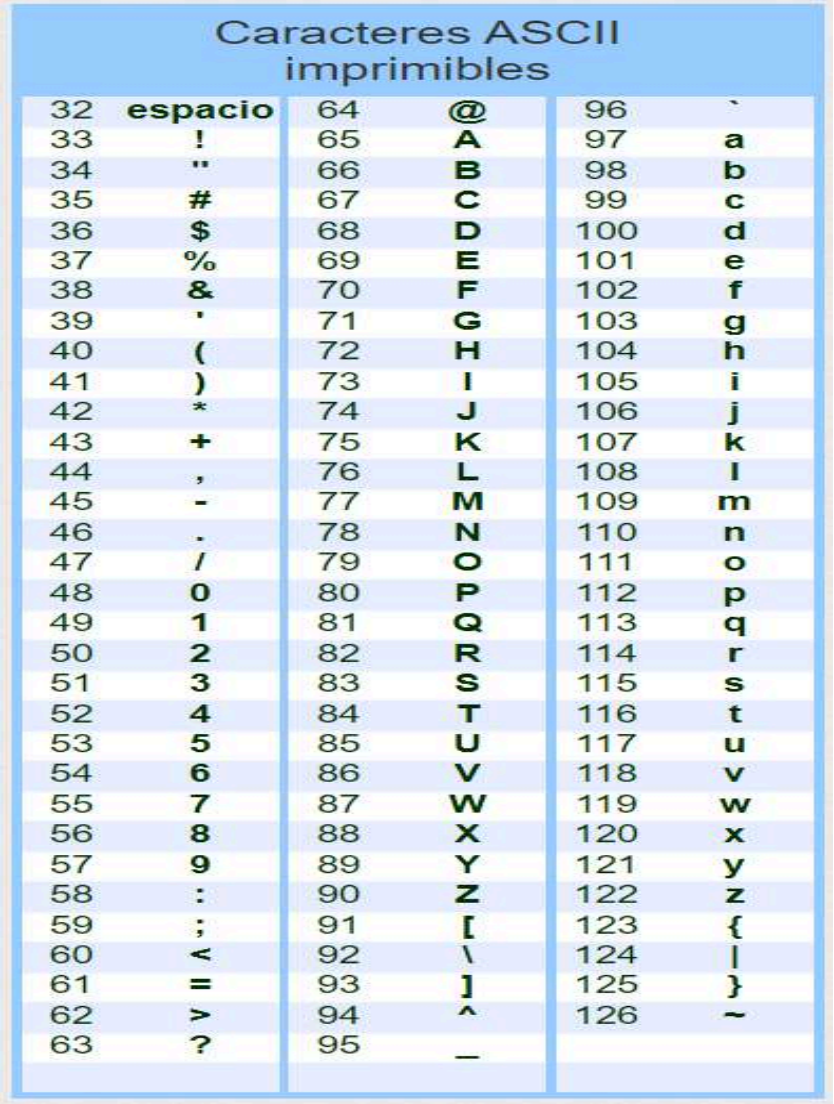
En el estándar ASCII cada carácter del alfabeto latino es representado numéricamente, ya sea en decimal u otra base numérica.

En decimal, los códigos del 32 al 126 se conocen como caracteres imprimibles, y representan letras, dígitos y algunos símbolos especiales.

El tratamiento de los caracteres no imprimibles y los del código extendido (superior a 126) considerados como caracteres especiales en general, suelen ser más difíciles de tratar, y es por ello que en distintos sistemas que requieren identificación encontramos que una de las restricciones es crear contraseñas y nombres de usuario que omitan varios de estos caracteres especiales, ya que de lo contrario no se podrá proceder a la creación de cuentas y emitir credenciales de acceso válidas.

La siguiente figura muestra los caracteres imprimibles del código ASCII[22]

[21] Ver más acerca del código ASCII en ASCII - Wikipedia, la enciclopedia libre
[22] Imagen tomada de: https://workshops.nuevofoundation.org/es/secret-messages/activity-5/



Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

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`***)

Tipos de datos primitivos en algunos lenguajes de programación

Los tamaños en general son muy similares en los distintos lenguajes de programación y dicha característica también depende de las características de la máquina. En los computadores comerciales la mayoría de lenguajes manejan los mismos tamaños para los tipos que implementan. El tipo cadena no está presente en el listado, ya que las cadenas de caracteres son tratadas como objetos de una clase (llamada **String**) en lenguajes como

Java y otros orientados a objetos, aunque sí en lenguajes como C/C++ conocido como el tipo char, donde las cadenas se definen como arreglos de este tipo de dato; cada carácter de una cadena ocupa un byte (1B) de memoria, esto es, es representado con una cadena de de ocho bits (8b).

- **byte**: representa un tipo de dato de 8 bits con signo. Puede almacenar valores numéricos en el rango de `-128 a 127`, incluyendo ambos extremos. Es útil para ahorrar memoria cuando se necesita almacenar valores pequeños.
- **short**: este tipo de dato utiliza 16 bits con signo y puede almacenar valores numéricos en el rango de `-32768 a 32767`. Se utiliza cuando se necesita un rango más amplio que el proporcionado por los bytes, pero aún se desea ahorrar memoria en comparación con los tipos de datos más grandes.
- **int**: es un tipo de dato de 32 bits con signo utilizado para almacenar valores numéricos. Su rango va desde `-2147483648 (-2 ^ 31) hasta 2147483647 (2 ^ 31 - 1)`. Es el tipo de dato más comúnmente utilizado para representar números enteros.
- **long**: este tipo de dato utiliza 64 bits con signo y puede almacenar valores numéricos en el rango de `-9223372036854775808` ($(-2)^{63}$) a `9223372036854775807` ($2^{63} - 1$). Se utiliza cuando se necesitan números enteros muy grandes.
- **float**: es un tipo de dato diseñado para almacenar números en punto o coma flotante con precisión simple de 32 bits. Se utiliza cuando se requieren números decimales con un grado de precisión adecuado para muchas aplicaciones. Está en un rango de `1.4 * 10 ^ −045 a 3.4 * 10 ^ 038`.
- **double**: este tipo de dato almacena números en coma flotante con doble precisión de 64 bits, lo que proporciona una mayor precisión qué el tipo float. Se usa en aplicaciones que requieren una alta precisión en cálculos numéricos. Está en un rango de `4.9 * 10 ** −324 a 1.8 * 10 ** 308`.
- **boolean**: sirve para definir tipos de datos booleanos que pueden tener solo dos valores: **`true`** o **`false`**. Aunque ocupa sólo 1 bit de memoria, generalmente se almacena en un byte completo por razones de eficiencia.
- **char**: es un tipo de datos que representa un carácter unicode sencillo de 16 bits. Se utiliza para almacenar caracteres individuales, como letras o símbolos en diferentes lenguajes y conjuntos de caracteres.

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 ∧ **	Alta - media
Raíz cuadrada	$\sqrt{x}$ |/ $raiz2(x)$ $sqrt(x)$	
Multiplicación	× * . ()()	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 div 2 = 8
c. 48 mod 4 = 0; 48 \ 4 = 12
d. 57 % 6 = 3; 57 // 6 = 9
e. 49 mod 5 = 4; 49 // 5 = 9
f. 9 mod 20 = 9; 9 \ 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	$\neq$ $<>$ $!=$	
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. $8 \neq 9 \rightarrow (V)$
b. $9 \geq 9 \rightarrow (V)$
c. $7 \neq 14 \div 2 \rightarrow (F)$
d. $9 \times 2 \leq 50 \div 10 \rightarrow (F)$
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	¬	~	–	!	*no*	*not*	Alta ↓ (mayor a menor)
Conjunción	∧	&&	•		*y*	*and*	
Disyunción	∨	||	+		*o*	*or*	
Disyunción exclusiva	V	⊕	*W*	*'o bien'*	*xor*	*eor*	

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 lógicos[23]

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

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

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 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: (←) 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:

$$constante \leftarrow valofijo$$
$$constante = valofijo$$
$$variable \leftarrow expresión$$
$$variable = expresión$$

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. $a * b + 3 * a * c \wedge 3$
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. $(a - 2 * b + 3 * c) / raizc(2);\ donde\ raizc()\ es\ una\ función$
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.

   $a >= 0 \,\&\&\, b <> (4 + 2 * a * (b \wedge 3)) \,||\, (!\,(a + 2 < b) \,\&\&\, (-9 = c))$

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 ***f*uertemente 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.

Comentarios de una línea

La forma para especificar comentarios que utilizaremos en algoritmia, será usando dos símbolos de **barra oblicua** (***slash***) **//**, y esto significa que todo lo que continúe hacia la derecha de dichos símbolos serán ignorados en la compilación y ejecución; esta forma es la utilizada en el lenguaje C/C++ y derivados de éste; también utilizaremos para comentarios de una línea el carácter **numeral** o **almohadilla #** utilizado en Python y otros entornos. En el ejemplo anterior ya ilustró el uso de comentarios, como puede observarse en la tercera línea (instrucción).

Sintaxis

```
// Texto del comentario
```

O también

```
# 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
```

Comentarios de varias líneas

Los lenguajes de programación también ofrecen sintaxis específica para crear comentarios que ocupen varios renglones del código, evitando así tener que comentar línea a línea. En este documento se usará la sintaxis implementada en el lenguaje C/C++ y derivados de éste encerrando el texto comentado entre los caracteres **barra oblicua asterisco - asterisco barra oblicua** (***slash asterisk***).

Sintaxis

```
/*
 Texto del comentario
 ocupando varias líneas
*/
```

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*[24]

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 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.`

[24] Puede consultar más sobre esta notación en: Camel case - Wikipedia, la enciclopedia libre

Notación *snake case*[25]

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 húngara[26]

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.

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

[25] Puede consultar más sobre esta notación en: Snake case - Wikipedia, la enciclopedia libre
[26] Puede consultar más sobre esta notación en: Convenciones de codificación de Windows - Win32 apps | Microsoft Learn y Notación húngara - Wikipedia, la enciclopedia libre

- Puntos suspensivos …: continúa de la misma forma que antecedan

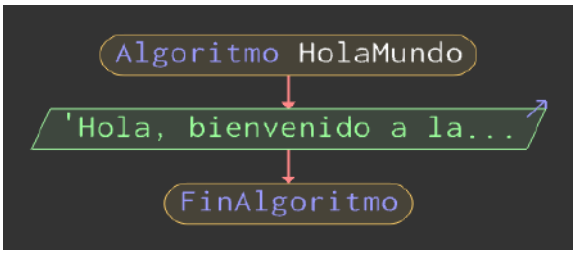
Problemas resueltos

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 libre[27]:



Pseudocódigo:

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

Diagrama Nassi-Schneiderman:

Inicio
Imprimir "Hola, bienvenido a la programación"
Fin

Pseudo programa:

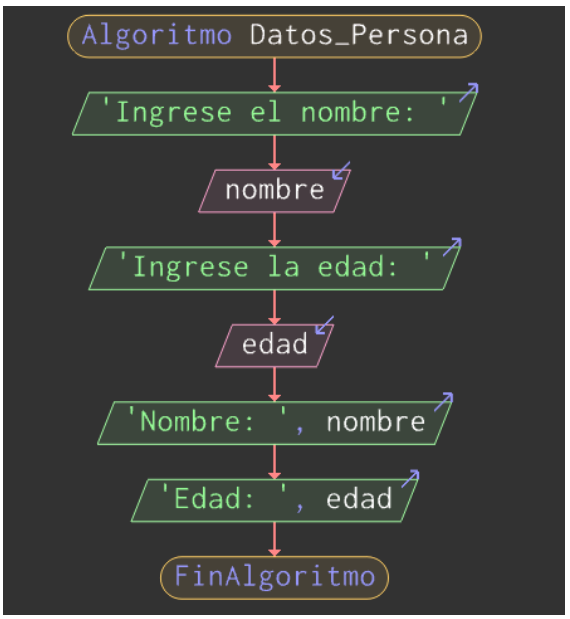
```
Algoritmo HolaMundo
      Imprimir "Hola, bienvenido a la programación"
FinAlgoritmo
```

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:

[27] 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.



Pseudocódigo:

```
Inicio
Leer nombre, edad
Imprimir nombre, edad
Fin
```

Diagrama Nassi-Schneiderman:

Inicio
Leer nombre, edad
Imprimir nombre, edad
Fin

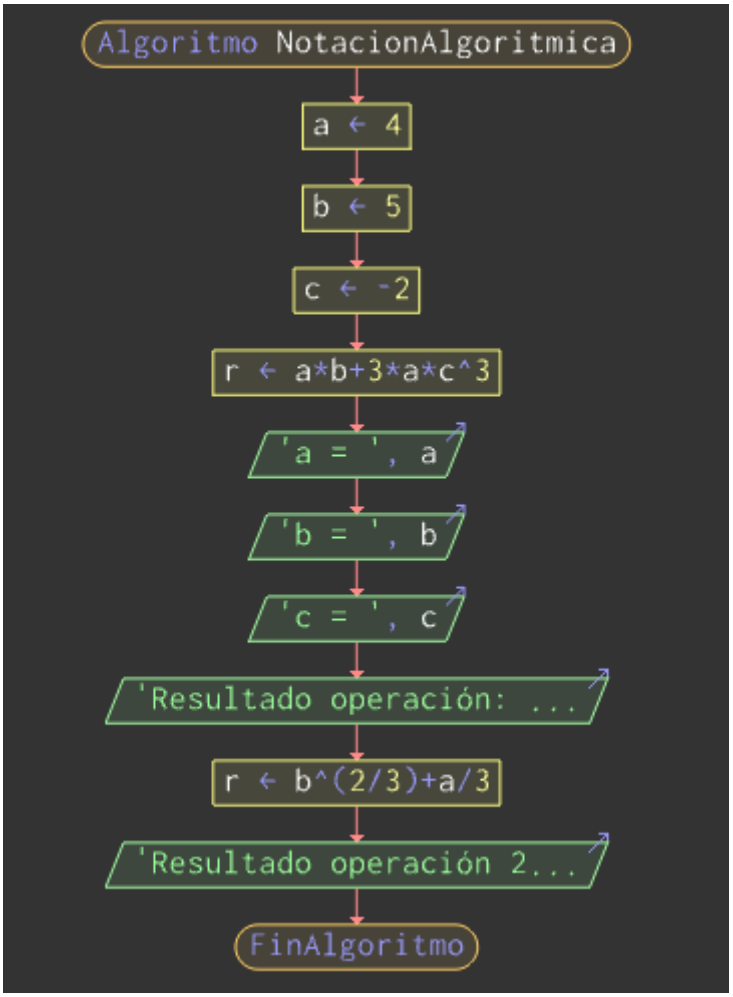
Pseudo programa:

```
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:

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
```

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 = Verdadero, q = Verdadero y r = Falso, entonces el resultado de la expresión: p Y No(q O (r Y No(p))) ¿qué da cómo resultado?
16. (F o V) La expresión: Falso O No(No("z" > "m")) 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. $\sqrt{ab - a} + 5d \div c - a^4be$
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 <= 5ae - b/2
    d. d <> 2ea
    e. 3/c >= 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 cuanto 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

Capítulo 3. 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**.

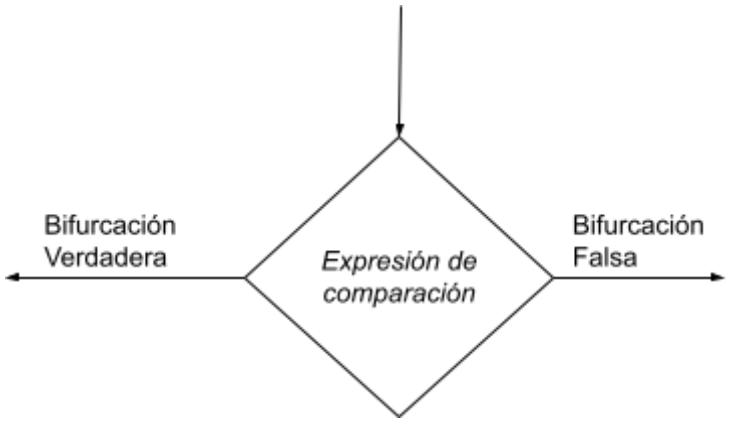
Condicional simple

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

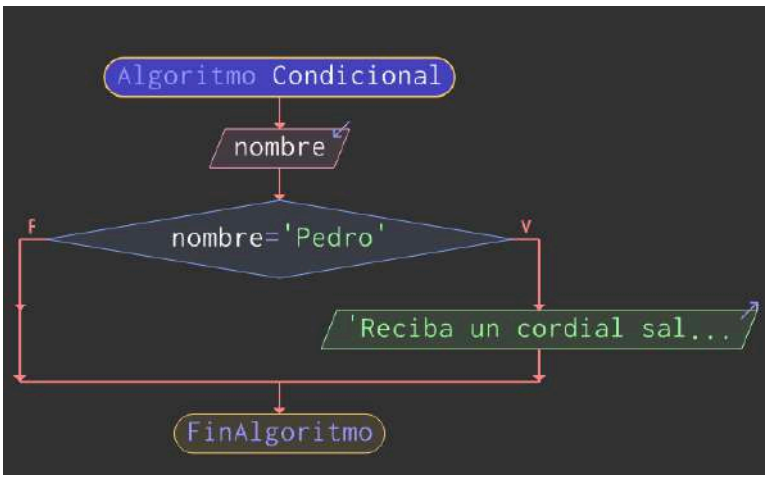
```
Si expresión_de_comparación [Entonces]
    Bloque de instrucciones si expresión_de_comparación es verdadera
FinSi
```

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:

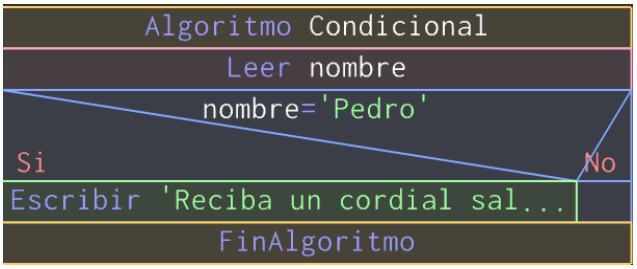


Pseudocódigo:

```
Inicio
Carácter: nombre
Leer 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.

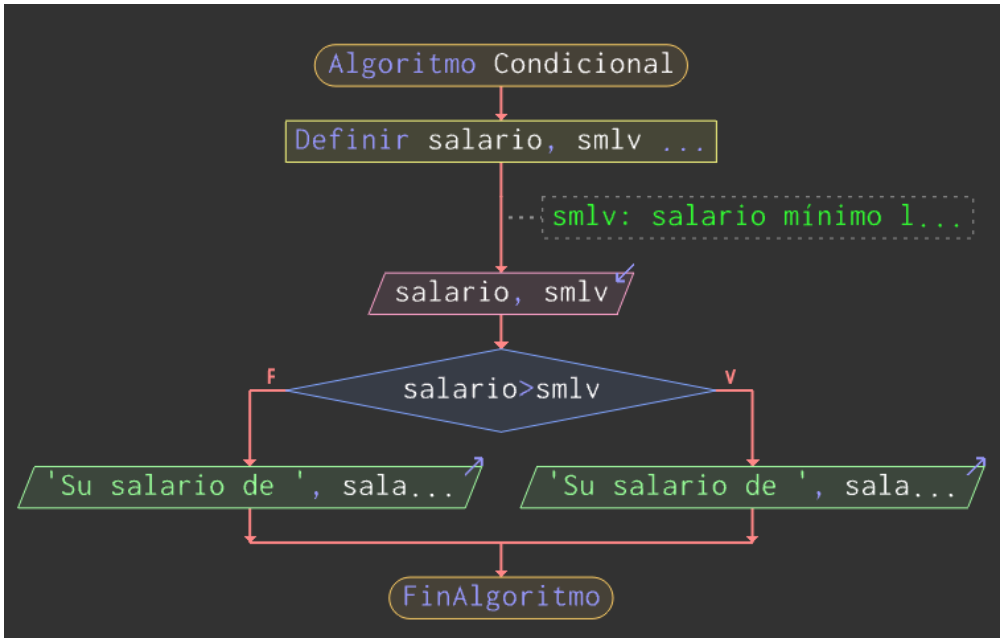
Sintaxis

```
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:

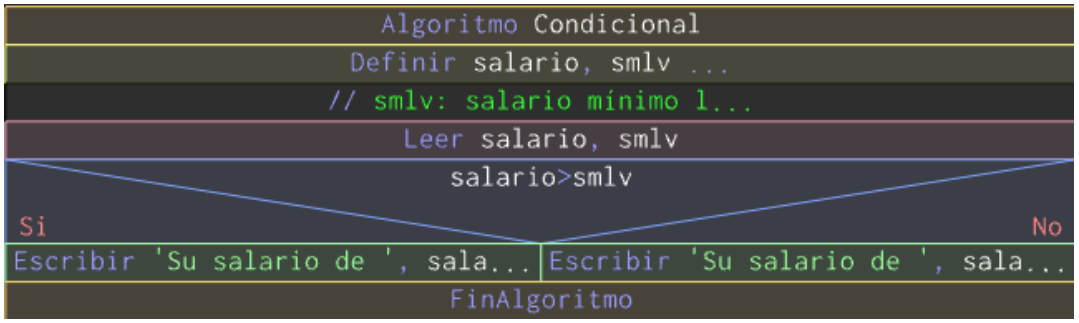


Pseudocódigo:

```
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:



Operador condicional ternario

Es un operador que tiene tres operandos. Este operador se usa como una forma de simplificar el condicional compuesto, esto es, la instrucción `Si-SiNo` (`if-else`) cuando se evalúa una expresión de comparación (booleana) para devolver un valor u otro del mismo tipo, en función de que la expresión booleana se evalúe como `Verdadera` (`true`) o `Falsa` (`false`).

Sintaxis

```
expresión_de_comparación ? valor si es verdadera : valor si es falsa
```

Ejemplo 2.2

Leer el salario de un empleado y el salario mínimo legal vigente (smlv). Calcule una retención del 10% si el salario del empleado supera en tres veces el smlv o del 5% en caso contrario.

Pseudocódigo:

```
Inicio
Reales: salario, retencion, smlv //smlv: salario mínimo legal vigente
Leer salario, smlv
Si salario > 3 * smlv Entonces
    retencion = salario * 0.1
SiNo
    retencion = salario * 0.05
FinSi
Imprimir “Retención: ”, retencion
Fin
```

Usando el operador condicional ternario, toda la sentencia condicional del ejemplo se reduce a una línea:

Pseudocódigo:

```
Inicio
```

```
Reales: salario, retencion, smlv //smlv: salario mínimo legal vigente
Leer salario, smlv
retencion = salario > 3 * smlv ? salario * 0.1 : salario * 0.05
Imprimir “Retención: ”, retencion
Fin
```

Nota

La sintaxis usada en el pseudocódigo que se implementa en este documento para el operador condicional ternario está basada en la utilizada por el lenguaje C/C++ y distintos lenguajes de programación provenientes de éste, aunque vale aclarar que Python utiliza una sintaxis diferente, pero con el mismo objetivo.

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.

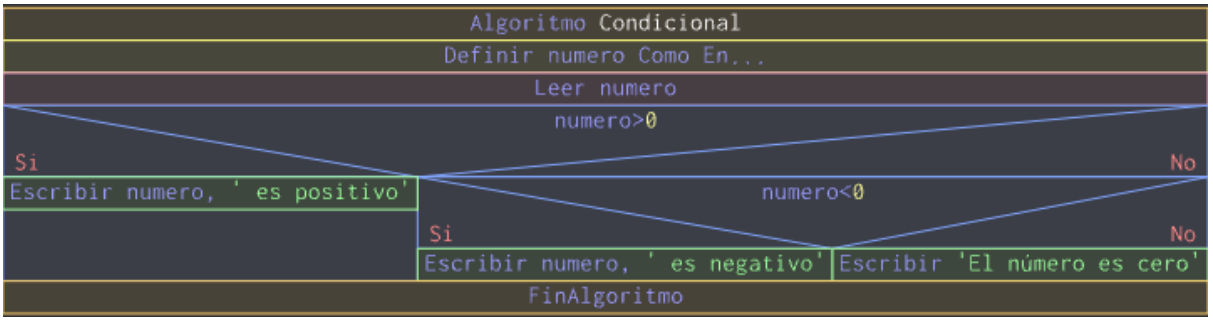
Sintaxis

```
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

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

Diagrama Nassi-Schneiderman:

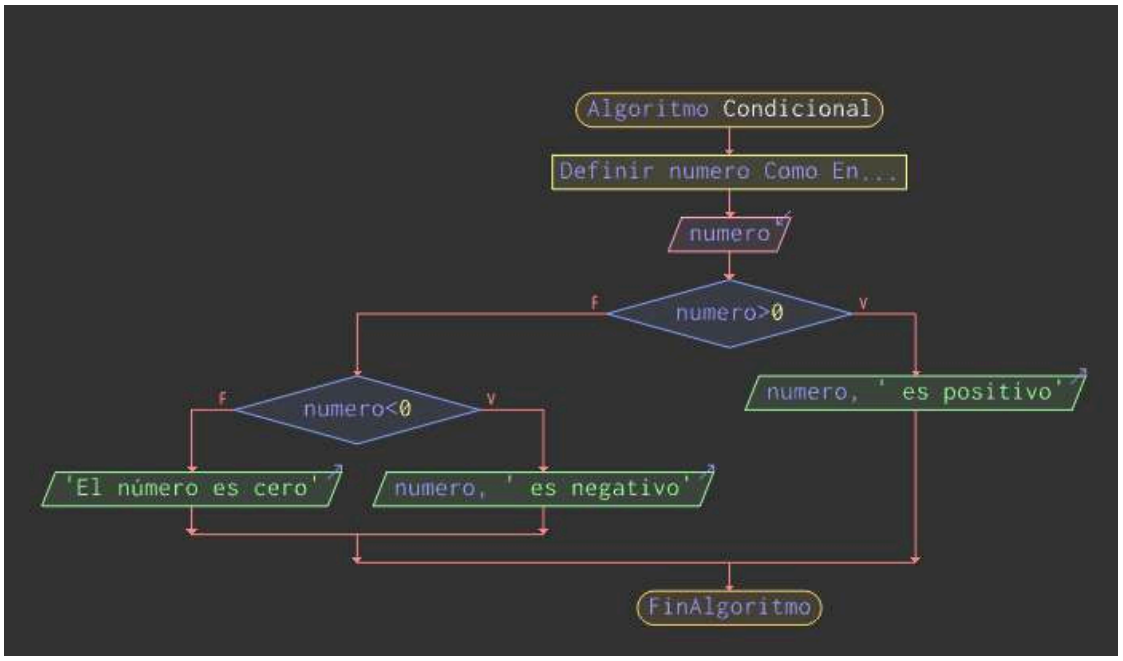


Pseudocódigo:

```
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 libre:



Condicional anidado Si - SiNo - Si (if - else - if)

Esta sintaxis es utilizada para abreviar y mejorar la escritura de condicionales anidados. Permite evaluar múltiples condiciones en secuencia para ejecutar un bloque de código específico; evalúa la primera condición `Si` (`if`), si es falsa, pasa a la siguiente `SiNo Si` (`else if`), y así sucesivamente, hasta encontrar la primera condición verdadera o ejecutar opcionalmente el bloque `SiNo` (`else`) final si ninguna se cumple, siendo una forma más limpia de manejar decisiones múltiples que el anidamiento típico.

Nota

Los lenguajes de programación manejan formas como `if else if`, `if elseif` y `if elif` como lo hace Python en su implementación. En PSeInt se implementa de igual forma a la que se ilustra en la siguiente sintaxis y ejemplo.

Sintaxis

```
Si expresión_de_comparación1 [Entonces]
```

```
        Instrucciones si expresión_de_comparación1 es verdadera
SiNo Si expresión_de_comparación2 [Entonces]
    Bloque de sentencias si expresión_de_comparación2 es verdadera]
...
[SiNo
    Bloque de sentencias si las expresiones_de_comparación son falsas]
FinSi
```

Ejemplo 2.3
Leer un número. Determinar si es positivo, negativo o cero.

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
Fin
```

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.

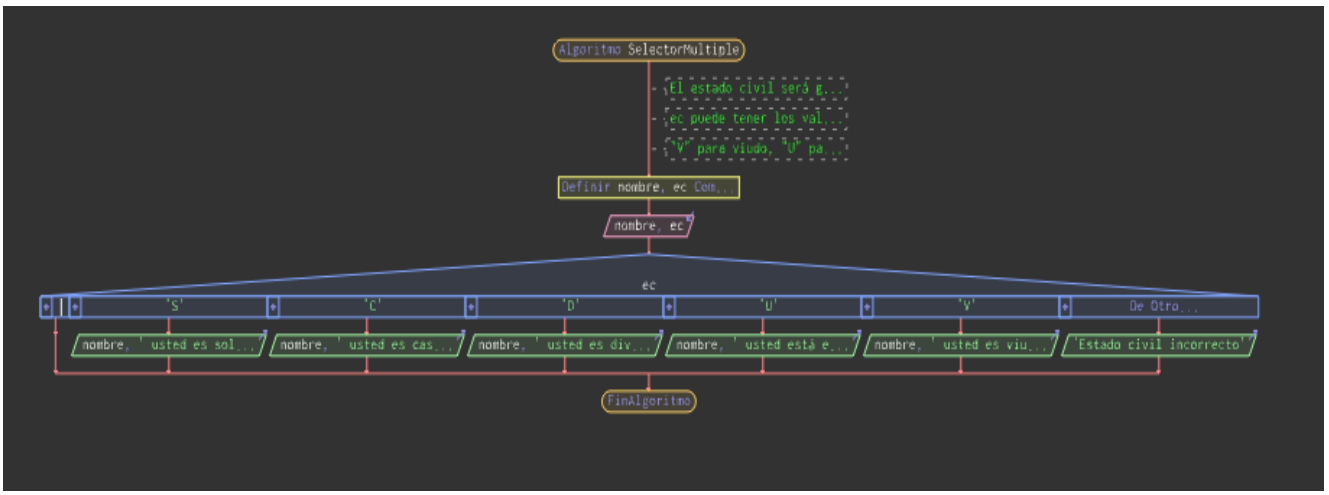
```
Según expresión [Hacer]
    Caso valor_1:
        Instrucciones si valor_1 coincide con expresión
    Caso valor_2:
        Instrucciones si valor_2 coincide con expresión
    …
    Caso valor_n:
        Instrucciones si valor_n coincide con expresión
    [DeOtroModo:
        Instrucciones si ningún 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:

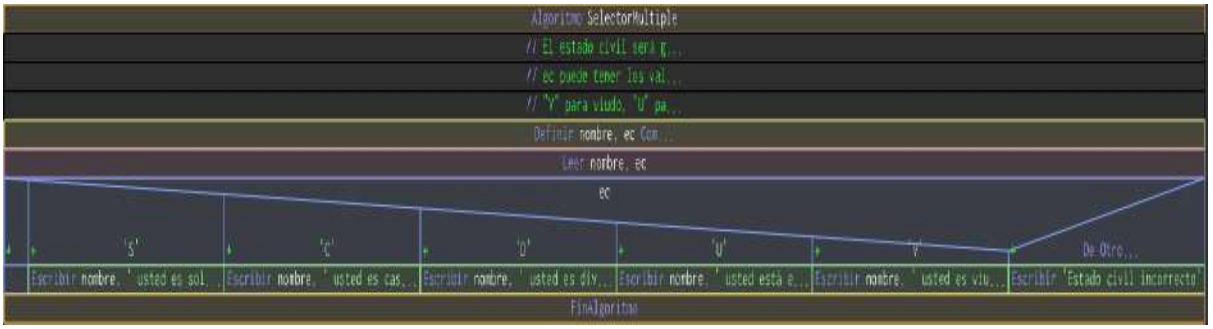


Pseudocódigo:

```
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 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
```

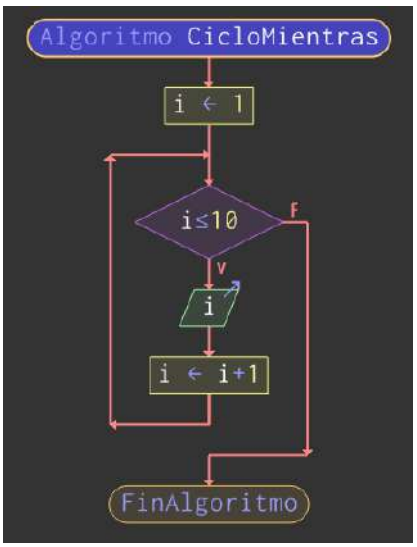
O también de esta forma

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.

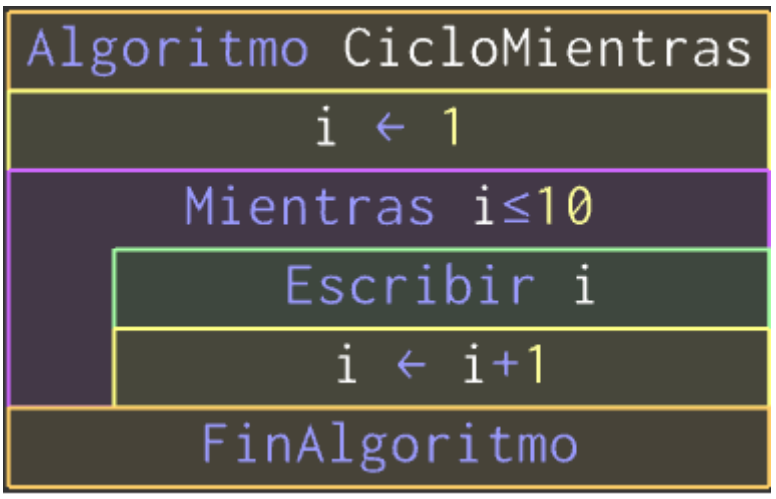
Diagrama libre:



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 Gauss[28] 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.:

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

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

Prueba de escritorio:

n	suma	Salida
5	15	15

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
Enteros: 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.

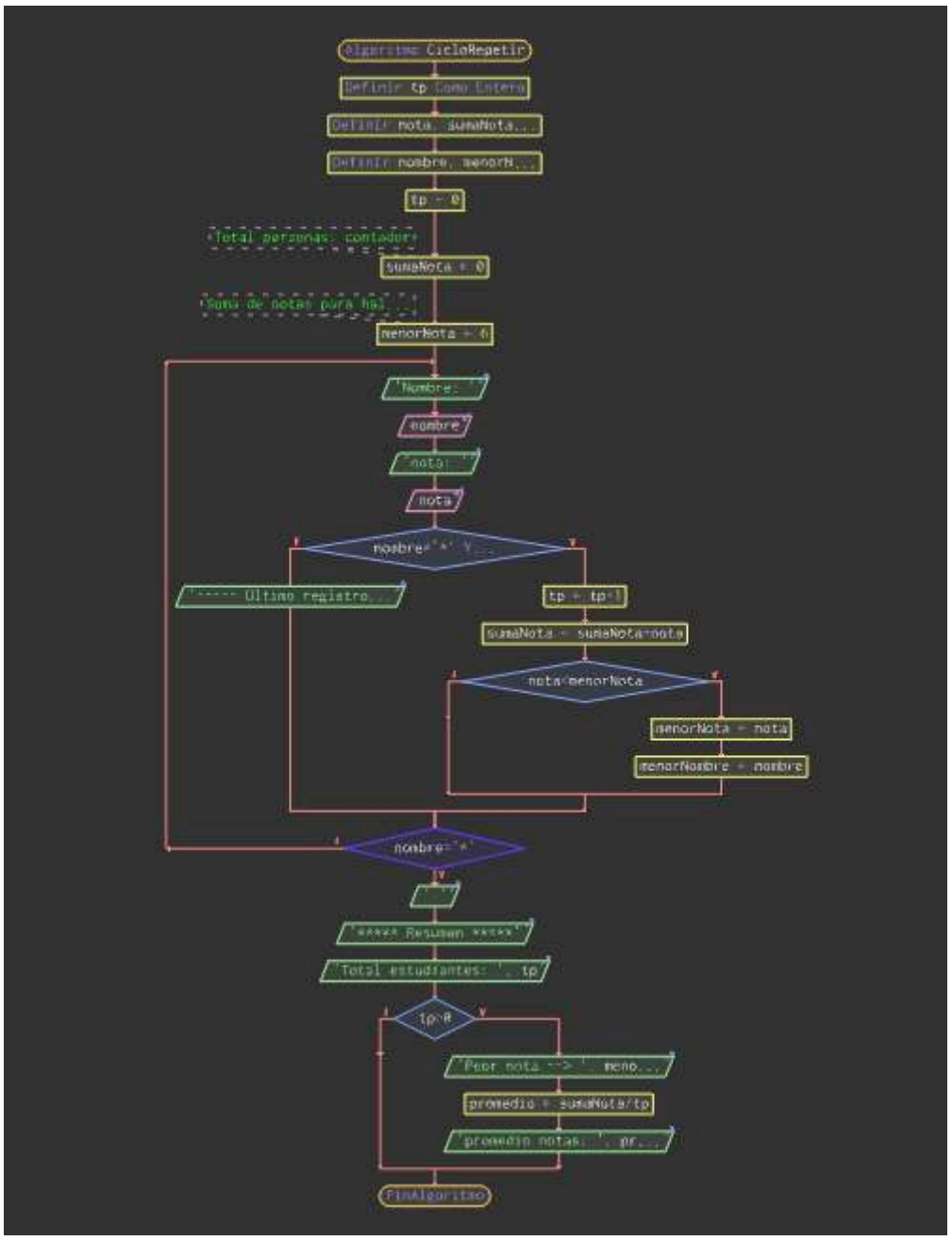
Sintaxis

```
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:

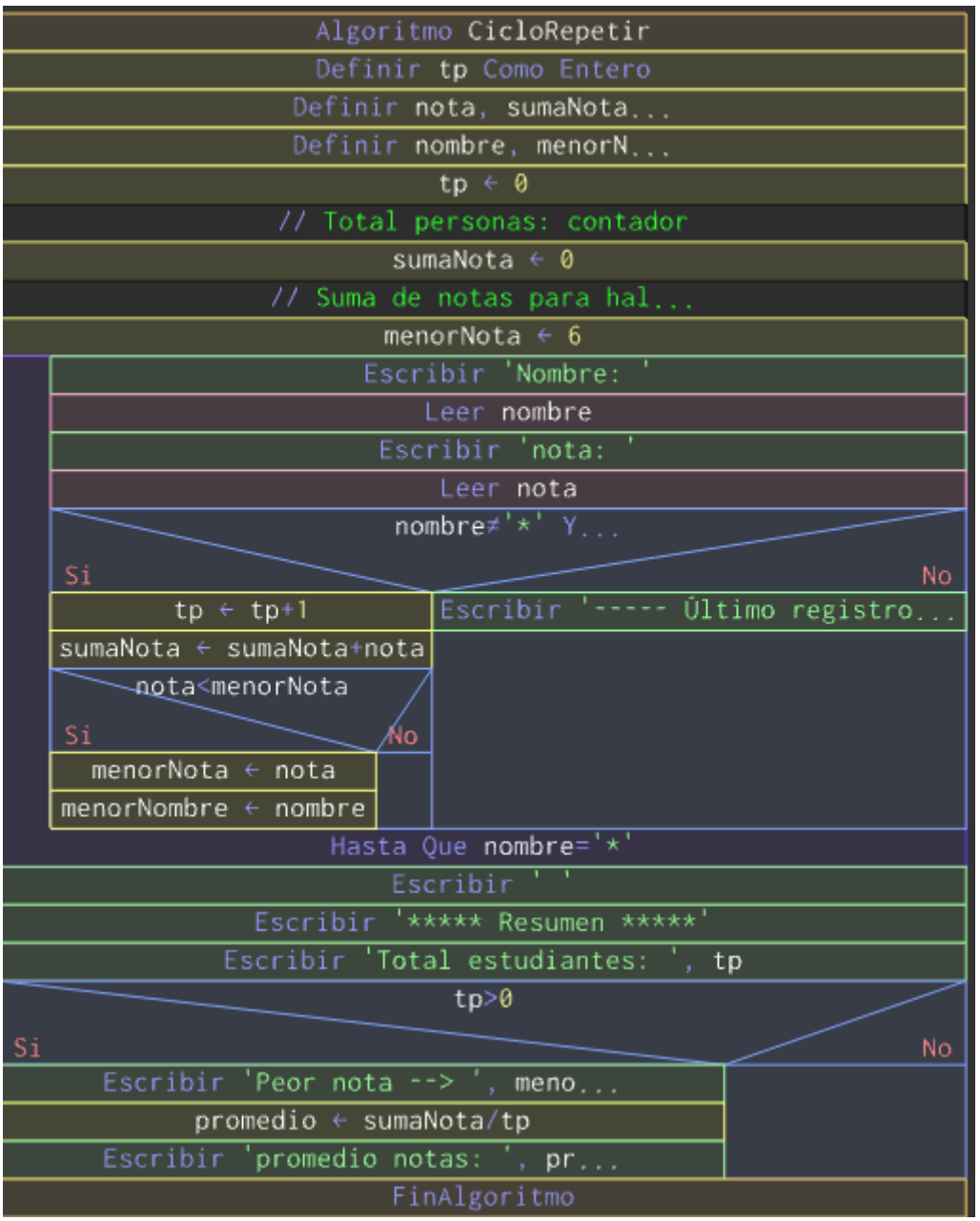


Pseudocódigo (PSeInt):

```
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

Leer 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 (PSeInt):

```
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)

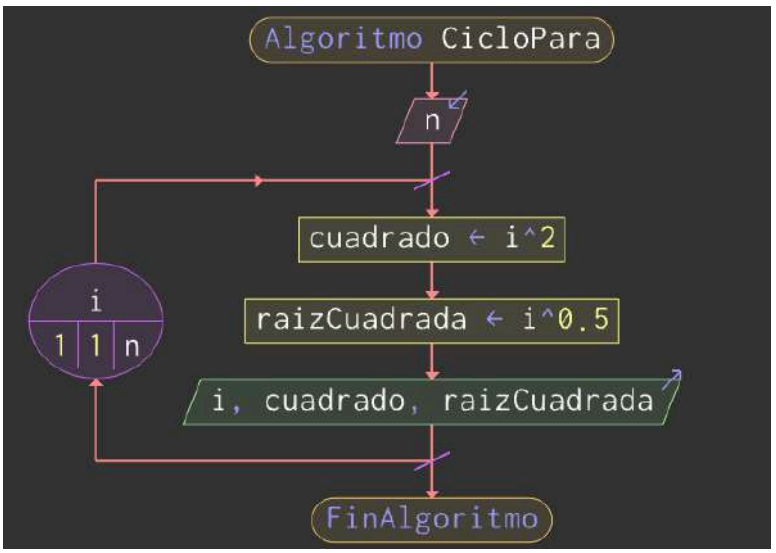
Notas

- 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.

