Lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Lógica de programación / Camilo Augusto Cardona Patiño, /
Bogotá D.C., Fundación Universitaria del Área Andina. 2017

978-958-5455-95-5

Catalogación en la fuente Fundación Universitaria del Área Andina (Bogotá).

Edición:
Fondo editorial Areandino
Fundación Universitaria del Área Andina
Calle 71 11-14, Bogotá D.C., Colombia
Tel.: (57-1) 7 42 19 64 ext. 1228
E-mail: publicaciones@areandina.edu.co
http://www.areandina.edu.co

Primera edición: noviembre de 2017

Corrección de estilo, diagramación y edición: Dirección Nacional de Operaciones virtuales
Diseño y compilación electrónica: Dirección Nacional de Investigación

Hecho en Colombia
Made in Colombia

Lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

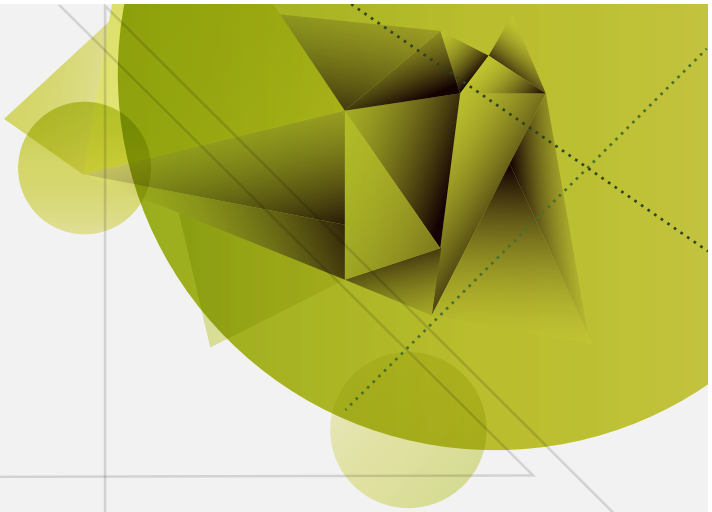
Índice

Índice

1

Unidad 1

Sistemas numéricos y conversiones entre sistemas



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Las posibilidades de manipulación de la información dependen de la manera de representarla. Dependiendo del contexto, un sistema de numeración puede representar ventajas sobre otro, tal es el caso del sistema binario, que está perfectamente adecuado para los cálculos realizados por el microprocesador, el cual puede procesar dos tipos de estados (Encendido/Apagado). En oposición, el sistema decimal es el sistema con el que los humanos estamos más familiarizados, por esta razón es importante aprender a identificar y manipular los sistemas de numeración así como también adquirir la capacidad para convertir entre sistemas.

El sistema binario es el sistema más sencillo, ya que su base solo tiene dos símbolos, sin embargo para las personas resulta un poco engorroso trabajarlo ya que para representar magnitudes relativamente grandes, es necesario emplear una gran cantidad de dígitos (bits). Por esta razón surgieron los sistemas Octal y Hexadecimal, de manera que podamos aprovechar las ventajas ofrecidas por el sistema binario, pero empleando una menor cantidad de dígitos y además poder convertir fácil y rápidamente entre sistemas.

U1

Metodología

Se invita al estudiante a leer completamente la guía, de manera que logre alcanzar gran solidez en cuanto a fundamentación teórica. De igual forma, se le recomienda analizar los ejemplos planteados, siguiendo paso a paso su desarrollo, a fin de lograr claridad y comprensión en el procedimiento efectuado; estar atento de la realización de las actividades; participar en los foros y formular todas las dudas que pueda tener en relación al tema tratado.

U1

Desarrollo temático

Sistemas numéricos y representación de la información

Un sistema numérico es un conjunto de reglas y de símbolos que se relacionan para expresar la relación existente entre la cantidad y la unidad. Debido a que un número es un símbolo, podemos encontrar diferentes representaciones para expresar una cantidad.

Podemos representar un sistema numérico como N = S + R, donde:

N: Sistema de Numeración.

S: Símbolos del sistema.

R: Reglas de generación que establecen la validez de los números en el sistema.

Cada sistema tiene reglas diferentes, únicamente tienen una regla en común y es que cada sistema está limitado al uso único de los símbolos permitidos en él.

Los sistemas de numeración pueden ser clasificados en 2 tipos:

Sistemas numéricos posicionales

En los sistemas posicionales, cada cifra de un valor numérico contribuye al valor final dependiendo de su valor y de la posición que ocupa dentro del valor relativo. En estos sistemas tenemos tantos símbolos como la base del sistema. Los números mayores a la base se representan por medio de varias cifras. El valor final será la suma de una serie de potencias de la base del sistema (B):

$$N = (Sn * Bn) + (Sn-1 * Bn-1) + \ldots + (S1 * B1) + (S0 * B0)$$

(Fórmula 1)

Donde Si son las distintas cifras "Símbolos" del valor numérico e I es su posición.

Sistemas numéricos no posicionales

En los sistemas no posicionales, contrario a los posicionales, la contribución de cada cifra no

depende del lugar que ocupa, un ejemplo serían los números romanos:

La combinación XXI equivale a 21. Podemos ver como la cifra X aparece 2 veces y siempre tiene el mismo valor: 10 unidades, independiente de su posición.

Una de las mayores complicaciones que presenta este tipo de sistemas es que para escribir valores numéricos grandes se requieren muchos símbolos, y además resulta difícil efectuar operaciones aritméticas con ellos, cosa que no sucede con los posicionales.

Los sistemas de numeración que veremos son todos sistemas posicionales.

Cuando se expresa un valor numérico, es necesario incluir un subíndice al final indicando la base en la que se expresa dicho valor, salvo que por el contexto quede suficientemente claro:

47 o 47_{10} = Base Decimal (cuando no se indica la base, se asume que es decimal)

110110_2 = Base Binaria

57_8 = Base Octal

$2F_{16}$ = Base Hexadecimal

Sistema Decimal

Es el sistema más usado, emplea 10 símbolos para representar cualquier cantidad posible.

Base: 10

Símbolos: 1-2-3-4-5-6-7-8-9-0

Para este caso B=10

$$N = (Sn * Bn) + (Sn-1 * Bn-1) + \ldots + (S1 * B1) + (S0 * B0)$$

La posición de cada dígito indica la magnitud de la cantidad que representa y al cual es posible fijarle un peso, el peso son las potencias de la base "10" que comienza con 100 = 1 y aumentan de derecha a izquierda.

Peso

		1000	100	10	1
10^n	...	10^3	10^2	10^1	10^0
n	...	3	2	1	0

Potencia

Figura 1.
Fuente: Propia.

Ejemplo: el número decimal 1253 puede obtenerse como la suma de:

N = (Sn * Bn) + (Sn-1 * Bn-1