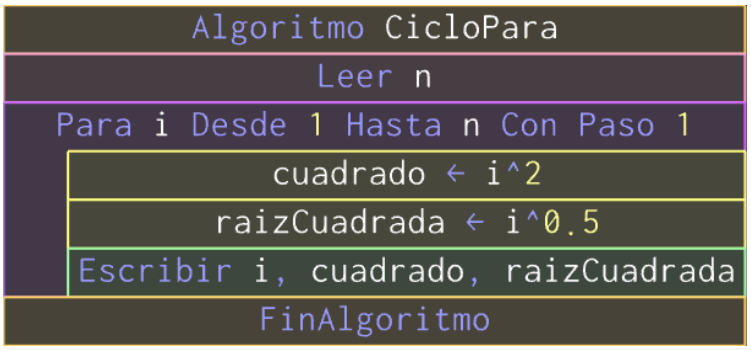
Diagrama libre:



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) * ... * 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
```

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
Enteros: 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 el siguiente capítulo se ilustra el uso de esta sentencia muy utilizada en funciones.

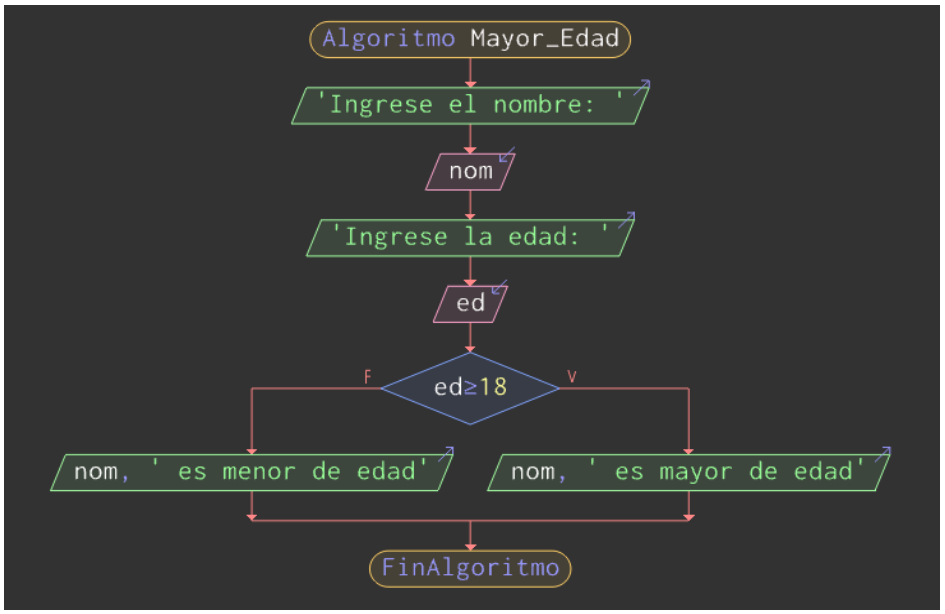
Problemas resueltos

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.

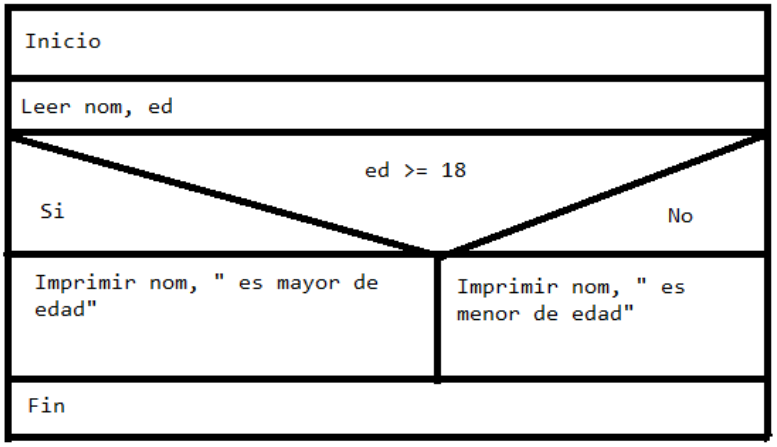
Diagrama libre:



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:

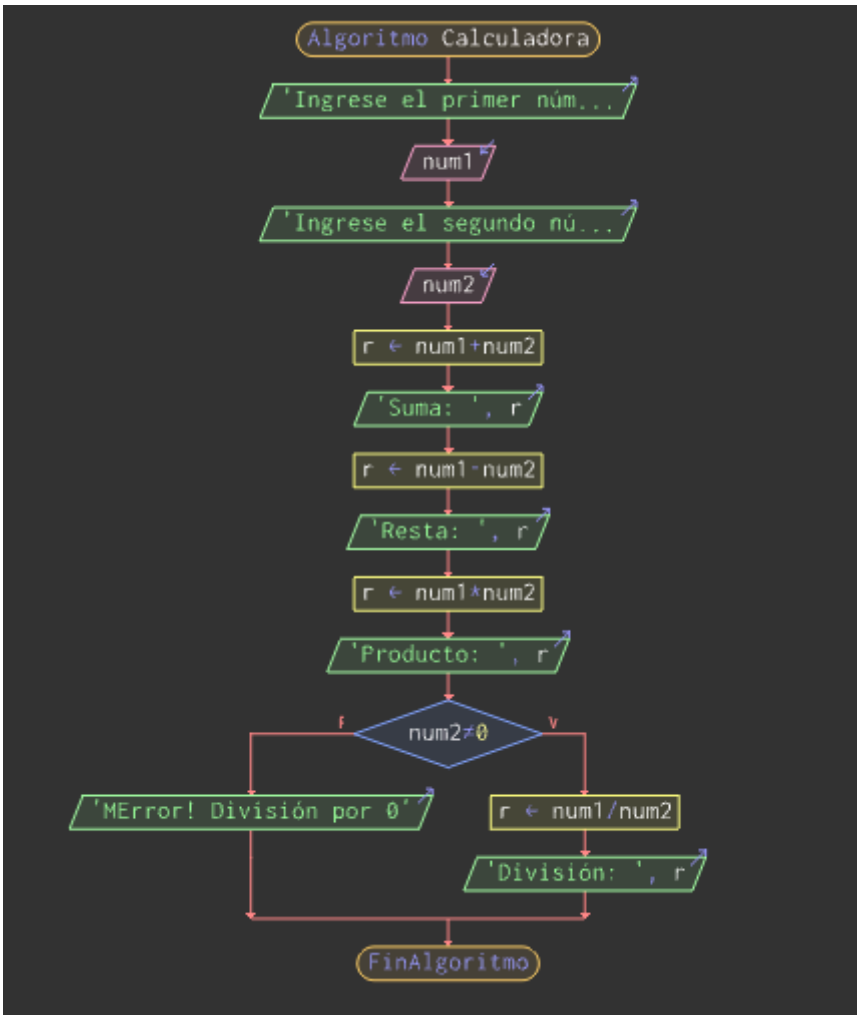
```
Algoritmo Mayor_Edad
```

```
      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
Enteros: 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 $(Un\ par\ se\ define\ como\ 2k\ y\ un\ impar\ como\ 2k\ +\ 1,\ k \in Z)$
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 Armónica[29] `1 + 1/2 + 1/3 + 1/4 + ... + 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) * ... * 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,...`
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.

[29] 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)

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

Capítulo 4. 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
`abs(x)`	valor absoluto de x	`abs(-8) //devuelve 8`
`trunc(x)`	valor truncado de x	`trunc(2.6) //devuelve 2`
`redondear(x)`	valor redondeado de x	`redondear(2.6) //devuelve 3`
`rc(x), raiz(x)`	raíz cuadrada de x	`raiz(9) //devuelve 3`
`sen(x)`	seno trigonométrico en radianes	`sen(0) //devuelve 0`
`cos(x)`	coseno trigonométrico en radianes	`cos(0) //devuelve 1`
`tan(x)`	tangente trigonométrica en radianes	`tan(0) //devuelve 0`
`asen(x)`	arcoseno de x	
`acos(x)`	arcocoseno de x	
`atan(x)`	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)`
`aleatorio(x, y)`	número aleatorio (al azar) entre x e y	`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 a mayúsculas	`mayusculas("abc") //devuelve "ABC"`
`minusculas(cadena)`	convierte cadena a minúsculas	`minusculas("ABC") //devuelve "abc"`
`subCadena(cadena, inicio, num_caract)`	extrae una cadena de otra comenzando desde *inicio*, extrayendo *num_caract* 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
```

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 cómo **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*[30] 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.

[30] 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.

- 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 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°) = 0.5`, `sen(-30°) = 0.5`, `cos(60°) = 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

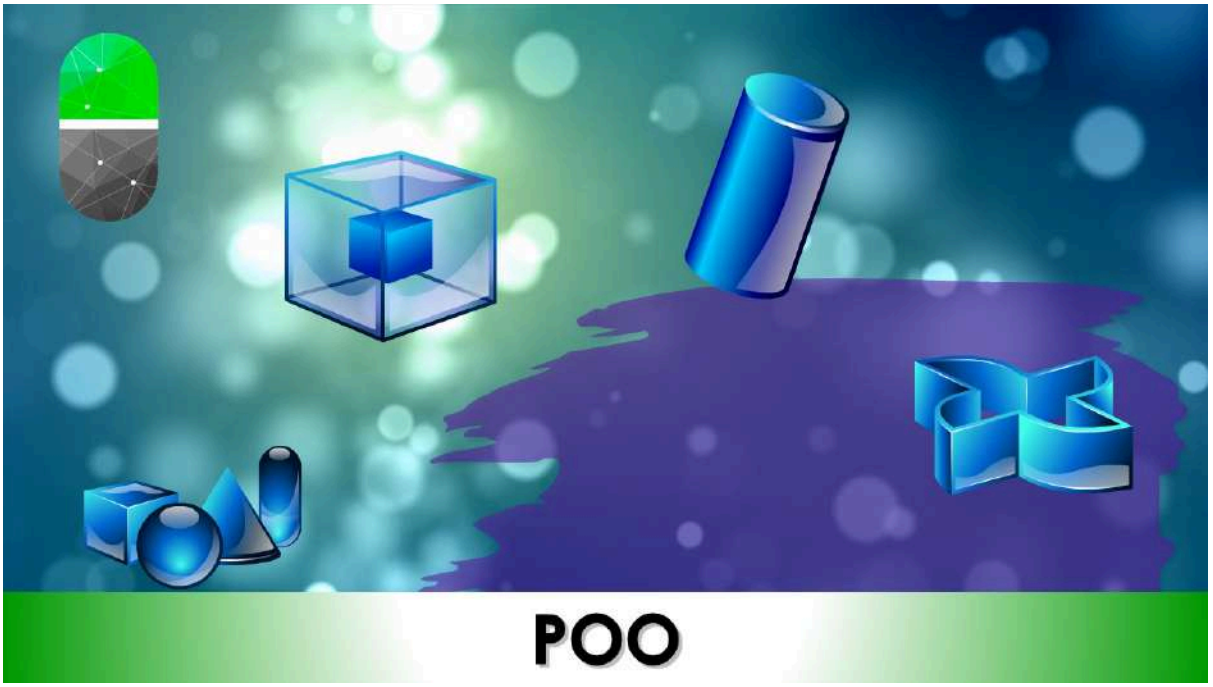
1. Cree un procedimiento para intercambiar el valor de dos variables. Las variables deben ser ingresadas desde un procedimiento. Mostrar los valores antes y después del intercambio en otro procedimiento.
2. Se tiene un grupo de estudiantes del TdeA, de los cuales se lee la nota del parcial y del final, 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 `2`$\boldsymbol{\pi}$, cree dos funciones para calcular el seno y el coseno trigonométricos en radianes usando la serie de Taylor para 100 términos: $sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$; $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$
6. Calcule el valor de **π** (**pi**) usando la **serie de Leibniz**[31] para un n grande: π/4 = `1 - 1/3 + 1/5 - 1/7 + 1/9 - ... + 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,` $\boldsymbol{\theta}$) a coordenadas cartesianas: `x = rcos(`$\boldsymbol{\theta}$`), y = rsen(`$\boldsymbol{\theta}$`)`, y otro procedimiento que convierta de coordenadas cartesianas a polares: $r = \sqrt{x^2 + y^2}$; $\boldsymbol{\theta} = tan^{-1}\frac{y}{x}$
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”
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”

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

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 (aquellos 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: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$
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.



SEGUNDA PARTE. PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Capítulo 5. Introducción a la 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 s*ubgrupo, 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 muestran 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.

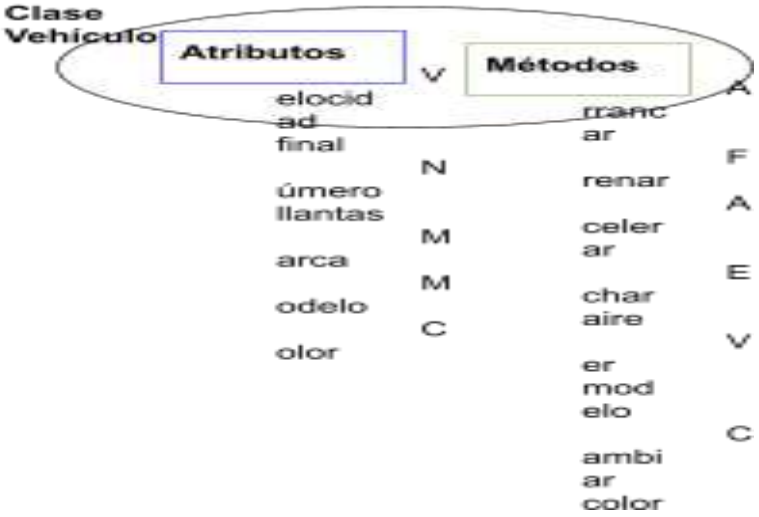


Imagen: representación de una clase para crear objetos de tipo Vehículo

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 creen
    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
```

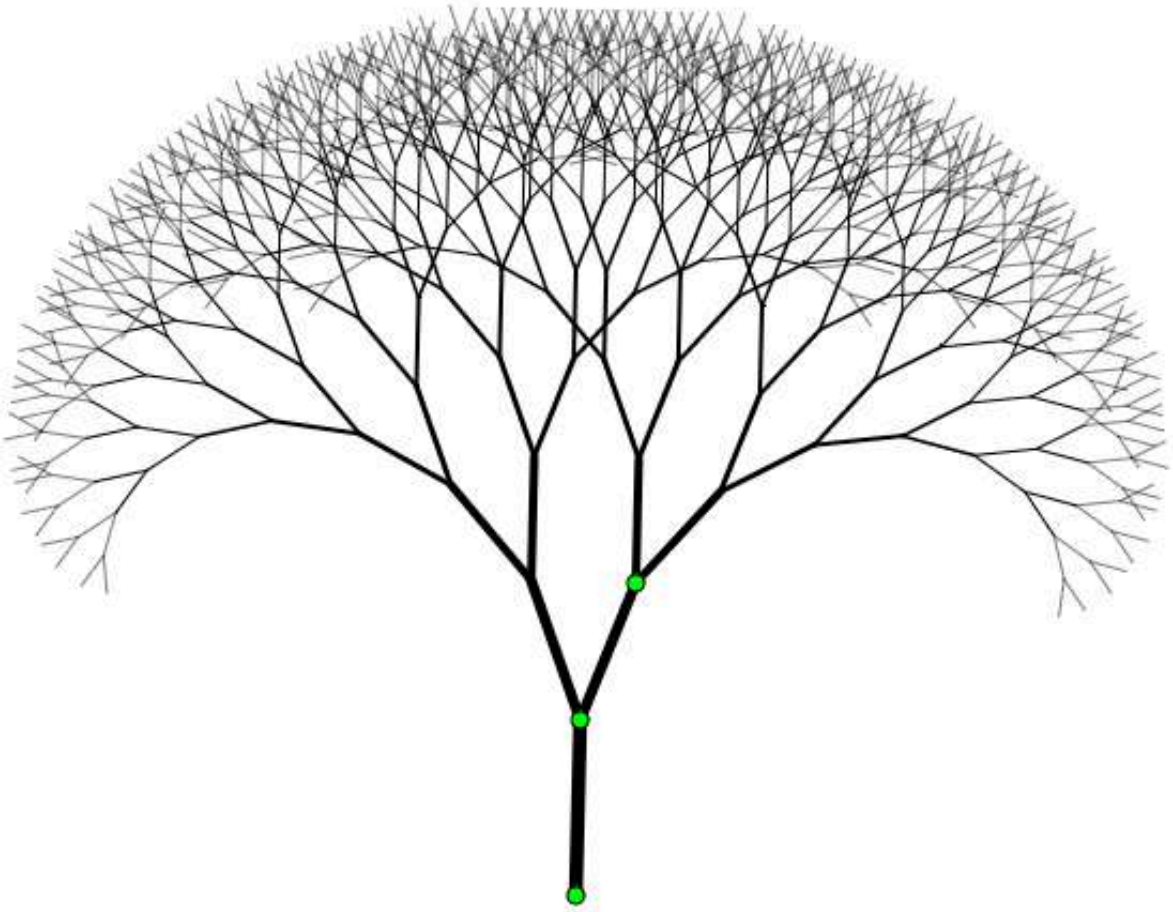
```
        SiNo
            Retornar Falso
        FinSi
    FinMetodo
FinClase

// 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



TERCERA PARTE. ESTRUCTURAS DE DATOS

Capítulo 6. Estructuras de datos fundamentales

Este capítulo comprende un repaso de conceptos que ya han sido vistos por los estudiantes en asignaturas como Lógica de Programación y Programación Orientada a Objetos, por lo que también puede considerarse un repaso y/o profundización en algunos aspectos relacionados con arreglos. Si el lector considera que se siente bien en dicha teoría, puede pasar sin problemas al capítulo 2, no sin antes dar una revisión general al material del capítulo con el fin de que esté seguro que ha estudiado anteriormente los temas aquí tratados.

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 (reales, enteros), lógicos y caracteres. En los **tipos estructurados**, tenemos varios tipos en dos categorías:

Estáticas

- Cadenas de caracteres
- Arreglos: vectores, matrices, arreglos multidimensionales
- Registros
- Conjuntos
- Archivos
- Pilas (tratadas como vectores)
- Colas (tratadas como vectores)

Dinámicas

- Pilas (tratadas como listas ligadas)
- Colas (tratadas como listas ligadas)
- 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 está basado en los tipos de datos primitivos, esto es, se forman a partir de los datos simples, así como posiblemente de otros datos compuestos.

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.

Cadenas de caracteres

Una **cadena de caracteres** es un conjunto de símbolos disponibles en un idioma, lengua, medio/dispositivo (como un computador), etc. y que se compone de letras (`a - z; A - Z` y posiblemente los carácteres de otros alfabetos) , caracteres numéricos (`0 - 9`) y símbolos especiales (`+, -, ¿, }, #, ”, ...`). A nivel de programación, una cadena se considera una **estructura de datos lineal estática** que almacena una secuencia de caracteres y se puede considerar como una **matriz** de caracteres; de ahí que lenguajes como C/C++ las traten como **vectores** (arreglos unidimensionales) en donde cada carácter ocupa una posición en el vector[32].

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 la informática y hasta muchos años después, el tratamiento de las cadenas de caracteres fue un asunto que traía bastantes dificultades a los programadores. Las funcionalidades de los lenguajes actuales facilitan la labor del desarrollador permitiendo la correcta manipulación de las cadenas de texto y concentrándose en la lógica del negocio que requiere solucionar y no en temas que deben resolver los lenguajes de programación.

Algorítmicamente, se definen las funciones más relevantes para ilustrar el tratamiento de cadenas, aunque los lenguajes disponen de una lista extensa para la manipulación de texto, cada una con sus propias especificaciones de implementación, pero en general, realizando las mismas funciones. A nivel de la lógica de programación, podemos definir las siguientes funciones que se muestran en la siguiente tabla y que son de común implementación.

Notas

- 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. En este curso en algunos casos algorítmicos, las cadenas se toman desde la posición uno.
- La mayoría de lenguajes son sensibles a los caracteres, esto es, distinguen entre mayúsculas y minúsculas, por lo que la variable “*a*” será diferente de la variable “*A*”. En este curso siempre consideraremos la sensibilidad de los caracteres.

Función	Descripción	Ejemplo
`longitud(cadena)`	longitud de la cadena	`longitud("hola") //devuelve 4`
`mayusculas(cadena)`	convierte cadena a mayúsculas	`mayusculas("abc") //devuelve "ABC"`

[32] El tema de vectores, matrices y arreglos en general se tratará más adelante en este capítulo.

minusculas(cadena)	convierte cadena a minúsculas	minusculas("ABC") //devuelve "abc"
subCadena(cadena, inicio, num_caract)	extrae una cadena de otra comenzando desde *inicio*, extrayendo *num_caract* 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")
caracterEn(cad, índice)	devuelve el carácter en posición índice. de la cadena cad. Si el índice es inválido, devuelve Nulo.	caracterEn("Hola", 2) //Devuelve o
caracterASCII(car)	Devuelve el valor ASCII del carácter car.	caracterASCII("a") //Devuelve 97
ASCIICaracter(valor_ASCII)	Devuelve el carácter correspondiente al valor ASCII especificado	ASCIICaracter(97) //Devuelve "a"
indiceEn(cad, sub_cad[, pos_ini[, pos_fin]])	Devuelve la posición de sub_cad dentro de cad; la búsqueda inicia en pos_ini y finaliza en pos_fin. Si se omiten los límites de búsqueda, comienza desde la primera posición hasta la última. Si la subcadena no encuentra, la función devuelve 0 (o -1 dependiendo en qué posición comienzan las cadenas)	indiceEn("Hola", "la", 1) //Devuelve 3
eliminaEspacios(cad)	Elimina los espacios al inicio y fin de la cadena cad	cad " Hoy es un día " eliminaEspacios(cad) //devuelve "Hoy es un día"

Nota

La comparación de cadenas se realiza con los operadores relacionales conocidos (= o ==, <> o !=, >, <, >=, <=), teniendo en cuenta que dicha comparación es alfabética en donde se tiene presente el valor ASCII de los caracteres.

Ejemplo 1.1

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
```

Cadenas de caracteres en Java

Java trata las cadenas de caracteres como **objetos** de la clase **String**, por lo que es posible acceder a sus métodos para manipular éstas[33].

Definir cadenas en Java

Se puede utilizar el constructor de la clase String o usar directamente un literal de cadena. Una cadena también puede ser leída desde el teclado usando el escáner, por ejemplo.

Ejemplo 1.2
Ilustración en la creación de cadenas en Java

Pseudo programa Java:

[33] La clase String de Java es una clase inmutable y no posee atributos, sólo dispone de métodos.

```
String literal = "Esto es un literal de cadena";

String keyboardInput = input.next();

String object = new String("Invocando al constructor de la clase String");
```

La clase String

La clase String de Java forma parte del paquete `java.lang` y es una clase inmutable que no posee atributos, sólo dispone de métodos; esto significa que una vez creado un objeto `String`, su valor no puede modificarse.

Algunos de los métodos más comunes se ilustran a continuación.

Longitud de una cadena de caracteres

La longitud de una cadena especifica el número de caracteres de ésta. El método `length()` devuelve un entero indicando la longitud de la cadena.

Ejemplo 1.3
Uso del método length.

Pseudo programa Java:

```
String firstName = "Flavio";
int length = firstName.length();
System.out.print(length); //Muestra: 6
```

Concatenar cadenas de caracteres

Hay dos formas de realizar esta operación consistente en unir cadenas en Java:

Con el operador + (recomendado):

Ejemplo 1.4
Concatenación de cadenas

Pseudo programa Java:

```
String firstName = "Flavio ";
String lastName = "Gil";
String fullName = firstName + lastName;
System.out.print(fullName); //Muestra: Flavio Gil
```

Con el método concat:

Pseudo programa Java:

```
String firstName = "Flavio ";
```

```
String lastName = "Gil";
String fullName = firstName.concat(lastName);
System.out.print(fullName); //Muestra: Flavio Gil
```

Nota
Algunos desarrollos (códigos) se presentan de forma abreviada obviando todo el código asociado al presentado con el fin de hacer énfasis en la lógica e implementación; en estos casos podemos considerar que estamos trabajando con código *pseudo Java*, que no afectará la lógica del problema.

Convertir cadenas de caracteres a mayúscula/minúscula

Utilizamos los métodos `toUpperCase()` para convertir a mayúsculas y `toLowerCase()` para convertir a minúsculas, respectivamente:

Ejemplo 1.5
Convertir a mayúscula y minúscula.

Pseudo programa Java:

```
String firstName = "Flavio ";
String lastName = "Gil";
String fullName = firstName.concat(lastName).toUpperCase();
System.out.print(fullName); //Muestra: FLAVIO GIL
fullName = firstName.concat(lastName).toLowerCase();
System.out.print(fullName); //Muestra: flavio gil
```

Comparar cadenas

Java dispone de varios métodos para comparar dos cadenas; estos son:

Método equals
El método permite comparar dos cadenas devolviendo `true` si son iguales o `false` en caso contrario: `cadena1.equals(cadena2)`.

Método equalsIgnoreCase
Método similar a `equals`, pero sin tener en cuenta las mayúsculas y minúsculas: `cadena1.equalsIgnoreCase(cadena2)`.

Método compareTo
Devuelve cero (`0`) si las dos cadenas son iguales. Devuelve un valor menor a cero (`< 0`) si la primera cadena es alfabéticamente menor que la segunda ó un valor mayor a cero (`> 0`) si la primera cadena es alfabéticamente mayor que la segunda: `cadena1.compareTo(cadena2)`.

Método compareToIgnoreCase

Similar a `compareTo`, pero sin tener en cuenta las mayúsculas y minúsculas: `cadena1.compareToIgnoreCase(cadena2)`.

Conversión de datos a String

El método `valueOf()` es un método estático que convierte un argumento de cualquier tipo de dato a String: `String.valueOf(cadena)`.

Obtener la posición de un carácter o subcadena dentro de una cadena

En muchas ocasiones necesitamos saber si un carácter o una subcadena se encuentran en una cadena; el método `indexOf()` permite encontrar la posición de la primera ocurrencia del carácter o subcadena dentro de la cadena dada. Si la búsqueda es infructuosa, el método devuelve -1.

Sintaxis

```
indexOf(subCadena[, posiciónInicial])
```

Donde:
subCadena: carácter o subcadena a buscar en la cadena dada
posiciónInicial: argumento opcional que permite especificar en dónde inicia la búsqueda dentro de la cadena

Ejemplo 1.6
Aplicación del método `indexOf`.

Pseudo programa Java:

```
String str1 = "Feria de las flores Medellín";
str2 = "Medellín";
int pos = str1.indexOf(str2, 0);
System.out.print(pos); //Muestra: 20
```

Eliminar espacios al inicio y fin de la cadena

Java dispone de la función `trim()` para eliminar los espacios en blanco que se encuentre tanto al inicio como al final de una cadena de caracteres:

```
System.out.println("Uso de trim():" + " Hola ".trim() + "a todos");
// Imprime: Uso de trim():Holaa todos
```

Devolver un carácter de una cadena a partir de un índice

El método `chartAt` permite obtener el carácter que se especifique según su posición dentro de la cadena mediante un índice; recordemos que las cadenas en Java inician en la posición `0` y finalizan en la posición `cadena.length() - 1`; así, para obtener un carácter de una cadena, podemos escribir: `cadena.charAt(índice)`.

Obtener el valor ASCII de un carácter

Java no dispone de una función directa para obtener el valor `ASCII` de un carácter, pero basta con convertir éste a entero, por lo que para realizar esta operación es necesario que el tipo de dato de la variable sea `char` para que pueda ser convertida al tipo `int`. Recuerde que un dato tipo `char` se encierra entre apóstrofos:

```
System.out.println("Ascii '/':" + (int)'/');
System.out.println("Ascii:" + (int)"hola".charAt(3));
```

Obtener un carácter a partir de su valor ASCII

Java no dispone de una función directa para obtener el carácter a partir de su valor `ASCII`, pero basta con convertir éste a carácter (`char`), por lo que para realizar esta operación es necesario que el tipo de dato de la variable sea `int` para que pueda ser convertida al tipo `char`. Recuerde que un dato tipo `char` se encierra entre apóstrofos:

```
System.out.println("Carácter ASCII(97): " + (char)97); // ASCII(97): a
```

Substraer una cadena (subcadena) de otra cadena

Otra operación común es la sustracción de cadenas de otra cadena. Java dispone del método `substring()` para substraer subcadenas a partir de una cadena dada.

Sintaxis

```
substring(inicio[, fin])
```

Donde:
inicio: posición a partir de la cual se sustraerá la subcadena de la cadena dada
fin: opcional. Argumento que permite especificar hasta dónde se sustrae la cadena. Si no se especifica, se sustrae hasta el final de la cadena. Este índice debe indicar uno más después del último carácter de la subcadena.

Ejemplo 1.7
Substraer una subcadena de una cadena dada

Pseudo programa Java:

```
String str1 = "Feria de las flores Medellín";
String str2 = str1.substring(13, 19);
System.out.print(str2); //Muestra: flores
```

Ejemplo 1.8

Determinar si una frase es un palíndromo[34]. Aplicar los métodos de cadenas de caracteres para solucionar este problema.

Programa Java:

```
package com.packages.strings;

public class Strings
{
    private String text;


    public Strings()
    {
        this.text = "";
    }

    public void setText(String str)
    {
        this.text = str;
    }

    public String getText()
    {
        return text;
    }

    public String palindrome(String text)
    {
        String message = "";
        text = text.toLowerCase();
        text = this.deleteSpaces(text);
        message = this.compareCharacters(text) ?
            " es palíndromo" : " no es palíndromo";
        return message;
    }

    public String deleteSpaces(String text)
    {
        int i = 0;
        text = text.trim();
        while (i < text.length()) {
            if (text.substring(i, i + 1).equals(" ")) {
                text = text.substring(0, i) +
                    text.substring(i + 1, text.length());
```

[34] Un palíndromo es una frase que puede leerse igual de izquierda a derecha y de derecha a izquierda; por ejemplo: amad a la dama, dábale arroz a la zorra el abad, anita lava la tina y anilina, entre otras.

```
            } else {
                i++;
            }
        }
        return text;
    }

    public boolean compareCharacters(String text)
    {
        boolean sw = true; //Supuesto: text es palíndromo
        int i = 0;
        while (i < text.length() / 2 && sw) {
            if (text.substring(i, i + 1).equals(
                text.substring(text.length() - i - 1, text.length() - i)
            )) {
                i++;
            } else {
                sw = false;
            }
        }
        return sw;
    }
}
```

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 o listas

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 1.1. 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

En pseudocódigo:

```
tipo_de_dato: nombre_vector[tamaño]
```

En Java:

```
tipo_de_dato nombre_vector[] = new tipo_de_dato[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. En el lenguaje Java la primera posición está determinada por el índice 0.

Ejemplo 1.9

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 1.10

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
FinFuncion

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
FinFuncion

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
```

Operaciones sobre un vector

Ya hemos visto algunas operaciones que pueden aplicarse sobre un vector, tales como el acceso a los elementos de este, ya sea para leerlos o mostrarlos. Las operaciones típicas sobre arreglos unidimensionales son:

- Llenar el vector
- Recorrer el vector
- Buscar un dato en el vector

- Insertar un dato en el vector (en una posición dada, o antes o después de una referencia)
- Eliminar un dato (de una posición dada)
- Ordenar el vector

Las dos primeras operaciones se realizaron en el ejemplo 1.2; veamos las demás.

Buscar un dato

La búsqueda es una de las operaciones más importantes y comunes en vectores y otras estructuras de datos, por lo que se han desarrollado distintas técnicas, unas más complejas que otras, aprovechando la capacidad de las máquinas para ejecutar instrucciones a altas velocidades, lo cual es aprovechado para buscar información en listas de datos mediante la creación de algoritmos para ello. La **búsqueda secuencial** es la más sencilla de todas y de las más utilizadas; los tipos de búsqueda sobre vectores y otras estructuras de datos, son:

- Búsqueda secuencial
- Búsqueda binaria
- Búsqueda por transformación de claves
- Árboles Binarios de Búsqueda (ABB)

En cursos posteriores se analizan los algoritmos que permiten realizar dichas búsquedas midiendo y comparando su eficiencia; veamos la búsqueda secuencial y binaria.

Ejemplo 1.11
Búsqueda secuencial de un dato en un vector.

Este tipo de búsqueda consiste en tomar un dato y compararlo elemento por elemento del vector hasta encontrarlo o hasta terminar de recorrer el vector. El tipo de dato que devuelve el método (función) puede ser un valor *lógico* (*booleano*) para indicar que el dato está o no en el arreglo; también puede devolver un valor *entero* con la posición donde el dato se encuentra, en cuyo caso se inicializa en un valor por fuera de los posibles valores que tome el índice del vector para indicar que el dato no se encuentra.

Programa Java:

```
public int secuencialSearchVector(int datum)
{
    int i, pos;
    i = 0;
    pos = -1;
    while (i < this.n && pos == -1) {
        if (this.vec[i] == datum) {
            pos = i;
        } else {
            i++;
        }
    }
```

```
    return pos;
}
```

Ejemplo 1.12
Búsqueda binaria de un dato en un vector.

Para implementar este tipo de búsqueda, **el vector debe estar ordenado**. Consiste definir el **límite inferior** y un **límite superior**, y a partir de estos datos, se toma el elemento central calculando la posición "**promedio**"; si el dato se encuentra en dicha posición, la búsqueda finaliza, de lo contrario, se evalúa si el dato es mayor o menor que el elemento en la posición y se redefinen los límites. El proceso finaliza si el dato es encontrado o si no hay un intervalo de búsqueda. Esta búsqueda reduce significativamente el número de comparaciones, lo cual puede verse en vectores grandes

Programa Java:

```
public int binarySearchVector(int datum)
{
    int lowerLimit, upperLimit, pos, centralPos;
    lowerLimit = 0;
    upperLimit = this.n;
    pos = -1;
    while (lowerLimit <= upperLimit && pos == -1) {
        centralPos = (upperLimit + lowerLimit) / 2;
        if (this.vec[centralPos] == datum) {
            pos = centralPos;
        } else {
            if (this.vec[centralPos] > datum) {
                upperLimit = centralPos - 1;
            } else {
                lowerLimit = centralPos + 1;
            }
        }
    }
    return pos;
}
```

Eliminar un dato

Esta operación consiste en "pararse" en la posición del elemento a eliminar y desplazar los siguientes `n - posición - 1` elementos hacia la "izquierda" una posición. Una vez se muevan los elementos (se creen las copias), se reduce el tamaño del vector en uno. Debe comprobarse que hayan elementos para poder eliminar.

Ejemplo 1.13
Eliminar un dato de un vector.

Programa Java:

```
public void deleteVector(int pos)
{
      int i;
      for (i = pos; i < n - 1; i++) {
            this.vec[i] = this.vec[i + 1];
      }
      this.n--;
}
```

Insertar un dato

Esta operación consiste en “pararse” en la última posición e ir moviendo (copiando) los anteriores *n - posición - 1* elementos hacia la “derecha” una posición para “abrir el espacio”. Una vez se muevan los elementos (se creen las copias), se guarda el nuevo dato en la posición y se aumenta el tamaño del vector en uno. Antes de realizar la inserción, debe verificarse que haya espacio disponible en el vector para evitar un error.

Ejemplo 1.14
Insertar un dato en un vector.

Programa Java:

```
public void insertVector(int pos, int datum)
{
      int i;
      for (i = this.n; i > pos; i--) {
            this.vec[i] = this.vec[i - 1];
      }
      this.vec[pos] = datum;
      n++;
}
```

Ordenar el vector

La ordenación de datos es una operación importante y requerida en distintas situaciones al operar con vectores y otras estructuras de datos, tanto a nivel interno como externo; consiste en clasificar los elementos en un orden determinado, facilitando así las tareas de búsqueda. Al igual que con la búsqueda de datos, con la ordenación se han desarrollado distintas técnicas, unas más complejas que otras que permiten la clasificación de la información. La ordenación por el **método de intercambio directo o burbuja** es la más sencilla de todas, de las más utilizadas, pero también la más ineficiente; los tipos de ordenación sobre vectores y otras estructuras de datos, son:

4. Ordenación directa (burbuja)
5. Ordenación por inserción directa (baraja)
6. Ordenación por selección directa
7. Ordenación por el método de Shell
8. Ordenación por el método de fusión (merge sort)

9. Ordenación por el método rápido (quicksort)
10. Ordenación por el método del montículo (heapsort)

En cursos posteriores se analizan los algoritmos que permiten realizar distintas formas de ordenaciones midiendo y comparando su eficiencia.

Ejemplo 1.15
Ordenación de un vector por el método de **intercambio directo o burbuja**.

Este método consiste en tomar cada elemento del vector y compararlo con los siguientes `n - i - 1` elementos; si alguno es mayor (o menor), se realiza un *intercambio directo* y se continúa comparando. Este método es lento comparado con otros, pero garantiza que deja ordenado cada elemento antes de pasar al siguiente.

Programa Java:

```
public void sortBubbleVector()
{
    int i, j, aux;

    for (i = 0; i < this.n - 1; i++) {
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[i] > this.vec[j]) {
                aux = this.vec[i];
                this.vec[i] = this.vec[j];
                 this.vec[j] = aux;
            }
        }
    }
}
```

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 1.2. Representación gráfica de un arreglo bidimensional: matriz A_{mxn}.[35]

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 1.3. Representación gráfica de un arreglo bidimensional en memoria

Nota

Observe que un vector es una matriz de `1 x n` ó `n x 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

Pseudocódigo

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

Java

```
tipo_de_dato nom_matriz[][] = new tipo_de_dato[tot_filas][tot_columnas]
```

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.

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

Sintaxis

Pseudocódigo

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

Java

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

Nota

En Java, tanto la primera posición para las filas como para las columnas, es la cero (0).

Ejemplo 1.16

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 1.17

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.

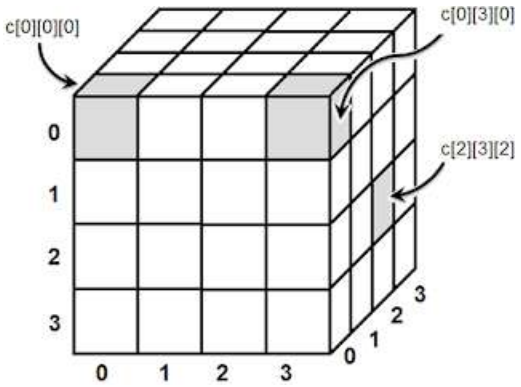


Figura 1.4. Representación gráfica de un arreglo tridimensional en forma de cubo. Aquí la primera posición para cada dimensión es la cero (0)[36]

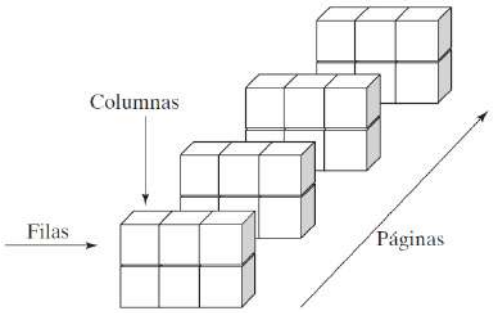


Figura 1.5. Representación gráfica de un arreglo tridimensional[37]

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

Pseudocódigo

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

Java

```
tipo_de_dato nom_arreglo[][]...[] = new tipo_de_dato[dim1][dim2]...[dimN]
```

Acceso a los elementos de un arreglo multidimensional

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

Sintaxis

[36] Imagen tomada de Arreglos de n dimensiones.
[37] Imagen tomada de ▹ Tipos de Arreglos en Matlab - [octubre, 2023]

Pseudocódigo

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

Java

```
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 1.18

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 `5x10x7`. 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
```

Registros

Un **registro** es un tipo estructurado de datos (compuesto), esto significa que se compone de datos primitivos y posiblemente otros tipos estructurados, los cuales se conocen como **campos** del registro y que deben tener un identificador (nombre) único. Algunos lenguajes estructurados (procedimentales) e híbridos soportan la creación de registros (como en C/C++ y en PHP); los lenguajes orientados a objetos no implementan un tipo especial para crear registros, pero esto se soluciona fácilmente creando objetos sin métodos, solo especificando propiedades (atributos) que representan los campos del registro.

Al poder definir registros en un lenguaje de programación, tenemos la posibilidad de crear nuevos tipos de datos, esta vez definidos por el usuario a partir de los datos primitivos y posiblemente otros datos estructurados de acuerdo a las necesidades que indique un problema.

La siguiente figura muestra la estructura de un registro de clientes con algunos campos definidos para éste.

<table>
<tr><th colspan="8">Registro Clientes</th></tr>
<tr><th>Nombre del campo</th><th>Identificación</th><th>Nombre</th><th>Ciudad</th><th colspan="3">Dirección</th><th>Teléfono</th></tr>
<tr><td rowspan="2">Tipo de dato</td><td rowspan="2">(carácter)</td><td rowspan="2">(carácter)</td><td rowspan="2">(carácter)</td><td>Barrio</td><td>Tipo vía</td><td>Número</td><td rowspan="2">(carácter)</td></tr>
<tr><td>(carácter</td><td>(carácter)</td><td>(entero)</td></tr>
</table>

Figura 1.6. Representación gráfica de un registro de clientes con algunos campos

Crear un registro

Algorítmicamente o en pseudocódigo, podemos definir un registro así:

Sintaxis

```
Registro nombre_registro
    tipo_dato campo1 [,
    tipo_dato campo2 [,
    ...]]
FinRegistro
```

Análogamente, en lógica orientada a objetos realizamos lo siguiente:

Sintaxis

```
Clase nombre_clase
    publico|privado|protegido tipo_dato atributo1 [,
    publico|privado|protegido tipo_dato atributo2 [,
    ...]]
FinClase
```

Crear variables de tipo registro

Para definir una variable de tipo registro realizamos lo siguiente.

Sintaxis (crear una variable de tipo registro)

```
Registro nombre_registro variable
```

Análogamente, en lógica orientada a objetos creamos un objeto, es decir, instanciamos una clase:

Sintaxis (instanciar una clase)

```
NombreClase nombre_objeto = Nuevo NombreClase
```

Acceso a los campos de un registro

Acceder a una variable de tipo registro consiste en acceder a sus campos. Dado que el lenguaje algorítmico permite flexibilidad, o más bien, definir un propio pseudolenguaje

independiente de los lenguajes existentes, pero “universal”, podemos establecer en este texto el operador **punto** (**.**) para acceder a los campos.

Sintaxis (variable de tipo registro)

```
variable.campo
```

Análogamente, en lógica orientada a objetos accedemos a las propiedades de un objeto de esta forma:

Sintaxis (objeto representando un registro)

```
objeto.campo
```

Ejemplo 1.19
Crear un registro de estudiantes con los campos identificación, nombre, edad y si trabaja o no. Leer los datos de un estudiante y mostrar su información. Presentar la solución en pseudocódigo, C/C++ y Java.

Pseudocódigo:

```
Inicio
Registro Estudiante
    Caracter: identificacion,
    Caracter: nombre,
    Enteros: edad,
    logico: trabaja
FinRegistro

Estudiante: est1
Leer est1.identificacion
Leer est1.nombre
Leer est1.edad
Leer est1.trabaja
Imprimir est1.identificacion
Imprimir est1.nombre
Imprimir est1.edad
Imprimir est1.trabaja
Fin
```

Programa C++:

```
#include <iostream>

struct Estudiante
{
    char identificacion[20];
    char nombre[50];
    int edad;
```

```
    bool trabaja;
};

int main(int argc, char** argv)
{
    struct Estudiante est1;
    printf("Identificación: ");
    cin.getline(est1.identificacion);
    printf("Nombre: ");
    cin.getline(est1.nombre);
    printf("Edad: ");
    cin >> est1.edad;
    printf("Trabaja [1->Sí | 2->No: ");
    cin >> est1.trabaja;
    printf(est1.identificacion);
    printf(est1.nombre);
    printf(est1.edad);
    printf(est1.trabaja);
}
```

La solución en Java requiere de un poco más de trabajo. Crearemos un paquete con dos clases, una contendrá la estructura del registro y la otra la interfaz de la aplicación

La estructura de carpetas es la siguiente para un SO Windows:

```
carpeta_principal\Registros.java
```

```
carpeta_principal\com\packages\Estudiante.java
```

Programa Java: Estudiante.java

```
public class Estudiante
{
    String identificacion;
    String nombre;
    int edad;
    boolean trabaja;
}
```

Programa Java: Registros.java

```
import java.util.Scanner;
import com.packages.Estudiante;

public class Registros
{
    public static Scanner input = new Scanner(System.in);

    public static void main(String[] args)
```

```
    {
        Estudiante est1 = new Estudiante();
        System.out.println("Identificación: ");
        est1.identificacion = input.next();
        System.out.printls("Nombre: ");
        est1.nombre = input.next();
        System.out.println("Edad: ");
        est1.edad = input.nextInt();
        System.out.println("Trabaja [true | false: ");
        est1.trabaja = input.nextBoolean();
        System.out.println(est1.identificacion);
        System.out.println(est1.nombre);
        System.out.println(est1.edad);
        System.out.println(est1.trabaja);
    }
}
```

Diferencias con objetos

Observe la similitud entre un registro y un objeto cuando se define su estructura básica, es decir, un objeto sin métodos; sin embargo, son elementos muy diferentes; un objeto generalmente contiene métodos y hace parte de un paradigma que va más allá de solo especificar atributos (campos) a una entidad. Los registros son más simples y anteriores a la POO, en la cual se trata el registro como un conjunto de atributos junto a un conjunto de métodos que permiten la manipulación de dichos atributos (propiedades).

Realmente, y de acuerdo a lo que en informática entendemos por registro, el conjunto de atributos o propiedades de un objeto ¡es un **registro**! y esta puede ser una de las razones por las que en los lenguajes orientados a objetos no existe una definición para el tipo `struct` como en C/C++ o PHP para evitar ser redundantes en cuestiones que son inherentes a los objetos. C es un lenguaje estructurado que solo soporta la creación de registros; C++ y PHP son lenguajes híbridos y soportan tanto la creación de registros como de objetos; Java y Python al ser lenguajes orientados a objetos, todo lo consideran como un objeto, por lo que un registro es simplemente otro objeto.

Diferencias con arreglos

Hay básicamente dos diferencias entre registros y arreglos:

- La información almacenada en arreglos es homogénea, esto es, del mismo tipo; mientras que en los registros se guarda información heterogénea, es decir, de distintos tipos de datos, tanto primitivos como estructurados.
- Para acceder a un elemento de un arreglo, se utiliza uno o varios índices representados por valores enteros, mientras que para acceder a los elementos de un registro se hace mención al nombre único o identificador del campo usando el operador punto (.).

Arreglos de registros, arreglos de objetos y arreglos paralelos

Los lenguajes permiten un nivel de abstracción para combinar arreglos con registros y objetos, lo que da la posibilidad de crear “arreglos heterogéneos” en el sentido que permitirán manipular información de distinto tipo, pero sin salir de la regla de ser homogéneos, ya que en realidad son definidos y están almacenando información de un mismo tipo estructurado. Un arreglo de registros, así como un arreglo de objetos no es es más que un arreglo que contendrá en cada posición un registro u objeto

En cuánto a los objetos, es de tener cuidado en cómo se realiza la implementación, ya que cada objeto es una instancia única de la clase, por lo que debe instanciarse un objeto por cada posición del arreglo donde deseemos almacenarlo.

Cuando un lenguaje no implementa registros ni el paradigma de la POO, es posible usar “arreglos paralelos” para solucionar problemas que involucren tratar varios datos de una misma entidad. Como en el ejemplo anterior, en lugar de usar registros u objetos, se pueden crear los arreglos `identificacion[], nombre[], edad[], trabaja[]`, todos del mismo tamaño y en donde una posición (índice) corresponde a la misma entidad, en este caso a un estudiante determinado.

Ejemplo 1.20

Teniendo presente el registro y la clase de Estudiantes del ejemplo anterior, crear un arreglo de registros en pseudocódigo y un arreglo de objetos en Java de tipo Estudiantes leyendo varias entradas y mostrando dicha información.

Pseudocódigo:

```
Inicio
Registro Estudiante
    caracter identificacion,
    caracter nombre,
    entero edad,
    logico trabaja
FinRegistro

Caracter: resp
Enteros: te = 0
Estudiante: est[]
Repetir
    te = te + 1
    Leer est[te].identificacion
    Leer est[te].nombre
    Leer est[te].edad
    Leer est[te].trabaja
    Leer resp
Hasta Que resp == minusculas("no")
```

```
Para i = 1, te, 1
    Imprimir est[i].identificacion
    Imprimir est[i].nombre
    Imprimir est[i].edad
    Imprimir est[i].trabaja
FinPara
Fin
```

Así como vimos anteriormente, la solución en Java requiere un poco más de trabajo. Crearemos un paquete con dos clases, una contendrá la estructura del registro y la otra métodos para manipularlo.

La estructura de carpetas y archivos es la siguiente para un SO Windows:

```
main_folder\MainMenu.java
```

```
main_folder\com\packages\records\Student.java
main_folder\com\packages\records\Records.java
```

Programa Java: Student.java

```
package com.packages.records;

public class Student
{
    public String id;
    public String name;
    public int age;
    public boolean work;
}
```

Programa Java: Records.java

```
package com.packages.records;

import java.util.Scanner;

public class Records
{
    private static Scanner input = new Scanner(System.in);
    private Student est[] = new Student[30];
    private int ts;

    public Records()
    {
        ts = 0;
    }
```

```
    public int getTs()
    {
        return ts;
    }

    public void setTs(int ts)
    {
        this.ts = ts;
    }

    public void createRecords()
    {
        String resp = "";
        do {
            est[ts] = new Student();
            System.out.println("Identificación: ");
            est[ts].id = input.next();
            System.out.println("Nombre: ");
            est[ts].name = input.next();
            System.out.println("Edad: ");
            est[ts].age = input.nextInt();
            System.out.println("Trabaja [true | false]: ");
            est[ts].work = input.nextBoolean();
            ts++;
            System.out.println("Continuar? [s/?]");
            resp = input.next();
        } while (resp.toLowerCase().equals("s"));
    }

    public void showRecords()
    {
        System.out.println("-----------");
        for (int i = 0; i < ts; i++) {
            System.out.println("Id: " + est[i].id);
            System.out.println("Nombre: " + est[i].name);
            System.out.println("Edad: " + est[i].age);
            System.out.println("Trabaja: " + est[i].work);
            System.out.println("-----------");
        }
    }
}
```

Conjuntos

Un conjunto es un elemento similar a un vector, pero que tiene la misma propiedad de los conjuntos matemáticos, y es que cada elemento que lo compone es único. Los conjuntos por su parte, admiten información heterogénea.

Muy pocos lenguajes tienen una estructura de datos explícita para su tratamiento, como el lenguaje Python, por ejemplo. En Java si se desea implementar una aplicación relacionada con conjuntos, se pueden usar vectores o listas ligadas (aunque es más cómodo trabajar estos elementos con arreglos) e implementar las reglas de conjuntos respectivas.

A nivel de lógica de programación, podemos definir un conjunto así:

Sintaxis

1.
```
variableConjunto = {elementos}
```

2.
```
variableConjunto = Conjunto(iterable)
```

En Python se hace de esta forma:

Sintaxis

1.
```
variableConjunto = {elementos}
```

2.
```
variableConjunto = set(iterable)
```

Donde:
elementos: es la secuencia o conjunto de elementos separados por comas. Si no se indica, el conjunto se crea vacío
iterable: objeto iterable reconocible por Python (o en lógica), tal como listas (vectores), tuplas o cadenas. Si no se especifica, el conjunto se inicializará vacío

Con los conjuntos de Python es posible realizar algunas de las operaciones matemáticas entre ellos.

Operaciones con conjuntos

Las operaciones fundamentales entre conjuntos son: Unión, Intersección y Diferencia; otras operaciones se pueden expresar en términos de estas operaciones y otras reglas. Estas operaciones crean nuevos conjuntos.

Operación	Símbolo operador	Prioridad	Uso

Unión	+ | Union	Baja	`A + B`
Intersección	* & Interseccion	Alta	`Interseccion(A, B)`
Diferencia	- Diferencia	Baja	`A - B`

Implementación de conjuntos en Python

El tratamiento de un conjunto en Python es similar al tratamiento de listas (vectores), aunque difiere en algunos métodos y también dispone de otros para las operaciones entre conjuntos.

Creación de conjuntos

Ejemplo 1.21
Creación de los conjuntos a y b:

```
>>> a = {1, 2, 3}
>>> a
{1, 2, 3}
```

```
>>> b = set([0, 5, 1])
>>> b
{0, 1, 5}
```

Ejemplo 1.21
A partir de los conjuntos a y b realizar las operaciones de Unión, intersección y diferencia de los conjuntos utilizando los métodos y operadores disponibles en Python:

```
>>> c = a.union(b)
>>> c
{0, 1, 2, 3, 5}
>>> c = a.intersection(b)
>>> c
{1}
```

```
>>> a
{0, 1, 3}
>>> b
{0, 1, 5}
>>> c = a | b
>>> c
{0, 1, 3, 5}
>>> c = a & b
>>> c
{0, 1}
>>> c = a - b
>>> c
{3}
```

Sean:
`a, b, c`: conjuntos
`dato`: una variable
`posicion`: variable que guarda un índice de la tupla

Se pueden aplicar los siguientes métodos a un conjunto en Python:

f. Obtener el total de elementos con la función `len`: `len(a)`.
g. Agregar un elemento: `a.add(dato)`.
h. Buscar con el operador `in`: `sw = dato in a # True: si se encuentra`.
i. Eliminar un elemento: `a.discard(dato)`.

Ejemplo 1.22
A partir de los conjuntos `a` y `b` dados anteriormente, realizar lo siguiente:

5. Encontrar la cardinalidad del conjunto
6. Adicionar un nuevo elemento al conjunto
7. Eliminar un elemento del conjunto

```
>>> len(a)
3
>>> a.add(2)
>>> a
{1, 2, 3}
>>> a.add(0)
>>> a
{0, 1, 2, 3}
>>> a.discard(2)
>>> a
{0, 1, 3}
```

Nota
En los ejercicios se propone desarrollar una aplicación que permita trabajar con conjuntos en Java.

Preguntas

Ejercicios

- Ingrese un texto por teclado. Cree un subprograma para determinar cuántas vocales se encuentran en éste
- Ingrese un texto y un carácter. Cree un subprograma para encontrar cuántas veces está dicho carácter en el texto
- Ingrese un texto. Cree un subprograma que indique cuántas palabras conforman el texto

- 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”
- 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”
- 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”
- 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”
- 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” (esta podría considerarse una forma básica de cifrar un mensaje)
- Ingrese el nombre de una persona. Utilice una función para validar que no ingresen números
- 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.
- Dada una cantidad numérica, use funciones para convertirla a letras
- Dado un número entero positivo, use funciones para convertirlo a número romano
- Dado un número, utilice subprogramas para determinar si es un número capicúa sin convertirlo a cadena (los números capicúas son aquellos que se leen igual de ambos sentidos, por ejemplo: 12321).
- Dada una frase, cree subprogramas para determinar si ésta es un palíndromo o no (los palíndromos son aquellas frases que se leen igual de ambos sentidos, por ejemplo: “*amad a la dama*”)
- Leer N números y guardar en una vector los números positivos. Crear subprogramas para mostrar el arreglo y mostrar el mayor número almacenado.
- Teniendo presente el ejemplo 1.2, agregue métodos a la clase Vector desarrollada en la aplicación de ejemplo:
    - Sumar los datos del vector (sumatoria) $s = \sum_{i=1}^{n} V_i$
    - Multiplicar los datos del vector (productoria) $p = \prod_{i=1}^{n} V_i$
    - Hallar el promedio $\bar{x} = \frac{\sum_{i=1}^{n} V_i}{n}$
    - Encontrar el mayor dato $max(V_i)$
    - Encontrar el menor dato $min(V_i)$
    - Encontrar la moda si existe (la moda es el dato que más se repite)
    - Encontrar la multimoda
    - La mediana (dato que con la lista ordenada, se encuentra en el centro de la muestra). $\hat{x} = x_{(n+1)/2}\ si\ n\ es\ impar; \hat{x} = \frac{x_{n/2}+x_{n/2+1}}{2}\ si\ n\ es\ par$

    - Varianza muestral $\sigma^2 = \sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n-1}$
    - Desviación estándar $\sigma = \sqrt{\sigma^2} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2}$
- 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:
    - 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`
    - Listar todos los estudiantes en forma tabulada y añadir una columna adicional que muestre si ganó o perdió la asignatura
    - Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
    - Mostrar el promedio de notas del grupo
    - Mostrar la mayor nota del grupo y a quien pertenece ésta
    - Mostrar la menor nota del grupo y a quien pertenece ésta
    - Permitir la eliminación de estudiantes
    - Permitir actualizar la nota de un estudiante
    - Informar si hay un estudiante de nombre Pedro Zapata o Ana Zapata
- 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:
    - 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`
    - Listar todos los estudiantes en forma tabulada junto con su promedio de notas
    - Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
    - Mostrar el promedio general de notas
    - Mostrar la mayor nota de un estudiante cualquiera
    - Mostrar la menor nota de un estudiante cualquiera
    - Mostrar la mayor nota de todos los estudiantes y a quien pertenece ésta
    - Mostrar la menor nota de todos los estudiantes y a quien pertenece ésta
    - Permitir la eliminación de estudiantes
    - Permitir actualizar la nota de un estudiante
- 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:

    - Agregar productos de forma ordenada por código, validando que éstos no se repitan
    - Mostrar el producto más costoso
    - Mostrar el producto más barato
    - Listar el inventario de forma tabulada y añadir una columna adicional que muestre un IVA del 19% para todos los artículos
    - Permitir buscar un producto por código
    - Listar los productos cuya cantidad sea inferior al *stock* mínimo y alertar en caso de encontrarlos
    - Permitir actualizar cualquier dato del producto, excepto el código, validando además que la cantidad no sea negativa y el precio mayor a cero
    - Permitir eliminar productos
    - Informar si hay un producto que tenga una existencia de 300 o de 400 unidades
- Crear una matriz `M` con `mxn` valores numéricos aleatorios entre 1 y 50. Usando subprogramas, realizar las siguientes operaciones sobre ella:
    - Mostrar el arreglo
    - $\sum_{i=1}^{n} M_{i,j}$ (sumatoria)
    - $\prod_{i=1}^{n} M_{i,j}$ (productoria)
    - $\overline{M}$ (media o promedio)
    - Encontrar el mayor dato
    - Encontrar el mayor dato de una fila o columna dada
    - Encontrar el menor dato
    - Encontrar el menor dato de una fila o columna dada
    - Buscar un elemento
    - Eliminar una fila o columna
    - Insertar una fila o columna dada una fila y/o columna de referencia
    - Ordenar una fila o columna dada
    - Ordenar la matriz
- 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:
    - Ingresar las ventas de totales de cada producto por día
    - Mostrar la matriz completa en forma de tabla
    - Buscar un producto y mostrar las ventas de la semana
    - Dado un día, mostrar las ventas de cada producto en éste
    - Mostrar el total de ventas por día
    - Mostrar el promedio de ventas por día
    - Mostrar el total de ventas por producto
    - Mostrar el promedio de ventas por producto
    - Mostrar el total de ventas de la semana
    - Mostrar el promedio de ventas de la semana

    - Mostrar la mejor venta del día
    - Mostrar el mejor día en ventas de un producto
    - Mostrar la peor venta del día
    - Mostrar el peor día en ventas de un producto
    - Mostrar la mejor venta e indicar a qué producto corresponde y en qué día se realizó
    - Mostrar la peor venta e indicar a qué producto corresponde y en qué día se realizó
    - Permitir actualizar los valores de las ventas
- Utilice el lenguaje Java para codificar el ejemplo 1.7 relacionado con arreglos multidimensionales
- Aplicaciones con conjuntos. Sean A y B dos conjuntos. Cree dos vectores y realice las operaciones básicas sobre conjuntos (recuerde que un conjunto no contiene elementos repetidos)
    - La cardinalidad de los conjuntos
    - Determine si A y B son iguales: A = B
    - Unión: $A \cup B$
    - Intersección: $A \cap B$
    - Diferencia: $A - B$
    - Diferencia simétrica: $A \Delta B$
    - Complemento de un conjunto cualquiera X ($X = A$ o $X = B$): X'
    - Conjunto potencia de un conjunto cualquiera X ($X = A$ o $X = B$): $P(X)$
    - Producto cartesiano: $A \times B$ y $B \times A$
- Dada una matriz cuadrada de orden n, imprima las áreas y/o elementos
    - De la diagonal principal y secundaria
    - Que están por debajo de la diagonal principal
    - Que están por encima de la diagonal principal
    - Que están por debajo de la diagonal secundaria
    - Que están por encima de la diagonal secundaria
    - Que forman un triángulo isósceles por cualquier lado
    - Que forman una onda sinusoidal
    - Que siguen la forma de un caracol

Capítulo 7. Listas Ligadas

Las **listas ligadas** son **estructuras de datos lineales dinámicas** conformadas por elementos llamados **nodos**. Un nodo es un *registro* formado básicamente con dos campos: uno para almacenar la **información** y otro para guardar una **referencia** (**liga**) a otro nodo.

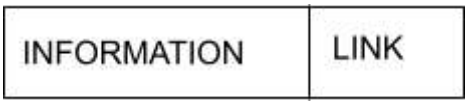


Figura 2.1. Estructura básica de un nodo

Este registro puede crearse fácilmente como una clase en Java; en el lenguaje C++ puede crearse como registro o bien como una clase, ya que éste soporta la creación de la estructura de datos **struct** para la creación de registros, mientras que en la versión del lenguaje C sólo puede crearse usando la estructura de datos **struct**. El campo **Información** puede a su vez estar compuesto de otro registro, arreglos, etc., mientras el campo **Liga** es una **referencia** (**dirección**) a otro nodo; si no hay una referencia a ningún otro nodo, dicho campo almacenará una marca de **NULO** (vacío) o **NIL** (**null** en Java).

Nota
La estructura del nodo que utilizaremos en los ejemplos que se presentan más adelante, tendrán una estructura muy similar a la mostrada en la figura 2.1 con los campos **info** de tipo *entero* y **link** de tipo *nodo*. Para acceder a un nodo vamos a requerir de una variable de tipo **apuntador** que lo pueda referenciar. Las variables tipo **puntero**, apuntan a la memoria y sus valores son **números hexadecimales** que especifican una **dirección** cualquiera de la memoria o un valor **nulo** (**null**) o vacío.

A diferencia de los arreglos, las listas ligadas son estructuras de datos **dinámicas**, ya que no se reserva un espacio determinado de memoria antes de iniciar el programa, sino que ésta se va solicitando al sistema de acuerdo a la necesidad del usuario y/o aplicación, así como de la disponibilidad que tenga de ésta en la máquina. El hardware actual supone que no hay problemas para la asignación dinámica de la memoria, sin embargo, es importante tener las precauciones correspondientes y no abusar de los recursos disponibles, buscando siempre el mejor uso posible de éstos.

Al igual que sucede con los vectores, las listas ligadas son estructuras de datos **lineales**, ya que después de un elemento encontramos a lo sumo otro elemento.

Lista Simplemente Ligada (LSL)

Una LSL es el tipo de lista ligada más sencilla, en el sentido que ésta se recorre en un solo sentido y no contiene referencias circulares. La figura 2.2 muestra la representación de una LSL.

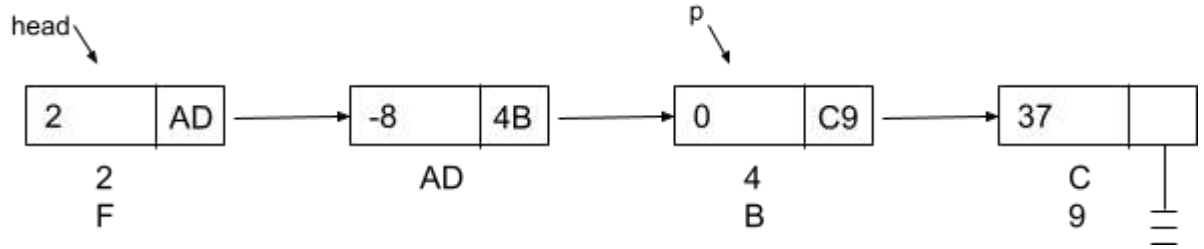


Figura 2.2. Representación de una LSL que contiene en su campo **info** números enteros y su primer registro es apuntado por la variable apuntador **head**. Un puntero *p* apunta a un nodo arbitrario de la lista. El campo **link** contiene la *dirección* (*referencia*) al siguiente nodo; el último nodo que no apunta a ningún otro, tiene una "*marca*" de **nulo** en este campo. Las direcciones de memoria están dadas en hexadecimal.

Una lista ligada, por regla general, debe contener un apuntador al primer nodo, con lo que garantizamos que la lista **no se pierda** y podamos acceder a su contenido. Dicho apuntador se suele conocer como **nodo (registro) cabeza**; en la gráfica aparece como **head**. En algunas aplicaciones, el registro cabeza se define como un registro independiente a los demás nodos de la lista, con el fin de establecer el inicio de ésta y la diferencia con los demás nodos; aquí no se considerarán nodos (registros) adicionales, ya que con el apuntador al primer nodo (cabeza) se puede implementar cualquier operación y no perder la información de la lista, basta con especificar correctamente las reglas en cada algoritmo de implementación sobre listas.

Creación de una LSL

Una lista ligada puede creerse por el **inicio** o por el **final** de ésta. Antes de realizar esto, se debe crear primero el **registro** correspondiente para cada **nodo** de la lista. La representación de los nodos depende de las necesidades de la aplicación. Para los ejemplos que se describen a continuación para los subtemas de LSL y LSLC se utilizará la representación dada en la nota anterior y que se muestra en el ejemplo 2.1, que describe la creación de una clase para crear nodos.

Ejemplo 2.1

Clase **`Node (Nodo)`** para crear nodos en una LSL; la clase básicamente es un registro que contiene los campos **`info`** de tipo **`int (enteros)`** y **`link (liga)`** de tipo **`Node (Nodo)`**.

Pseudocódigo:

```
Clase Nodo
    publico enteros info;
    publico Nodo liga;
FinClase
```

Programa Java:

```
public class Node
{
    public int info;
    public Node link;
}
```

Asignar y eliminar memoria

Los lenguajes de programación disponen de mecanismos para traer y liberar la memoria. En pseudocódigo se acostumbra a usar una funciones como **`TRAER(x)`** y **`LIBERAR(x)`** para asignar memoria dinámicamente y eliminarla cuando ésta ya no se requiera.

Ejemplo 2.2
Asignar memoria dinámicamente y eliminarla posteriormente y acceder a los campos del nodo.

Pseudocódigo:

```
…
punteros p, q // p, q: variables de tipo apuntador

// La función TRAER asigna memoria a las variables tipo puntero p y q
TRAER(p) // En forma de procedimiento o función sin valor de retorno
q = TRAER() // En forma de función

// Acceder a los campos del nodo
Leer p->info // Forma de acceder al campo info del nodo con el puntero p
p->liga = nulo // Guarda un nulo en el campo liga (link) del nodo

…

// La función LIBERAR(x) elimina de la memoria la referencia x
LIBERAR(p)
LIBERAR(q)
…
```

En el lenguaje C/C++ se utiliza la función **`malloc(x)`** para la asignación dinámica de memoria; en Java, al usar clases para definir el nodo, simplemente instanciamos la clase que los crea con la sentencia **`new()`**.

La función utilizada en C/C++ para liberar la memoria es **`free(x)`**; en Java no se requiere de una función para liberar memoria, ya que al trabajar con objetos, estos son eliminados automáticamente por el ***garbage collector*** (**recolector de basura**) cuando se pierden las referencias o ya no son usados, facilitando aún más el trabajo con este tipo de estructura de datos y con la gestión de la memoria.

Las referencias a los nodos están dadas por direcciones a la memoria expresadas en **números en hexadecimal**; al trabajar con objetos, estamos trabajando con referencias, por lo que el trabajo con listas ligadas en Java se hace relativamente cómodo comparado con la complejidad que ofrece al respecto un lenguaje como C/C++.

Ejemplo 2.3

Crear la clase para gestionar la LSL. La clase contendrá inicialmente las propiedades **head** y **n**; `head` será el puntero al nodo cabeza (primer nodo de la lista) y también permitirá establecer si la lista está vacía (`head = null`); `n` será usado para tener el total de nodos en la lista. Los métodos de esta clase se especifican en desarrollos más adelante, sólo se indica el constructor.

Pseudocódigo:

```
Clase ListaSimplementeLigada
    publico Nodo cab; // O también: punteros p
    publico enteros n;

    Constructor ListaSimplementeLigada()
        cab = nulo;
        n = 0;
    FinConstructor
FinClase
```

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class LinkedList
{
    public static Scanner input = new Scanner(System.in);
    public Node head;
    public int n;

    public LinkedList()
    {
        head = null;
        n = 0;
    }
}
```

Creación de una LSL por el inicio

En este método, los nodos quedan almacenados en **orden inverso** al orden en que se ingresan. En términos gráficos, podemos decir que la lista se crea de "**derecha a izquierda**".

Ejemplo 2.4

Crear una LSL por el inicio

Pseudocódigo:

```
publico Metodo crearInicioLSL()
    punteros p; // O también: Nodo p
```

```
    cadenas resp;
    Repetir
        p = TRAER(); // O también: p = nuevo Nodo()
        Imprimir "Ingrese un número a la LSL: ";
        Leer p->info;
        p->liga = cab;
        cab = p;
        n++;
        Imprimir "¿Agregar más nodos?: ";
        Leer resp;
    Hasta Que resp <> "s";
FinMetodo
```

Programa Java:

```
public void createStartLSL()
{
    Node p;
    String resp;
    do {
        p = new Node();
        System.out.print("Ingrese un número a la LSL: ");
        p.info = input.nextInt();
        p.link = head;
        this.head = p;
        this.n++;
        System.out.print("¿Agregar más nodos?: ");
        resp = input.next();
    } while (resp.equals("s"));
}
```

Creación de una LSL por el final

En este método, los nodos quedan almacenados en el **orden natural** en que se ingresan. En términos gráficos, podemos decir que la lista se crea de "**izquierda a derecha**".

Ejemplo 2.5
Crear una LSL por el final

Programa Java:

```
public void createEndLSL()
{
    Node p;
    String resp;
    do {
        p = new Node();
        System.out.print("Ingrese un número a la LSL: ");
        p.info = input.nextInt();
```

```
        p.link = null;
        this.n++;
        if (this.head == null) {
            this.head = p;
        } else {
            this.last.link = p;
        }
        this.last = p;
        System.out.print("¿Agregar más nodos?: ");
        resp = input.next();
    } while (resp.equals("s"));
}
```

Operaciones sobre listas

Son varias las operaciones que pueden realizarse sobre listas ligadas en general, independientemente del tipo de información que éstas contengan y del tipo de lista ligada que se esté tratando. Además de las operaciones de creación por el inicio y final de la lista vistas en los ejemplos anteriores, existen otras operaciones fundamentales sobre ellas:

19. Recorrer la lista
20. Buscar un dato
21. Modificar un dato
22. Eliminar un nodo
23. Insertar un nodo antes de una referencia dada
24. Insertar un nodo después de una referencia dada

Recorrer la lista

Esta operación, similar a la usada en arreglos unidimensionales, consiste en visitar cada elemento de la lista, en otras palabras, desplazarnos sobre cada nodo de la lista. Al recorrerla, podemos realizar distintas operaciones sobre la información contenida en los nodos, como imprimirla por ejemplo.

Ejemplo 2.6
Recorrer la lista para mostrar sus elementos (información contenida en cada nodo) e informar cuantos nodos hay en ella.

Pseudocódigo:

```
publico Metodo recorrerLSL()
    punteros p = cab; // O también: Nodo p
    Mientras p <> nulo
        Imprimir p->info;
        p = p->liga;
    FinMientras
    Imprimir "Total nodos: ", n;
```

```
FinMetodo
```

Programa Java:

```
public void scrollLSL()
{
    Node p = this.head;
    while (p != null) {
        System.out.print(p.info + "\t");
        p = p.link;
    }
    System.out.println("\nTotal nodos: " + this.n);
}
```

Buscar un dato en la lista

La función en este caso puede devolver la referencia (puntero) al nodo o un valor booleano para indicar que el dato fue o no encontrado. En el ejemplo a continuación, implementaremos la primera opción.

De igual forma que con los arreglos (vectores), esta búsqueda será útil tanto para la modificación de información como la inserción y eliminación de nodos.

Ejemplo 2.7
Buscar un dato en la lista

Programa Java:

```
public Node searchLSL(int datum)
{
    Node p = this.head;
    Node pSearch = null;
    while (p != null && pSearch == null) {
        if (p.info == datum) {
            pSearch = p;
        } else {
            p = p.link;
        }
    }
    return pSearch;
}
```

Modificar un dato de la lista

En esta operación se busca el dato a modificar, se solicita el nuevo dato y se procede al cambio. Para ello, utilizaremos la búsqueda sobre la LSL.

Ejemplo 2.8
Modificar un dato de la lista

Programa Java:

```
public void updateNodeLSL(Node p, int datum)
{
    p.info = datum;
}
```

Insertar un dato en la lista

En esta operación se busca un dato de referencia para insertar antes o después de él, el nuevo dato; se debe solicitar también el nuevo dato para ser agregado a la lista si la referencia fue encontrada. Veamos los dos casos.

Ejemplo 2.9
Insertar un dato en la lista después de otro dato dado como referencia

Programa Java:

```
public boolean insertAfterNode(int dr, int di)
{
    boolean sw = false;
    Node p, z;
    p = this.head;
    while (p != null && !sw) {
        if (p.info == dr) {
            z = new Node();
            z.info = di;
            z.link = p.link;
            p.link = z;
            this.n++;
            sw = true;
        } else {
            p = p.link;
        }
    }
    return sw;
}
```

Ejemplo 2.10
Insertar un dato en la lista antes de otro dato dado como referencia

Programa Java:

```
public boolean insertBeforeNode(int dr, int di)
{
    boolean sw = false;
    Node z;
```

```
    if (this.head.info == dr) {
        this.n++;
        sw = true;
        z = new Node();
        z.info = di;
        z.link = this.head;
        this.head = z;
    } else {
        Node p = this.head.link;
        Node q = this.head;
        while (p != null && !sw) {
            if ( p.info == dr) {
                this.n++;
                sw = true;
                z = new Node();
                z.info = di;
                z.link = p;
                q.link = z;
            } else {
                p = p.link;
                q = q.link;
            }
        }
    }
    return sw;
}
```

Eliminar un dato de la lista

En esta operación se busca el dato a eliminar; si se encuentra, se procede a eliminarlo.

Ejemplo 2.11
Eliminar un dato de la lista

Programa Java:

```
public boolean deleteNode(int datum)
{
    boolean sw = false;
    if (this.head.info == datum) {
        this.n--;
        sw = true;
        this.head = this.head.link;
    } else {
        Node p = this.head.link;
        Node q = this.head;
        while (p != null && !sw) {
            if ( p.info == datum) {
```

```
                this.n--;
                sw = true;
                q.link = p.link;
            } else {
                p = p.link;
                q = q.link;
            }
        }
    }
    return sw;
}
```

Lista Simplemente Ligada Circular (LSLC)

Una LSLC es muy similar a una LSL, la única diferencia es que el nodo que sigue al último, es el primero, esto es, el último nodo de la lista apunta al registro **cabeza**.

La gráfica 2.3 muestra una LSLC que contiene en su campo *info* números enteros y su primer registro (registro cabeza) es apuntado por la variable apuntador *head*. Un puntero *p* apunta a un nodo arbitrario de la lista. El campo *link* contiene la *dirección* (*referencia*) al siguiente nodo; al ser circular la lista, el siguiente nodo al último, es el primero.

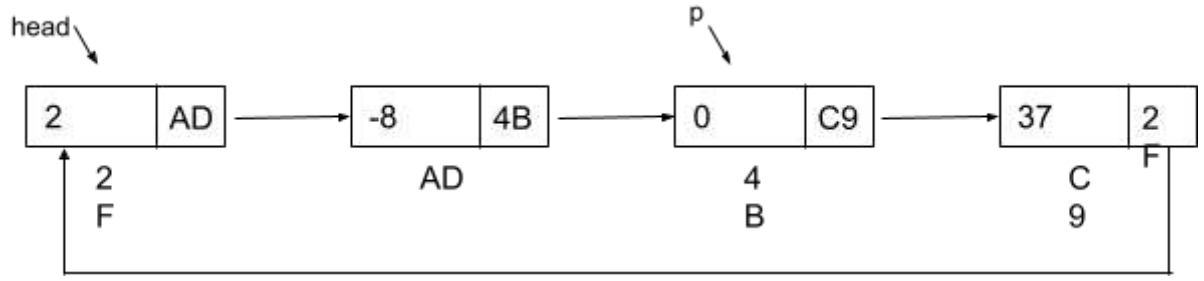


Figura 2.3. Representación de una LSLC que contiene en su campo *info* números enteros y su primer registro es apuntado por la variable apuntador *head*. Un puntero *p* apunta a un nodo arbitrario de la lista. El campo *link* contiene la *dirección* (*referencia*) al siguiente nodo; al ser circular la lista, el siguiente nodo al último, es el primero.

Para el caso de una LSLC con un solo nodo, éste se apunta a sí mismo, tal y como se muestra en la figura 2.4.

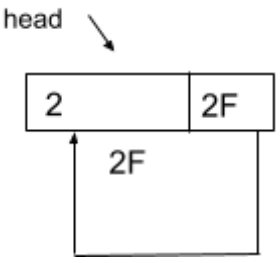


Figura 2.4. Representación de una LSLC con un solo nodo que apunta a sí mismo.

Las operaciones sobre una LSLC son las mismas a las vistas para una LSL, pero deben realizarse algunos ajustes en la implementación de los algoritmos, teniendo en cuenta que ya se está tratando con una LSLC y deben tenerse presente todos los casos posibles. Se ilustrarán algunas operaciones y las demás se dejan como ejercicio a los estudiantes.

Ejemplo 2.12

Crear una clase para gestionar LSLC. La clase es similar a la creada para LSL; se ilustra aquí la creación de la lista por el final y el desplazamiento por cada nodo de la lista para mostrar su contenido; las demás operaciones se dejan como ejercicio para el estudiante. Se mantendrá un puntero al último registro para facilitar la creación de la lista.

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class LinkedListCircular
{
    public static Scanner input = new Scanner(System.in);
    public Node head, last;
    public int n;

    public LinkedListCircular()
    {
        this.head = null;
        this.last = null;
        this.n = 0;
    }

    public void createEndLSLC()
    {
        Node p;
        String resp;
        do {
            p = new Node();
            System.out.print("Ingrese un dato en la LSLC: ");
            p.info = input.nextInt();
            if (this.head == null) {
                this.head = p;
            } else {
                last.link = p;
            }
            p.link = this.head;
            last = p;
            this.n++;
```

```
            System.out.print("¿Desea agregar más nodos? [s/?]: ");
            resp = input.next();
        } while (resp.equals("s"));
    }

    public void scrollLSLC()
    {
        Node p;
        System.out.print(this.head.info + "\t");
        p = this.head.link;
        while (p != this.head) {
            System.out.print(p.info + "\t");
            p = p.link;
        }
    }
}
```

Listas Doblemente Ligadas (LDL)

Un nodo en una LDL está formado al menos por tres campos:

- **Liga Izquierda LI (Left Link LL)**: contiene la referencia al nodo anterior o un valor nulo en su defecto
- **Liga Derecha LD (Right Link RL)**: contiene la referencia al nodo siguiente o un valor nulo en su defecto
- **Información (Information)**: contiene los datos que se almacenan en el nodo. Al igual que con las LSL y LSLC, este "campo" puede ser en realidad un registro tan complejo como se quiera; para los ejemplos a tratar, continuaremos con un campo de tipo entero

La figura 2.5 muestra la estructura de un nodo de una LDL

Figura 2.5. Estructura de un nodo en una LDL

Al tener dos punteros en cada nodo, se puede avanzar o retroceder sobre la lista, esto es, ir hacia adelante o hacia atrás de ésta.

Las operaciones sobre LDL son las mismas que las implementadas en LSL, pero al poder avanzar hacia atrás o adelante de la lista, éstas operaciones se hacen más sencillas. Veremos algunos ejemplos de éstas en los ejemplos presentados más adelante.

Primero vamos a definir la estructura del nodo para una LDL.

Ejemplo 2.13

Clase **NodeLDL (NodoLDL)** para crear nodos en una LDL; la clase básicamente es un registro que contiene los campos **info** de tipo **int (enteros)**, **ll (left link - liga izquierda li)** y **rl (right link - liga derecha ld)** de tipo **Node (Nodo)**.

Pseudocódigo:

```
Clase NodoLDL
    publico enteros info
    publico NodoLDL li
    publico NodoLDL ld
FinClase
```

Programa Java:

```
package com.packages.linked_list;

public class NodeLDL
{
    public int info;
    public NodeLDL ll;
    public NodeLDL rl;
}
```

Ejemplo 2.14

Crear una clase para gestionar LDL. La clase es similar a la creada para LSL; se ilustran varios métodos para procesar la información de la lista, las demás operaciones se dejan como ejercicio para el estudiante. Se mantendrá un puntero al último registro para facilitar la creación y procesamiento de datos en la lista.

Pseudocódigo:

```
Clase ListaDoblementeLigada
    publico NodoLDL cab, ult // O también: publico punteros cab, ult
    publico enteros n

    Constructor ListaDoblementeLigada()
        cab = nulo
        ult = nulo
        n = 0
    FinConstructor

    publico Metodo crearFinalLDL()
        NodoLDL p // O también: punteros p
        cadenas resp
        Repetir
            p = nuevo NodoLDL(); // O también: TRAER(p); p=TRAER()
            Imprimir "Ingrese un dato en la LDL: "
            Leer p->info
            p->ld = nulo
```

```
            p->li = ult
            Si cab == nulo Entonces
                cab = p
            SiNo
                ult->ld = p
            }
            ult = p
            n++
            Imprimir "¿Desea agregar más nodos? [s/?]: "
            Leer resp
        Hasta Que resp <> "s"
    FinMetodo

    publico Metodo recorrerLDL()
        NodoLDL p
        p = cab
        Mientras p <> null
            Imprimir p->info
            p = p->ld
        FinMientras
    FinMetodo

    publico Metodo buscarLDL(Enteros d)
        NodoLDL p = cab
        Logicos sw = Falso
        Mientras p <> null && !sw
            Si p->info == d
                sw = Verdadero
            SiNo
                p = p->ld
            FinSi
        FinMientras
        Retornar p
    FinMetodo

    publico Metodo eliminarLDL(NodoLDL p)
        Si p == cab
            cab = cab->ld
            Si cab <> nulo Entonces
                cab->li = nulo
            FinSi
        SiNo
            Si p == ult
                ult = ult->li
                ult->ld = nulo
            SiNo
                (p->li)->ld = p->ld;
```

```
                (p->ld)->li = p->li;
            FinSi
        FinSi
        n--;
    FinMetodo

    publico Metodo insertAntesLDL(NodoLDL p, Enteros d)
        NodoLDL q = nuevo NodoLDL()
        q->info = d
        q->ld = p;
        Si p == cab
            q->li = nulo
            cab = q;
        SiNo
            (p->li)->ld = q
            q->li = p->li
        FinSi
        p->li = q
        n++
    FinMetodo
FinClase
```

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class DoubleLinkedList
{
    public static Scanner input = new Scanner(System.in);
    public NodeLDL head, last;
    public int n;

    public DoubleLinkedList()
    {
        this.head = null;
        this.last = null;
        this.n = 0;
    }

    public void createEndLDL()
    {
        NodeLDL p;
        String resp;
        do {
            p = new NodeLDL();
```

```
            System.out.print("Ingrese un dato en la LDL: ");
            p.info = input.nextInt();
            p.rl = null;
            p.ll = last;
            if (this.head == null) {
                this.head = p;
            } else {
                last.rl = p;
            }
            last = p;
            this.n++;
            System.out.print("¿Desea agregar más nodos? [s/?]: ");
            resp = input.next();
        } while (resp.equals("s"));
    }

    public void scrollLDL()
    {
        NodeLDL p;
        p = this.head;
        while (p != null) {
            System.out.print(p.info + "\t");
            p = p.rl;
        }
    }

    /**
     * @param d
     * @return p
     */
    public NodeLDL searchLDL(int d)
    {
        NodeLDL p = this.head;
        boolean sw = false;
        while (p != null && !sw) {
            if (p.info == d) {
                sw = true;
            } else {
                p = p.rl;
            }
        }
        return p;
    }

    /**
     * @param p
     */
```

```
    public void deleteLDL(NodeLDL p)
    {
        if (p == this.head) {
            this.head = this.head.rl;
            if (this.head != null) {
                this.head.ll = null;
            }
        } else if (p == this.last) {
            this.last = this.last.ll;
            this.last.rl = null;
        } else {
            (p.ll).rl = p.rl;
            (p.rl).ll = p.ll;
        }
        this.n--;
    }

    public void insertBeforeLDL(NodeLDL p, int d)
    {
        NodeLDL q = new NodeLDL();
        q.info = d;
        q.rl = p;
        if (p == this.head) {
            q.ll = null;
            this.head = q;
        } else {
            (p.ll).rl = q;
            q.ll = p.ll;
        }
        p.ll = q;
        this.n++;
    }
}
```

Lista Doblemente Ligada Circular (LDLC)

Una LDLC es muy similar a una LDL, la única diferencia es que el nodo que sigue al último, es el primero, esto es, el último nodo de la lista apunta al registro **cabeza**, además, ésta también apunta al último registro, es decir, el nodo anterior al nodo cabeza es el último.

Preguntas

- ¿Qué es una lista ligada y pará qué sirve?
- ¿Qué es un nodo cabeza y cuál es su importancia?
- ¿Son mejores las listas ligadas qué los vectores?

- ¿Cuáles son las diferencias entre LSL, LSLC, LDL y LDLC?
- ¿En qué casos debemos utilizar listas ligadas o arreglos unidimensionales?
- ¿En qué otros lenguajes, además de Java, se pueden implementar listas ligadas?
- ¿Por qué se dice que las listas ligadas son estructuras de datos dinámicas lineales y cuál es su diferencia con las llamadas estáticas?
- ¿Qué otras estructuras de datos dinámicas existen, además de las listas ligadas?
- ¿Qué otras estructuras de datos lineales existen, además de las listas ligadas?
- ¿Qué tan compleja puede ser la definición de un nodo?
- ¿Por qué en Java no liberamos memoria luego de utilizar variables de tipo apuntador?
- ¿Qué es un puntero y para qué se utilizan?

Ejercicios

- Leer una cantidad indeterminada de números y guardarlos en una LSL los números positivos y en otra los negativos. Encuentre en ambas listas:
  - Tamaño de cada lista
  - Cual lista es la más extensa
  - Suma de datos (sumatoria)
  - Producto de datos (productoria)
  - Promedio de datos
  - Mayor dato
  - Menor dato
  - Moda
  - Mediana
- Implemente el algoritmo de la burbuja para ordenar los datos en una lista ligada. Considere todos los tipos de listas
- Implemente el uso de LSL para representar polinomios y realizar operaciones sobre ellos. Un polinomio es una expresión algebraica de la forma: $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$. con $a_i \in \mathbf{R}$. El nodo debe estar configurado para permitir agregar el coeficiente a_i ($i = 0, \ldots, n$) y la potencia i. El siguiente es un ejemplo de un polinomio con los coeficientes $10, 5, -7, 0$ y 2 y las potencias $0, 1, 2, 3,$ y 4 respectivamente: $2x^4 - 7x^2 + 5x + 10$. Realice las siguientes operaciones:
  - Sumar dos polinomios
  - Restar dos polinomios
  - Multiplicar dos polinomios
- Implemente las operaciones realizadas en LSL sobre LSLC, LDL y LDLC para:
  - Recorrer la lista
  - Ordenar la lista
  - Buscar un dato
  - Modificar un dato
  - Eliminar un nodo
  - Insertar un nodo antes de una referencia dada
  - Insertar un nodo después de una referencia dada

- Inserte en un dato en una lista ligada que se encuentra ordenada. Considere todos los tipos de listas.
- Aplicaciones con conjuntos. Sean A y B dos conjuntos. Cree dos listas ligadas y realice las operaciones básicas sobre conjuntos (recuerde que un conjunto no contiene elementos repetidos)
    - La cardinalidad de los conjuntos
    - Determine si A y B son iguales: A = B
    - Unión: $A \cup B$
    - Intersección: $A \cap B$
    - Diferencia: $A - B$
    - Diferencia simétrica: $A \Delta B$
    - Complemento de un conjunto cualquiera X ($X = A$ o $X = B$): X'
    - Conjunto potencia de un conjunto cualquiera X ($X = A$ o $X = B$): $P(X)$
    - Producto cartesiano: $A \times B$ y $B \times A$
- Elimine todas las ocurrencias de un dato dado en una lista ligada. Considere todas las variantes de listas
- Elimine los datos repetidos en una lista ligada. Considere todos los tipos de listas
- En un arreglo se almacena el nombre de distintas recetas de comidas. Cada posición del arreglo contiene dos campos: un campo para guardar el nombre de la receta y otro con una dirección a una lista ligada, la cual contiene los ingredientes de la receta. Cree un programa que permita modelar esta situación y además pueda realizar las siguientes operaciones:
    - Crear las recetas
    - Mostrar los ingredientes de una receta dada
    - Agregar ingredientes a una receta dada
    - Quitar ingredientes a una receta dada
    - Agregar nuevas recetas
    - Eliminar recetas
    - ¿Cuál receta contiene más ingredientes?
- En una LDL se almacena el nombre y edad de un grupo de personas. Realice lo siguiente:
    - Cree un menú de opciones para gestionar la información de las personas, según los siguientes literales. Adicione una opción para finalizar el programa.
    - Registrar personas
    - Listado general de personas con todos sus datos
    - Total de personas y promedio de edades
    - Convertir la LDL en LDLC. Si la lista ya fue convertida a LDLC, permitir convertirla de nuevo en LDL. Muestre el tipo de lista actual

Capítulo 8. Pilas y Colas

Introducción

Las **pilas** y **colas** son estructuras de datos abstractas, en el sentido que no existe como tal un tipo de representación directa en los lenguajes de programación; ambas estructuras de datos se representan entonces usando arreglos (vectores) o listas ligadas (en cualquiera de sus tipos), ya que también son estructuras de datos lineales, pues sus elementos ocupan lugares sucesivos en la estructura.

Tanto en el uso de arreglos unidimensionales como listas ligadas, vimos que se pueden agregar o quitar elementos que se encuentren en cualquier parte de estas estructuras; para el caso de las pilas y colas, la inserción y la eliminación deben realizarse por alguno de sus extremos.

Pilas

Una **pila** es una estructura de datos en donde los elementos se insertan y eliminan por uno de los extremos. Los elementos se eliminan en orden inverso al ingresado, esto es, el último elemento en agregarse, será el primero en eliminarse. Debido a esto, una pila es conocida como una estructura **LIFO** (**Last In First Out** -**Último en Entrar Primero en Salir**-).

Ejemplos de pilas en la vida real hay muchos: una pila de platos, las cajas arrumadas en una bodega, productos acomodados en estanterías uno debajo del otro en estanterías, etc. En informática las pilas se aplican en el manejo de la memoria RAM, el tratamiento de expresiones aritméticas y la llamada a subprogramas, entre otras aplicaciones.

Si tratamos de ingresar elementos a una pila llena, se obtendrá un error de **desbordamiento** (**overflow**). En caso de querer eliminar un elemento de una pila vacía, se obtiene un error de **subdesbordamiento** (**underflow**).

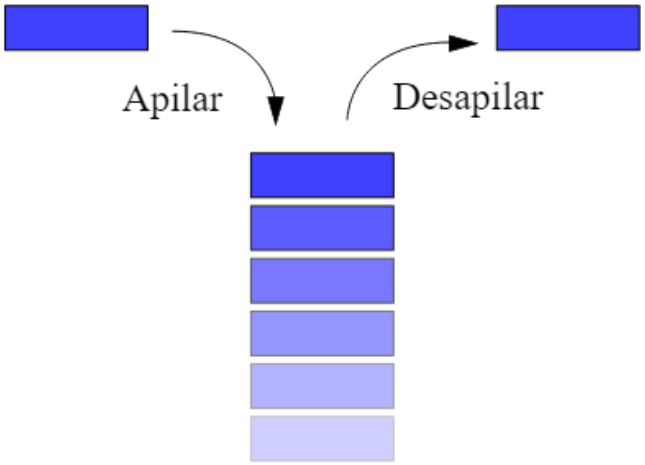


Figura 3.1. Representación de una pila[38]

Operaciones con pilas

Las dos operaciones fundamentales en pilas son **apilar** (agregar un elemento) y **desapilar** (quitar un elemento) siguiendo la metodología LIFO. En los ejemplos presentados más adelante se ilustran estas sencillas operaciones.

Aplicaciones con pilas

Son varios los casos donde las pilas tienen uso a nivel informático, algunos de estos son:

31. **Llamadas a subprogramas**: los lenguajes guardan en una pila las llamadas a subprogramas, esto con el fin de poder controlar el **punto de retorno** de éstos una vez finalizan. Los algoritmos recursivos son un ejemplo claro del uso de pilas cuando éstos se ejecutan; estos algoritmos tienen la particularidad de invocarse a sí mismos.
32. **Manejo de la memoria principal (RAM)**: a medida que se abren aplicaciones en un computador, el sistema operativo los guarda en una pila. Si excedemos la capacidad de la pila, esto es, agotamos la memoria, la máquina se bloqueará
33. **Tratamiento de expresiones aritméticas y lógicas**: una operación común en computación, es convertir expresiones aritméticas y lógicas en notación **infija** a notación **prefija** o **postfija**. Los lenguajes de programación traducen las expresiones en notación infija (las que escribimos los humanos) a notación prefija o postfija; con esto, se eliminan los signos de agrupación (paréntesis), ya que el orden de los operadores y operandos indica cómo deben realizarse las operaciones. La notación prefija y postfija se conoce también como **notación polaca**. Veamos algunos ejemplos, teniendo presente la prioridad natural de cada operador usado (aritmético o lógico):

Ejemplo 3.1

[38] Imagen tomada de Pila (informática) - Wikipedia, la enciclopedia libre

Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación prefija.

a. Si `a + b` es una expresión algebraica cualquiera:
   Expresión infija: `a + b`
   Expresión prefija: `+ab`
   Expresión postfija: `ab+`
b. Si `a + bc = a + b * c` es una expresión algebraica cualquiera:
   Expresión infija: `a + b * c`
   Expresión prefija: `a + *bc = +a*bc`
   Expresión postfija: `a + bc* = abc*+`
c. Si `a * (b + c) - a / d` es una expresión algebraica cualquiera:
   Expresión infija: `a * (b + c) - a / d`
   Expresión prefija: `a * +bc - a / d = *a+bc = *a+bc - /ad = -/ad*a+bc`
   Expresión postfija: `a * bc+ - a / d = abc+* - a / d = abc+* - ad/ = abc+*ad/-`

Ejemplo 3.2
Crear una clase para implementar el manejo de pilas

Programa Java:

```
package com.packages.stacks_tails;

public class Stacks
{
    public int stack[];
    public int top;
    public final int MAX = 50;

    public Stacks()
    {
        this.stack = new int[MAX];
        this.top = 0;
    }

    public void stacking(int datum)
    {
        if (this.top < this.MAX) {
            this.stack[this.top] = datum;
            this.top++;
        } else {
            System.err.println("Error de desbordamiento: pila llena
(overflow)");
        }
    }

    public void unStacking()
```

```
    {
        if (this.top > 0) {
            this.stack[this.top] = 0;
            this.top--;
        } else {
            System.err.println("Error de subdesbordamiento: pila vacía
(underflow)");
        }
    }

    public void showStack()
    {
        int i;
        for (i = this.top - 1; i >= 0; i--) {
            System.out.print("\n___\n" + this.stack[i]);
            // System.out.print(this.stack[i] + " | ");
        }
    }
}
```

Colas

Una **cola** es una estructura de datos en donde los elementos se insertan por un extremo y se eliminan por el otro. Los elementos se eliminan en el mismo orden al ingresado, esto es, el primer elemento en agregarse, será el primero en eliminarse. Debido a esto, una cola es conocida como una estructura **FIFO** (**First In First Out** -**Primero en Entrar Primero en Salir**-).

Ejemplos de colas en la vida real hay muchos: una cola para ingresar al cine, una fila para retirar dinero en un cajero electrónico, la fila para ingresar a un parqueadero, etc. En informática las colas se aplican cuando se comparten recursos, como por ejemplo en una red se cuenta solo con una impresora y se envían varios trabajos a imprimir; de acuerdo a los tiempos de llegada de cada trabajo, éstos entran en la *cola de impresión* y serán atendidos en orden de llegada. Otro caso es cuando se comparten recursos como memoria, procesador u otros, donde se asignarán a medida que vayan entrando solicitudes, si suponemos que todos tienen la misma prioridad.

Análogamente al caso de las pilas, si tratamos de ingresar elementos a una cola llena, se obtendrá un error de **desbordamiento** (**overflow**). En caso de querer eliminar un elemento de una pila vacía, se obtiene un error de **subdesbordamiento** (**underflow**).

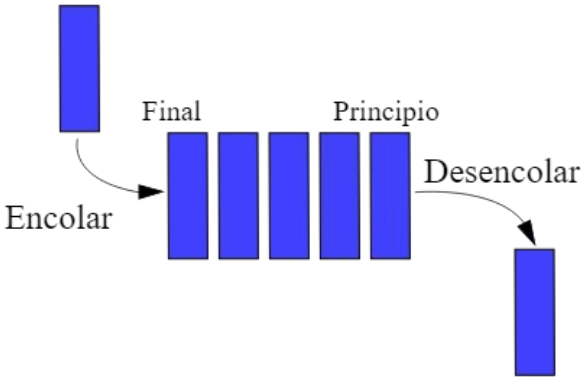


Figura 3.2. Representación de una cola[39]

Operaciones con colas

Las dos operaciones fundamentales en colas son **encolar** (agregar un elemento) y **desencolar** (quitar un elemento) siguiendo la metodología FIFO. En los ejemplos presentados más adelante se ilustran estas sencillas operaciones.

Aplicaciones con colas

Son varios los casos donde las colas tienen aplicaciones en informática. El caso más conocido son las **colas de impresión**, en donde varias estaciones de trabajo comparten una sola impresora; cuando se envían trabajos para imprimir, éstos entran en cola y son atendidos una vez sale el que ya fue atendido (impreso). Otro caso se da cuando se comparten recursos de un servidor, tales como la memoria, el procesador u otros, en donde se atienden de acuerdo al orden de llegada, suponiendo que todas las solicitudes tienen la misma prioridad.

Ejemplo 3.3
Crear una clase para implementar el manejo de colas

Programa Java:

```
package com.packages.stacks_tails;

public class Tails
{
    public int tail[];
    public int end;
    public final int MAX = 50;

    public Tails()
    {
        this.tail = new int[MAX];
        this.end = 0;
```

[39] Imagen tomada de Cola (informática) - Wikipedia, la enciclopedia libre

```
    }

    public void pushTail(int datum)
    {
        if (this.end < this.MAX) {
            this.tail[this.end] = datum;
            this.end++;
        } else {
            System.err.println("Error de desbordamiento: cola llena
(overflow)");
        }
    }

    public void popTail()
    {
        if (this.end > 0) {
            this.deleteElementTail(0);
            this.end--;
        } else {
            System.err.println("Error de subdesbordamiento: cola vacía
(underflow)");
        }
    }

    public void deleteElementTail(int pos)
    {
        int i;
        for (i = pos; i < this.end - 1; i++) {
            this.tail[i] = this.tail[i + 1];
        }
    }

    public void showTail()
    {
        int i;
        for (i = 0; i < this.end; i++) {
            System.out.print(this.tail[i] + " | ");
        }
    }
}
```

Preguntas

Ejercicios

34. Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación infija.
    a. x ^ y + z * w
    b. x - (w / z) * y
    c. a / (a - b) * (c + b)
    d. m / (n - q) ^ p
    e. a * (b + (x * y)) ^ c
    f. z ^ (x + y + z) / w
    g. (a + b) - c * (d + e ^ x)
    h. a + b + c + d + e
    i. (x + y + z) ^ e / a + (b * c)
    j. ((m - n) * (p + q)) ^ r / s
35. Escriba funciones para convertir expresiones escritas en notación infija a notación polaca prefija y postfija, respectivamente.
36. Problema sobre Colas. Una institución educativa universitaria requiere para el departamento de Admisiones y Registro un control básico de las personas que atiende en el día por ventanilla. Las personas se atienden una a una a medida que van llegando, y se guarda de ellas el documento, el nombre y por defecto se establece que no ha sido atendida. A medida que llegan personas, se ubican en la fila y van saliendo de ésta una vez son atendidas.
    a. Cree un menú de opciones para gestionar la atención de personas de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    b. Registrar el ingreso a la fila de una persona
    c. Mostrar la fila de personas a atender
    d. Cuántas personas hay en espera
    e. Atender personas
37. Problema sobre Pilas. Una tienda vende dos tipos de productos (ratón -mouse- y teclados) que llegan en cajas y los organizan en estanterías; de cada producto se conoce su código, nombre y precio, y ambos tipos de productos se organizan de forma independiente de acuerdo con su tipo.
    a. Cree un menú de opciones para gestionar el movimiento del inventario de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    b. Acomode productos en las estanterías de acuerdo con su tipo registrando su ingreso al inventario
    c. Liste los productos disponibles de acuerdo con el tipo seleccionado con todos sus datos e indique cuántos hay en dicha estantería
    d. Muestre el producto de mayor valor de acuerdo a su tipo
    e. Aliste productos para la venta. Los productos que se alistan para la venta salen del inventario; el usuario debe tener la posibilidad de indicar el tipo de producto a alistar y cuántos productos va a alistar

Capítulo 9. Recursividad

La **recursión** o **recursividad** la encontramos en distintas situaciones de la vida cotidiana, por ejemplo cuando se hace una llamada internacional a través de un operador de llamadas y éste transfiere la llamada a otro país. Tomar una foto a otra foto también es un caso de recursión. Un ejemplo de recursión infinita se da cuando se sobreponen dos espejos, donde vemos que la imagen se extiende indefinidamente sobre un pasillo. A nivel matemático también se encuentran muchos ejemplos, muchos de ellos llevados a la aplicación computacional.

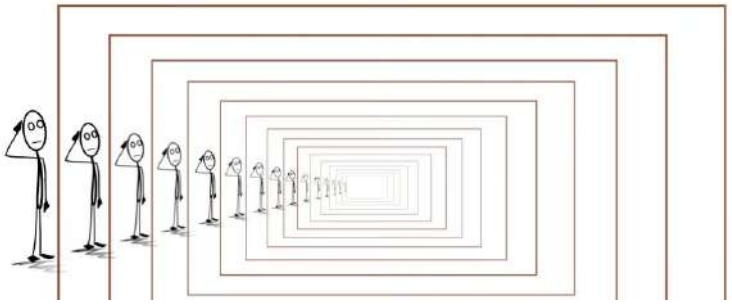


Figura 4.1. Representación de la recursión[40]

Matemáticamente hablando, y por tanto aplicado a la computación, la recursión es una característica de ciertas funciones capaces de definirse en términos de sí mismas. Una función recursiva debe tener o establecer un **estado básico** que no se define de manera recursiva y al cual se van acercando las distintas entradas recursivas de la función; si esto no sucede, se entraría en un ciclo infinito. Por tanto, si un problema cumple con estas características, entonces puede definirse de manera recursiva. Algunos casos de este tipo son el factorial de un número entero, el término n-ésimo de la serie de Fibonacci, el capital acumulado en n años en un banco, etc.

Las listas ligadas pueden ser tratadas de forma recursiva, ya que es posible establecer un estado básico y acercarse a él; otro caso son los árboles, una estructura de datos no lineal que trataremos más adelante.

A pesar de la simpleza con que se pueden resolver algunos problemas de forma recursiva, es claro que su entendimiento requiere un mayor esfuerzo para comprender cómo funciona dicho mecanismo. Uno de los aspectos que ayudan a entender el funcionamiento de la recursión, es realizar la prueba de escritorio en donde se utilizan pilas para guardar los estados de las variables y los puntos de retorno.

Es importante tener presente que el uso de la recursión en computación puede conllevar a altos consumos de memoria, por lo que es importante determinar si de verdad se requiere de su implementación en la solución de problemas; por otro lado, algunos problemas que pueden ser tratados de esta forma, es preferible hacerlos en su correspondiente forma iterativa, dado la complejidad que encierra la recursividad. Sin embargo, y como se verá en el tema de **árboles**, la recursión es imprescindible.

[40] Imagen tomada de Reasons To Use Recursion and How It Works - DEV Community

Ejemplo 4.1

Crear una clase para implementar algunos ejemplos de funciones recursivas en distintos campos, así como casos conocidos ya resueltos de forma iterativa o secuencial.

28. Mostrar los primeros n números naturales
29. Sumar los primeros n números naturales
30. Factorial de un número n
31. Término n-ésimo de la serie de Fibonacci
32. Capital acumulado en n años en un banco.

Pseudocódigo:

```
Clase Recursion
{
    Publico Entero mostrarNumerosNaturales(Entero n)
        Imprimir n
        Si n == 1 Entonces
            Retornar 1;
        SiNo
            Retornar mostrarNumerosNaturales(n - 1);
        FinSi
    FinMetodo

    Publico Entero sumaNumerosNaturales(Entero n)
        Entero sum = n;
        Si n == 1 Entonces
            Retornar 1;
        SiNo
            Retornar sum + sumaNumerosNaturales(n - 1);
        FinSi
    FinMetodo

    Publico Entero factorial(Entero n)
        Si n == 0 Entonces
            Retornar 1
        SiNo
            Retornar n * factorial(n - 1)
        FinSi
    FinMetodo

    Publico Entero fibonacci(Entero n)
        Enteros f
        Si n < 2 Entonces
            f = n
        SiNo
            f = fibonacci(n - 1) + fibonacci(n - 2)
        FinSi
```

```
        Retornar f
    FinMetodo

    Publico Real capital(Real cant, Entero numA, Real ptje)
        Real: capitalActual
        Si  numA == 0 Entonces
            capitalActual = cant
        SiNo
            capitalActual = (1 + ptje) * capital(cant, numA - 1, ptje)
        FinSi
        Retornar capitalActual
    FinMetodo
FinClase
```

Programa Java:

```
package com.packages.recursivity;

public class Recursion
{
    public int showNaturalNumbers(int n)
    {
        System.out.println(n);
        if (n == 1) {
            return 1;
        } else {
            return showNaturalNumbers(n - 1);
        }
    }

    public int sumNaturalNumbers(int n)
    {
        int sum = n;
        if (n == 1) {
            return 1;
        } else {
            return sum + sumNaturalNumbers(n - 1);
        }
    }

    public int factorial(int n)
    {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
```

```
        }
    }

    public int fibonacci(int n)
    {
        int f;
        if (n < 2) {
            f = n;
        } else {
            f = fibonacci(n - 1) + fibonacci(n - 2);
        }
        return f;
    }

    public double capital(double amount, int numYear, double percentage)
    {
        double currentCapital;
        if (numYear == 0) {
            currentCapital = amount;
        } else {
            currentCapital = (1 + percentage) * capital(amount, numYear -
1, percentage);
        }
        return currentCapital;
    }
}
```

Preguntas

Ejercicios

- Recorrer listas LSL, LSLC, LDL y LDLC de forma recursiva
- La *función de Ackermann*[41] se define como:
  $A(m,n) = n + 1,\ si\ m = 0$
  $A(m,n) = A(m-1,1),\ si\ n = 0$
  $A(m,n) = A(m-1, A(m,n-1)),\ en\ otro\ caso$
  Esta función toma dos números enteros positivos como argumentos y devuelve un único entero positivo. Escriba una función recursiva para calcular la función de Ackermann $A(M,N)\ para\ M,N \geq 0$
- El *algoritmo de Euclides* para el cálculo del *Máximo Común Divisor* (*MCD*) se define de la siguiente forma:

[41] Esta función se debe al matemático alemán Wilhelm Ackermann y tiene interés en las ciencias de la computación. En Función de Ackermann - Wikipedia, la enciclopedia libre se habla más sobre esta función, así como en otras fuentes de literatura escrita y digital.

$MCD(m,n) = m,\ si\ n = 0$
$MCD(m,n) = MCD(n, m\ MOD\ n),\ si\ n > 0$
Escriba una función recursiva para calcular $MCD(m,n)\ para\ m,n \geq 0$

- Invierta una palabra de forma recursiva
- Escriba funciones no recursivas para solucionar los problemas 2, 3 y 4
- Muestre el cuadrado de los primeros n números naturales
- Muestre la suma de los cuadrados de los primeros n números naturales de forma recursiva
- Escriba un programa para simular el problema de *Las Torres de Hanoi*[42]. Resuelva de forma iterativa y recursiva. Este problema fue ideado por el matemático francés Edouard Lucas en el año de 1883 inspirado en una leyenda hindú. El problema consiste en resolver lo siguiente: se tienen tres torres a las que llamamos *origen*, *destino* y *auxiliar*; en la torre de origen se encuentran n discos todos de distintos tamaños, ordenados de mayor a menor tamaño desde la base de la torre; se deben pasar los discos a la torre de destino usando como apoyo la torre auxiliar con las siguientes condiciones: 1) solo puede pasarse un disco a la vez; 2) no puede quedar un disco de mayor tamaño sobre uno de menor tamaño. Después de muchas pruebas y de analizar este problema, se ha encontrado que el número de movimientos a efectuar es $2^n - 1$.

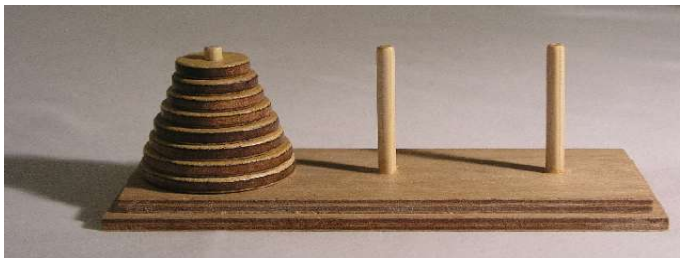


Figura 3.4. Las Torres de Hanoi como juego infantil[43]

[42] Puede encontrar más detalles sobre el curioso problema de Las torres de Hanoi en distinta literatura, tanto de matemáticas como de computación. La siguiente fuente habla un poco acerca de la leyenda hindú y una solución usando Python
4.10. Las torres de Hanoi — Solución de problemas con algoritmos y estructuras de datos
[43] Imagen tomada de: Torres de Hanói - Wikipedia, la enciclopedia libre

Capítulo 10. Árboles

Las estructuras de datos estudiadas hasta ahora son **estructuras lineales**, tanto estáticas como dinámicas. Se dice que éstas son lineales, porque a un elemento solo le puede anteceder o suceder otro elementos, esto es, antes o después de un elemento, solo se encuentra, posiblemente, otro elemento.

Un **árbol** es una estructura de datos **no lineal** y **dinámica** compuesta de elementos que pueden ramificarse, esto es, después de un elemento pueden haber otros elementos. Los árboles mantienen una estructura jerárquica sobre un conjunto de elementos conocidos como **nodos**. El nodo principal, y de cual se genera todo el árbol, se conoce como **nodo raíz**. Esta estructura jerárquica permite usar los conceptos de antecesor, sucesor, padre, hijo, hermano, entre otros.

La siguiente tabla muestra las estructuras de datos lineales y no lineales disponibles en computación, algunas ya conocidas:

Estáticas	Dinámicas	Característica	Observación
Arreglos	Listas ligadas	Lineal	
Conjuntos (tratados con vectores)	Pilas	Lineal	Como lista
Registros (tratados como clases sin métodos)	Colas	Lineal	Como lista
Pilas		Lineal	Como vector
Colas		Lineal	Como vector
	Árboles	No Lineal	Es un tipo especial de grafo
	Grafos	No Lineal	

El nombre de árbol a este tipo de estructura de datos, se debe a su similitud con los árboles naturales, como se verá más adelante en las representaciones gráficas. Matemáticamente, un árbol es un tipo especial de **grafo**, y ambos objetos son tema de estudio permanente en este campo, además del computacional.

Árboles en general

Sea el árbol **`A`** de cualquier tipo. La estructura vacía es considerada un árbol, esto es **`A = nulo`** es un árbol. En general, un árbol A es una estructura homogénea conformada por la unión finita de otros elementos de tipo A, llamados subárboles y que se encuentran disjuntos.

La recursión es una característica inherente a los árboles, por lo que se utilizará esta técnica algorítmica para el tratamiento de éstos.

Los árboles tienen diversas aplicaciones en el mundo real, convirtiendo a este tipo de estructura de datos en una de las más utilizadas para resolver problemas. Un caso muy actual, es el uso de algoritmos basados en árboles para la IA (Inteligencia Artificial); también están las aplicaciones en circuitos, árboles genealógicos, fórmulas matemáticas, indexación de información (como la utilizada en bases de datos -BD-) entre otras.

Un árbol puede representarse de distintas formas, tanto gráficas como tipo conjunto, todas equivalentes. Veamos la siguiente figura:

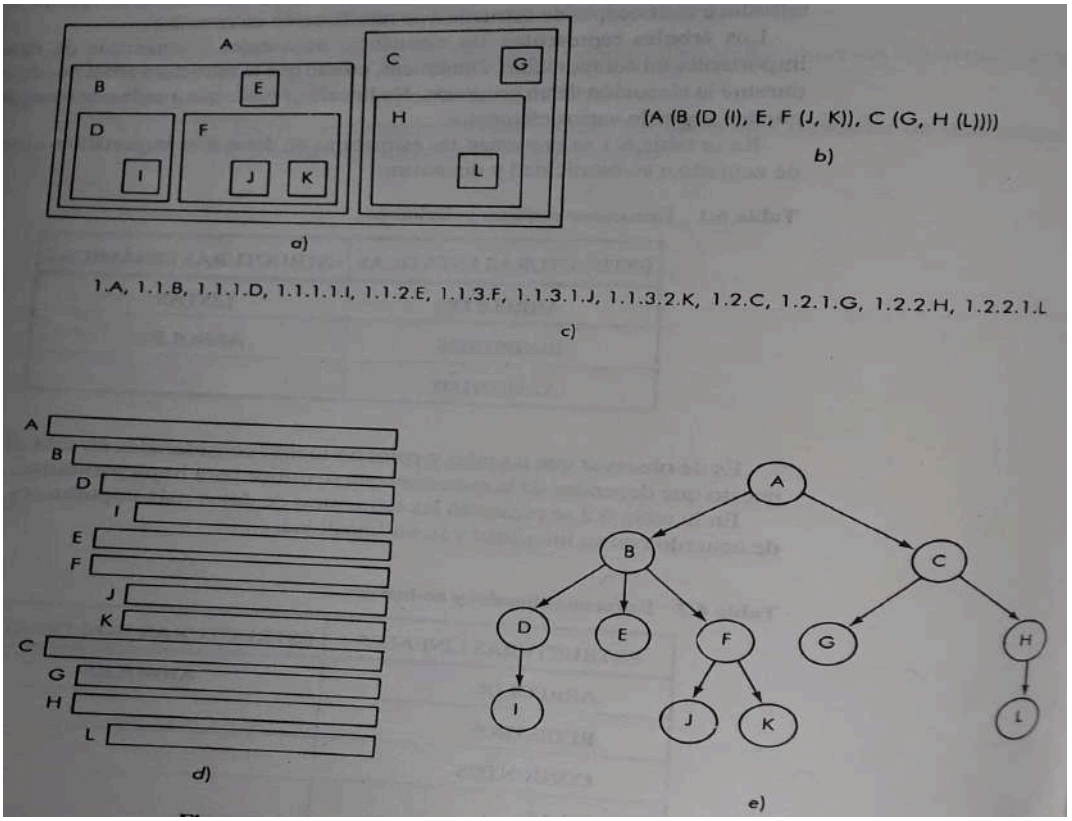


Figura 5.1. Diferentes representaciones de un mismo árbol[44]

- Figura 5.1a) representación por medio de **diagramas de Venn**
- Figura 5.1b) representación por medio de **anidación de paréntesis**
- Figura 5.1c) representación por medio de la **notación decimal de Dewey**[45]
- Figura 5.1d) representación por medio de la **notación indentada**

[44] Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 182.

[45] El Sistema de Clasificación Decimal Dewey es un sistema simple, sólido, lógico y fácil de interpretar, muy útil en bibliotecas y lugares que utilicen clasificaciones de material de manera similar (ver más en Sistema Decimal Dewey). Este nombre se debe a su creador, un bibliotecario llamado Melvil Dewey (1851-1931), que 1875-1876 creó un sistema numérico decimal para organizar los libros de la biblioteca escolar en la que trabajaba (ver más en Sistema de Clasificación Dewey | Biblioteca Pública de Denver)

- Figura 5.1e) representación por medio de **grafos**

Los grafos son los más utilizados en la representación de árboles, y es debido al parecido (abstracto) con este tipo de planta, que recibe este nombre. En los grafos la raíz del árbol se encuentra en la parte superior, como tomando el árbol invertido.

Características y propiedades de los árboles

Sea el árbol `A` de cualquier tipo; se cumplen las siguientes propiedades/características en el árbol:

- Si `A ≠ nulo`, entonces tiene un único **nodo raíz**
- Un nodo cualquiera `N` es descendiente directo de un nodo `M`, si el nodo `N` es apuntado por el nodo `M`: **N es hijo de M**
- Un nodo cualquiera `N` es antecesor directo de un nodo `M`, si el nodo `N` apunta al nodo `M`: **N es padre de M**
- Los nodos descendientes directos del mismo nodo `N` son **hermanos**
- Los nodos sin ramificaciones (sin hijos), reciben el nombre de **hojas** o nodos **terminales**
- Los nodos que no son raíz ni hojas se conocen como nodos **interiores**
- El número de descendientes directos de un nodo cualquiera se conoce como **grado**
- El máximo grado de todos los nodos del árbol es el **grado del árbol**
- Una **arista** (**camino**) es el enlace (liga) entre dos nodos consecutivos
- Una **rama** está compuesta por un conjunto de aristas (caminos) que termina en una hoja
- El número de nodos (o aristas + 1) que deben ser recorridos para llegar a un determinado nodo desde la raíz se conoce como **nivel**. Por definición, la raíz tiene nivel 1 (algunos textos toman la raíz en el nivel 0, siendo el nivel igual al número de aristas desde la raíz hasta el nodo)
- El máximo número de niveles de todos los nodos del árbol se conoce como **altura** o **profundidad** del árbol. También se puede definir como el número máximo de nodos de una rama
- El **peso** del árbol es el número de nodos terminales u hojas

Ejemplo 5.1
Verificar las propiedades y características del árbol T mostrado en la siguiente figura

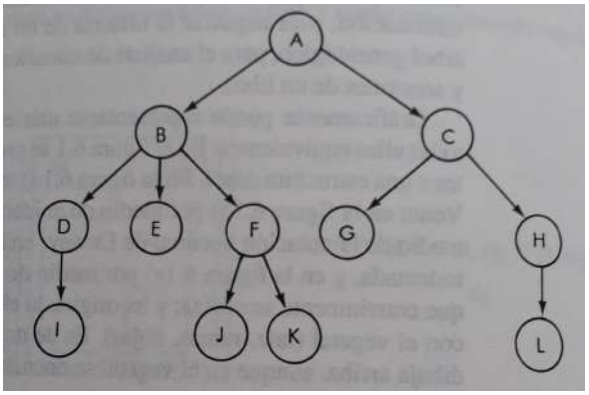


Figura 5.2. Un árbol T cualquiera[46]

- Nodo raíz: `A`, con `A ≠ nulo`
- Relaciones entre nodos
    - B y C son hijos de A (o equivalentemente, A es el padre de B y C), por tanto B y C son hermanos
    - D, E y F son hijos de B y por tanto hermanos; también son *nietos* de de A
    - F es el padre de J y K
    - F y G son *primos* y tienen como abuelo a A
    - L es bisnieto de A
- Hojas o nodos terminales: I, E, J, K, G, L
- Peso del árbol: 6
- Nodos interiores: B, D, F, C, H
- Grado de algunos nodos
    - Grado de A: 2
    - Grado de B: 3
    - Grado de C: 2
    - Grado de D:1
    - Grado de E: 0
- Grado del árbol: 3
- Niveles de los nodos
    - Nivel de A: 1
    - Nivel de B, C: 2
    - Nivel de D, E, F, G, H: 3
    - Nivel de I, J, K, L: 4
- Altura del árbol: 4

Longitud de camino interno y externo

El nivel de los nodos nos ayuda a encontrar la **Longitud de Camino LC** como el número de nodos que deben ser recorridos para llegar a él desde la raíz; así, para el árbol de la figura 5.2 la raíz tiene una `LC = 1`, sus descendientes directos será una `LC = 2`, el nivel 3 tendrá una `LC = 3`, y análogamente con los demás niveles.

[46] Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 184.

Longitud de Camino Interno (LCI)

Se define la **Longitud de Camino Interno (LCI)** como la suma de las longitudes de camino (LC) de todos los nodos del árbol.

$$LCI = \sum_{i=1}^{h} n_i * i$$

Donde:
h: altura del árbol
i: nivel actual en el árbol
n_i: número de nodos en el nivel i

Ejemplo 5.2
Encontrar la LCI del árbol de la figura 5.2

$$LCI = \sum_{i=1}^{h} n_i * i$$

$$h = 4;\ LCI = \sum_{i=1}^{4} n_i * i = 1 * 1 + 2 * 2 + 5 * 3 + 4 * 4 = 36$$

Se define también la **Longitud de Camino Interno Medio (LCIM)** como

$$LCIM = LCI/n$$

Donde n es el número de nodos que hay en el árbol.

Esta medida indica la longitud media (promedio) para visitar un nodo cualquiera desde la raíz.

Ejemplo 5.3
Encontrar la LCIM del árbol de la figura 5.2

$$n = 12; LCI = 36; LCIM = LCI/n = 36/12 = 3$$

Longitud de Camino Externo (LCE)

Para hallar la **Longitud de Camino Externo** de un árbol, primero debemos encontrar su correspondiente **árbol extendido**, el cual se forma a partir del árbol original añadiendo una serie de **nodos especiales**.

Árbol extendido
Es un árbol en el que cada nodo tiene un número de hijos igual al grado del árbol.

Nodo especial
Es aquel que debe agregarse al árbol cuando algunos nodos de éste no cumplen con la condición de árbol extendido. Un nodo cualquiera puede requerir agregar varios nodos

especiales de ser necesario para que cumpla dicha condición. Estos nodos especiales no pueden tener hijos (descendientes) y su objetivo es sustituir **ramas** vacías o nulas.

Ejemplo 5.4
Extender el árbol de la figura 5.2.

La siguiente figura muestra el árbol extendido, donde cada nodo tiene un número de hijos igual al grado del árbol. El grado del árbol original es 3, por consiguiente tenemos:

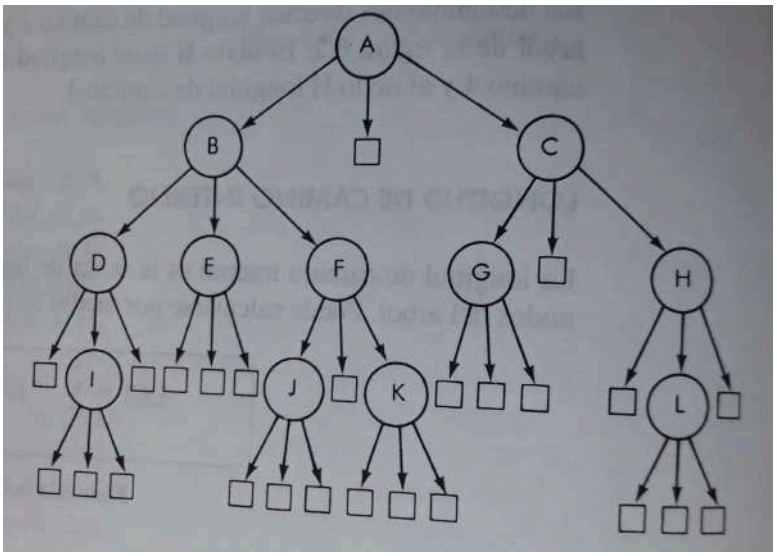


Figura 5.3. Un árbol extendido correspondiente al árbol de la figura 4.2[47]

Observe que estos nodos especiales se representan con cuadrados pequeños.

Para este árbol, hubo que adicionar 25 nodos especiales.

Se define la **Longitud de Camino Externo (LCE)** como la suma de las longitudes de todos los nodos especiales del árbol:

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

Donde:
h: altura del árbol
i: nivel actual en el árbol
ne_i: número de nodos especiales en el nivel i

Ejemplo 5.5
Encontrar la LCE del árbol de la figura 5.3

[47] Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 186.

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

$$h = 4;\ LCI = \sum_{i=2}^{5} ne_i * i = 1 * 2 + 1 * 3 + 11 * 4 + 12 * 5 = 2 + 3 + 44 + 60 = 109$$

Se define también la **Longitud de Camino Externo Medio (LCEM)** como

$$LCEM = LCE/ne$$

Donde ne es el número de nodos especiales que hay en el árbol.

Esta medida indica la longitud media (promedio) para visitar un nodo especial cualquiera desde la raíz.

Ejemplo 5.6
Encontrar la LCEM del árbol de la figura 5.3

$$ne = 25; LCE = 109; LCEM = LCE/ne = 109/25 = 4.36$$

Árboles binarios

Un árbol de grado 2 se conoce como **árbol binario** y goza de particular atención en las ciencias computacionales gracias a sus características y fácil tratamiento.

En un árbol binario cada nodo tiene a lo sumo dos descendientes directos, y podemos distinguirlos como la **rama (subárbol) izquierda** y la **rama (subárbol) derecha**. Con un árbol binario se pueden realizar diversas operaciones, como por ejemplo escribir operaciones algebraicas, representar árboles genealógicos, búsquedas sobre información clasificada (**árboles binarios de búsqueda**), entre otras.

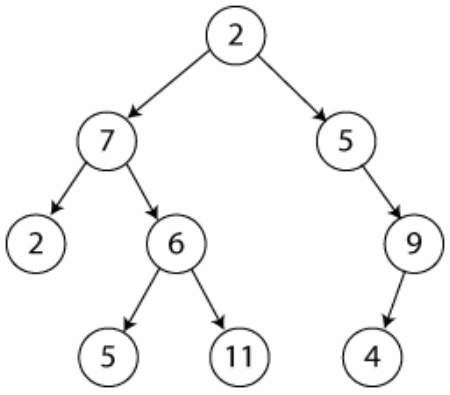


Figura 5.4. Un árbol binario[48]

Ejemplo 5.7

[48] Imagen tomada de Árbol binario - Wikipedia, la enciclopedia libre

En la figura 5.5 se tiene un árbol binario almacenando una expresión aritmética. Escribir la expresión equivalente y mostrar el resultado.

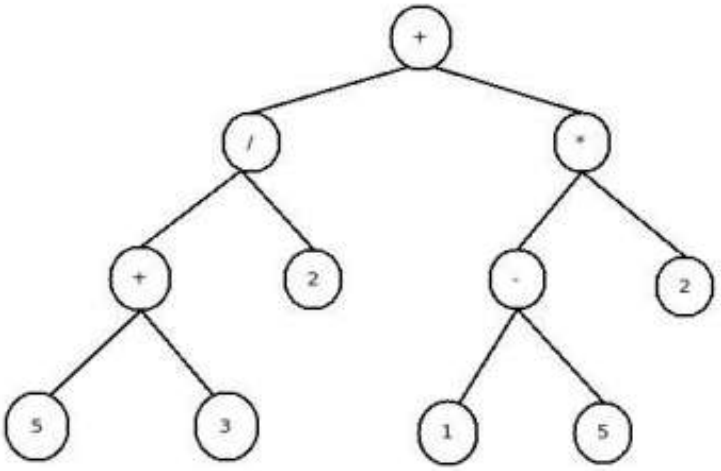


Figura 5.5. Un árbol binario para calcular una expresión aritmética[49]

Solución

Las operaciones de mayor prioridad se encuentran en el nivel más profundo del árbol, dando prioridad al subárbol izquierdo y siguiendo esta mismas condiciones en cada nivel, en el orden de abajo hacia arriba.

```
(((5 + 3) / 2) + ((1 - 5) * 2))
= ((5 + 3) / 2) + ((1 - 5) * 2)
= (5 + 3) / 2 + (1 - 5) * 2
= 4 + (-8)
= -4
```

Nota

Observe que los paréntesis no son requeridos en una representación por medio de árboles binarios de una expresión aritmética.

Ejemplo 5.8

Represente mediante un árbol binario la siguiente expresión algebraica

$(a + 3 * b * c)/(x\text{^}2 - y\%5)$

Solución

La figura 5.6 muestra la representación como árbol binario de la expresión en cuestión

[49] Imagen tomada de Arbol binario para calcular operaciones simples - Stack Overflow en español

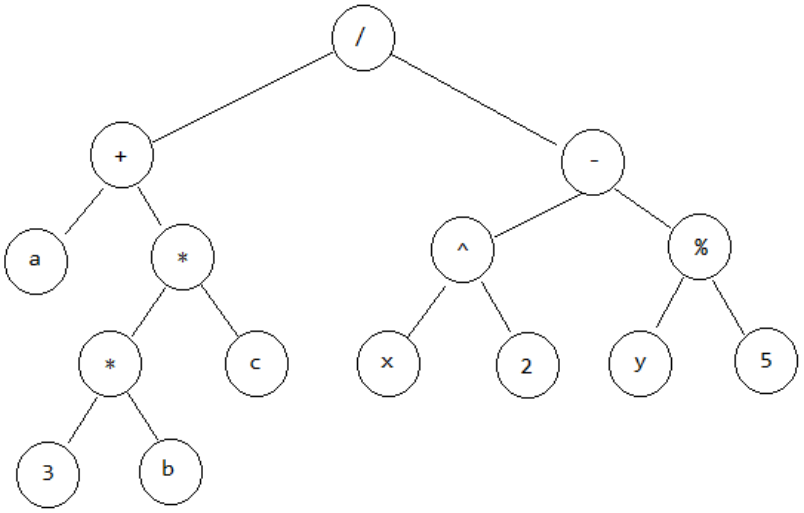


Figura 5.6. Representación de la expresión aritmética del ejemplo 4.8 como un árbol binario

Árboles binarios distintos

Se dice que dos árboles binarios son **distintos** si difieren en su estructura. En la figura 5.7 se muestran dos casos.

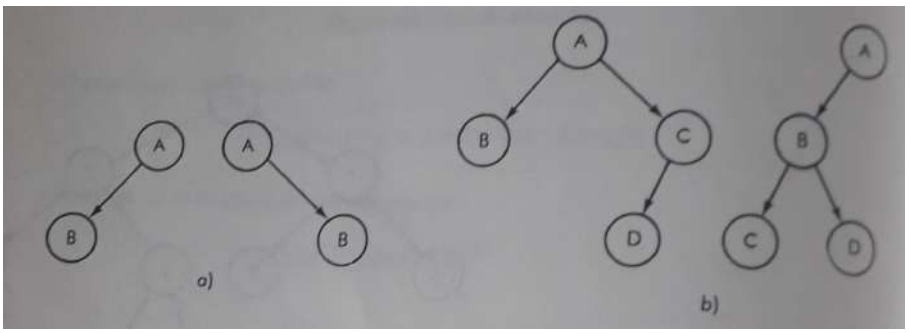


Figura 5.7. Árboles binarios distintos[50]

Árboles binarios similares

Dos árboles binarios son **similares** si sus estructuras son iguales, pero la información contenida en los nodos difiere. La figura 5.8 muestra un par de casos.

[50] Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 190.

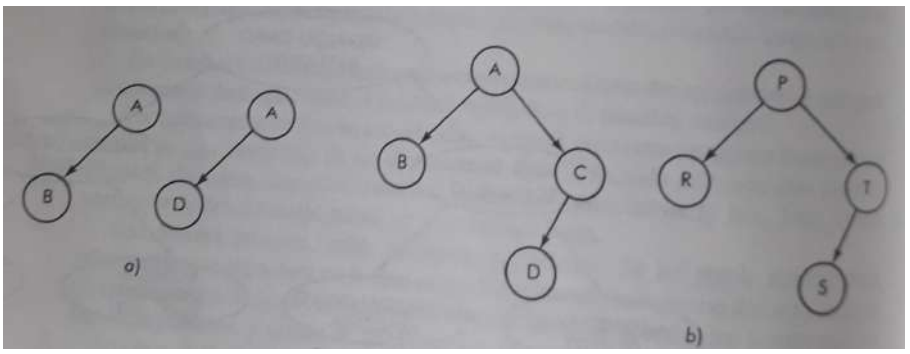


Figura 5.8. Árboles binarios similares[51]

Árboles binarios equivalentes

Dos árboles binarios son **equivalentes** si son similares y además tienen la misma información en cada nodo correspondiente.

Árboles binarios completos

Un árbol binario es **completo**, si todos los nodos, excepto los del último nivel, tienen exactamente dos hijos. Los nodos del último nivel son hojas. La figura 5.9 muestra dos árboles binarios completos.

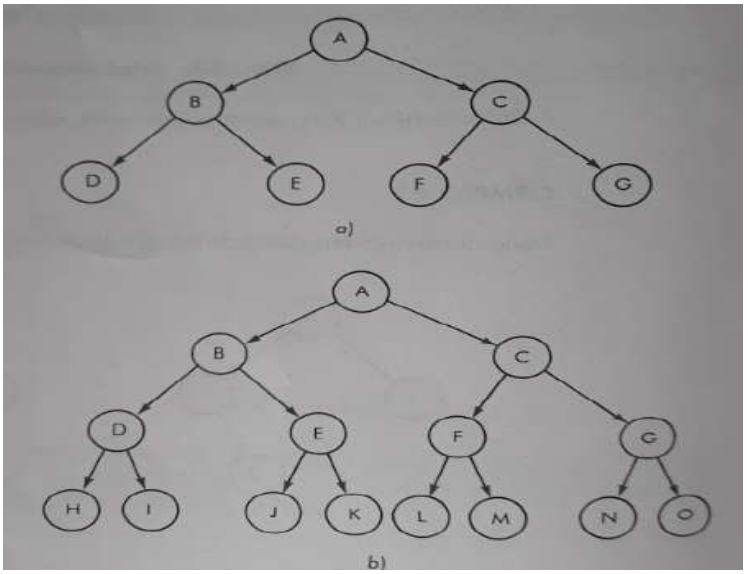


Figura 5.9. Árboles binarios completos[52]

Árboles binarios degenerados o patológicos

Un árbol binario se considera **degenerado**, si todos los nodos tienen solamente un subárbol, esto es, tienen un solo hijo, excepto el último. En un árbol binario degenerado, la altura del árbol es igual al número total de nodos.

[51] Ibídem
[52] Ídem, pág 192

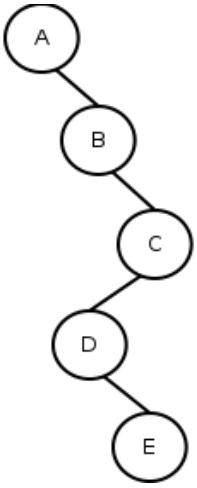


Figura 5.10. Árbol binario degenerado[53]

El número de nodos en un árbol binario completo de altura *h* se calcula mediante la expresión:

$$n_{abc} = 2^h - 1$$

Donde:
n_{abc}*: Número de nodos del árbol binario completo*
h: altura del árbol binario completo

Si conocemos el número de nodos de un árbol binario completo, podemos encontrar su altura *h*:

$$n_{abc} + 1 = 2^h \Rightarrow lg(n_{abc} + 1) = lg(2^h) \Rightarrow lg(n_{abc} + 1) = hlg(2) \Rightarrow lg(n_{abc} + 1) = h$$

Donde:
n_{abc}*: Número de nodos del árbol binario completo*
h: altura del árbol binario completo
lg: logaritmo en base 2

También podemos conocer el número de nodos de un nivel determinado mediante la expresión:

$$n_l = 2^{l-1}$$

Donde:
n_{nl}*: Número de nodos en el nivel l del árbol binario completo*
l: nivel en el árbol binario completo

[53] Imagen tomada de: https://elarbolinformatico.blogspot.com/2015/

Ejemplo 5.9
Encontrar el n_{abc} para los árboles binarios completos de la figura 5.9

Para el árbol de la figura a) se tiene
$h = 3$
$n_{abc} = 2^3 - 1 = 7$ nodos
$n_3 = 2^{3-1} = 2^2 = 4$ nodos en el nivel 3

Para el árbol de la figura b) se tiene
$h = 4$
$n_{abc} = 2^4 - 1 = 15$ nodos
$n_3 = 2^{3-1} = 2^2 = 4$ nodos en el nivel 3
$n_4 = 2^{4-1} = 2^3 = 8$ nodos en el nivel 4

Nota
Algunos textos pueden referirse a los árboles binarios completos como **llenos**.

Representación de árboles binarios en memoria

Hay dos formas de representar árboles binarios, mediante punteros o con el uso de arreglos, sin embargo, la naturaleza dinámica de los árboles hace más conveniente el uso de punteros.

Un nodo en un árbol binario es representado como un registro (o clase sin métodos en los lenguajes OO), similar al definido para una LDL, que contiene al menos tres campos (o propiedades en OO). La figura 5.11 muestra dicha representación como un registro:

Figura 5.11. Estructura de un nodo en un árbol binario como registro

Los campos de este registro almacenan lo siguiente:

h. **INFORMATION - INFORMACIÓN**: en su forma más simple, almacena un dato primitivo (entero, real, lógico, carácter), pero puede ser tan complejo como se quiera para que contenga arreglos, otros registros, objetos, etc.
i. **LB (Left Branch) - IZQ (Subárbol Izquierdo)**: campo de tipo puntero que guardará la referencia a un nodo apuntado en el subárbol izquierdo o nulo sino apunta a ningún subárbol (árbol vacío)
j. **RB (Right Branch) - DER (Subárbol Derecho)**: campo de tipo puntero que guardará la referencia a un nodo apuntado en el subárbol derecho o nulo sino apunta a ningún subárbol (árbol vacío)

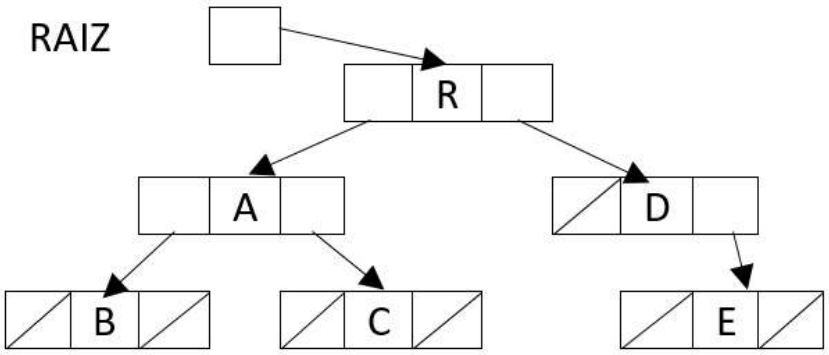


Figura 5.12. Representación de un árbol binario en memoria[54]

Operaciones con árboles binarios

Las operaciones fundamentales en árboles binarios están relacionadas con su creación, inserción y borrado de nodos, así como visitar sus elementos (recorrer el árbol. Veamos la creación del árbol y como recorrerlo; las otras dos operaciones se tratarán con los árboles binarios de búsqueda.

Creación de un árbol binario

Esta operación, de manera similar al manejo de listas ligadas, consiste en cargar los nodos que se requieran en la memoria, solo que aquí se hará de forma recursiva. En los ejemplos que siguen a continuación, se ilustra ésta y otras operaciones, entre ellas el desarrollo algorítmico del problema.

Recorrido en árboles binarios

Esta operación consiste en visitar cada nodo del árbol y efectuar alguna operación sobre la información de éste: mostrarla, usarla en cálculos, etc. Hay tres formas de naturaleza recursiva para realizar esto:

f. **Recorrido en preorden**
   i. Visitar la raíz
   ii. Recorrer el subárbol izquierdo
   iii. Recorrer el subárbol derecho

g. **Recorrido en inorden**
   i. Recorrer el subárbol izquierdo
   ii. Visitar la raíz
   iii. Recorrer el subárbol derecho

h. **Recorrido en postorden**
   i. Recorrer el subárbol izquierdo
   ii. Recorrer el subárbol derecho

[54] Imagen tomada de: Arboles Binarios de Búsqueda (ABB) – DEFINICIÓN, ESTRUCTURA, REPRESENTACIÓN EN MEMORIA DINÁMICA, OPERACIONES, RECORRIDOS – – Programación C++

iii. Visitar la raíz

Ejemplo 5.10
Mostrar como son los recorridos en el árbol de la figura 5.6

Solución

Preorden: /+a**3bc-^x2%y5
Inorden: a+3*b*c/x^2-y%5
Postorden: a3b*c*+x2^y5%-/

Ejemplo 5.11
Clase **NodeTree (NodoArbol)** para crear nodos en un árbol binario; la clase, sin métodos, básicamente es un registro que contiene los campos **info** de tipo **int (enteros)**, **lb (left branch - subárbol izquierdo izq)** y **rb (right branch - subárbol derecho der)** de tipo **Node (Nodo)**.

Pseudocódigo:
```
Clase NodoArbol
    publico enteros info
    publico NodoArbol izq
    publico NodoArbol der
FinClase
```

Programa Java:
```
package com.packages.trees;

public class NodeTree
{
    public int info;
    public NodeTree left;
    public NodeTree right;
}
```

Ejemplo 5.12
Clase **BinaryTree (ArbolBinario)** para realizar distintas operaciones sobre árboles binarios, entre ellas las siguientes:

Crear el árbol
Recorrer el árbol: preorden, inorden y postorden
Encontrar el total de nodos

Programa Java:
```
package com.packages.trees;

```

```
import java.util.Scanner;

public class BinaryTree
{
    public static Scanner input = new Scanner(System.in);

    public BinaryTree()
    {

    }

    public void loadNode(NodeTree node)
    {
        NodeTree p;
        String resp;

        System.out.print("Ingrese un dato para el árbol: ");
        node.info = input.nextInt();

        System.out.print("¿Desea agregar nodos por el subárbol izquierdo?
[s/?]: ");
        resp = input.next().toLowerCase();
        if (resp.equals("s")) {
            p = new NodeTree();
            node.left = p;
            loadNode(node.left);
        } else {
            node.left = null;
        }

        System.out.print("¿Desea agregar nodos por el subárbol derecho?
[s/?]: ");
        resp = input.next().toLowerCase();
        if (resp.equals("s")) {
            p = new NodeTree();
            node.right = p;
            loadNode(node.right);
        } else {
            node.right = null;
        }
    }

    public void traversePreorder(NodeTree node)
    {
        if (node != null) {
            System.out.println("Nodo: " + node.info);
            traversePreorder(node.left);
```

```
            traversePreorder(node.right);
        }
    }

    public void traverseInorder(NodeTree node)
    {
        if (node != null) {
            traverseInorder(node.left);
            System.out.println("Nodo: " + node.info);
            traverseInorder(node.right);
        }
    }

    public void traversePostorder(NodeTree node)
    {
        if (node != null) {
            traversePostorder(node.left);
            traversePostorder(node.right);
            System.out.println("Nodo: " + node.info);
        }
    }


    public int totalNodes(NodeTree node)
    {
        if (node != null) {
            return 1 + totalNodes(node.left) + totalNodes(node.right);
        } else {
            return 0;
        }
    }
}
```

Representación de árboles generales como árboles binarios

Un árbol general puede convertirse en un árbol binario. El alcance del curso no cubre este aspecto, pero se deja al lector consultar el algoritmo respectivo para realizar esta operación; en los textos de Estructuras de Datos mencionados en la bibliografía se encuentra dicho desarrollo.

Bosque de árboles

Un conjunto de 2 o más árboles se conoce como Bosque de árboles. Éstos también pueden ser representados mediante árboles binarios. Se deja como ejercicio consultar o desarrollar el algoritmo respectivo.

Árboles Binarios de Búsqueda (ABB)

Este tipo especial de árbol facilita las operaciones de acceso a los elementos, tales como la búsqueda, inserción y eliminación de nodos. Generalmente, los árboles binarios de búsqueda no contienen información repetida.

La definición formal de un **árbol binario de búsqueda** es la siguiente:

Definición: árbol binario de búsqueda
Para todo nodo N del árbol debe cumplirse que todos los valores del subárbol izquierdo deben ser menores o iguales al valor de N; análogamente, se debe cumplir que todos los valores del subárbol derecho deben ser mayores o iguales al valor de N.

La siguiente figura muestra un árbol binario de búsqueda.

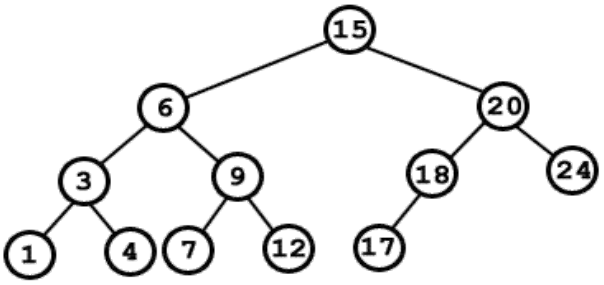


Figura 5.13. Representación de un árbol binario de búsqueda[55]

Ejemplo 5.13
Crear un método para buscar un elemento en un árbol binario de búsqueda en la Clase **`BinaryTree`**.

Programa Java:

```
public void searchBinary(NodeTree node, int datum)
{
    if (node != null) {
        if (datum < node.info) {
            searchBinary(node.left, datum);
        } else if (datum > node.info) {
```

[55] Imagen tomada de: Operaciones básicas de los árboles binarios de búsqueda

```
            searchBinary(node.right, datum);
        } else {
            System.out.println("Nodo encontrado en el árbol");
        }
    } else {
        System.out.println("Nodo no encontrado en el árbol");
    }
}
```

Ejemplo 5.14

Crear un método para insertar un elemento en un árbol binario de búsqueda en la Clase **`BinaryTree`**. Los datos en el árbol se suponen únicos.

Programa Java:

```
public void insertBinary(NodeTree node, int datum)
{
    NodeTree p;
    if (datum < node.info) {
        if (node.left == null) {
            p = new NodeTree();
            p.info = datum;
            p.left = null;
            p.right = null;
            node.left = p;
        } else {
            insertBinary(node.left, datum);
        }
     } else if (datum > node.info) {
        if (node.right == null) {
            p = new NodeTree();
            p.info = datum;
            p.left = null;
            p.right = null;
            node.right = p;
        } else {
            insertBinary(node.right, datum);
        }
    } else {
        System.out.println("El nodo ya se encuentra en el árbol");
    }
}
```

Preguntas

- ¿Qué es un árbol?
- ¿Qué es un árbol binario?

- ¿Qué es un árbol completo?
- ¿Qué es un árbol AVL?
- ¿Qué es un grafo?

Ejercicios

- Realice los siguientes cálculos para encontrar el valor numérico de cada expresión si $a = 3$, $b = 4$, $c = -1$, $d = -5$, $e = 2$, reescriba cada expresión en notación algorítmica y muestre su representación mediante árboles binarios
  - $ab^2 + 3c - \sqrt{b}$
  - $5e - 2bcd + 4(d^3 - 2a + c) + a \,\%\, e$
  - $(\frac{b}{e} + \frac{e}{c}) - 6(\frac{c^2}{a})$
  - $\sqrt[3]{d^6} + \frac{a+b+c}{e} + e - b \,\%\, 2$
  - $\sqrt{ab - a} + 5d \div c - a^4be$
- Muestre los árboles del punto anterior utilizando la notación de Dewey, notación indentada, notación de paréntesis y mediante diagramas de Venn
- En un vector se almacena el número de nodos en cada nivel correspondiente a un árbol general. Cree una función que calcule la LCI
- Si se sabe que el número de nodos de un árbol general es n, cree una función para calcular la LCIM usando el resultado del ejercicio anterior
- Calcule el NN_{abc} para árboles binarios completos de altura 2, 3, 4, 5, 6, 7, 8, 9 y 10, respectivamente
- Realice las siguientes operaciones sobre un árbol binario de raíz root que contiene números enteros:
  - Muestre los números pares e impares
  - Muestre el total de números pares
  - Muestre el total de números impares
  - Encuentre la suma de los números
  - Encuentre el promedio de los números
  - Encuentre el mayor de los números
  - Encuentre el menor de los números
  - Muestre los números primos
  - Encuentre el total de números primos
  - Muestre solo los nodos del subárbol izquierdo
  - Muestre solo los nodos del subárbol derecho
  - Muestre y cuente las hojas del árbol
  - Muestre solo los nodos que tengan exactamente dos hijos
  - Buscar un nodo
  - Insertar un nodo
  - Eliminar un nodo
- Implemente un algoritmo para crear árboles generales
- Implemente un algoritmo para convertir árboles generales en árboles binarios

Capítulo 11. Grafos

En matemáticas y en ciencias de la computación, la teoría de grafos (también llamada teoría de las gráficas) estudia las propiedades de los **grafos** (también llamadas **gráficas**). Un grafo es una estructura discreta compuesta y no vacía, por un conjunto de objetos llamados **vértices** (o **nodos**) y una selección de pares de vértices llamados **aristas** (o **arcos**) (*edges* en inglés) que pueden ser orientados o no. Típicamente, un grafo se representa mediante una serie de puntos (los vértices) conectados por líneas (las aristas).

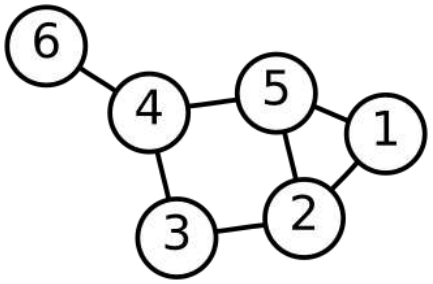


Figura 6.1. Grafo con seis vértices y siete aristas

Definiciones fundamentales

Vértice

Los **vértices** constituyen uno de los dos elementos que forman un grafo. Como ocurre con el resto de las ramas de las matemáticas, a la Teoría de Grafos no le interesa saber qué son los vértices, sin embargo también los podemos llamar **nodos**, que se representan como "**puntos**" en la gráfica.

Diferentes situaciones en las que pueden identificarse objetos y relaciones que satisfagan la definición de grafo pueden verse como grafos y así aplicar la Teoría de Grafos en ellos.

Arista

Una **arista** es una **línea**, orientada o no, que une dos vértices. Corresponde a una **relación** entre dos vértices de un grafo. En un grafo no dirigido, se trata de relaciones simétricas sin dirección, mientras que en un grafo dirigido son relaciones direccionales, también conocidas como **arcos**.

Grafo

Un **grafo** es una **relación** entre dos **conjuntos** `G = (V, A)`, donde `V` es el conjunto de **vértices** y `A` es el conjunto de **aristas**; este último es un conjunto de pares de la forma `(u, v)` tal que `u, v ∈ V`. También se pueden usar otras notaciones equivalentes para indicar

una arista: `(a, b) = ab = a → b`; la última forma se aplica cuando el grafo es dirigido; las otras dos pueden usarse en grafos no dirigidos, aunque el contexto también puede indicar lo contrario.

En teoría de grafos, sólo queda lo esencial del dibujo: la forma de las aristas no son relevantes, sólo importa a qué vértices están unidas. La posición de los vértices tampoco importa, y se puede variar para obtener un dibujo más claro.

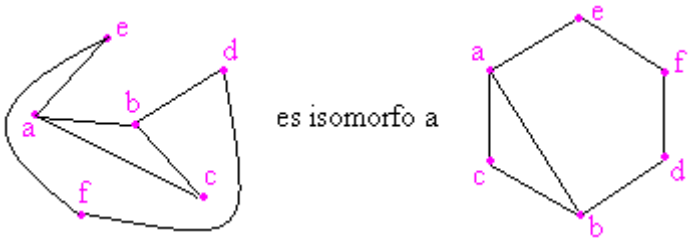


Figura 6.2. Grafos isomorfos[56]. V = { a, b, c, d, e, f }, y A = { ab, ac, ae, bc, bd, df, ef }.

Muchas situaciones pueden ser modeladas con un grafo: redes de uso cotidiano, una red de carreteras que conecta ciudades, una red eléctrica o la red de gas de una ciudad, el control de flujo de un programa (los vértices pueden ser sentencias de control y los arcos la transferencia del flujo), etc.

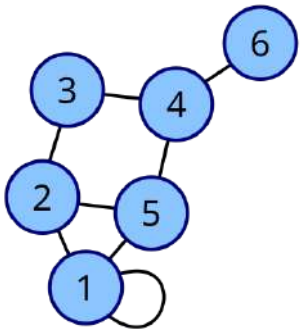


Figura 6.3. Grafo no dirigido

Subgrafo

Un **subgrafo** de un grafo `G` es un grafo cuyos conjuntos de vértices y aristas son subconjuntos de los de `G`. Se dice que un grafo `G` contiene a otro grafo `H` si algún subgrafo de `G` es `H` o es isomorfo a `H` (dependiendo de las necesidades de la situación).

El **subgrafo inducido** de `G` es un subgrafo `G'` de `G` tal que contiene todas las aristas adyacentes al subconjunto de vértices de `G`.

[56] "Isomorfo" se refiere a objetos que tienen la misma forma o estructura, incluso si están compuestos de diferentes materiales o tienen propiedades distintas. En varios contextos, el concepto de "isomorfismo" describe una relación de equivalencia estructural.

Definición

Sea G = (V, A). G'=(V', A') se dice subgrafo de G si:

- V' ⊆ V
- A' ⊆ A
- (V', A') es un grafo

Si G' = (V', A') es subgrafo de G, para todo v ∈ G se cumple gr(G',v) ≤ gr(G, v)

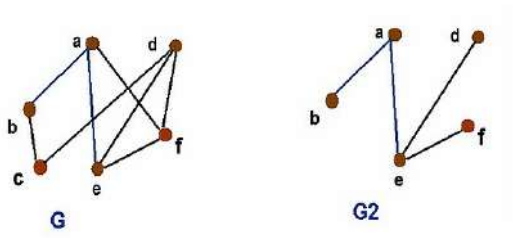


Figura 6.4. G2 es un subgrafo de G.

Grafo dirigido

Un **grafo dirigido** G consiste en un conjunto de vértices V y un conjunto de arcos dirigidos A. Un arco o arista es un **par ordenado** de vértices (v, w); v es la *cola* y w es la *cabeza* de la arista.

Figura 6.5. Representación gráfica del arco (v, w) = v → w.

En la figura puede verse como la "cola" está en el vértice v, mientras que la punta de la flecha indica la "cabeza" en el vértice w del par ordenado. Se dice que el arco v → w va de v a w, y que w es adyacente a v.

Los vértices y los arcos pueden tener una **etiqueta**, la cual puede representar un nombre o un valor. En la figura 6.5 los nodos están etiquetados como v y w respectivamente.

Ejemplo 6.1

Grafo dirigido con 6 vértices (nodos) y 8 aristas (arcos). Cada vértice y cada arista está etiquetada

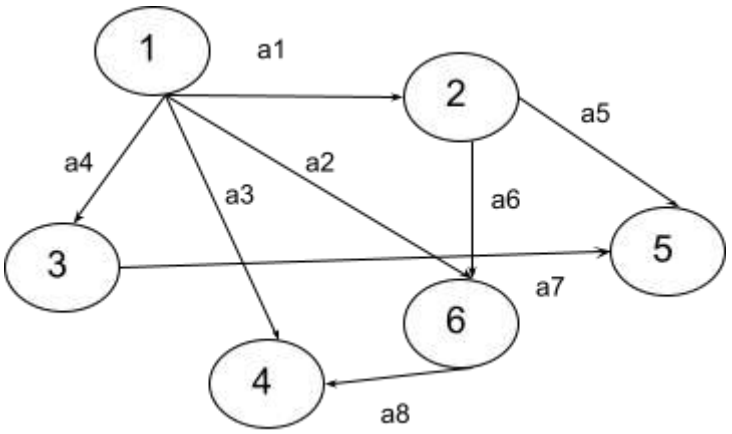


Figura 6.6. Grafo dirigido con 6 vértices (nodos) y 8 aristas (arcos)

```
V = {1, 2, 3, 4, 5, 6}
A = {a1, a2, a3, a4, a5, a6, a7, a8} = {(1, 2), (1, 6), (14), (1,
3), (2, 5), (2, 6), (3, 5), (6, 4)}
```

Observe cómo el orden de las parejas ordenadas en el conjunto A de las aristas especifican su dirección.

Camino en un grafo

Un **camino en un grafo** es una secuencia de vértices $v_1, v_2, \ldots, v_n$ tal que $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3, \ldots, v_{n-1} \rightarrow v_n$ son arcos. Este camino va del vértice v_1 al vértice v_n, pasa por los vértices $v_2, v_3, \ldots, v_{n-1}$ y termina en el vértice v_n.

Longitud de un camino

La **longitud de un camino** es el número de arcos en ese camino. Como consecuencia, se tiene que la longitud de un camino $v \rightarrow v$ es 0.

Camino simple

Un camino es simple si todos sus vértices, excepto quizá el primero y el último son distintos.

Ciclo simple

Es un camino simple de longitud por lo menos uno, que empieza y termina en el mismo vértice.

Camino hamiltoniano
En teoría de grafos es una trayectoria que visita cada vértice de un grafo exactamente una vez.

Ciclo hamiltoniano

Si en un camino hamiltoniano, el recorrido comienza y termina en el mismo vértice, se llama **ciclo hamiltoniano** o **circuito hamiltoniano**. Es un ciclo donde se recorren todos los vértices exactamente una vez, excepto quizá el vértice del que parte y al cual llega.

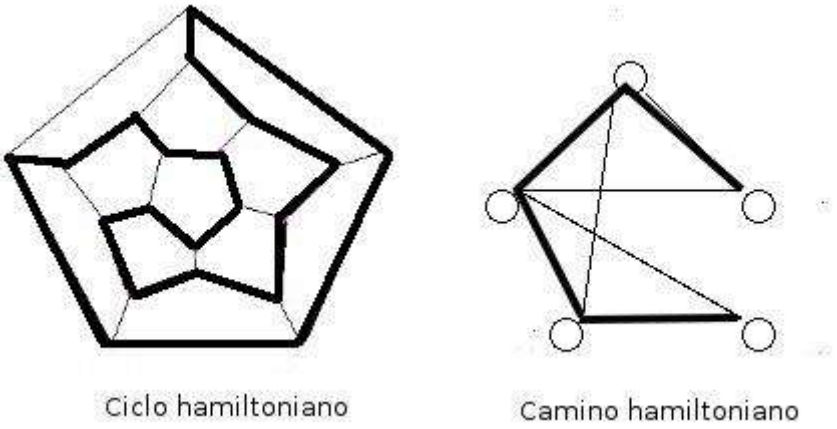


Figura 6.7. Ciclo y camino hamiltoniano[57]

Ejemplo 6.2

En el grafo de la figura 6.6 la secuencia (camino) `1, 2, 6, 4` es tal que `1 → 2, 2 → 6, 6 → 4` son arcos y tiene una longitud de camino igual a 3; la secuencia `1, 3, 5` con los arcos `1 → 3, 3 → 5`, tiene una longitud igual a 2. Se observa que no hay ciclos simples en este grafo.

Grafos no dirigidos

Un grafo no dirigido (muchas veces descrito simplemente como grafo) G = (V, A) consta de un conjunto finito de vértices V y de un conjunto de aristas A. Se diferencia de un grafo dirigido en que cada arista en A es un par no ordenado de vértices.

Estructuras de datos en la representación de grafos

Existen diferentes formas de almacenar grafos en un computador. La estructura de datos depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

[57] Imagen tomada de: https://poliformat.upv.es

Estructura de lista

2. **Lista de incidencia**: las aristas son representadas con un vector de pares (ordenados, si el grafo es dirigido), donde cada par representa una de las aristas.

3. **Lista de adyacencia**: cada vértice tiene una lista de vértices los cuales son adyacentes a él. Esto causa redundancia en un grafo no dirigido (ya que `A` existe en la lista de adyacencia de `B` y viceversa), pero las búsquedas son más rápidas.

En esta estructura de datos la idea es asociar a cada vértice `i` del grafo una lista que contenga todos aquellos vértices `j` que sean adyacentes a él. De esta forma sólo reservará memoria para los arcos adyacentes a `i` y no para todos los posibles arcos que pudieran tener como origen `i`. El grafo, por tanto, se representa por medio de un vector de `n` componentes (`|V| = n`) donde cada componente va a ser una lista de adyacencia correspondiente a cada uno de los vértices del grafo. Cada elemento de la lista consta de un campo indicando el vértice adyacente. En caso de que el grafo sea etiquetado, habrá que añadir un segundo campo para mostrar el valor de la etiqueta.

Estructuras matriciales

- **Matriz de incidencia**: el grafo está representado por una matriz de `A` (aristas) por `V` (vértices), donde `[arista, vértice]` contiene la información de la arista (1 - conectado, 0 - no conectado). También puede representarse con una matriz de `V` (vértices) por `A` (aristas), donde `[vértice, arista]` contiene la información de la arista (1 - conectado, 0 - no conectado).

- **Matriz de adyacencia**: el grafo está representado por una matriz cuadrada `M` de tamaño n^2, donde `n` es el número de vértices. Si hay una arista entre un vértice `x` y un vértice `y`, entonces el elemento $m_{x,y}$ es `1`, de lo contrario, es `0`.

Operaciones básicas con grafos en computación

Las operaciones básicas con grafos en computación incluyen la comprobación de la existencia de una arista entre dos vértices, la obtención de vértices adyacentes, la inserción y eliminación de vértices y aristas, y la determinación de la adyacencia de un vértice a otro. Aquí un listado de las más comunes:

- Inserción de un vértice: agrega un nuevo vértice al grafo.

- Recorrido del grafo: permite visitar todos los vértices y aristas del grafo, normalmente en un orden determinado.

- Determinación de la adyacencia: verifica si un vértice es adyacente a otro.

- Búsqueda de un vértice: permite encontrar un vértice específico en el grafo.

- Comprobación de la existencia de una arista: determina si existe una conexión (arista) entre dos vértices específicos.

- Obtención de vértices adyacentes: devuelve la lista de vértices a los que un vértice dado está conectado.

- Eliminación de un vértice: borra un vértice y todas las aristas que lo conectan a otros vértices.

- Inserción de una arista: añade una nueva conexión entre dos vértices existentes.

- Eliminación de una arista: elimina la conexión entre dos vértices.

Ejemplo 6.3

La siguiente figura muestra un grafo no dirigido con 6 vértices y 8 arcos. Encontrar y representar:

- Matriz de incidencia
- Matriz de adyacencia
- Lista de incidencia
- Lista de adyacencia

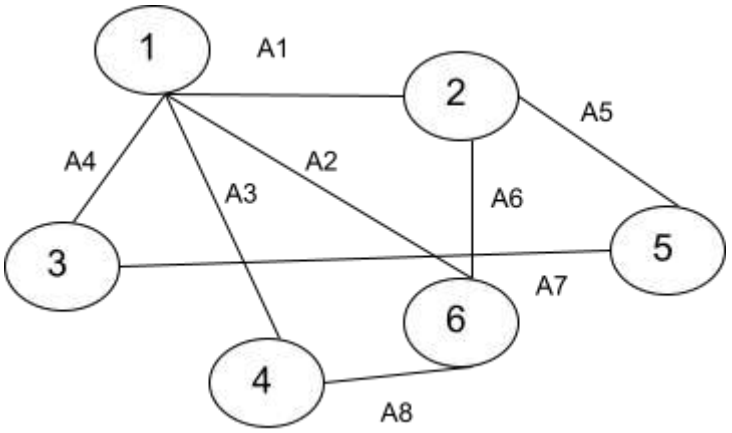


Figura. Grafo no dirigido con 6 vértices (nodos) y 8 aristas (arcos)

Matriz de incidencia

Esta matriz está conformada de `m` vértices y `n` aristas. En un vector columna almacenamos los vértices del grafo y en un vector fila almacenamos sus aristas. Si `matriz[vértice, arista] = 1` es por que el vértice está conectado con la arista, y es igual a cero en caso contrario.

Vertice / Arista	A1	A2	A3	A4	A5	A6	A7	A8
1	1	1	1	1	0	0	0	0
2	1	0	0	0	1	1	0	0

3	0	0	0	1	0	0	1	0
4	0	0	1	0	0	0	0	1
5	0	0	0	0	1	0	1	0
6	0	1	0	0	0	1	0	1

Matriz de adyacencia

Esta matriz está conformada de m filas por m columnas, donde m es el número de vértices del grafo, esto es, es una matriz cuadrada. Si `matriz[i, j] = 1` es por que el vértice `i` está conectado con el vértice `j`, y es igual a cero en caso contrario. Usamos el vector de vértices implementado para la matriz de incidencia y lo utilizamos tanto como vector fila como vector columna.

Vértice	1	2	3	4	5	6
1	0	1	1	1	0	1
2	1	0	0	0	1	1
3	1	0	0	0	1	0
4	1	0	0	0	0	1
5	0	1	1	0	0	0
6	1	1	0	1	0	0

Lista de incidencia

Creamos un vector para almacenar las aristas del grafo, donde cada arista es una pareja (ordenada si el grafo es dirigido) (v_i, v_j), $0 \leq i, j < n$ (n total de vértices), $v_i, v_j \in V$.

(1, 2)	(1, 6)	(1, 4)	(1, 3)	(2, 5)	(2, 6)	(5,3)	(6, 4)

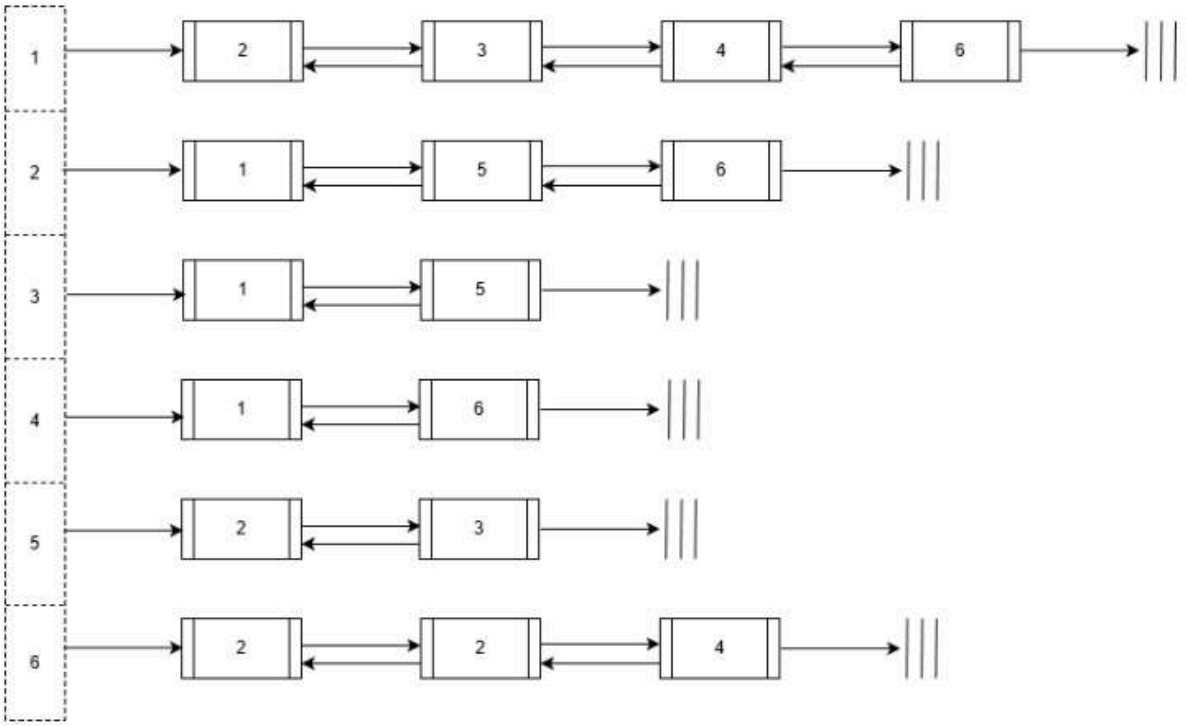
Lista de adyacencia

Creamos un vector (o lista) para almacenar los vértices y de cada uno de éstos se desprende una lista con los vértices adyacentes a éstos.

1	2	3	4	5	6
2	1	1	1	2	1
3	5	5	6	3	2
4	6	0	0	0	4
6	0	0	0	0	0

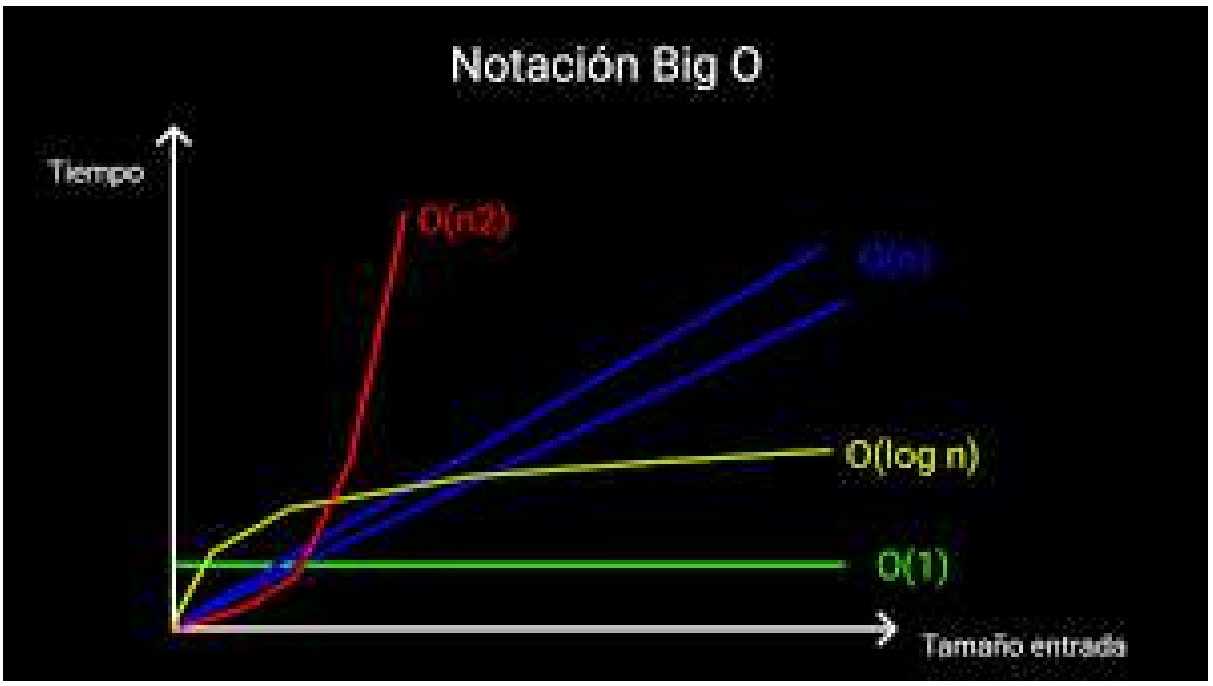
Esta representación puede verse como un vector que contiene cada vértice (nodo) del grafo, y de cada posición de éste, se desprende otro vector con los vértices adyacentes al vértice correspondiente. Esta representación puede llevarse a cabo con una matriz de n columnas, donde n es el número de nodos, sin embargo, es necesario rellenar con ceros los lugares vacíos donde hay adyacencia con un nodo cualquiera.

La representación con arreglos (vectores) y listas ligadas se puede ver como en la siguiente figura (aquí se usa una LDL, pero se puede usar cualquier tipo de lista ligada):



Ejemplo 6.4

Operaciones sobre grafos. Realizar las principales operaciones sobre grafos dirigidos y no dirigidos



CUARTA PARTE. ANÁLISIS DE ALGORITMOS

Capítulo 12. Algoritmia elemental. Notación asintótica

Problemas y ejemplares

Por ejemplo, si tenemos dos enteros positivos como 56 y 84, y queremos multiplicarlos, podemos usar alguno de los métodos o algoritmos disponibles para hacerlo; sin embargo, el método usado debería funcionar para cualquier par de enteros positivos y no solo para los casos en particular. Decimos entonces que multiplicar enteros positivos (sin importar qué método se use) es el **problema**, mientras que `(56, 84)` es un **ejemplar** de este problema[58]. Otro ejemplar para este problema es `(632, 1584)`, sin embargo `(-3, 7.65)`, no lo es, ya que `-3` no es un entero positivo y `7.65` no es un número entero; es claro que `(-3, 7.65)` es un ejemplar de otro problema, correspondiente a una multiplicación más general.

Un algoritmo debe funcionar correctamente con todos los ejemplares del problema y que corresponden al ámbito de éste. De forma similar a las demostraciones en matemáticas, donde basta con encontrar un contraejemplo para invalidar un teorema, determinar si un algoritmo es incorrecto consiste en encontrar si para un ejemplar, el algoritmo no es capaz de entregar una respuesta correcta. Por tanto, basta con un resultado incorrecto que arroje el algoritmo para descartar éste.

Cuando se especifica un problema, es muy importante indicar el **dominio de definición**, esto es, el conjunto de casos que deben considerarse. Por ejemplo, el algoritmo que multiplica dos números enteros positivos no tiene porqué funcionar para números negativos y fraccionarios, ya que éstos no se encuentran en el dominio de definición del problema.

Las máquinas de cálculo tienen un límite que afectan los resultados de las operaciones. Por ejemplo, puede suceder que no haya espacio disponible para realizar el cálculo o que el resultado supere la capacidad definida para el tipo de dato dado. Los algoritmos que se desarrollarán serán correctos en abstracto, sin preocuparnos por estas limitantes, pero al realizar laboratorios con los lenguajes Java y Python debemos tener los cuidados pertinentes según las reglas que éstos imponen, incluso habría que tener otros aspectos en cuenta, como el sistema operativo, el hardware disponible, etc., pero solo nos concentraremos por ahora en el tamaño de los tipos de datos y la memoria disponible.

Operaciones elementales

Son instrucciones simples que incluyen declaraciones, asignaciones, lecturas, impresiones, operaciones aritméticas y lógicas, así como comparaciones, entre otras. El número de operaciones elementales que se requieren para obtener la solución de un caso ayuda a determinar la complejidad del algoritmo.

[58] Los ejemplares también pueden ser llamados **instancias** o **casos** del problema.

Es de anotar que en ocasiones estas “operaciones elementales” no son totalmente sencillas, por ejemplo la suma y la multiplicación se hacen lentas a medida que aumenta el tamaño de los ejemplares (operandos). Otro problema a considerar en la práctica, es que al realizar operaciones aritméticas con operandos muy grandes, se excederá la capacidad de la máquina para el resultado, por lo cual estas operaciones pueden resultar ser “no elementales” y requerir un tratamiento especial para resolver el caso.

Es claro que las operaciones elementales incluirán a los ejemplares del problema.

Eficiencia de los algoritmos

Para un mismo problema, es posible que existan varias soluciones a éste, es decir, se dispone de varios algoritmos. Surge la pregunta *¿cuál es el mejor*? Al desarrollar programas, muchas veces se escoge la solución más fácil de programar, pero ¿realmente es la mejor?

El enfoque **empírico** (o **a posteriori**) para seleccionar un algoritmo consiste en programar las técnicas e ir probándolas en distintos casos con ayuda de un computador. El enfoque **teórico** (o **a priori** *-independiente de la experiencia-*) consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos *como función del tamaño de los casos considerados*. Los recursos más interesantes son el tiempo de computación y el espacio de almacenamiento, siendo el primero el más relevante en general. Por tanto, cuando hablemos de la **eficiencia** de un algoritmo, nos estaremos refiriendo a qué tan rápido se ejecuta. En ocasiones se tendrá en cuenta almacenamiento y otros recursos como el procesador.

El *tamaño* de un ejemplar se corresponde formalmente con el número de *bits* que se necesitan para representar el ejemplar en un computador.

Evaluación de algoritmos

Es el análisis que consiste en medir la eficiencia de los algoritmos en cuanto a consumo de memoria y tiempo de ejecución.

Anteriormente era crucial medir el consumo de memoria debido a las restricciones que existían en el hardware. Por ejemplo, el computador con microprocesador 8086 de finales de la década de 1970 tenía capacidad para acceder a una memoria RAM de `1MB = 1048576B`. Un vector de enteros de dimensión `10000` para el tipo de dato entero típico de un tamaño de `2B`, ocupa en memoria `20000B = 0.0191MB`. Esto equivalía al 1.91% de la memoria para esta máquina cuyos recursos eran limitados, un consumo relativamente alto para un caso de un vector pequeño, y eso sin considerar las demás variables que pudiese tener el programa, junto con los demás programas que se estuvieran ejecutando en la máquina, razones que hacían que tener presente el diseño de los programas fuera un aspecto muy importante.

Actualmente y gracias a los avances significativos del hardware, esto ya es no considerado un problema en la mayoría de los casos, por lo que ha pasado a un segundo plano, y ahora la preocupación del análisis de algoritmos está enfocada en mejorar los tiempos de ejecución, en particular cuando se trata de entradas grandes.

Los sistemas operativos y los lenguajes suministran funciones para medir los tiempos de ejecución de los programas, pero esto en ocasiones no nos sirve de mucho, ya que cuando ejecutamos el mismo algoritmos en distintas máquinas, en realidad estamos midiendo otros factores como el lenguaje de programación, el sistema operativo y el hardware entre otros aspectos.

Este tipo de evaluación del algoritmo es la que llamamos **a posteriori** o **empírica**. Pero nos interesa poder analizar la eficiencia de un algoritmo independiente de estos aspectos, esto es, **a priori**, para lo cual requerimos de herramientas matemáticas que permitan argumentar cuál algoritmo es mejor que otro para un problema dado.

Medir la cantidad de espacio que utiliza un algoritmo

Es posible medir la cantidad de espacio que utiliza un algoritmo en función del tamaño de los ejemplares. La unidad natural en informática utilizada para calcular el espacio utilizado, independiente de la máquina, es el **bit**, por lo que es fácil establecer cuánta memoria requiere un algoritmo para solucionar un caso.

En la actualidad el consumo de espacio no es un problema que preocupe en el análisis de algoritmos, tal y como sucedía hace varios años donde era preciso tener cuidado con el abuso del uso de variables y estructuras de datos debido a que los recursos de máquina eran muy limitados.

Ejemplo 1.1

Determinar el espacio ocupado por el siguiente programa y encontrar el número de operaciones elementales.

Programa Java:

```
public int sumNaturals(int n)
{
    int i, s;
    i = 1;
    s = 0;
    while (i <= n) {
        s += i;
        i++;
    }
    return s;
}
```

El programa cuenta con tres variables de tipo entero (int), el cual ocupa en Java y los lenguajes comunes `2B = 16b`. Por tanto el programa ocupa en memoria al menos `3 * 2B = 6B = 48b`, lo que realmente es un consumo de espacio muy bajo.

El programa cuenta con diez operaciones elementales, entre las que se cuentan declaraciones, asignaciones y comparaciones.

Medir la cantidad de tiempo que utiliza un algoritmo

Medir el tiempo que utiliza un algoritmo no tiene opciones tan obvias como en el caso de medir el espacio que ocupa éste. Pensar en una unidad de tiempo específica, como el segundo por ejemplo, no proporciona necesariamente una medida confiable, porque dependería del hardware donde se ejecute el programa, o el lenguaje de programación u otros factores, en otras palabras, no contamos con un computador estándar que permita establecer estas mediciones usando las unidades convencionales.

Una respuesta a este problema viene dado en el **principio de invariancia** que se enuncia a continuación, junto con otros conceptos que también se describen más adelante.

Principio de invariancia

Este principio afirma que dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en de más de alguna constante multiplicativa, esto es, si dos implementaciones requieren $t_1(n)$ y $t_2(n)$ segundos, respectivamente, para un resolver un caso de tamaño n, entonces existen dos constantes positivas, conocidas como **constantes ocultas**, tales que $t_1(n) \leq at_2(n)$, $t_2(n) \leq bt_1(n)$, siempre que n sea suficientemente grande. Es decir, el tiempo de ejecución de una implementación está acotado por el tiempo de ejecución de otra implementación multiplicado por una constante positiva. Así por ejemplo, si una de las constantes fuese 5, entonces sabríamos que una implementación en una máquina (o lenguaje de programación) tardaría un segundo, mientras que en otra máquina tardaría a lo sumo 5 segundos más.

Este principio no es demostrable, es un hecho confirmado por la observación y se cumple sea cual sea el computador (siempre y cuando sea de diseño convencional -clásico-), el lenguaje de programación y el compilador utilizado, o el sistema operativo donde se implemente la solución, entre otros aspectos. De esta forma, un cambio de lenguaje o de máquina puede significar aumentar la velocidad de ejecución de un algoritmo en el doble, 10 veces, 20 veces o más veces, siempre en un factor constante.

Contador de Frecuencias (`CF`)

Es una expresión algebraica que indica el número de veces que se ejecutan las instrucciones de un algoritmo (programa). Para determinar el `CF` es importante identificar las operaciones elementales. Es una herramienta muy útil que permite determinar el tiempo de ejecución de un algoritmo en “el orden de” en las notaciones asintóticas.

Ejemplo 1.2
Establecer el contador de frecuencias de cada uno de los siguientes algoritmos.

Al lado de cada operación elemental (instrucción) indicaremos el número de veces que se ejecuta ésta; al final sumamos estas cantidades para obtener el CF. La sentencia de inicio puede omitirse en el conteo, a no ser que sea una función, la cual generalmente involucra parámetros.

Pseudo programa Java a)

```
public static void main(String args[])
{
    int s, a, b; _____________________________ 1
    a = 10;      _____________________________ 1
    b = 15;      _____________________________ 1
    s = (a + b) / 2; __________________________ 1
    print(s);        __________________________ 1
}   ___________________________________________ 1

                                               _____
Contador de frecuencias CF =                     6
```

Pseudo programa Java b)

```
public static void main(String args[])
{
    int i, s, n;_____________________________ 1
    i = 1;    _____________________________ 1
    s = 0;    _____________________________ 1
    read(n);  _____________________________ 1
    while (i <= n) { ________________________ n + 1
        s += i;      ________________________ n
        i++;         ________________________ n
    }   ____________________________________ n
    print(s); ______________________________ 1
}   ___________________________________________ 1

                                              ________
Contador de frecuencias CF =                   4n + 7
```

Pseudo programa Java c)

```
public static void main(String args[])
{
    int i, m, n;_____________________________ 1
    i = 1;    _____________________________ 1
    read(m);  _____________________________ 1
    while (i <= m) { ________________________ m + 1
        print(i * i); __________________________ m
        print(1 / i); __________________________ m
        i++;         ___________________________ m
```

```
    }   _____________________________ m
    i = 1;    _____________________________ 1
    read(n);  _____________________________ 1
    while (i <= n) { _____________________ n + 1
        print(2 * i); _____________________ n
        i++;          _____________________ n
    }   _____________________________ n
}   _____________________________ 1

                                  ____________
Contador de frecuencias CF =       5m + 4n + 8
```

Pseudo programa Java d)

```
public static void main(String args[])
{
    int i, j, n, s; _____________________ 1
    i = 1;    _____________________________ 1
    read(n);  _____________________________ 1
    while (i <= n) {  ____________________ n + 1
        s = 0;        ____________________ n
        j = 1;        ____________________ n
        while (j <= n) { ________________ n * n + n
            s = s + i + j; _____________ n * n
            j++;      ____________________ n * n
        }   _____________________________ n * n
        i++;          ____________________ n
        print(s);     ____________________ n
    }   _____________________________ n
    print(s);         ____________________ 1
}   _____________________________ 1

                                  ____________
Contador de frecuencias CF =       4n² + 7n + 6
```

Pseudo programa Java e)

```
public static void main(String args[])
{
    int i, j, n, s; _____________________ 1
    i = 1;    _____________________________ 1
    read(m);  _____________________________ 1
    while (i <= m) {  ____________________ m + 1
        read(n);      ____________________ m
        s = 0;        ____________________ m
        j = 1;        ____________________ m
        while (j <= n) { ________________ n * m + m
            s = s + i + j; _____________ n * m
            j++;      ____________________ n * m
        }   _____________________________ n * m
```

```
        i++;          _____________________ m
        print(s);     _____________________ m
    }   ___________________________________ m
    print(s);         _____________________ 1
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4mn+ 8m + 6
```

Pseudo programa Java f)

```
public static void main(String args[])
{
    int i, n;   _____________________________ 1
    read(n);  _____________________________ 1
    i = n;    _____________________________ 1
    while (i > 1) {  ____________________ lg(n) + 1
        print(i);    ____________________ lg(n)
        i /= 2;      ____________________ lg(n)
    }   ___________________________________ lg(n)
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4lg(n) + 5
```

Pseudo programa Java g)

```
public static void main(String args[])
{
    int i, n;   _____________________________ 1
    read(n);  _____________________________ 1
    i = 1;    _____________________________ 1
    while (i <= n) { ____________________ log5(n) + 1
        print(i);    ____________________ log5(n)
        i *= 5;      ____________________ log5(n)
    }   ___________________________________ log5(n)
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4log5(n) + 5
```

Pseudo programa Java h)

```
public static void main(String args[])
{
    int i, j, n; _____________________________ 1
    read(n);  _____________________________ 1
    i = 1;    _____________________________ 1
    while (i < = n) { ___________________ n + 1
        j = n;       ____________________ n
        while (j > 1) {  _______________ nlg(n) + n
            print(i, j); _______________ nlg(n)
            j /= 2;      _______________ nlg(n)
```

```
        }   _____________________________ nlg(n)
        i++;  _____________________________ n
    } _____________________________________ n
}   _____________________________________ 1

                                          _________________
Contador de frecuencias CF =               5n + 4nlg(n) + 5
```

Notación Asintótica

Es una forma de representar matemáticamente el comportamiento de un algoritmo y analizar su eficiencia a medida que aumenta el tamaño de la entrada.

Aplicaciones de la notación asintótica

En computación científica tenemos entre otras aplicaciones, las siguientes:

1. Analizar el tiempo de ejecución de un algoritmo
2. Entender cómo crecen los tiempos de ejecución y el uso de memoria
3. Comprender la eficiencia de un algoritmo cuando la entrada es muy grande
4. Aproximar la complejidad temporal o espacial de un algoritmo

Funcionamiento básico

La notación asintótica es relativamente fácil de aplicar y realiza lo siguiente con el CF y los recursos a analizar en detalle:

- Se descartan los coeficientes constantes y los términos menos significativos
- Se enfoca en la parte importante del tiempo de ejecución de un algoritmo y su tasa de crecimiento

Notaciones asintóticas comunes

En el análisis de algoritmos se cuenta con las siguientes notaciones asintóticas, de las cuales revisaremos las tres primeras de la lista que son las más implementadas:

- Big O (O Grande)
- Big Omega (Ω Grande)
- Big Theta (Θ Grande)
- Condicional
- Little o (o pequeña)
- Little omega (ω pequeña)
- Landau Notation (notación Landau)

Notación Asintótica O Grande (*Big O*)

Representada con la letra **O** mayúscula, define la eficiencia del algoritmo a partir del contador de frecuencias. Para esto, se toma la expresión dada por el `CF` y se eliminan los coeficientes, las constantes y los términos negativos; de los términos resultantes, si son dependientes entre sí, se toma el mayor de ellos y este será el orden de magnitud del algoritmo; en otro caso, el orden de magnitud estará determinado por la suma de los términos restantes.

De manera más formal, la notación **O Grande** proporciona una **cota superior** e indica que si el tiempo de ejecución de un algoritmo es $O(f(n))$, entonces para n suficientemente grande, el tiempo de ejecución es a lo sumo $cf(n)$: $t \leq cf(n)$ para alguna constante $c \in \mathbf{R}^{\geq 0}$. La condición de un n grande no es realmente necesaria, como se verá en la "regla del umbral", pero permite hallar constantes más pequeñas de las que se encontrarían en caso contrario, lo cual es útil cuando se buscan buenas cotas sobre el tiempo de ejecución de una implementación cuando se conoce el tiempo de otra.

Gráficamente se muestra como la notación O Grande proporciona una cota superior, esto es, un tiempo máximo de ejecución del algoritmo:

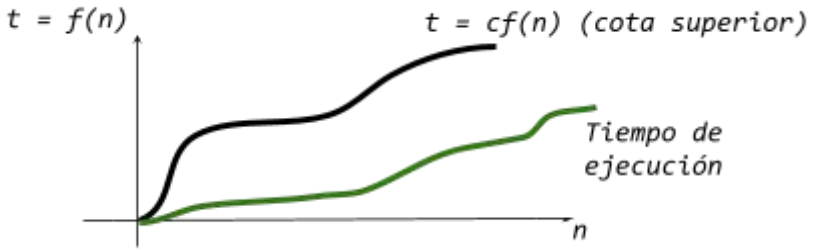


Figura 1.1. Comparación del tiempo de ejecución "normal" con el máximo posible determinado por la cota superior

Órdenes de magnitud en notación asintótica O Grande

Los órdenes de magnitud más frecuentes en informática, expresados en notación asintótica O Grande, se ilustran a continuación.

Orden de magnitud constante `O(1)`

Si el `CF` de un algoritmo es una constante, entonces su orden de magnitud es constante y lo denotamos como `O(1)`. Este tipo de algoritmos carece de ciclos o procesos repetitivos. Los algoritmos de magnitud constante son los ideales, ya que su eficiencia es la mejor, pero no siempre es fácil o posible obtener algoritmos con un `CF` constante.

Orden de magnitud lineal O(n)

El CF se obtiene en términos del valor de un n dado como entrada del algoritmo; generalmente interesan casos de entradas grandes, pero en casos de un n pequeño o "normal" también se cumple la definición (este valor de n puede representar el número de elementos de un arreglo, los registros de un archivo, datos arbitrarios ingresados por el usuario, etc.). Denotamos este orden de magnitud como O(n). Este tipo de algoritmos posee uno o más ciclos independientes.

Orden de magnitud lineal O(m + n)

El CF se obtiene en términos de los valores de m, n, dados como entradas del algoritmo, siendo independientes entre sí. Denotamos este orden de magnitud como O(m + n). Este tipo de algoritmos posee uno o más ciclos independientes y se reduce al caso anterior cuando m = n, por lo que O(m + n) = O(n + n) = O(2n) = O(n). En realidad, este es un caso derivado del orden de magnitud lineal O(n).

Orden de magnitud cuadrático $O(n^2)$

Este tipo de algoritmos posee un ciclo que se ejecuta n veces, y en su interior se encuentra un ciclo (anidado) que también se ejecuta n veces; dichos ciclos son dependientes y como veremos, los términos cuadrático y lineal del CF también lo serán. Denotamos este orden de magnitud como $O(n^2)$.

Orden de magnitud cuadrático O(m * n)

El CF se obtiene en términos de los valores de m, n, dados como entradas del algoritmo, siendo dependientes entre sí. Denotamos este orden de magnitud como O(m * n). Este tipo de algoritmos posee uno o más ciclos dependientes y se reduce al caso anterior cuando m = n, por lo que O(m 2 n) = O(n * n) = $O(n^2)$. En realidad, este es un caso derivado del orden de magnitud cuadrático $O(n^2)$.

Orden de magnitud cúbico $O(n^3)$

Similar al orden de magnitud cuadrático, pero en su interior se encuentran dos ciclos, uno anidado dentro de otro, y el tercero anidado en el segundo, todos ejecutándose n veces; dichos ciclos son dependientes entre sí, y como veremos, el término cúbico del CF predominará al medir la eficiencia del algoritmo. Denotamos este orden de magnitud como $O(n^3)$.

Orden de magnitud logarítmico O(log(n))

Este orden de magnitud se obtiene cuando en un ciclo cuya entrada final es n, se avanza en cada iteración en función de un cociente que afecta la variable controladora del ciclo. Denotamos este orden de magnitud como O(log(n)).

Orden de magnitud semilogarítmico O(nlog(n))

Se obtiene este orden de magnitud cuando en un ciclo de magnitud lineal O(n) se encuentra un ciclo de magnitud O(log(n)).

Orden de magnitud exponencial $O(x^n)$

Aunque no es tan recurrente, hay casos de algoritmos con una eficiencia muy deficiente, tal y como se presenta en los programas con orden de magnitud exponencial y que se denota por $O(x^n)$, $x \in \mathbf{Z}$, $x > 1$.

Clasificación de los algoritmos según su eficiencia

En la siguiente tabla se listan los principales órdenes de magnitud que se encuentran en informática según su eficiencia en orden ascendente, siendo 1 el más eficiente y 7 el más ineficiente.

Eficiencia	Orden de magnitud	Representación	Ejemplo
1	Constante	O(1)	Ver ejemplo 1.2 a)
2	Logarítmico	O(log(n))	Ver ejemplo 1.2 f) y g)
3	Raíz m-ésima	$O(\sqrt[m]{n})$	Ver ejemplo números primos mejorado
4	Lineal	O(n)	Ver ejemplo 1.2 b)
5	Semilogarítmico	O(nlog(n))	Ver ejemplo 1.2 h)
6	Cuadrático	$O(n^2)$	Ver ejemplo 1.2 d)
7	Cúbico	$O(n^3)$	Multiplicación de matrices. Simulación de un reloj (hh:mm:ss)
8	Polinomial	$O(n^k)$, $k \in \mathbf{Z}$, $k > 3$	
9	Exponencial	$O(x^n)$ $x \in \mathbf{Z}$, $x > 1$ ó más común $O(2^n)$	
10	Factorial	O(n!)	

Tabla 1.1. Clasificación de los algoritmos según su eficiencia

Regla del umbral

Sean $f, t: \mathbf{N} \rightarrow \mathbf{R}^+$ dos funciones arbitrarias de los naturales en los reales estrictamente positivos, entonces $t(n) \in O(f(n))$ si y sólo si existe una constante real positiva tal que $t(n) \in O(f(n))$ para cada número natural n.

Regla del máximo

Esta regla permite demostrar que una función es del orden de otra. Sean $f, g: \mathbf{N} \rightarrow \mathbf{R}^+$ dos funciones arbitrarias de los naturales en los reales no negativos. La regla del máximo dice que $O(f(n) + g(n)) = \max(f(n), g(n))$, o si se quiere igual a $\max(O(f(n)), O(g(n))$.

Operaciones sobre notación asintótica

Para simplificar cálculos, es posible manipular la notación asintótica mediante el uso de operadores aritméticos, así por ejemplo $O(f(n)) + O(g(n))$ representa el conjunto de operaciones obtenidas sumando punto a punto en las funciones $O(f(n))$ y $O(g(n))$ para cada $n \in N$. Intuitivamente se puede observar que $f(n)$ es el tiempo que requiere una primera fase del algoritmo para ejecutarse y $g(n)$ el tiempo que requiere una segunda parte para ejecutarse después de haber ejecutado la primera.

Proposición

El tiempo de ejecución de un algoritmo está dado por $f(n) + g(n)$, es decir : $t(n) = O(f(n) + g(n))$. Es válida la expresión $O(f(n) + g(n)) = O(f(n)) + O(g(n))$.

Demostración

Sea $f(n) = m$, $g(n) = n$ los tiempos de ejecución que toman dos partes de un algoritmo, entonces:

$O(f(n) + g(n)) = O(m + n)$ (1).

Por la regla del máximo esto es idéntico a:

$O(max(f(n), g(n))) = O(max(m, n)) = O(m + n)$ (2).

Ahora $O(f(n)) + O(g(n)) = O(m) + O(n)$ (3). Calculando el máximo de esta última igualdad: $max(O(m), O(n)) = O(max(m, n))$.

Pero por la ecuación (2): $O(max(m, n)) = O(m + n)$ (4). Por tanto tenemos por transitividad que $O(f(n) + g(n)) = O(f(n)) + O(g(n))$.

Esta demostración justifica porqué $O(m + n)$ es lineal usando la regla del máximo que dice que esta expresión es equivalente a $O(máx(m, n))$.

Ejemplo 1.3

Supongamos que un algoritmo tiene dos partes de las cuales a la primera le toma un tiempo de $O(n)$ para ejecutarse y a la segunda le toma un tiempo de $O(m)$, por tanto el algoritmo requiere un tiempo de $O(O(n) + O(m)) = max(O(n), O(m))$.

Ejemplo 1.4

Supongamos que un algoritmo ejecuta tres partes, con tiempos de $O(n^2)$, $O(n^3)$ y $O(nlog(n))$, respectivamente. Es claro que el algoritmo completo requiere un tiempo de $O(n^2 + n^3 + nlog(n)) = max(O(n^2), O(n^3), O(nlog(n))) = O(n^3)$.

Aun cuando el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeridos por cada parte separada, dicho tiempo es del orden del tiempo requerido por la parte que más consuma tiempo, siempre y cuando el número de partes sea una constante, independientemente del tamaño de la entrada.

La regla del máximo nos dice que si $t(n)$ es una función complicada tal como $t(n) = 15n^3log(n) - 5n^2 + log^2(n) + 27$ y si $f(n)$ es el término más significativo de $t(n)$, entonces $O(t(n)) = O(f(n))$, lo cual permite una simplificación significativa y casi que inmediata en la notación asintótica. Para el caso del ejemplo, es fácil ver que $f(n) = 15n^3log(n)$ es el término más significativo, y descartando el coeficiente, se tiene que $O(t(n)) = O(f(n)) = O(n^3log(n))$.

En otras palabras, se pueden eliminar los términos de orden inferior porque son despreciables en comparación con el término más significativo para entradas de n suficientemente grandes.

Regla del límite

Es considerada la herramienta más potente y versátil para demostrar que algunas funciones están en el orden de otras.

Sean $f, g: \mathbf{N} \rightarrow \mathbf{R}^{\geq 0}$ dos funciones arbitrarias de los naturales en los reales no negativos, se cumple lo siguiente:

1. Si $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in R^+$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$.
2. Si $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$.
3. Si $\lim_{n \to \infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ pero $g(n) \in O(f(n))$.

Ejemplo 1.5

Sea $f(n) = log(n)$, $g(n) = n^{1/2}$. Queremos determinar el orden relativo de estas funciones. Ambas funciones tienden al infinito cuando n tiende al infinito, por tanto utilizaremos la regla de L'Hopital para hallar el límite:

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(n)}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$ Esto significa que `f(n)` $\in$ `O(g(n))` pero `g(n)` $\notin$ `O(f(n))`, entras palabras, `g(n)` = $n^{1/2}$ crece asintóticamente más rápido que `f(n) = log(n)`.

Ejemplo 1.6

Tomemos la función arbitraria cuadrática `f(n)` = $2n^2$ + `10n + 50`. Veamos cómo un simple análisis de los términos de esta función, nos permite ilustrar cómo el término cuadrático predomina en los resultados de la salida final para entradas de `n` grandes.

Si tomamos `g(n)` = $2n^2$, `h(n) = 10n + 50`, también es fácil ver que el término más significativo de `f(n)` es $2n^2$, y si aplicamos la regla del límite, encontramos igual resultado:

$\lim_{n\to\infty} \frac{h(n)}{g(n)} = \lim_{n\to\infty} \frac{10n+50}{2n^2} = 0$ Por tanto `O(h(n))` $\in$ `O(g(n))`.

La siguiente tabla ilustra algunas entradas para `n`; la gráfica muestra el comportamiento de las funciones a medida que la entrada se hace más grande.

f(n) = 2 * n ** 2 + 10 * n + 50 f(n) = f1(n) + f2(n)		
n	f1(n) = 2 * n ** 2	f2(n) = 10 * n + 50
0	0	50
20	800	250
40	3200	450
60	7200	650
80	12800	850
100	20000	1050
120	28800	1250
140	39200	1450
160	51200	1650
180	64800	1850
200	80000	2050

Tabla 1.2. Tabla de valores para n, $f_1(n)$ y $f_2(n)$

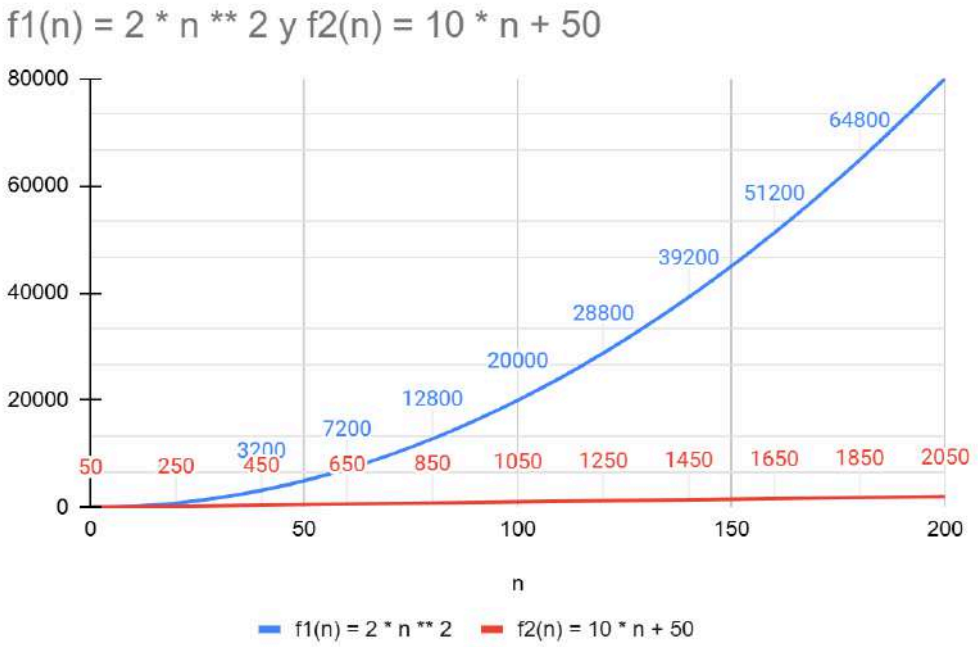


Figura 1.2. Comparación del comportamiento algorítmico entre el orden de magnitud lineal y cuadrático

Notación Asintótica Omega (Ω) Grande (*Big Omega*)

La notación O Grande solo proporciona **cotas superiores** sobre la cantidad de recursos requeridos; mediante la notación asintótica **Omega Grande** (o simplemente **Omega**) es posible obtener una notación que proporciona **cotas inferiores**. La notación Omega permite establecer la cantidad mínima de tiempo que puede tomar un algoritmo en ejecutarse.

De manera más formal, la notación **Ω** Grande proporciona una **cota inferior** e indica que si el tiempo de ejecución de un algoritmo es $\Omega(f(n))$, entonces para n suficientemente, el tiempo de ejecución es a lo menos $cf(n)$: $t \geq cf(n)$ para alguna constante $c \in \mathbf{R}^{\geq 0}$.

Gráficamente se muestra como la notación **Ω** Grande proporciona una cota inferior, esto es, un tiempo mínimo de ejecución del algoritmo:

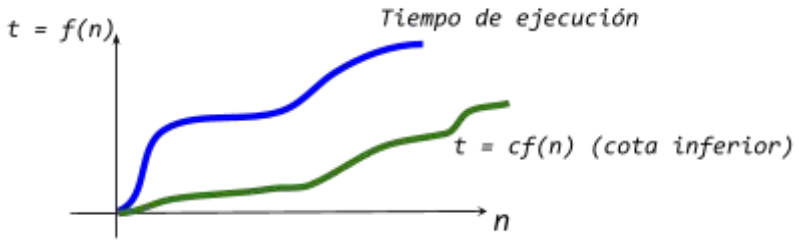


Figura 1.3. Comparación del tiempo de ejecución “normal” con el mínimo posible determinado por la cota inferior

Notación Asintótica Theta (Θ) Grande (*Big Theta*)

La notación **Theta Grande** proporciona una **cota inferior** y otra **cota superior** como límites del tiempo de ejecución de un algoritmo; esto significa que la ejecución del algoritmo no tardará menos de cierto tiempo ni más que otro cierto tiempo. Para ser más precisos, para dos constantes positivas, se tiene que $c_1 f(n) \leq t \leq c_2 f(n)$ para constantes arbitrarias $c_1, c_2 \in \mathbf{R}^{\geq 0}$. Esto es, el tiempo de ejecución está acotado por arriba y por abajo.

La notación Θ Grande indica que se cuenta con una **cota asintóticamente ajustada** sobre el tiempo de ejecución. "Asintóticamente" porque importa en general para valores grandes de *n*. "Cota ajustada" porque se ajusta el tiempo de ejecución dentro de un rango determinado por un par de constantes positivas hacia arriba y hacia abajo.

Gráficamente se muestra como la notación Θ Grande proporciona una cota inferior y otra superior, esto es, suministra un tiempo mínimo y un tiempo máximo de ejecución del algoritmo:

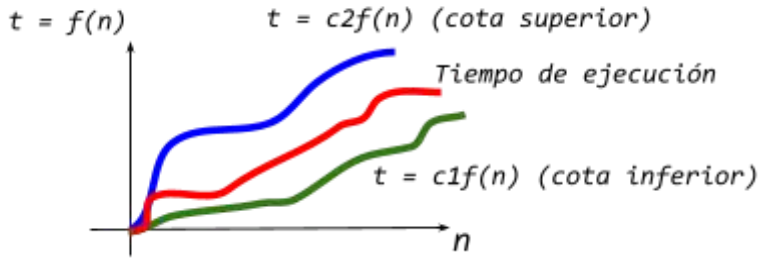


Figura 1.4. Comparación del tiempo de ejecución “normal” con el mínimo y máximo posible determinado por las cotas inferior y superior respectivamente

Preguntas

Ejercicios

1. Utilice la regla del límite para encontrar en qué orden de magnitud relativo está cada función. En donde sea necesario utilice la suma de funciones para separar éstas y hacer el análisis
   a. $f(n) = 10n + 7$
   b. $f(n) = 7n^2 + 2n + 7$
   c. $f(n) = 8n^3 + 5n + 10$
   d. $f(n) = 3n^3 + 4n^2$
   e. $f(n) = 8n^3 + 4n^2 + 5n + 1$

f. $f(n) = 8n^3 + 2nlg(n) + 3n + 6$
g. $f(n) = 4m^2 + 10mn + 20$
h. $f(n) = m^2 + 2mn + n^2$
i. $f(n) = 2^{3n} + 3n^2 + 12$
j. $f(n) = n^4 + 4n^3 + 5n + 10$
k. $f(n) = 2\log_4(n^2) + n + 3$
l. $f(n) = nlog(n) + 23n + 12$
m. $f(n) = 2^{3n} + 3n^2$; $g(n) = log(n) + 3n$
n. $f(n) = 5n^3 + n^2$; $g(n) = n^4 + 1$
o. $f(n) = lg(n^2)$; $g(n) = 2lg(n)$

2. Grafique los pares de funciones correspondientes del punto 1) a partir de una tabla de valores; compare los rendimientos de cada parte del algoritmo y verifique que coincidan con lo encontrado anteriormente
3. Encuentre el contador de frecuencias y el orden de magnitud de los siguientes algoritmos. Indique además la memoria que consume cada uno y si contienen operaciones no elementales:

Algoritmo a)

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

b. Algoritmo b)

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

Algoritmo c):

```
Inicio
Imprimir “Hoy es martes”
Escribir “Hola mundo”
Fin
```

Algoritmo d)

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

Algoritmo e)

```
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
```

Algoritmo f)

```
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
```

Algoritmo g)

```
Inicio
Carácter: nombre
Leer nombre
Si nombre = “Pedro” Entonces
    Imprimir “Reciba un cordial saludo, señor “, nombre
FinSi
Fin
```

Algoritmo h)

```
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
```

Algoritmo i)

```
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 unión libre“
    Caso “V”:
        Imprimir nombre, “ usted es viudo(a)“
    EnOtroCaso:
        Imprimir “Estado civil incorrecto“
FinCaso
Fin
```

Algoritmo j)

```
Inicio
Enteros: i, n
i = 1
Leer n
Mientras i <= n Hacer
    Imprimir i
    i = i + 1
FinMientras
Fin
```

Algoritmo k.1)

```
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
```

Algoritmo k.2)

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

¿Qué hacen este par de algoritmos? ¿Cuál es más eficiente?

Algoritmo l)

```
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
```

Algoritmo m)

```
Inicio
Enteros: 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
```

Algoritmo n)

```
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
```

Algoritmo ñ)

```
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
```

Algoritmo o)

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

Algoritmo p)

```
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
```

Algoritmo q)

```
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
```

Algoritmo r)

```
// 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
```

Algoritmo s)

```
Inicio
Enteros: c, i, n, num
Leer n
c = 0
i = 1
Mientras i <= n Hacer
    Imprimir "Ingrese un número: "
    Leer num
    Si num == 0 Entonces
        c = c + 1
    FinSi
    i = i * 2
FinMientras
```

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

¿Qué hace este algoritmo?

Algoritmo t)

```
Inicio
Enteros: c, i, n, num
Leer n
c = 0
i = n
Mientras i > 1 Hacer
    c = c + i
    i = i / 4
FinMientras
Imprimir "Valor de c: ", c
Fin
```

¿Qué hace este algoritmo?

Algoritmo u)

```
Inicio
Enteros: c, i, j, n, num
Leer n
i = 1
Mientras i <= n
    j = n
    c = 0
    Mientras j > 1 Hacer
        leer num
        c = c + num * j ^ 2
        j = j / 2
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo v)

```
Inicio
Enteros: c, i, j, n, num
Leer n
i = 1
Mientras i <= n
    j = i
    c = 0
```

```
    Mientras j > 1 Hacer
        leer num
        c = c + num * j ^ 2
        j = j / 2
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo w)

```
Inicio
Enteros: c, i, j, n, m, num
Leer n
i = 1
Mientras i <= m
    j = 1
    Leer m
    c = 1
    Mientras j <= m Hacer
        leer num
        c = c * num
        j = j + 1
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo x)

```
Inicio
Enteros: h, m, s
h = 0
Mientras h < 24
    m = 0
    Mientras m < 60 Hacer
        s = 0
        Mientras s < 60 Hacer
            Imprimir h + “:” + m + “:” + s + “\n”
            s = s + 1
        FinMientras
        m = m + 1
    FinMientras
    h = h + 1
FinMientras
```

```
Fin
```

¿Qué hace este algoritmo?

Capítulo 13. Búsqueda y ordenamiento

Búsqueda

La búsqueda es una de las operaciones más importantes y comunes en vectores, otras estructuras de datos y otros objetos computacionales, por lo que se han desarrollado distintas técnicas, unas más complejas que otras para la localización de información; los tipos de búsqueda sobre vectores y otras estructuras de datos más conocidos, son:

8. Búsqueda secuencial
9. Búsqueda binaria
10. Búsqueda por transformación de claves
11. Árboles de búsqueda

A continuación se analizan los algoritmos que permiten realizar dichas búsquedas midiendo y comparando su eficiencia, tanto interna como externa.

Búsqueda interna

Búsqueda secuencial

Es el tipo de búsqueda más sencilla y de las más utilizadas; consiste en tomar un dato y compararlo elemento por elemento del vector hasta encontrarlo o hasta terminar de recorrer el vector. El tipo de dato que devuelve el método (función) puede ser un valor *lógico* (*booleano*) para indicar que el dato está o no en el arreglo; también puede devolver un valor *entero* con la posición donde el dato se encuentra, en cuyo caso se inicializa en un valor por fuera de los posibles valores que tome el índice del vector para indicar que el dato no se encuentra.

Ejemplo 1.
Búsqueda secuencial de un dato en un vector. Este tipo de búsqueda también puede implementarse sobre una lista ligada devolviendo preferiblemente un valor booleano.

Programa Java:

```
public int secuencialSearchVector(int datum)
{
    int i, pos;
    i = 0;
    pos = -1;
    while (i < this.n && pos == -1) {
        if (this.vec[i] == datum) {
            pos = i;
        } else {
```

```
            i++;
        }
    }
    return pos;
}
```

Búsqueda binaria

Para implementar este tipo de búsqueda, el requisito es que **el vector debe estar ordenado**. El método consiste en establecer un **límite inferior** y un **límite superior**, y a partir de estos datos, se toma el elemento central calculando la posición "**promedio**"; si el dato se encuentra en dicha posición, la búsqueda finaliza, de lo contrario, se evalúa si el dato es mayor o menor que el elemento en la posición y se redefinen los límites. El proceso finaliza si el dato es encontrado o si no hay un intervalo de búsqueda. Esta búsqueda reduce significativamente el número de comparaciones, lo cual puede verse en vectores grandes.

Ejemplo 1.
Búsqueda binaria de un dato en un vector.

Programa Java:

```
public int binarySearchVector(int datum)
{
    int lowerLimit, upperLimit, pos, centralPos;
    lowerLimit = 0;
    upperLimit = this.n;
    pos = -1;
    while (lowerLimit <= upperLimit && pos == -1) {
        centralPos = (upperLimit + lowerLimit) / 2;
        if (this.vec[centralPos] == datum) {
            pos = centralPos;
        } else {
            if (this.vec[centralPos] > datum) {
                upperLimit = centralPos - 1;
            } else {
                lowerLimit = centralPos + 1;
            }
        }
    }
    return pos;
}
```

Búsqueda externa

Concepto

Ordenamiento

La ordenación de datos es una operación importante y requerida en distintas situaciones al operar con vectores y otras estructuras de datos; consiste en clasificar los elementos en un orden determinado, facilitando así las tareas de búsqueda. Al igual que con la búsqueda de datos, con la ordenación se han desarrollado distintas técnicas, unas más complejas que otras que permiten la clasificación de la información. Los tipos de ordenación más conocidos sobre vectores y otras estructuras de datos, son:

- Ordenación por intercambio directo o burbuja
- Ordenación por inserción directa o método de la baraja
- Ordenación por selección directa
- Ordenación por el método de *Shell*
- Ordenación por el método *quicksort*
- Ordenación por el método del montículo (*heapsort*)

A continuación se analizan los algoritmos que permiten realizar dichas ordenaciones midiendo y comparando su eficiencia interna y externa.

Ordenación interna

Concepto

Ordenación por intercambio directo o burbuja

Es el tipo de ordenación más sencillo de todos, de los más utilizados, pero también el más ineficiente. Este método consiste en tomar cada elemento del vector y compararlo con los siguientes $n - i - 1$ elementos; si alguno es mayor (o menor), se realiza un *intercambio directo* y se continúa comparando. Este método es lento comparado con otros, pero garantiza que deja ordenado cada elemento antes de pasar al siguiente.

Ejemplo 1.
Ordenación de un vector por el método de **intercambio directo o burbuja**.

Programa Java:

```
public void sortBubbleVector()
{
    int i, j, aux;

    for (i = 0; i < this.n - 1; i++) {
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[i] > this.vec[j]) {
                aux = this.vec[i];
                this.vec[i] = this.vec[j];
                 this.vec[j] = aux;
            }
        }
```

```
    }
}
```

Ordenación por inserción directa o método de la baraja

Este método es tomado de los juegos de cartas y cómo los jugadores ordenan éstas ubicándolas de menor a mayor de izquierda a derecha; de ahí que se conozca con el nombre de método de la baraja.

Se trata de insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada; el proceso se repite hasta el n-ésimo elemento.

Ejemplo 1.
Ordenación de un vector por el método de **inserción directa o método de la baraja**.

Supongamos que se tiene el siguiente vector:

vec

15	*67*	*8*	*16*	*44*	*27*	*12*	*35*

Se deben realizar las siguientes comparaciones:

Primera pasada
vec[1] < vec[0] → 67 < 15: no hay intercambio y va a la siguiente pasada
vec

15	*67*	*8*	*16*	*44*	*27*	*12*	*35*

Segunda pasada
vec[2] < vec[1] → 8 < 67: sí hay intercambio
vec[1] < vec[0] → 8 < 15: sí hay intercambio
vec

8	*15*	*67*	*16*	*44*	*27*	*12*	*35*

Tercera pasada
vec[3] < vec[2] → 16 < 67: sí hay intercambio
vec[2] < vec[1] → 16 < 15: no hay intercambio y va a la siguiente pasada
vec

8	*15*	*16*	*67*	*44*	*27*	*12*	*35*

Una vez se determina la posición del elemento, se terminan las comparaciones, como en el caso de la tercera pasada, pues los elementos van quedando organizados en cada pasada a la izquierda del arreglo de menor a mayor.

Se deja como ejercicio verificar las demás pasadas que realiza el algoritmo y hacer el análisis de eficiencia de éste.

Ordenación por el método de selección directa

En este método se localiza el menor elemento y se intercambia con la primera posición; luego se busca el segundo en elemento menor en los n - 1 elementos restantes y se intercambia con la segunda, y así sucesivamente con los demás elementos, es decir, se busca el menor en los primeros n elementos, luego en los n - 1, luego en los n - 2, hasta llegar al penúltimo elemento.

Ejemplo 1.
Ordenación de un vector por el método de **selección directa**.

Programa Java:

```
public void sortSelectionVector()
{
    int i, j, k, min;
    for (i = 0; i < this.n - 1; i++) {
        min = this.vec[i];
        k = i;
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[j] < min) {
                min = this.vec[j];
                k = j;
            }
        }
        this.vec[k] = this.vec[i];
        this.vec[i] = min;
    }
}
```

Ordenación por fusión (merge sort)

El enfoque para la solución de este problema pertenece a la técnica **Divide y Vencerás (VyC)**, de la cual se hablará más adelante. Para ordenar una lista dada de n números, se divide en dos listas de aproximadamente n/2 números cada una, se ordenan cada una por turno y se intercalan ambos resultados adecuadamente para obtener la versión ordenada de la lista dada (ver figura). Este enfoque se conoce como algoritmo de **ordenación por fusión** (***Merge Sort***)[59].

[59] El algoritmo de ordenación por fusión se debe al científico húngaro John von Neumann, quien lo propuso en 1945 y a quien le debemos también la llamada arquitectura “von Neumann”, base de las máquinas que utilizamos actualmente.

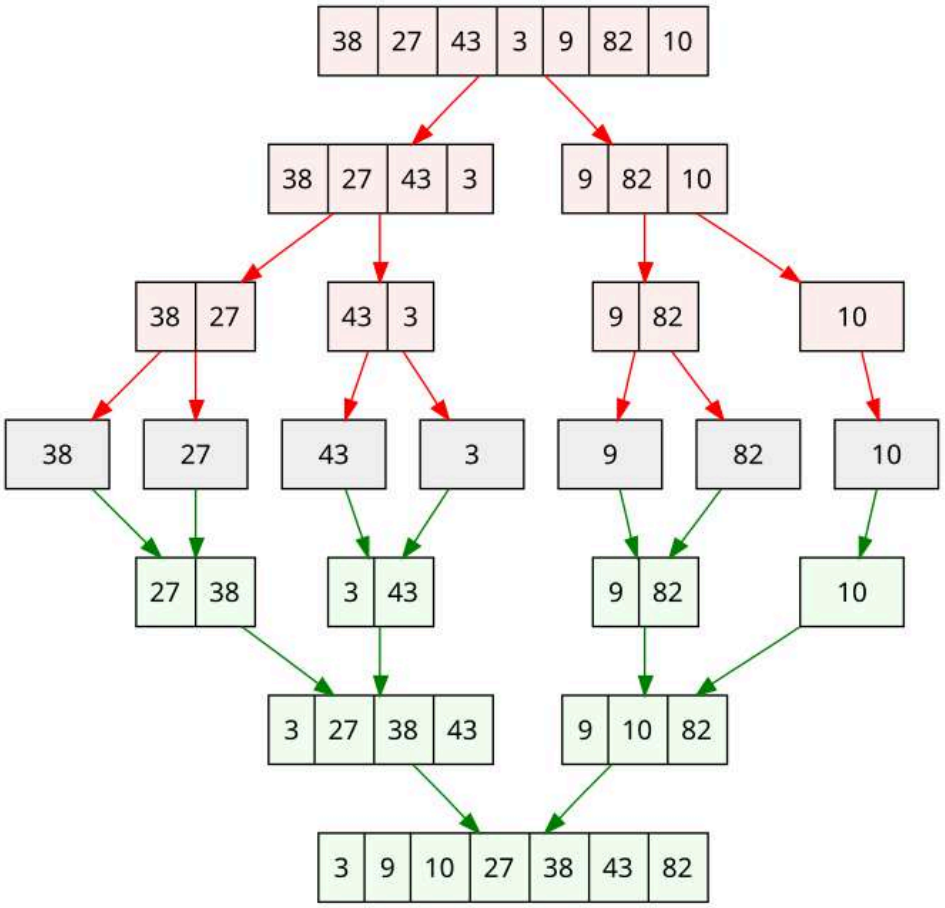


Figura. Método de divide y vencerás para ordenar la lista (38, 27, 43, 3, 9, 82, 10) en orden creciente. Mitad superior: división en sublistas; mitad media: ordenación trivial de una lista de un elemento; mitad inferior: composición de sublistas ordenadas.[60]

Merge Sort es un algoritmo de ordenación altamente eficiente y fiable, ideal para manejar grandes volúmenes de datos debido a su complejidad temporal consistente de `O(nlg(n))`. Aunque su implementación puede ser ligeramente más compleja que otros algoritmos más simples de ordenación, los beneficios de su rendimiento y fiabilidad justifican su uso en entornos que demandan alta eficiencia y precisión. Merge Sort proporciona una solución robusta y eficaz en el campo de los algoritmos de ordenación.

En esencia, Merge Sort funciona descomponiendo recursivamente un vector sin ordenar en vectores más pequeños, ordenándolos y luego fusionándolos de nuevo de forma ordenada. Este proceso se basa en el paradigma de "divide y vencerás", una potente técnica algorítmica que consta de tres pasos fundamentales:

33. Dividir: divide el vector original en dos mitades.
34. Conquistar: ordena recursivamente cada mitad.
35. Combinar: fusiona las dos mitades ordenadas para crear un vector ordenado.

[60] Imagen tomada de https://en-m-wikipedia-org.translate.goog/wiki/Divide-and-conquer_algorithm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge#:~:text=A%20divide%2Dand%2Dconquer%20algorithm,solution%20to%20the%20original%20problem.

Esta descomposición recursiva continúa hasta alcanzar el caso base: vectores con un solo elemento, que están ordenados de forma inherente. El proceso de fusión reconstruye entonces el vector ordenado paso a paso.

Ejemplo
Algoritmo “mergesort” (ordenación por fusión).

Este algoritmo se implementa sobre la clase Vector del proyecto

Programa Java

```
   public void merge(int arr[], int l, int m, int r)
    {
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int[n1];
        int R[] = new int[n2];

        for (int i = 0; i < n1; ++i) {
            L[i] = arr[l + i];
        }

        for (int j = 0; j < n2; ++j) {
            R[j] = arr[m + 1 + j];
        }

        int i = 0;
        j = 0;
        int k = l;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
```

```
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    public void sort(int arr[], int l, int r)
    {
        if (l < r) {
            int m = l + (r - l) / 2;

            sort(arr, l, m);
            sort(arr, m + 1, r);

            merge(arr, l, m, r);
        }
    }

// Para hacer uso de la ordenación por fusión:
objVec.sort(objVec.getVec(), 0 , objVec.getN() - 1);
```

Ordenación externa

Concepto

Preguntas

Ejercicios

Capítulo 14. Complejidad algorítmica en la recursión y en estructuras de datos

La complejidad de algoritmos que utilizan estructuras de datos o recursividad también es estudiada en la computación científica en donde se analiza tanto su complejidad temporal como espacial.

Complejidad algorítmica en arreglos

La complejidad de los algoritmos en arreglos, tanto en tiempo como en espacio, varía según la operación que se realice y las dimensiones del arreglo.

Arreglos unidimensionales (vectores)

Ya se vio como los vectores tienen una complejidad temporal en operaciones como la búsqueda secuencial de O(n), así como la inserción o eliminación de elementos, mientras que la búsqueda binaria tiene un orden de magnitud O(lg(n)). Agregar elementos por el final del vector tiene un tiempo de O(1) y recorrerlo de O(n). La complejidad espacial también es del mismo orden.

Arreglos bidimensionales (matrices)

Los algoritmos sobre matrices más comunes como llenarla o recorrerla, tienen en general un orden de magnitud de O(m*n) o de $O(n^2)$ en caso de matrices cuadradas. La complejidad espacial también es del mismo orden.

Arreglos multidimensionales (n > 2)

En general, para realizar operaciones sobre arreglos, se requerirá un orden de magnitud proporcional al producto de sus dimensiones, esto es, un orden de magnitud de $O(n_1*n_2*n_3*...*n_m)$ o de $O(n^m)$ (n_1, n_2, n_3, ..., n_m, m, $\in$ **N**) en caso de dimensiones del mismo orden; por ejemplo, para recorrer una estructura en forma de cubo (tridimensional), se tiene una complejidad algorítmica de $O(n^3)$. La complejidad espacial también es del mismo orden.

Complejidad algorítmica en listas ligadas

La complejidad de los algoritmos en listas ligadas[61], tanto en tiempo como en espacio, varía según la operación que se realice y es similar al tratamiento con vectores. La complejidad temporal de operaciones como la búsqueda lineal es O(n), mientras que la inserción o eliminación en la cabeza o final es O(1), sin embargo, eliminar el último también puede tener un orden de magnitud O(n),dependiendo de cómo se esté tratando la lista, si es simplemente ligada, doblemente ligada o circular. La complejidad espacial es generalmente O(n) para almacenar los nodos y sus referencias, aunque también puede ser O(1) si cada nodo se agrega uno a uno.

Complejidad Temporal

Se analizan las operaciones fundamentales sobre listas para determinar el tiempo que requiere la ejecución de cada una de estas.

Búsqueda

La búsqueda de un elemento en una lista ligada requiere, en el peor caso, recorrer todos los nodos (O(n)). Si la lista está ordenada, la búsqueda binaria (O(log n)) podría ser una alternativa en la cual pensar, pero por las características del algoritmo, se hace casi que inaplicable en este caso.

Inserción

La inserción en la cabeza o final es de complejidad O(1) porque no requiere recorrer la lista; al final es aplicable esto si se tiene un puntero que apunta al último nodo o es circular, ya que de lo contrario sería necesario ir hasta dicho nodo para hacer la inserción. Insertar en una posición cualquiera puede requerir en general tener que recorrer la lista hasta esa posición (O(n)), dado que implica realizar una búsqueda de la referencia donde se va a insertar. En realidad, la inserción requiere de un búsqueda previa, que hace que su orden de magnitud sea O(n), aunque si se analiza desde la operación como tal suponiendo que ya se realizó la búsqueda, el orden de magnitud es O(1).

Eliminación

Eliminar un elemento de la lista puede requerir recorrerla para encontrar la posición del elemento, esto es, buscarlo, por lo que requiere en general un tiempo de O(n). Esta operación es similar a la inserción y aplica la misma observación hecha allí.

Nota: Inserción/Eliminación en una lista doblemente ligada:
Las operaciones de inserción y eliminación son de complejidad O(1) si se conoce la posición o el elemento a eliminar/insertar. De igual manera, si no hay un puntero apuntando al nodo requerido, será necesario realizar la búsqueda, que requiere un tiempo O(n).

[61] Al hablar de listas ligadas, se consideran todos los tipos: simplemente ligadas (LSL), simplemente ligada circular (LSLC), doblemente ligada (LDL) y doblemente ligada circular (LDLC).

Modificación

Esta operación es más simple, ya que solo es cambiar la información del nodo (no se considera la complejidad del registro que compone el nodo como tal), por lo que el orden de magnitud es el mismo que en la búsqueda, es decir, O(n). Si se cuenta con la referencia del nodo, el algoritmo solo requiere un tiempo de O(1).

Recorrido

Recorrer la lista (visitar cada nodo) tiene una complejidad de O(n).

Complejidad Espacial

La complejidad espacial es O(n), esto es, es proporcional al número de nodos de la lista. Cada nodo requiere un espacio para almacenar la información de acuerdo a su tipo de dato y una referencia al siguiente nodo (y anterior de ser doblemente ligada). El espacio requerido por cada nodo es proporcional al tamaño de los datos almacenados en el nodo y al tamaño de la referencia/puntero; en el caso más simple, el nodo puede estar compuesto de un registro con un dato primitivo y uno o dos datos tipo puntero (referencia). El tratamiento de listas ligadas requerirá también de un puntero que apunte al primer nodo (cabeza) de la lista.

Complejidad algorítmica en pilas y colas

Las pilas y colas son estructuras de datos abstractas que no tienen implementación directa en los lenguajes de programación, pero que pueden simularse mediante vectores o listas ligadas.

Complejidad temporal

Se podría decir que las operaciones fundamentales sobre pilas y colas tendrían una complejidad algorítmica O(1), sin embargo, por el tratamiento mismo de cada estructura de datos con que se implementan las pilas y colas, es necesario considerar cada caso.

Pilas

Las pilas obedecen a la metodología conocida como **LIFO** (*Last In, First Out*), por lo que las operaciones se hacen por un extremo. Las operaciones más importantes sobre pilas son conocidas como **apilar** y **desapilar**, las cuales consisten en agregar y quitar elementos respectivamente; ambas tienen un orden de magnitud O(1) tanto si se trata con vectores o con listas ligadas. Otras operaciones, como buscar un elemento, insertar o eliminar, requieren, en el peor de los casos, mover todos los elementos (o n-1 elementos) a una

nueva pila, insertar o quitar el elemento, y luego regresar los elementos a la pila original, lo que representa 2n movimientos, obteniendo así una complejidad algorítmica de O(n).

Colas

Las colas obedecen a la metodología conocida como **FIFO** (*First In, First Out*), por lo que las operaciones se hacen por los extremos. Las operaciones más importantes sobre colas son conocidas como **encolar** y **desencolar**, las cuales consisten en agregar y quitar elementos respectivamente.

Agregar un elemento a la cola (encolar) en un vector tiene una complejidad de O(1); en una LSL o LSLC es O(1) si se tiene un puntero en el último nodo, de lo contrario, será necesario recorrer la lista hasta éste, lo que implica una complejidad de O(n). En una LDL o LDLC la complejidad es O(1).

Eliminar un elemento de la cola (desencolar) en un vector tendrá un orden de magnitud O(n), ya que implica mover los n-1 elementos que le siguen hacia la derecha; si se trata con listas ligadas, su complejidad es de O(1).

Otras operaciones, como buscar un elemento, insertar o eliminar, requieren, en el peor de los casos, mover todos los elementos (o n-1 elementos) a una nueva cola, insertar o quitar el elemento, y luego regresar los elementos a la cola original, lo que representa 2n movimientos, obteniendo así una complejidad algorítmica de O(n).

Complejidad espacial

La complejidad espacial tanto para pilas como para colas es O(n), ya sean tratadas con arreglos unidimensionales o con listas ligadas.

Complejidad algorítmica en la recursión

El análisis temporal de un algoritmo iterativo es simple con base en la operación básica de este, para los algoritmos recursivos hay una dificultad añadida, pues la función que establece su tiempo de ejecución viene dada por una ecuación en recurrencia, es decir, $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{E}(\boldsymbol{n})$, en donde en la expresión E aparece la propia función $\boldsymbol{T}$.

Ecuaciones de recurrencia

Cuando se quiere calcular la demanda de recursos de un algoritmo definido recursivamente, la función complejidad que resulta no está definida sólo en términos del tamaño del problema y algunas constantes, sino en términos de la función de complejidad misma.

Además no es una sola ecuación, dado que existen otras (al menos una) que determinan la cantidad de recursos para los casos base de los algoritmos recursivos. Dada esta situación,

para poder obtener el comportamiento del algoritmo, es necesario resolver el sistema recurrente obtenido.

Ejemplo
Calcular el factorial de un número n.

Programa Java
```
public int factorial(int n)
{
    if (n == 0) {
        return 1;
    } else {
         return n * factorial(n - 1);
    }
}
```

Se considera el producto `n * factorial(n - 1)` como operación básica y el costo del caso base como 1, esto es `T(0) = 1`; así, se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = 1 + T(n - 1).`
Donde 1 es el costo de la multiplicación y `T(n - 1)` es el costo de la llamada a `factorial(n - 1)`.
`T(0) = 1`. Costo del caso base.

Ejemplo
Encontrar el n-ésimo término de la serie de Fibonacci.

Programa Java
```
public static int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 0;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Consideremos la suma `fibonacci(n - 1) + fibonacci(n - 2)` como operación básica y el costo 1 de los dos casos base, así se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = T(n - 1) + 1 + T(n - 2).`
Donde 1 es el costo de la suma, `T(n - 1)` es el costo de la llamada a `fibonacci(n - 1)` y `T(n - 2)` es el costo de la llamada a `fibonacci(n - 2)`.

`T(0) = 1`, `T(1) = 1`. Costo del caso base para `n = 0` y `n = 1`.

Ejemplo

Encontrar el MCD de dos números naturales (este ejemplo también se ilustra más adelante en la sección de problemas NP).

Programa Python

```
def mcdRecursive(self, m, n) -> int:
    if n == 0:
        return m
    else:
        return self.mcdRecursive(n, m % n)
```

Consideremos el caso base (n = 0) con un coste de 1; `mcdRecursive(n, m % n)` como operación básica y el costo 1 de los dos casos base, así se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = T(m % n) + 1`.

Donde 1 es el costo del caso base y `T(m % n)` es el costo de la llamada a `mcdRecursive(n, m % n)`.

`T(0) = m`, costo del caso base para `n = 0`.

Complejidad algorítmica en árboles y grafos

En matemáticas y en ciencias de la computación, la teoría de grafos (también llamada teoría de las gráficas) estudia las propiedades de los **grafos** (también llamadas gráficas). Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (*edges* en inglés) que pueden ser orientados o no. Típicamente, un grafo se representa mediante una serie de puntos (los vértices) conectados por líneas (las aristas).

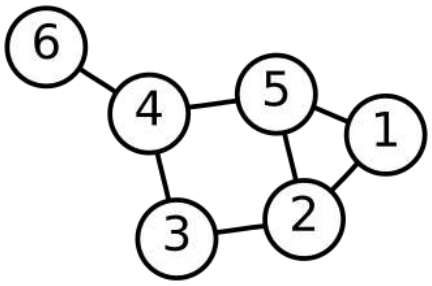


Figura. Grafo con seis vértices y siete aristas

Preguntas

-

Ejercicios

-

Capítulo 15. Técnicas de diseño de algoritmos

Problemas P, NP y NP-completos

En la teoría de la complejidad computacional, los problemas P, NP y NP-completos se refieren a la dificultad de resolver problemas computacionales. P (problemas polinomiales) son problemas deterministas que pueden resolverse en tiempo polinomial. NP (problemas no deterministas polinomiales) son problemas cuya solución se puede verificar en tiempo polinomial. Los problemas NP-completos son los más difíciles en NP, y si se encuentra un algoritmo eficiente para uno de ellos, se puede encontrar uno eficiente para todos los problemas en NP.

La relación entre las clases de complejidad NP y P la formuló el científico computacional Stephen Cook y que está aún sin resolver por la teoría de la complejidad computacional. En esencia, la pregunta "**¿es P = NP completo?**" significa: "Si es posible verificar rápidamente las soluciones de un problema de tipo NP ¿eso implica que también es posible 'obtener' las respuestas con la misma rapidez? (es decir, es un problema de tipo P)", donde "rápidamente" significa "en tiempo polinómico".

Los recursos comúnmente estudiados en complejidad computacional son:

- **Tiempo**: mediante una aproximación al número de pasos de ejecución que un algoritmo emplea para resolver el problema.

- **Espacio**: mediante una aproximación a la cantidad de memoria utilizada para resolver el problema.

Los problemas se clasifican en conjuntos o clases de complejidad (L, NL, P, P-Completo, NP, NP-Completo, NP Duro, etc.)[62].

Ejemplo
Suma de subconjuntos. Este problema consiste en hallar un subconjunto no vacío a partir de un conjunto de enteros, cuya suma de elementos sea cero.

Sea A = {−2, −3, 15, 14, 7, −10} un conjunto de números enteros. ¿Existe algún subconjunto de A cuyos elementos sumen 0?

Podemos ver que el subconjunto B = {−2, −3, 15, −10} cumple con esta condición. Sin embargo, encontrar el subconjunto tarda más que encontrar la suma de sus elementos.

La información necesaria para verificar un resultado positivo/afirmativo es llamada **certificado**. Podemos concluir entonces que dado los certificados apropiados, es posible

[62] El Clay Mathematics Institute ha ofrecido un premio de un millón de dólares estadounidenses para la persona que demuestre la solución de este problema (fuente: Wikipedia).

verificar rápidamente las respuestas afirmativas de nuestro problema (en tiempo polinomial) y es esta la razón por la que el problema se encuentra en NP.

Una respuesta a la pregunta P = NP sería determinar si en problemas del tipo SUMA-SUBCONJUNTO es tan fácil hallar la solución como verificarla. Si se encuentra que P no es igual a NP, significa que algunos problemas NP serían significativamente más difíciles de hallar su solución que verificar la misma. La respuesta es aplicable a todo este tipo de problemas y no solo al ejemplo específico citado.

La restricción a problemas de tipo SÍ/NO realmente no es importante; aún si se permiten respuestas más complicadas, el problema resultante es equivalente.

En este tipo de análisis, se requiere un modelo de computador para la que desea estudiar el requerimiento en términos de tiempo. Típicamente, dichos modelos suponen que la computadora es determinista (dado el estado actual de la máquina y las variables de entrada, existe una única acción posible que ésta puede tomar) y secuencial (realiza las acciones una después de la otra). Estas suposiciones son adecuadas para representar el comportamiento de todas las computadoras existentes, incluyendo a las máquinas con computación en paralelo.

En esta teoría, la clase P consiste de todos aquellos problemas de decisión que pueden ser resueltos en una máquina determinista secuencial en un período de tiempo polinomial en función a los datos de entrada; en la teoría de complejidad computacional, la clase P es una de las más importantes.

La clase NP consiste de todos aquellos problemas de decisión cuyas soluciones positivas/afirmativas pueden ser verificadas en tiempo polinómico a partir de ser alimentadas con la información apropiada, o en forma equivalente, cuya solución puede ser hallada en tiempo polinómico en una máquina no determinista. Por lo tanto, la principal pregunta aún sin respuesta en la teoría de la computación está referida a la relación entre estas dos clases:

¿Es **P** igual a **NP**?[63]

[63] En una encuesta realizada en el 2002 entre 100 investigadores, 61 creían que la respuesta era NO, 9 creían que la respuesta era SI, 22 no estaban seguros, y 8 creían que la pregunta podía ser independiente de los axiomas actualmente aceptados, y por lo tanto imposible de demostrar por el SI o por el NO (fuente: Wikipedia)

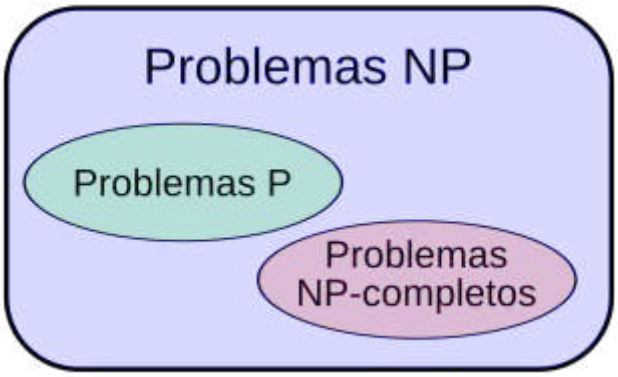


Figura. Diagrama de clases de complejidad para el caso en que P ≠ NP. La existencia de problemas fuera tanto de P como de NP-completos, fue determinada por Pichard T. Ledner.[64]

Problema de decisión

Es un problema que especifica una cadena de caracteres de datos de entrada y requiere como solución una respuesta por el SI o por el NO. Si existe un algoritmo (por ejemplo una máquina de Turing, o un programa en lenguajes Lisp o Pascal con memoria irrestricta) que es capaz de entregar la respuesta correcta para toda cadena de datos de longitud n en a lo sumo cn^k pasos, donde k y c son constantes independientes del conjunto de datos, entonces se dice que el problema puede ser resuelto en tiempo polinómico y lo clasificamos como perteneciente a la clase P. En forma intuitiva, consideramos que los problemas contenidos en P son aquellos que pueden ser resueltos en forma razonablemente rápida.

P es la clase de problemas computacionales que son “eficientemente resolubles” o “tratables”, aunque también existen problemas en P que no son tratables en términos prácticos; por ejemplo, unos requieren al menos $n^{1000000}$ operaciones.

NP es un problema de decisión que es difícil de resolver si no se posee ningún otro dato o información adicional. Sin embargo, si se recibe la información adicional llamada “certificado”, entonces el problema puede ser resuelto fácilmente. Por ejemplo, si se tiene el número 323 y se pregunta si 323 es un número factorizable, sin ningún dato o información adicional, podemos hallar la raíz cuadrada de 323 (≈17.9722), redondear al entero menor en (17) y dividir desde 17 a 2 hasta encontrar una división exacta. Sin embargo, si se da el número 17, podemos dividir 323 por 17 y rápidamente verificar que 323 es factorizable. El número 17 es llamado “un certificado”. El proceso de división para verificar que 323 es factorizable es en esencia una máquina Turing y en este caso es denominado el verificador para 323.

Técnicamente se dice que el problema es fácil si se puede resolver en tiempo polinómico y que es difícil si se resuelve en tiempo exponencial.

[64] Tomado de: Clases de complejidad P y NP - Wikipedia, la enciclopedia libre

La clase P (Problemas Polinomiales)

Son problemas que pueden ser resueltos por un algoritmo en tiempo polinomial, es decir, el tiempo que tarda en ejecutarse el algoritmo aumenta de manera polinomial con respecto al tamaño de la entrada. Por ejemplo ordenar un vector, buscar un elemento en una lista ligada, etc.

P es conocido por contener muchos problemas naturales, incluyendo las versiones de decisión de programa lineal, cálculo del máximo común divisor (MCD), y encontrar una correspondencia máxima.

Problemas notables en P

Algunos problemas naturales son completos para P, incluyendo la conectividad (o la accesibilidad) en grafos no dirigidos.

Una generalización de P es NP, que es la clase de lenguajes decidibles en tiempo polinómico sobre una máquina de Turing no determinista. De forma trivial, tenemos que P es un subconjunto de NP, aunque este hecho no está demostrado, pero la mayor parte de la comunidad científica creen que es un subconjunto estricto.

Nota
Los problemas más difíciles en **P** son los problemas **P-completos**.

Propiedades

Los algoritmos de tiempo polinómico son cerrados respecto a la composición. Intuitivamente, esto quiere decir que si uno escribe una función con un determinado tiempo polinómico y consideramos que las llamadas a esa misma función son constantes y, de tiempo polinómico, entonces el algoritmo completo es de tiempo polinómico. Esto es uno de los motivos principales por los que **P** se considera una máquina independiente; algunos rasgos de esta máquina, como el acceso aleatorio, es que puede calcular en tiempo polinómico el tiempo polinómico del algoritmo principal reduciéndolo a una máquina más básica.

Nota: pruebas existenciales de algoritmos de tiempo polinómico
Se conoce que algunos problemas son resolubles en tiempo polinómico, pero no se conoce ningún algoritmo concreto para solucionarlos. Por ejemplo, el teorema Robertson-Seymour garantiza que hay una lista finita de los menores permitidos que compone (por ejemplo) el conjunto de los grafos que pueden ser integrados sobre un toroide; además, Robertson y Seymour demostraron que hay una complejidad $O(n^3)$ en el algoritmo para determinar si un grafo tiene un grafo incluido. Esto nos da una prueba no constructiva de que hay un algoritmo de tiempo polinómico para determinar si dado un grafo puede ser integrado sobre un toroide, a pesar de no conocerse ningún algoritmo concreto para este problema.

Ejemplos

- Camino Mínimo: encontrar el camino mínimo desde un vértice origen al resto de los vértices.

- Ciclo Euleriano: Encontrar un ciclo que pase por cada arco de un grafo una única vez.

La clase NP (Problemas No Deterministas Polinomiales):

Son problemas para los cuales, dada una solución candidata, existe un algoritmo que puede verificar si esa solución es correcta en tiempo polinomial. Sin embargo, encontrar la solución en sí puede no ser posible en tiempo polinomial. Ejemplos: el problema del viajero, el problema de la mochila, etc.

La clase NP está compuesta por los problemas que tienen un certificado sucinto (también llamado testigo polinómico) para todas las instancias cuya respuesta es un SÍ. La única forma de que tengan un tiempo polinomial es realizando una etapa aleatoria, incluyendo el azar de alguna manera para elegir una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta.

Completitud de NP

Para analizar la pregunta **P = NP**, resulta muy útil el concepto de completitud **NP**. De manera informal, los problemas de completitud **NP** son los problemas más "difíciles" en **P** en el sentido de que ellos son los que son más probables no se encuentren en P. Problemas **P**-difíciles son aquellos para los cuales cualquier problema en **P** puede ser reducido en tiempo polinómico. Los problemas de completitud **P** son aquellos problemas **P**-difícil que se encuentran en **P**. Si cualquier problema P-completo se encuentra contenido en **NP**, entonces se verificaría que **P = NP**. Desafortunadamente, se ha demostrado que muchos problemas importantes son **P**-completos y no se conoce la existencia de ningún algoritmo rápido para ellos.

Ejemplos

2. Camino Máximo: Dados dos vértices de un grafo encontrar el camino (simple) máximo.

3. Ciclo Hamiltoniano: Ciclo simple que contiene cada vértice del grafo.

NP-Completo

Son problemas en **NP** que son "los más difíciles" en **NP**. Si se encuentra un algoritmo polinomial para cualquier problema **NP**-completo, se puede encontrar un algoritmo polinomial para todos los problemas en **NP**.

Para abordar la pregunta de si P=NP, el concepto de la completitud de NP es muy útil. Informalmente, los problemas de NP-completos son los problemas más difíciles de NP, en el sentido de que son los más probables de no encontrarse en P. Los problemas de

NP-completos son esos problemas NP-duros que están contenidos en NP, donde los problemas NP-duros son estos que cualquier problema en P puede ser reducido a complejidad polinomial. Por ejemplo, la decisión del Problema del viajero de comercio es NP-completo, así que cualquier caso de cualquier problema en NP puede ser transformado mecánicamente en un caso del Problema del viajero de comercio, de complejidad polinomial. El Problema del viajero de comercio es de los muchos problemas NP-completos existentes.

Si cualquier problema NP-completo estuviera en P, entonces indicaría que P=NP. Desafortunadamente, se sabe que muchos problemas importantes son NP-completos y a fecha de hoy, no se conoce ningún algoritmo rápido para ninguno de ellos. Según esto, no es obvio que exista un problema NP-completo.

Un problema NP-completo trivial e ideado, se puede formular como: Dada una descripción de una máquina de Turing M que se detiene en tiempo polinómico, ¿existe una entrada de tamaño polinómico que M acepte? Es NP porque, dada una entrada, es simple comprobar si M acepta o no la entrada simulando M, es NP-duros porque el verificador para cualquier caso particular de un problema en NP puede ser codificado como una máquina M de tiempo polinomial que toma la solución para ser verificada como entrada. Entonces la pregunta de si el caso es o no un caso, está determinado por la existencia de una entrada válida. El primer problema natural que se demostró ser NP-completo fue el Problema booleano de satisfacibilidad. Este resultado es conocido como el teorema de Cook-Levin; su prueba de que la satisfacibilidad es NP-completo contiene los detalles técnicos sobre máquinas de Turing y como se relacionan con la definición de NP. Sin embargo, después se demostró que el problema era NP-completo, la prueba por reducción proporcionó una manera más simple de demostrar que muchos otros problemas están en esta clase. Así, una clase extensa de problemas aparentemente sin relación es reducible a otra, y son en este sentido el mismo problema.

El Problema P vs NP

Es uno de los problemas abiertos más importantes en matemáticas y computación. Se pregunta si P es igual a NP, es decir, si todos los problemas cuya solución puede verificarse rápidamente (NP) también pueden resolverse rápidamente (P).

Ejemplo

Algoritmo de Euclides para hallar el Máximo Común Divisor (MCD) de dos números naturales.

Matemáticamente, se buscan los divisores comunes de ambos números y se escoge el mayor de éstos.

Recursivamente, el MCD se define así:

$$MCD(m, n) = m,\ si\ n = 0$$
$$MCD(m, n) = MCD(n, m\ MOD\ n),\ si\ n > 0$$

Programa Python

```
class MaxCommonDivisor:

    # Constructor
    def __init__(self) -> None:
        pass

    def mcd(self, m, n) -> int:
        i = m if m < n else n
        while m % i != 0 or n % i != 0:
            i -= 1
        return i

    def mcdEuclides(self, m, n) -> int:
        while n > 0:
            aux = m
            m = n
            n = aux % n
        return m

    def mcdRecursive(self, m, n) -> int:
        if n == 0:
            return m
        else:
            return self.mcdRecursive(n, m % n)
```

Algoritmos de fuerza bruta

También conocidos como **algoritmos exhaustivos**, es una estrategia de resolución de problemas que intenta todas las posibles soluciones hasta encontrar la correcta o demostrar que no hay solución. Es simple de implementar, pero puede ser ineficiente para problemas grandes debido a su alta complejidad temporal.

Es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de fuerza bruta para encontrar el divisor de un número natural n consistiría en enumerar todos los enteros desde 1 hasta n, chequeando si cada uno de ellos divide n exactamente. Otro ejemplo de búsqueda por fuerza bruta, en este caso para solucionar el problema de las ocho reinas (posicionar ocho reinas en el tablero de ajedrez de forma que ninguna de ellas ataque al resto), consistiría en examinar todas las combinaciones de posición para las 8 reinas (en total 64!/8!(64-8)! = 4.426.165.368 posiciones diferentes), comprobando en cada una de ellas si las reinas se atacan mutuamente.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema.

Es un método utilizado también cuando es más importante una implementación sencilla que una mayor rapidez. Este puede ser el caso en aplicaciones críticas donde cualquier error en el algoritmo puede acarrear serias consecuencias; también es útil como método "base" cuando se desea comparar el desempeño de otros algoritmos metaheurísticos. La búsqueda de fuerza bruta puede ser vista como el método metaheurístico más simple.

Algoritmo básico de fuerza bruta

Para poder utilizar la fuerza bruta a un tipo específico de problema, se deben implementar las funciones **primero**, **siguiente**, **válido**, y **mostrar**. Todas recogerán el parámetro P indicando una instancia en particular del problema:

1. **primero (P)**: genera la primera solución candidata para P.
2. **siguiente (P, c)**: genera la siguiente solución candidata para P después de una solución candidata c.
3. **válido (P, c)**: chequea si una solución candidata c es una solución correcta de P.
4. **mostrar (P, c)**: informa que la solución c es una solución correcta de P.

La función **siguiente** debe indicar de alguna forma cuándo no existen más soluciones candidatas para el problema P después de la última. Una forma de realizar esto consiste en devolver un valor "nulo". De esta misma forma, la función **primero** devolverá un valor "nulo" cuando no exista ninguna solución candidata al problema P.

Pseudocódigo

```
c = primero(P)
Mientras c != nulo
    Si valido(P,c) Entonces
        mostrar(P, c)
        c = siguiente(P,c)
    FinSi
FinMientras
```

Por ejemplo, para buscar los divisores de un entero n, la instancia del problema P es el propio número n. La llamada **primero(n)** devolverá 1 siempre y cuando n ≥ 1, y "nulo" en otro caso; la función **siguiente(n, c)** debe devolver c + 1 si c < n, y "nulo" en caso contrario; **válido(n, c)** devolverá verdadero si y sólo si c es un divisor de n.

Nota

El algoritmo descrito anteriormente llama a la función **mostrar** para cada solución al problema. Este puede ser fácilmente modificado de forma que termine una vez encuentre la primera solución, o bien después de encontrar un determinado número de soluciones, después de probar con un número específico de soluciones candidatas, o después de haber consumido una cantidad fija de tiempo de CPU.

Explosión combinacional

La principal desventaja del método de fuerza bruta es que, para la mayoría de problemas reales, el número de soluciones candidatas es altamente elevado.

Por ejemplo, para buscar los divisores de un número n tal y como se describe anteriormente, el número de soluciones candidatas a probar será de n. Por tanto, si n consta de, digamos, 16 dígitos, la búsqueda requerirá de al menos 1015 comparaciones computacionales, tarea que puede tardar varios días en un ordenador personal. Si n es un bit de 64 dígitos, que aproximadamente puede tener hasta 19 dígitos decimales, la búsqueda puede tardar del orden de 10 años.

Este crecimiento exponencial en el número de candidatos, cuando crece el tamaño del problema ocurre en todo tipo de problemas. Por ejemplo, si buscamos una combinación particular de 10 elementos entonces tendremos que considerar 10! = 3,628,800 candidatos diferentes, lo cual en un PC habitual puede ser generado y probado en menos de un segundo. Sin embargo, añadir un único elemento más —lo cual supone solo un 10% más en el tamaño del problema— multiplicará el número de candidatos 11 veces — lo que supone un 1000% de incremento. Para 20 elementos el número de candidatos diferentes es 20!, es decir, aproximadamente 2.4×1018 o 2.4 millones de millones de millones; la búsqueda podría tardar unos 10000 años. A este fenómeno no deseado se le denomina **explosión combinacional**.

Fuerza bruta lógica

La fuerza bruta lógica, consiste básicamente en lo mismo, pero evitando casos que por razones demasiado obvias se sabe que quedan fuera de la solución buscada.

Este método sólo se usa cuando el número de posibilidades a evitar es lo suficientemente grande y que contando con el tiempo de ejecución del código necesario para implementarlo, reduzca ampliamente el tiempo necesario para encontrar el resultado esperado.

Cuando se habla de **fuerza bruta lógica**, por contraste, la contrapuesta es llamada **fuerza bruta ciega**.

Con el ejemplo de la factorización, cuando tratamos por fuerza bruta de encontrar el divisor de un número natural n enumeraríamos todos los enteros desde 1 hasta n, chequeando si cada uno de ellos divide a n sin generar resto. La fuerza bruta lógica haría lo mismo pero solo con los números primos, dada una tabla de primos, o solo con los impares y el 2 si no poseemos una tabla de primos. Dado que sabemos que cualquier número está compuesto de primos en cualquier cantidad y esto es demasiado obvio, no es absolutamente necesario revisar el 4,6,8,10,12,14,15,16 si ya hemos mirado el 2,3,5,7...

La fuerza bruta lógica sigue siendo fuerza bruta, ya que recorre sin estrategia el espacio de posibilidades pero descartando posibilidades muy obvias y relativamente fáciles de implementar.

En *criptografía*, se denomina **ataque de fuerza bruta** a la forma de recuperar una clave probando todas las combinaciones posibles hasta encontrar aquella que permite el acceso.

Dicho de otro modo, define al procedimiento por el cual a partir del conocimiento del algoritmo de cifrado empleado y de un par texto claro/texto cifrado, se realiza el cifrado (respectivamente, descifrado) de uno de los miembros del par con cada una de las posibles combinaciones de clave, hasta obtener el otro miembro del par. El esfuerzo requerido para que la búsqueda sea exitosa con probabilidad mejor que la par será 10^n-1 operaciones, donde n es la longitud de la clave (también conocido como el *espacio de claves*). Este enfoque no depende de tácticas intelectuales, se basa en hacer muchos intentos.

Otro factor determinante en el coste de realizar un ataque de fuerza bruta es el juego de caracteres que se pueden utilizar en la clave. Contraseñas que sólo utilicen dígitos numéricos serán más fáciles de descifrar que aquellas que incluyen otros caracteres como letras, así como las que están compuestas por menos caracteres serán también más fáciles de descifrar. La complejidad impuesta por la cantidad de caracteres en una contraseña es logarítmica.

La fuerza bruta suele combinarse con un ataque de diccionario, en el que se encuentran diferentes palabras para ir probando con ellas.

Estos tipos de ataques, no son rápidos, para una contraseña compleja, puede tardar siglos (aunque también depende de la capacidad de operación del ordenador que lo ejecute).

En la actualidad este tipo de ataques son usados para hackear Facebook, correos electrónicos, auditar redes WiFi y otras redes sociales.

Se puede calcular la cantidad de posibles contraseñas. De esto depende la cantidad de caracteres de la contraseña en cuestión, y el conjunto de caracteres.

Divide y vencerás

En informática , **"divide y vencerás"** (**DyV**) es un paradigma de diseño de algoritmos . Un algoritmo de "divide y vencerás" descompone recursivamente un problema en dos o más subproblemas del mismo tipo o de tipo relacionado, hasta que estos se vuelven lo suficientemente simples como para resolverse directamente. Las soluciones de los subproblemas se combinan para obtener una solución al problema original.

La técnica de divide y vencerás es la base de algoritmos eficientes para muchos problemas, como la ordenación (por ejemplo, quicksort, mergesort), la multiplicación de números grandes (por ejemplo, el algoritmo Karatsuba), la búsqueda del par de puntos más cercano, el análisis sintáctico (por ejemplo, los analizadores de arriba hacia abajo) y el cálculo de la

transformada de Fourier discreta (FFT), así como el máximo común divisor y la búsqueda binaria[65], entre otros ejemplos.

Diseñar algoritmos eficientes de divide y vencerás puede ser difícil. Al igual que en la inducción matemática, a menudo es necesario generalizar el problema para que sea susceptible de una solución recursiva. La exactitud de un algoritmo de divide y vencerás suele demostrarse mediante inducción matemática, y su coste computacional suele determinarse resolviendo relaciones de recurrencia.

El término "divide y vencerás" se aplica a veces a algoritmos que reducen cada problema a un solo subproblema, como el algoritmo de búsqueda binaria para encontrar un registro en una lista ordenada (o su análogo en computación numérica , el algoritmo de bisección para la búsqueda de raíces). Estos algoritmos pueden implementarse de forma más eficiente que los algoritmos generales de divide y vencerás; en particular, si utilizan recursión de cola, pueden convertirse en bucles simples. Sin embargo, bajo esta definición amplia, todo algoritmo que utilice recursión o bucles podría considerarse un "algoritmo de divide y vencerás". Por lo tanto, algunos autores consideran que el término "divide y vencerás" debería utilizarse sólo cuando cada problema pueda generar dos o más subproblemas. Se ha propuesto el término "decrecimiento y vencerás" para la clase de un solo subproblema.

Una aplicación importante de dividir y vencer es en la optimización, donde si el espacio de búsqueda se reduce ("poda") por un factor constante en cada paso, el algoritmo general tiene la misma complejidad asintótica que el paso de poda, y la constante depende del factor de poda (sumando las series geométricas); esto se conoce como podar y buscar.

El paradigma de divide y vencerás se utiliza a menudo para encontrar la solución óptima de un problema. Su idea básica es descomponer un problema dado en dos o más subproblemas similares, pero más simples, resolverlos uno por uno y componer sus soluciones para resolver el problema dado. Los problemas con suficiente simplicidad se resuelven directamente.

El término "divide y vencerás" se aplica a veces a algoritmos que reducen cada problema a un solo subproblema, como el algoritmo de búsqueda binaria para encontrar un registro en una lista ordenada (o su análogo en computación numérica , el algoritmo de bisección para la búsqueda de raíces). Estos algoritmos pueden implementarse de forma más eficiente que los algoritmos generales de divide y vencerás; en particular, si utilizan recursión de cola, pueden convertirse en bucles simples. Sin embargo, bajo esta definición amplia, todo algoritmo que utilice recursión o bucles podría considerarse un "algoritmo de divide y vencerás". Algunos autores consideran que el término "divide y vencerás" debería utilizarse sólo cuando cada problema pueda generar dos o más subproblemas. Se ha propuesto el término "decrecimiento y vencerás" para la clase de un solo subproblema.

[65] La búsqueda binaria es un algoritmo de decrecimiento y conquista donde los subproblemas tienen aproximadamente la mitad del tamaño original, no es precisamente nuevo. La descripción clara del algoritmo en computación la da John Mauchly en un artículo de 1946, sin embargo, la idea de usar una lista ordenada de elementos para facilitar la búsqueda se remonta al menos a Babilonia en el año 200 a. C. Otro antiguo algoritmo de decrecimiento y conquista es el algoritmo euclidiano para calcular el máximo común divisor de dos números reduciendo los números a subproblemas equivalentes cada vez más pequeños, y que data de varios siglos antes de Cristo.

Diseño e implementación

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

5. Se plantea el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n / k, y donde $0 \leq n / k < n$. A esta tarea se le conoce como división.
6. Luego se resuelven independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema garantiza la convergencia hacia los casos elementales, también denominados casos base.
7. Al final se combinan las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Los algoritmos divide y vencerás (o *divide and conquer*, en inglés), se diseñan como procedimientos generalmente recursivos.

La corrección de un algoritmo de “divide y vencerás”, está habitualmente probada por inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.

Un algoritmo genérico de implementación de DyV puede ser el siguiente:

```
AlgoritmoDyV (p: TipoProblema): TipoSolucion
   si esCasoBase(p)
      retorne resuelve(p)
   sino
      subproblemas: arreglo de TipoProblema
      subproblemas = divideEnSubproblemas(p)
      solucionesParciales: arreglo de TipoSolucion

      para cada sp en subproblemas
         solucionesParciales.agregar(AlgoritmoDYV(sp))
      finPara

      retorne mezcla(solucionesParciales)
   finSi
finAlgoritmoDyV
```

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

- El diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
- Los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Sin embargo, este tipo de algoritmos también se pueden implementar como un algoritmo no recursivo que almacene las soluciones parciales en una estructura de datos explícita, como puede ser una pila, cola, o cola de prioridad. Esta aproximación da mayor libertad al diseñador, de forma que se pueda escoger qué subproblema es el que se va a resolver a continuación, lo que puede ser importante en el caso de usar técnicas como ramificación y acotación o de optimización.

Recursión

Los algoritmos de “divide y vencerás” están naturalmente implementados, como procesos recursivos. En ese caso, los subproblemas parciales encabezados por aquel que ya ha sido resuelto se almacenan en la pila de llamadas de procedimientos.

Pila explícita

Los algoritmos de divide y vencerás también pueden ser implementados por un programa no recursivo que almacena los subproblemas parciales en alguna estructura de datos explícita, tales como una pila, una cola, o una cola de prioridad. Este enfoque permite más libertad a la hora de elegir los subproblemas a resolver después, una característica que es importante en algunas aplicaciones, por ejemplo en la búsqueda en anchura y en el método de ramificación y poda para optimización de subproblemas. Este enfoque es también la solución estándar en lenguajes de programación que no permiten procedimientos recursivos.

Elección de los casos base

En los algoritmos recursivos hay una libertad considerable para elegir los casos bases, los subproblemas pequeños que son resueltos directamente para acabar con la recursión.

Elegir los casos base más pequeños y simples posibles es más elegante y normalmente nos da lugar a programas más simples, porque hay menos casos a considerar y son más fáciles de resolver. Por ejemplo, el algoritmo quicksort de ordenación podría parar cuando la entrada es una lista vacía. Sólo hay que considerar un caso base, que no se define en términos de si mismo.

Por otra parte, la eficiencia normalmente mejora si la recursión se para en casos relativamente grandes, y estos son resueltos no recursivamente. Esta estrategia evita la sobrecarga de llamadas recursivas que hacen poco o ningún trabajo, y pueden también permitir el uso de algoritmos especializados no recursivos que, para esos casos base, son

más eficientes que la recursión explícita. Ya que un algoritmo de DyV reduce cada instancia del problema o subproblema a un gran número de instancias base, éstas habitualmente dominan el coste general del algoritmo, especialmente cuando la sobrecarga de separación/unión es baja. Es de observar que estas consideraciones no dependen de si la recursión está implementada por compilador o por pila explícita.

Programación dinámica

Concepto

Memorización y tabulación

Concepto

Algoritmos de programación dinámica

Concepto

Algoritmos probabilistas

Un **algoritmo probabilístico** utiliza decisiones aleatorias en su funcionamiento para alcanzar un objetivo, obteniendo resultados correctos en promedio para una amplia variedad de entradas. A diferencia de los algoritmos determinísticos, que siempre producen el mismo resultado para una entrada dada, los algoritmos probabilísticos pueden variar su comportamiento y resultados debido a la introducción de aleatoriedad en su proceso.

Repasemos algunos conceptos básicos de la teoría de la probabilidad.

Combinatoria

Definición (permutación)
Una **permutación** de `n` objetos es una disposición ordenada de los objetos.

Esta definición nos habla del número de formas en que se pueden ordenar `n` objetos. Por ejemplo, si tenemos tres objetos `a, b c`, los podemos ordenar de la siguiente forma: `abc, acb, bac, bca, cab, cba`.

La experiencia muestra que el número de permutaciones de `n` objetos es `n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1`. Para el caso de los tres objetos `a, b c`, vemos que el número de permutaciones es `3! = 3 * 2 * 1 = 6`.

Definición (combinación)
Una **combinación** de *r* objetos de entre *n* objetos es una selección de *r* objetos sin tener en cuenta el orden.

Por ejemplo, si tenemos los objetos *a, b, c, d*, podemos seleccionar dos objetos de *6* maneras diferentes, siempre y cuando no se tenga en cuenta el orden: *ab, ac, ad, bc, bd, cd*.

Si se tiene en cuenta el orden, el número de formas de seleccionar *r* objetos de *n* es: *n (n - 1) (n - 2) ... (n - r + 1)*.

Si no se tiene en cuenta el orden, el número de formas de seleccionar *r* objetos de entre *n* es[66]:

$C_r^n = \frac{n!}{r!(n-r)!}$; $si\ 0 \leq r \leq n$.

Si *r = 0*, sólo tenemos una forma de no seleccionar ningún objeto, por tanto tomamos: $C_0^n = 1$.

Si *r < 0*, lo cual no tiene sentido en términos combinatorios, se define $C_r^n = 0$.

Si *r > n*, se está diciendo que van a elegir más de n objetos de n disponibles, lo cual es imposible, por tanto $C_r^n = 0$, ya que hay cero formas de hacer dicha elección.

Probabilidad

La teoría de la **probabilidad** se encarga del estudio de fenómenos **aleatorios**, esto es, de analizar eventos en los que su futuro no se puede conocer con certeza. Para aplicar esta teoría, se consideran estos fenómenos como **experimentos aleatorios**, cuyo resultado no se conoce con certeza. Los experimentos pueden ser de muchos tipos, como lanzar una moneda o dado, hasta esperar los resultados que pueda arrojar un sistema informático a partir de unas entradas. Cada dato obtenido es anotado y se denomina **resultado**.

El **espacio muestral** S es el conjunto de todos los resultados posibles de un experimento aleatorio. Cada resultado individual se conoce como **punto muestral** o **suceso elemental**. Un espacio muestral puede ser finito o infinito. El espacio muestral es *discreto* si el número de puntos muestrales (elementos del conjunto) es finito o si se pueden rotular éstos utilizando los número enteros; de lo contrario, diremos que es *continuo*.

Si el experimento consiste en lanzar un dado, el espacio muestral será *S = {1, 2, 3, 4, 5, 6}*, el cual es finito y discreto. Si el experimento aleatorio consiste en medir los tiempos de respuesta de un sistema informático, el espacio muestral es *S = {t | t > 0}*, el cual

[66] En los textos de estadística se encuentran otras notaciones para expresar el número de combinaciones de r elementos tomados de n disponibles.

es continuo. Si el experimento aleatorio consiste en contar las aves que pasan por un cierto lugar en un tiempo determinado, el espacio muestral es *S = {0, 1, 2, 3, 4, 5, ...}*, el cual es infinito, pero discreto, ya que puede asignarse un número entero a cada ave.

Un **suceso** es un subconjunto de un espacio muestral. Un suceso A *sucede (ocurre)* si el experimento aleatorio arroja un resultado que se hace parte de *A*. Por ejemplo un suceso A al lanzar un dado puede ser obtener un número par, por tanto *A = {2, 4, 6}*. Decimos que el conjunto vacío ∅ es un **suceso imposible** y el espacio muestral es un **suceso universal**.

Dado que un espacio muestral es un conjunto y los sucesos subconjuntos, son válidas las operaciones entre ellos, como unión, intersección, etc., por lo que es posible formar nuevos sucesos a partir de éstas. Por ejemplo, decir que el suceso "A no sucede", corresponde a decir "todos los sucesos ocurren excepto el suceso A", que en conjuntos equivale al complemento de *A* y que representamos así *A'*[67]. Decir que "sucede *A* ó *B*, o ambos", representa la unión entre los sucesos *A* y *B*: A ∪ B. De manera análoga, la frase "sucede A y B" corresponde a la intersección de los sucesos *A* y *B*: A ∩ B. Dos sucesos son **mutuamente excluyentes** si A ∩ B = ∅**.**

Una **medida de probabilidad** es una función que asigna un valor numérico *Pr[A]* a todo suceso A del espacio muestral. Ésta debe cumplir estos tres axiomas:

25. Para todo suceso *A, Pr[A] >= 0*
26. *Pr[S] = 1*
27. Si los sucesos *A* y *B* son mutuamente excluyentes (A ∩ B = ∅), entonces *Pr*[A ∪ B] = *Pr[A] + Pr[B]*. Este axioma puede extenderse por medio de inducción matemática para n sucesos.

De estos axiomas se deduce:

1. *Pr[A'] = 1 - Pr[A]*, para todo suceso *A*.
2. *Pr[A ∪ B] = Pr[A] + Pr[B] - Pr[A ∩ B]*, para todo suceso *A* y *B*.

Enfoque básico para resolver problemas de probabilidad

Aunque en la práctica y en los casos reales es posible tomar distintos caminos, los siguientes pasos muestran una guía para abordar problemas relacionados con probabilidad:

- Identificar el espacio muestral
- Asignar probabilidades a los elementos del espacio muestral
- Identificar los sucesos de interés
- Calcular las probabilidades deseadas

[67] En los textos de matemáticas es posible que encuentre otras notaciones para el complemento de un conjunto, como poner una barra encima del nombre de éste .

Tipos de algoritmos de probabilidad

Los algoritmos probabilísticos se dividen en varias categorías según su método y aplicación. Estos incluyen algoritmos de inferencia bayesianos, que se basan en teoremas bayesianos para actualizar las creencias a la luz de nueva evidencia, y son fundamentales para la clasificación y la predicción. Los métodos Monte Carlo de cadena de Markov (MCMC) son otra categoría utilizada para muestrear distribuciones complejas generando secuencias de números aleatorios. Los algoritmos de regresión logística también son importantes en problemas de clasificación binaria, donde estiman la probabilidad de que una observación caiga en una categoría determinada. Los algoritmos de árboles de decisión, por otro lado, utilizan probabilidades condicionales para dividir los datos en subconjuntos homogéneos. Estos diferentes tipos de algoritmos probabilísticos tienen una variedad de aplicaciones, desde inteligencia artificial hasta ciencia de datos y toma de decisiones.

Algoritmos de regresión logística

La **regresión logística** es una técnica de análisis de datos que utiliza las matemáticas para encontrar las relaciones entre dos factores de datos. Luego, utiliza esta relación para predecir el valor de uno de esos factores basándose en el otro. Normalmente, la predicción tiene un número finito de resultados, como un sí o un no.

Por ejemplo, supongamos que desea adivinar si el visitante de su sitio web va a hacer clic en el botón de pago de su carrito de compras o no. El análisis de regresión logística analiza el comportamiento de los visitantes anteriores, como el tiempo que permanecen en el sitio web y la cantidad de artículos que hay en el carrito. Determina que, si anteriormente los visitantes pasaban más de cinco minutos en el sitio y agregaban más de tres artículos al carrito, hacían clic en el botón de pago. Con esta información, la función de regresión logística puede predecir el comportamiento de un nuevo visitante en el sitio web.

La **regresión logística** implementa un algoritmo de clasificación, específicamente para problemas binarios, que predice la probabilidad de que un evento pertenezca a una de dos categorías. Es un algoritmo estadístico que, a través de una función logística (conocida como *sigmoide*), modela la relación entre las variables independientes (o predictores) y la variable dependiente binaria.

La regresión logística es una técnica importante en el campo de la inteligencia artificial y el machine learning (AI/ML). Los modelos de ML son programas de software que se pueden entrenar para realizar tareas complejas de procesamiento de datos sin intervención humana. Los modelos de ML creados mediante regresión logística ayudan a las organizaciones a obtener información procesable a partir de sus datos empresariales. Pueden usar esta información para el análisis predictivo a fin de reducir los costos operativos, aumentar la eficiencia y escalar más rápido. Por ejemplo, las empresas pueden descubrir patrones que mejoran la retención de los empleados o conducen a un diseño de productos más rentable.

Proceso general de la regresión logística

1. Definición del problema: se identifica la variable dependiente binaria que se quiere predecir (por ejemplo: sí/no, 0/1) y las variables independientes que se creen que influyen en ella.
2. Recopilación y preparación de datos: se recopilan datos históricos y se preparan para el análisis, incluyendo limpieza, preprocesamiento y transformación de variables.
3. Modelo de regresión: se construye el modelo de regresión logística, que utiliza la función logística para modelar la relación entre las variables independientes y la probabilidad de la variable dependiente.
4. Ajuste del modelo: se ajustan los parámetros del modelo (pesos o coeficientes) mediante un proceso de optimización, generalmente utilizando métodos de máxima verosimilitud, para minimizar la diferencia entre los valores predichos y los reales.
5. Evaluación del modelo: se evalúa la precisión del modelo utilizando métricas como la precisión, la sensibilidad, la especificidad, el área bajo la curva ROC (AUC) y otras.
6. Predicción: una vez ajustado y evaluado, el modelo puede utilizarse para predecir la probabilidad de que un nuevo evento pertenezca a una de las dos categorías.

Ejemplos de aplicaciones

Hay una variada gama de aplicaciones de la regresión logística, algunas de ellas son:

- Predicción de spam: determinar si un correo electrónico es spam o no spam.
- Diagnóstico médico: predecir si un paciente tiene o no una enfermedad específica.
- Detección de fraude: identificar transacciones fraudulentas.
- Marketing: predecir si un cliente va a responder a una campaña de marketing.
- Fabricación: estimar la probabilidad de fallo de las piezas en la maquinaria.
- Finanzas: analizar las transacciones financieras en busca de fraudes y evaluar las solicitudes de préstamos y seguros en busca de riesgos.
- Transporte: analizar las distintas situaciones relacionadas con el transporte, tales como accidentalidad, velocidad de flujo en horas pico, etc.

Algoritmos de árboles de decisión

Un **árbol de decisión** es un algoritmo de aprendizaje supervisado no paramétrico, que se utiliza tanto para tareas de clasificación como de regresión. Tiene una estructura jerárquica de árbol, que consta de un nodo raíz, ramas, nodos internos y nodos terminales u hojas.

Un árbol de decisión tiene la estructura de un árbol binario y el aprendizaje en ellos emplea una estrategia de divide y vencerás realizando una búsqueda codiciosa para identificar los puntos de división óptimos dentro de un árbol. Este proceso de división se repite de forma descendente y recursiva hasta que todos o la mayoría de los registros se hayan clasificado con etiquetas de clase específicas.

El hecho de que todos los puntos de datos se clasifiquen como conjuntos homogéneos depende en gran medida de la complejidad del árbol de decisión. Los árboles más pequeños tienen más facilidad para alcanzar nodos de hojas puras, es decir, puntos de datos en una sola clase. Sin embargo, a medida que un árbol crece en tamaño, se vuelve

cada vez más difícil mantener esta pureza y, por lo general, da como resultado que caigan muy pocos datos dentro de un subárbol determinado. Cuando esto ocurre, se conoce como fragmentación de datos y, a menudo, puede conducir a un sobreajuste.

Como resultado, los árboles de decisión tienen preferencia por los árboles pequeños, lo que es coherente con el principio de parsimonia de la Navaja de Occam; es decir, "las entidades no deben multiplicarse más allá de lo necesario". Dicho de otra manera, los árboles de decisión deben agregar complejidad solo si es necesario, ya que la explicación más simple suele ser la mejor. Para reducir la complejidad y evitar el sobreajuste, se suele emplear la poda; se trata de un proceso que elimina las ramas que se dividen en características con poca importancia. A continuación, se puede evaluar el ajuste del modelo mediante el proceso de validación cruzada.

Otra forma en que los árboles de decisión pueden mantener su precisión es formando un conjunto mediante un algoritmo bosque aleatorio; este clasificador predice resultados más precisos, sobre todo cuando los árboles individuales no están correlacionados entre sí.

Tipos de árboles de decisión

El algoritmo de Hunt, desarrollado en la década de 1960 para modelar el aprendizaje humano en Psicología, es la base de muchos algoritmos populares de árboles de decisión, como los siguientes:

- ID3: a Ross Quinlan se le atribuye el desarrollo de ID3, que es la abreviatura de "Iterative Dichotomiser 3". Este algoritmo aprovecha la entropía y la ganancia de información como métricas para evaluar las divisiones de los candidatos.

- C4.5: este algoritmo se considera una iteración posterior de ID3, que también fue desarrollado por Quinlan. Puede utilizar la ganancia de información o los ratios de ganancia para evaluar los puntos de división dentro de los árboles de decisión.

- CART: el término CART es una abreviatura de "árboles de clasificación y regresión" y fue introducido por Leo Breiman. Este algoritmo suele utilizar la impureza de Gini para identificar el atributo ideal para dividir. La impureza de Gini mide la frecuencia con la que se clasifica erróneamente un atributo elegido al azar. Al evaluar utilizando la impureza de Gini, un valor más bajo es más ideal.

Ventajas y desventajas de los árboles de decisión

Aunque los árboles de decisión se pueden utilizar en una variedad de casos de uso, otros algoritmos generalmente superan a los algoritmos de árboles de decisión. Los árboles de decisión son particularmente útiles para las tareas de minería de datos y descubrimiento de conocimiento.

Ventajas

- Fácil de interpretar: la lógica booleana y las representaciones visuales de los árboles de decisión facilitan su comprensión y consumo. La naturaleza jerárquica de un árbol de decisión también hace que sea fácil ver qué atributos son los más importantes, lo que no siempre está claro con otros algoritmos, como las redes neuronales.

- Requiere poca o ninguna preparación de datos: los árboles de decisión tienen una serie de características que los hacen más flexibles que otros clasificadores. Puede manejar varios tipos de datos, es decir, valores discretos o continuos, y los valores continuos se pueden convertir en valores categóricos mediante el uso de umbrales. Además, también puede manejar valores con valores faltantes, lo que puede ser problemático para otros clasificadores, como Naive Bayes.

- Más flexibles: los árboles de decisión pueden utilizarse tanto para tareas de clasificación como de regresión, lo que los hace más flexibles que otros algoritmos. También es insensible a las relaciones subyacentes entre atributos; esto significa que si dos variables están muy correlacionadas, el algoritmo sólo elegirá una de las características para dividir.

Desventajas

- Propenso al sobreajuste: los árboles de decisión complejos tienden a sobreajustarse y no se generalizan bien a los nuevos datos. Este escenario se puede evitar mediante los procesos de poda previa o posterior a la poda. La poda previa detiene el crecimiento de los árboles cuando no hay datos suficientes, mientras que la poda elimina los subárboles con datos inadecuados tras la construcción del árbol.

- Estimadores de alta varianza: pequeñas variaciones dentro de los datos pueden producir un árbol de decisión muy diferente. Embolsado, o el promedio de estimaciones, puede ser un método para reducir la varianza de árboles de decisión. Sin embargo, este enfoque es limitado, ya que puede conducir a predictores altamente correlacionados. Es fácil que el árbol se degenere con cierto datos ingresados.

- Más costoso: dado que los árboles de decisión adoptan un enfoque de búsqueda codicioso durante la construcción, su entrenamiento puede resultar más costoso en comparación con otros algoritmos.

Tratamiento de la probabilidad en los lenguajes de programación

Los lenguajes de programación disponen a su vez de funciones para generar números **pseudoaleatorios**; esto es realizado por un algoritmo basado en **reglas matemáticas**, además de una **semilla**, la cual es utilizada por dichas reglas para generar los números. Actualmente los lenguajes tienen la semilla implícita en la funciones de generación, generalmente asociada al tiempo del sistema, por lo que no es necesario especificarla, aunque en algunos lenguajes se puede establecer de forma explícita. El nombre de pseudoaleatorios proviene de que son números que se generan a partir de un algoritmo con unas reglas matemáticas, pero que en realidad no son aleatorios, y es ahí donde entra en juego y se manifiesta la importancia de la semilla, ya que ésta se encarga de inicializar el algoritmo generador. Una semilla constante, siempre generará la misma secuencia de números pseudoaleatorios, por lo que tomar el tiempo como semilla es una opción apropiada, ya que éste cambia siempre, y así el algoritmo basará sus cálculos en valores iniciales diferentes.

Algunos lenguajes tienen funciones que generan un número pseudoaleatorio entero comprendido en un intervalo (`[límite_inferior, límite_superior]`); algunos de estos lenguajes permiten que el segundo parámetro sea opcional, siendo el límite superior igual en este caso al mayor entero disponible en el lenguaje. Otros lenguajes entregan un número real en el intervalo `[0, 1]`.

La clase Math de Java dispone de un método para generar números pseudo aleatorios: **`random`**, el cual devuelve un real entre 0 y 1, con una aproximación de 17 cifras significativas. Por ejemplo: *`Math.random()`* puede devolver algo como `0.28737539213034247` ó `0.03293366594495917`. Si queremos obtener un entero comprendido en un intervalo, por ejemplo `[1, 10]`, podemos hacer lo siguiente, tal y como se ilustra en el siguiente ejemplo.

Ejemplo

Si queremos obtener en Java un entero comprendido en un intervalo, por ejemplo entre 1 y 10 `[1, 10]`, podemos hacer la siguiente operación aritmética usando `Math.random()`.

```
class Maths
{
    public static void main(String[] args)
    {
        double randNumb = Math.random() * 10 + 1;
        System.out.println("Number random:" + randNumb);
        System.out.println("Number random int:" + (int) randNumb);
    }
}
```

Nota

Observe que la fórmula general para obtener números enteros en Java en un intervalo cualquiera se establece así:

```
(int) (Math.random() * (límSuperior - límInferior + 1) + límInferior).
```

Por tanto, podemos reescribir nuestro programa anterior para generar un número aleatorio usando una función parametrizada, así:

```
public class Maths
{
    public int randomNumber(int minLim, int maxLim)
    {
        double randNumb = Math.random() * (maxLim - minLim + 1) + minLim;
        return (int) randNumb;
    }
}
```

Podemos llamar la función anterior generando `n` (en este caso `n = 10`) números aleatorios, así:

```
public static void main(String[] args)
{
    Maths mt = new Maths();
    for (int i = 1; i <= 10; i++) {
        System.out.println("Number random int:" + mt.randomNumber(1, 10));
    }
}
```

Java también dispone de la clase **Random** para trabajar con números pseudoaleatorios generados en un rango determinado, como lo veremos en un ejemplo más adelante.

Asignar una probabilidad mayor a un dato de un conjunto

Es posible que en un fenómeno distintos eventos tengan distintas probabilidades de suceder, como por ejemplo que en un dado cargado uno de los lados caiga con mayor probabilidad que los demás. En los lenguajes de programación se puede simular esto aplicado tanto a valores individuales como a listas, que serán las que definen el universo de eventos o sucesos.

Los pasos para realizar esto pueden ser los siguientes:

i. Generar un número pseudoaleatorio: usando alguna de las funciones disponibles en los lenguajes o usando un generador congruencial.
j. Definir un rango de probabilidades: determinar el rango de números a usar para representar las probabilidades. Por ejemplo, en el rango de 1 a 10, el 100% equivale a 10.
k. Asignar probabilidades a los datos: asignar a cada dato un rango de números que representan su probabilidad. Por ejemplo, un dato que tenga un 60% de probabilidad, se le puede asignar el rango del 0 al 7.
l. Comparar con el número aleatorio: si el número aleatorio cae dentro del rango de probabilidad asignado a un dato, entonces ese dato es elegido.

Ejemplo

Sea U = {A, B}; la probabilidad de que suceda A es del 70% y la de B es del 30%. Simular esta situación en Java generando un número aleatorio entre uno y diez usando la clase Random.

El método `nextInt` de la clase `Random` puede ser invocado sin parámetros (`Random.nextInt()`), con lo cual se genera un número pseudoaleatorio comprendido en el rango de los enteros, incluidos tanto valores negativos como positivos. Si se especifica un argumento entero (`Random.nextInt(n)`), se genera entonces un número pseudoaleatorio entre cero y el número dado menos uno (`[0, n - 1]`); por ejemplo, si `n = 10`, entonces el método `nextInt` generará pseudoaleatorios entre 0 y 9.

```
import java.util.Random;
import java.util.Scanner;

class Main
{
    public static void main(String[] args)
    {
        Random random = new Random();
        Scanner sc = new Scanner(System.in);

        // Definir las probabilidades de cada dato
        int probA = 7; // 70% de probabilidad
        int probB = 3; // 30% de probabilidad

        // Generar eventos con distinta probabilidad de ocurrencia
        String op;
        do {
            int numRandom = random.nextInt(10);

            // Determinar qué dato se elige
            if (numRandom < probA) {
                    System.out.println("Dato seleccionado al azar:
A");
            } else {
                    System.out.println("Dato seleccionado al azar:
B");
            }
            System.out.println("Seguir [s/?]");
            op = sc.next().toLowerCase();
        } while (op.equals("s"));
    }
}
```

Python por su lado incluye una librería llamada **`random`** que dispone de una serie de métodos para trabajar con números pseudo aleatorios[68]. Por ejemplo, para generar un aleatorio (entero) entre 1 y 10, indicamos el método randint de la clase random especificando el intervalo respectivo por medio de dos parámetros: `random.randint(1, 10)`.

Ejemplo

Otros métodos interesantes de esta clase son los que operan sobre secuencias: **`choice`**, el cual elige aleatoriamente un elemento de una secuencia; **`shuffle`**, que devuelve una lista con los elementos distribuidos aleatoriamente:

```
import random
```

[68] Puede consultar más acerca de esta librería en: Generate pseudo-random numbers — Python 3.13.1 documentation y en Python Random Module

```
print(f"Número aleatorio entre 1 y 10: {random.randint(1, 10)}")
data_list = [0, 5, 4, 8]
print(f"Lista: {data_list}")
print(f"choice: {random.choice(data_list)}")
random.shuffle(data_list)
print(f"Lista shuffle: {data_list}")
```

Generación de números aleatorios: generadores congruenciales

Un número pseudoaleatorio representa la probabilidad de un suceso cualquiera y son la base de la simulación. Los números son generados suponiendo que las variables aleatorias son independientes e idénticamente distribuidas (i.i.d.) $U(0, 1)$.

Los principales generadores de números pseudo-aleatorios utilizados hoy en día son los llamados **generadores congruenciales lineales**, introducidos por Lehmer en 1951. Un método congruencial comienza con un valor inicial (**semilla**) x_0, y los sucesivos valores x_n, $n >= 1$ se obtienen recursivamente así:

$$x_n = (ax_{n-1} + b) \% m$$

Donde a, b, m son enteros positivos conocidos como el *multiplicador*, el *incremento* y el *módulo*, respectivamente.

La sucesión de números pseudoaleatorios U_n $(n >= 1)$ se obtienen así:

$$U_i = x_i / m$$

Ejemplo

Generadores congruenciales lineales. Este ejemplo muestra una forma de generar números pseudoaleatorios en Python con un método congruencial simple.

```
def pseudo_random(n, x0, a, b, m) -> None:
    u = []
    x = []
    seed = x0
    for i in range(1, n, 1):
        seed = (a * seed + b) % m
        x.append(seed)
        u.append(seed / m)
    print(x)
    print(u)

pseudo_random(5, 50, 7 ** 5, 0, 2 ** 31 -1)
```

Al ser la semilla constante, la secuencia de números se repite cada ejecutamos el programa, pero si indicamos una semilla apropiada que varíe, podemos obtener una

secuencia distinta cada vez en cada ejecución, simulando adecuadamente los números pseudoaleatorios. Es claro que un dato que siempre cambia es el tiempo, por lo que podemos usarlo como semilla. Veamos cómo queda la solución indicando como semilla al **timestamp** del sistema, el cual devuelve un entero que indica el tiempo del sistema.

```
from datetime import datetime

def pseudo_random(n, x0, a, b, m) -> None:
    u = []
    x = []
    seed = x0
    for i in range(1, n + 1, 1):
        seed= (a * seed + b) % m
        x.append(seed)
        u.append(x[i - 1] / m)
    print(x)
    print(u)

now = datetime.now()
pseudo_random(5, datetime.timestamp(now), 7 ** 5, 0, 2 ** 31 - 1)
```

Una adecuada elección de los parámetros permite obtener números aparentemente pseudoaleatorios de manera eficiente. Park y Miller (1988) propusieron en el ***minimal standard*** los siguientes valores para las constantes: $a = 7^5, b = 0, m = 2^{31} - 1$.

Algoritmos voraces

En informática, un **algoritmo voraz** (también llamado ávido, codicioso, devorador) es aquel que encuentra la solución a un problema en el menor tiempo posible. Elige la ruta que parece óptima en ese momento, sin tener en cuenta la optimización global de la solución resultante.

Edsger Dijkstra, informático y matemático que deseaba calcular un árbol de expansión mínima, introdujo el término "algoritmo voraz". Prim y Kruskal desarrollaron técnicas de optimización para minimizar el coste de los grafos.

Algoritmos codiciosos frente a algoritmos no codiciosos

Un algoritmo es voraz cuando la ruta elegida se considera la mejor opción según un criterio específico, sin tener en cuenta las consecuencias futuras. Sin embargo, normalmente evalúa la viabilidad antes de tomar una decisión final. La corrección de la solución depende del problema y de los criterios utilizados.

Por ejemplo, en un grafo que tiene distintos valores, se debe determinar el valor máximo en éste. Se empezaría buscando en cada nodo y comprobando su peso para ver si es el valor más alto.

Existen dos enfoques para resolver este problema: un enfoque voraz y un enfoque no voraz.

En el enfoque codicioso el algoritmo se detiene una vez que obtiene una solución óptima, sin ser necesariamente la mejor, y al ejecutarse una vez, no tiene opción de realizar correcciones; mientras que en el enfoque no codicioso se revisan todas las opciones antes de concluir el resultado final.

Características de un algoritmo voraz

- El algoritmo resuelve el problema encontrando una solución óptima. Esta solución puede ser un valor máximo o mínimo. Toma decisiones basándose en la mejor opción disponible.
- El algoritmo es rápido y eficiente, con una complejidad temporal de O(nlog n) u O(n). Por lo tanto, se aplica en la resolución de problemas a gran escala.
- La búsqueda de la solución óptima se realiza sin repetición: el algoritmo se ejecuta una sola vez.
- Es sencillo y fácil de implementar.

Los algoritmos voraces emplean un procedimiento de resolución de problemas para construir progresivamente soluciones candidatas, aproximando el óptimo global mediante la obtención de soluciones óptimas locales cada vez mejores en cada etapa. En general, los algoritmos voraces no pueden generar una solución óptima global , pero sí pueden producir buenas soluciones óptimas locales en un tiempo razonable y con menor esfuerzo computacional . GRASP (Feo y Resende, 1989) es un conocido algoritmo voraz iterativo basado en búsqueda local que implica varias iteraciones para construir soluciones voraces aleatorias y mejorarlas sucesivamente. El algoritmo consta de dos etapas principales: construcción y búsqueda local. La primera etapa consiste en construir una solución y la segunda en mejorarla para lograr la factibilidad. El algoritmo genera soluciones voraces aleatorias seleccionando nuevos elementos de un conjunto de soluciones candidatas ya construidas, basándose en el grado de mejora de la solución parcial en construcción, mediante una función de evaluación voraz. Los elementos seleccionados se incorporan a la solución parcial actual sin comprometer la factibilidad (si esta se ve comprometida, se selecciona un nuevo elemento), hasta obtener una solución factible completa , cuyo entorno se explora hasta encontrar un óptimo local mediante el procedimiento de búsqueda local. Se genera una lista, denominada lista de candidatos restringidos, con los mejores elementos, mediante la evaluación de los elementos con la función de evaluación voraz. Finalmente, se elige un elemento aleatoriamente de la lista restringida. Se crea una lista de candidatos para incorporarlo a la solución parcial, y una vez que el nuevo elemento se incluye en la solución parcial, el conjunto de candidatos de soluciones voraces se actualiza junto con la función de evaluación voraz.

¿Cómo funcionan los algoritmos voraces?

Los algoritmos voraces resuelven problemas siguiendo un enfoque simple, paso a paso:

1. Identificar el problema: El primer paso es comprender el problema y determinar si se puede resolver mediante un enfoque voraz. Esto suele implicar reconocer que el problema se puede descomponer en una serie de decisiones.
2. Tomar la decisión óptima: En cada paso del problema, el algoritmo toma la mejor decisión que parece óptima en ese momento. Esta decisión se basa en la propiedad de elección óptima, lo que significa seleccionar la opción que ofrece el beneficio más inmediato.
3. Resolver los subproblemas: Tras realizar una elección voraz, el problema se reduce a un subproblema más pequeño. El algoritmo repite entonces el proceso, realizando otra elección voraz para el problema más pequeño.
4. Combinar las soluciones: El algoritmo continúa tomando decisiones ávidas y resolviendo subproblemas hasta que se resuelve el problema completo. La solución al problema global es simplemente la combinación de todas las decisiones ávidas tomadas en cada paso.

Algoritmos voraces comunes

Los algoritmos ávidos más conocidos son:

- El problema de selección de actividades consiste en seleccionar el número máximo de actividades que no se solapen, dados sus horarios de inicio y finalización. El objetivo es maximizar el número de actividades a las que se puede asistir.
- El árbol de expansión mínima (MST) utiliza el algoritmo de Prim; no contiene ciclos y tiene el peso total mínimo posible en sus aristas. Este árbol se deriva de un grafo no dirigido conexo con pesos asignados.
- El algoritmo de Dijkstra para encontrar el camino más corto es un algoritmo de búsqueda que encuentra el camino más corto entre un vértice y otros vértices en un grafo ponderado.
- El problema del viajante consiste en encontrar la ruta más corta que visite cada lugar solo una vez y regrese al punto de partida.
- La codificación Huffman asigna un código más corto a los símbolos que aparecen con frecuencia y un código más largo a los que aparecen con menos frecuencia. Se utiliza para codificar datos de forma eficiente.

Tipos de algoritmos voraces

Algoritmos voraces fraccionarios

Estos algoritmos funcionan tomando la mejor decisión posible en cada paso basándose en fracciones. Se suelen utilizar en problemas donde los elementos se pueden dividir en partes más pequeñas.

Ejemplo: Problema de la mochila fraccionada, cuyo objetivo es maximizar el valor total de la mochila tomando fracciones de los artículos si es necesario.

Algoritmos puramente voraces

Estos algoritmos toman la mejor decisión posible en cada paso sin considerar las consecuencias futuras ni la posibilidad de revisar decisiones anteriores. Una vez tomada la decisión, es definitiva.

Ejemplo: Problema de selección de actividades, en donde las actividades se seleccionan en función de sus tiempos de finalización para maximizar el número de actividades que no se superponen.

Algoritmos voraces recursivos

Estos algoritmos utilizan la recursión para realizar una serie de elecciones voraces. El problema se divide en subproblemas, y el algoritmo realiza una elección voraz y luego resuelve recursivamente el subproblema.

Ejemplo: Algoritmo de Prim, que construye un árbol de expansión mínima tomando la decisión voraz de agregar la arista más pequeña en cada paso y continuando recursivamente este proceso.

Algoritmos de elección voraces

Estos algoritmos se basan en la propiedad de elección voraz, que garantiza que se puede alcanzar una solución óptima global tomando decisiones óptimas locales.

Ejemplo: Codificación Huffman, donde a los caracteres se les asignan códigos de longitud variable en función de sus frecuencias, con el objetivo de minimizar la longitud total del mensaje codificado.

Algoritmos voraces adaptativos

Estos algoritmos adaptan sus estrategias en función del estado actual del problema. Pueden reconsiderar decisiones anteriores si es necesario para mejorar la solución global.

Ejemplo: Coloreado de grafos mediante un algoritmo voraz; aquí el algoritmo asigna colores a los vértices de un grafo, ajustando potencialmente las elecciones a medida que se colorean nuevos vértices.

Algoritmos voraces constructivos

Estos algoritmos construyen la solución paso a paso, añadiendo elementos al conjunto de soluciones en función de un criterio específico.

Ejemplo: Algoritmo de Kruskal, que construye un árbol de expansión mínima añadiendo las aristas más cortas que no forman un ciclo.

Algoritmos voraces no adaptativos

Estos algoritmos toman decisiones definitivas que no se adaptan ni cambian una vez tomada la decisión.

Ejemplo: Algoritmo de Dijkstra – que encuentra el camino más corto desde un origen a todos los demás vértices de un grafo tomando decisiones localmente óptimas en cada paso.

Complejidad temporal de los algoritmos voraces

La complejidad temporal de los algoritmos voraces comunes es mayoritariamente O(nlog n) u O(n).

Algoritmo	Complejidad temporal	Complejidad espacial
Problema de selección de actividades	O(nlog(n))	O(1)
Codificación Huffman	O(nlog(n))	O(n)
Problema de la mochila fraccionaria	O(nlog(n))	O(1)
Problema con el cambio de monedas	O(n)	O(1)
Secuenciación de tareas con plazos de entrega	O(nlog(n))	O(n)

Aplicaciones de los algoritmos voraces

- **Enrutamiento de red**: Se utilizan algoritmos voraces como el algoritmo de Dijkstra para encontrar la ruta más corta en el enrutamiento de red, optimizando la entrega de paquetes de datos a través de Internet.
- **Planificación de tareas**: Se utiliza en problemas de secuenciación de trabajos para maximizar las ganancias o minimizar los plazos de entrega mediante la selección de trabajos en orden de prioridad.
- **Asignación de recursos**: Se aplica en escenarios como el problema de la mochila fraccionada para asignar recursos de manera eficiente en función de las relaciones valor-peso.
- **Compresión de datos**: La codificación Huffman utiliza un enfoque voraz para comprimir los datos de manera eficiente mediante la codificación de los caracteres de uso frecuente con códigos más cortos.
- **Cambio de monedas**: A menudo se utiliza un enfoque codicioso para proporcionar el número mínimo de monedas por una cantidad determinada de dinero,

especialmente cuando las denominaciones son estándar (como en la mayoría de las monedas).
- **Coloración voraz**: Se aplica en teoría de grafos para problemas como la coloración de grafos, donde el objetivo es minimizar el número de colores necesarios para colorear un grafo asegurando al mismo tiempo que ningún par de vértices adyacentes compartan el mismo color.
- **Heurística para el Problema del Viajante (TSP)**: Se utilizan heurísticas voraces para aproximar las soluciones al TSP seleccionando la ciudad no visitada más cercana en cada paso.

Ventajas de los algoritmos voraces

- **Sencillez**: Los algoritmos voraces son fáciles de entender e implementar porque siguen un enfoque directo para tomar la mejor decisión en cada paso.
- **Eficacia**: Suelen tener una complejidad temporal menor en comparación con otros algoritmos como la programación dinámica, lo que los hace más rápidos para ciertos problemas.
- **Decisiones óptimas locales**: Al tomar decisiones óptimas locales, los algoritmos voraces pueden llegar rápidamente a una solución sin necesidad de un retroceso complejo.
- **Funciona bien para ciertos problemas**: Los algoritmos voraces son particularmente efectivos para problemas que exhiben la propiedad de elección voraz y subestructura óptima, como el árbol de expansión mínima y la codificación Huffman.

Desventajas de los algoritmos voraces

4. **Puede que no siempre produzcan soluciones óptimas**: Los algoritmos voraces se centran en los beneficios inmediatos, lo que a veces puede conducir a soluciones subóptimas para problemas en los que se necesita una perspectiva global.
5. **Aplicabilidad limitada**: Los algoritmos voraces solo funcionan bien para problemas que satisfacen la propiedad de elección voraz y la subestructura óptima; de lo contrario, pueden fallar o dar resultados incorrectos.
6. **Sin posibilidad de rectificar**: Una vez tomada una decisión, no se puede deshacer, lo que puede conducir a resultados incorrectos si se toma la decisión equivocada al principio.
7. **Falta de flexibilidad**: Los algoritmos voraces son rígidos en su enfoque, y modificarlos para tener en cuenta diferentes restricciones del problema puede resultar complicado.

Preguntas

13. ¿Qué es un árbol?
14. ¿Qué es un árbol binario?

15. ¿Qué es un árbol completo?
16. ¿Qué es un árbol AVL?
17. ¿Qué es un grafo?

Ejercicios

- Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación infija.
    - x ^ y + z * w
    - x - (w / z) * y
    - a / (a - b) * (c + b)
    - m / (n - q) ^ p
    - a * (b + (x * y)) ^ c
    - z ^ (x + y + z) / w
    - (a + b) - c * (d + e ^ x)
    - a + b + c + d + e
    - (x + y + z) ^ e / a + (b * c)
    - ((m - n) * (p + q)) ^ r / s
- Escriba funciones para convertir expresiones escritas en notación infija a notación polaca prefija y postfija, respectivamente.
- Problema sobre Colas. Una institución educativa universitaria requiere para el departamento de Admisiones y Registro un control básico de las personas que atiende en el día por ventanilla. Las personas se atienden una a una a medida que van llegando, y se guarda de ellas el documento, el nombre y por defecto se establece que no ha sido atendida. A medida que llegan personas, se ubican en la fila y van saliendo de ésta una vez son atendidas.
    - Cree un menú de opciones para gestionar la atención de personas de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    - Registrar el ingreso a la fila de una persona
    - Mostrar la fila de personas a atender
    - Cuántas personas hay en espera
    - Atender personas
- Problema sobre Pilas. Una tienda vende dos tipos de productos (ratón -mouse- y teclados) que llegan en cajas y los organizan en estanterías; de cada producto se conoce su código, nombre y precio, y ambos tipos de productos se organizan de forma independiente de acuerdo con su tipo.
    - Cree un menú de opciones para gestionar el movimiento del inventario de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    - Acomode productos en las estanterías de acuerdo con su tipo registrando su ingreso al inventario
    - Liste los productos disponibles de acuerdo con el tipo seleccionado con todos sus datos e indique cuántos hay en dicha estantería
    - Muestre el producto de mayor valor de acuerdo a su tipo
    - Aliste productos para la venta. Los productos que se alistan para la venta salen del inventario; el usuario debe tener la posibilidad de indicar el tipo de producto a alistar y cuántos productos va a alistar

- Recorrer listas LSL, LSLC, LDL y LDLC de forma recursiva
- La *función de Ackermann*[69] se define como:
  $A(m,n) = n + 1,\ si\ m = 0$
  $A(m,n) = A(m-1,1),\ si\ n = 0$
  $A(m,n) = A(m-1, A(m,n-1)),\ en\ otro\ caso$
  Esta función toma dos números enteros positivos como argumentos y devuelve un único entero positivo. Escriba una función recursiva para calcular la función de Ackermann $A(M,N)\ para\ M,N \geq 0$
- El *algoritmo de Euclides* para el cálculo del *Máximo Común Divisor* (*MCD*) se define de la siguiente forma:
  $MCD(m,n) = m,\ si\ n = 0$
  $MCD(m,n) = MCD(n, m\ MOD\ n),\ si\ n > 0$
  Escriba una función recursiva para calcular $MCD(m,n)\ para\ m,n \geq 0$
- Invierta una palabra de forma recursiva
- Escriba funciones no recursivas para solucionar los problemas 6, 7 y 8
- Muestre el cuadrado de los primeros n números naturales
- Muestre la suma de los cuadrados de los primeros n números naturales
- Escriba un programa para simular el problema de *Las Torres de Hanoi*[70]. Resuelva de forma iterativa y recursiva. Este problema fue ideado por el matemático francés Edouard Lucas en el año de 1883 inspirado en una leyenda hindú. El problema consiste en resolver lo siguiente: se tienen tres torres a las que llamamos *origen*, *destino* y *auxiliar*; en la torre de origen se encuentran n discos todos de distintos tamaños, ordenados de mayor a menor tamaño desde la base de la torre; se deben pasar los discos a la torre de destino usando como apoyo la torre auxiliar con las siguientes condiciones: 1) solo puede pasarse un disco a la vez; 2) no puede quedar un disco de mayor tamaño sobre uno de menor tamaño. Después de muchas pruebas y de analizar este problema, se ha encontrado que el número de movimientos a efectuar es $2^n - 1$.

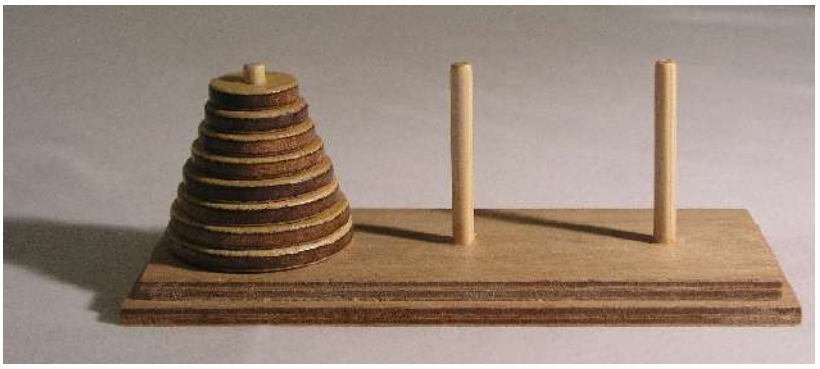


Figura 3.4. Las Torres de Hanoi como juego infantil[71]

[69] Esta función se debe al matemático alemán Wilhelm Ackermann y tiene interés en las ciencias de la computación. En Función de Ackermann - Wikipedia, la enciclopedia libre se habla más sobre esta función, así como en otras fuentes de literatura escrita y digital.
[70] Puede encontrar más detalles sobre el curioso problema de Las torres de Hanoi en distinta literatura, tanto de matemáticas como de computación. La siguiente fuente habla un poco acerca de la leyenda hindú y una solución usando Python
4.10. Las torres de Hanoi — Solución de problemas con algoritmos y estructuras de datos
[71] Imagen tomada de: Torres de Hanói - Wikipedia, la enciclopedia libre

Capítulo 16. Algoritmos heurísticos y metaheurísticos

Concepto

Problemas de optimización

Concepto

Problemas de programación lineal entera

Concepto

Herramientas de optimización

Concepto

Solver

Concepto

OR-Tools

Concepto

Algoritmos heurísticos y metaheurísticos o aproximados

Concepto

Métricas de eficacia de algoritmos aproximados

Concepto

Preguntas

Ejercicios

Fuentes y referencias adicionales

El presente documento está disponible en Internet en la url:
https://github.com/innovasistemas/textos/blob/master/fundamentos-programacion-v1-2026.pdf

Textos impresos

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

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

CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996.

AHO, ALFRED V., HOPCROFT, JOHN E., ULLMAN, JEFFREY D. Estructuras de datos y algoritmos. Addison-Wesley Iberoamericana. Wilmington, Delaware, E.U.A., 1988.

FLÓREZ R., ROBERTO. Algoritmos y Estructuras de Datos. Universidad de Antioquia. Medellín, 1996.

BRASSARD, G., BRATLEY, T. Fundamentos de Algoritmia. Prentice Hall. Madrid, 1996.

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

Aho, A. (1995). Foundations of computer science. Computer science press.

Cormen, T.; Lieserson, C.; Rivest, R. (2009). Introduction to algorithms. Ed. the mit press

Flórez, R. (2005) Algoritmos, estructuras de datos y programación orientada a objetos. Bogotá, ecoe ediciones.

Skiena, S. (2008). The algorithm design manual. Springer; 2nd edition.

Weiss, M. (1992). Estructuras de datos y algoritmos. Addison – Wesley Iberoamericana. wilmington, delaware, e.u.a.

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. (En pdf).

Leithold, Louis. El Cálculo con Geometría Analítica. Sexta edición. Harla. México D.F., 1992.

Páginas web

Árboles Balanceados AVL. Universidad Don Bosco, Facultad de Ingeniería, Escuela de Computación, Asignatura Programación con Estructuras de Datos
En Internet (en pdf)
Tema: “Árboles Balanceados AVL”

Teoría de grafos. Universidad de Pamplona
En Internet (en pdf)
Teoría de grafos

CASTILLO SUAZO, ROMMEL. Programación en: PSInt. Original para LPP. Implementado por: CARO, ALEJANDRO. Documento PDF.
En Internet (recuperado: 08/10/2023)
Manual PSeInt - Programación en

Árboles Balanceados AVL. Universidad Don Bosco, Facultad de Ingeniería, Escuela de Computación, Asignatura Programación con Estructuras de Datos
En Internet (en pdf)
Tema: “Árboles Balanceados AVL”

Generación de números aleatorios
Tema 1 Generación de números aleatorios

Generadores congruenciales lineales. Técnicas de Simulación y Remuestreo
2.1 Generadores congruenciales lineales | Técnicas de Simulación y Remuestreo

Ciencias de la computación
Ciencias de la computación | Khan Academy

Notación asintótica
Notación asintótica (artículo) | Algoritmos | Khan Academy

Clases de complejidad P y NP
Clases de complejidad P y NP - Wikipedia, la enciclopedia libre

Búsqueda de fuerza bruta
Búsqueda de fuerza bruta - Wikipedia, la enciclopedia libre

Ataque de fuerza bruta
Ataque de fuerza bruta - Wikipedia, la enciclopedia libre

Algoritmo de divide y vencerás

https://en-m-wikipedia-org.translate.goog/wiki/Divide-and-conquer_algorithm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge#:~:text=A%20divide%2Dand%2Dconquer%20algorithm,solution%20to%20the%20original%20problem.

Introducción al Algoritmo de Ordenación Merge Sort
Introducción al Algoritmo de Ordenación Merge Sort - Blog de SW Hosting

Algoritmo divide y vencerás
Algoritmo divide y vencerás - Wikipedia, la enciclopedia libre

Análisis de algoritmos recursivos.
Tema 06 "Análisis de algoritmos recursivos"

¿Qué es la regresión logística?
¿Qué es la regresión logística? - Explicación del modelo de regresión logística - AWS

¿Qué es un árbol de decisión?
¿Qué es un árbol de decisión? | IBM.

What is a Greedy Algorithm? Examples of Greedy Algorithms
What is a Greedy Algorithm? Examples of Greedy Algorithms

Greedy Algorithms: Examples, Types, Complexity
Greedy Algorithms: Examples, Types, Complexity

El método de la inducción matemática
El método de la inducción matemática.

La integral definida como el límite de una suma de Riemann
La integral definida como el límite de una suma de Riemann (artículo) | Khan Academy

Apéndice

A continuación se presentan aplicaciones de programación para solucionar algunos problemas matemáticos presentados en pseudocódigo y en distintos lenguajes de programación como Python, Java, PHP, C/C++, Javascript, C# y PSeInt entre otros.

Apéndice A. Aplicaciones con aritmética y álgebra

Este apéndice contiene desarrollos relacionados con los temas vistos en la Unidad 0.

Raíces de la ecuación cuadrática

Programa Python:

```
import math
print("Raíces de la ecuación cuadrática a * x ** 2 + b*x + c = 0")
a = float(input("a: "))
b = float(input("b: "))
c = float(input("c: "))
if a != 0:
    discrim = b ** 2 - 4 * a * c
    if discrim > 0:
        x1 = (-b + math.sqrt(discrim)) / (2 * a)
        x2 = (-b - math.sqrt(discrim)) / (2 * a)
        print(f"x1: {x1}")
        print(f"x2: {x2}")
    elif discrim == 0:
        x1 = -b / (2 * a)
        print(f"Raíces iguales: x: {x1}")
    else:
        print("Raíces imaginarias")
else:
    print("No es una ecuación cuadrática")
```

Cálculo del factorial

Programa PHP:

```
public function factorial($n)
{
    $f = 1;
    for ($i = 1; $i <= $n; $i++) {
        $f *= $i;
    }
    return $f;
```

```
}
```

Determinar números primos

Programa PHP:

```
public function prime($n)
{
    $i = 2;
    $sw = TRUE;
    while ($i <= $n / 2 && $sw) {
        if ($n % $i == 0) {
            $sw = FALSE;
        } else {
            $i++;
        }
    }
    return $sw;
}
```

Encontrar números perfectos

Programa PHP:

```
public function perfect($n)
{
    $sum = 0;
    for ($i = 1; $i <= $n / 2; $i++) {
        if ($n % $i == 0) {
            $sum += $i;
        }
    }
    $sw = $sum == $n ? TRUE : FALSE;
    return $sw;
}
```

Determinar la paridad de un número

Programa PHP:

```
public function even($n)
{
    $k = 0;
    $sw = FALSE;
    while ($k <= $n && !$sw) {
```

```
        if (2 * $k == $n) {
            $sw = TRUE;
        } else {
            $k++;
        }
    }
    return $sw;
}
```

Apéndice B. Aplicaciones con sistemas numéricos

Los ejemplos a continuación son métodos de una clase que a su vez dispone de la propiedad `digitHex`, la cual es un arreglo asociativo, tal y como se muestra a continuación.

Programa PHP:

```
private array $digitHex;

public function __construct()
{
    $this->digitHex = [
        'A' => '10',
        'B' => '11',
        'C' => '12',
        'D' => '13',
        'E' => '14',
        'F' => '15'
    ];
}
```

Convertir un número en base 10 a una base cualquiera

Programa PHP:

```
public function base10ToN($number, $base)
{
    $div = $number;
    $stringNumber = $number == 0 ? "0" : "";
    while ($div > 0) {
        $res = $div % $base;
        $res = $res > 9 ? array_search($res, $this->digitHex) : $res;
        $div = (int)($div / $base);
        $stringNumber = $res . $stringNumber;
    }
    return $stringNumber;
}
```

Convertir un número en una base cualquiera a base 10

Programa PHP:

```
public function baseNTo10($stringNumber, $base)
{
    $sum = 0;
    $n = strlen($stringNumber);
    for ($i = 0; $i < $n; $i++) {
        $digit = strtoupper(substr($stringNumber, $i, 1));
        $digit = is_numeric($digit) ? $digit : $this->digitHex[$digit];
        $sum += $digit * pow($base, $n - $i - 1);
    }
    return $sum;
}
```

Apéndice C. Aplicaciones con conjuntos

Este apéndice contiene desarrollos relacionados con los temas vistos en la Unidad 2.

Operaciones entre conjuntos

Programa Javascript:

```
class Sets
{
    union(arrayA, arrayB)
    {
        let arrayResult = arrayA.slice();
        arrayB.forEach ((element) => {
            if (this.findElement(arrayA, element) === -1) {
                arrayResult[arrayResult.length] = element;
            }
        });
        return arrayResult;
    }

    intersection(arrayA, arrayB)
    {
        let arrayResult = [];
        arrayA.forEach ((element) => {
            if (this.findElement(arrayB, element) > -1) {
                arrayResult[arrayResult.length] = element;
            }
```

```
        });
        return arrayResult;
    }

    minus(arrayA, arrayB)
    {
        let arrayResult = [];
        arrayA.forEach ((element) => {
            if (this.findElement(arrayB, element) === -1) {
                arrayResult[arrayResult.length] = element;
            }
        });
        return arrayResult;
    }

    cartesianProduct(arrayA, arrayB)
    {
        let i = 0;
        let arrayResult = [];
        arrayA.forEach ((elementA) => {
            arrayB.forEach ((elementB) => {
                arrayResult[i] = `(${elementA}, ${elementB})`;
                i++;
            });
        });
        return arrayResult;
    }

    findElement(array, element, property = '')
    {
        let i = 0;
        let pos = -1;
        let elemArray;
        while (i < array.length && pos == -1) {
            elemArray = property === '' ? array[i] : array[i][property];
            if (elemArray === element) {
                pos = i;
            } else {
                i++;
            }
        }
        return pos;
    }
}
```

Apéndice D. Aplicaciones con compuertas lógicas

Este apéndice contiene desarrollos relacionados con los temas vistos en la Unidad 3

Apéndice E. Aplicaciones con mapas de Karnaugh

Este apéndice contiene desarrollos relacionados con los temas vistos en la Unidad 4.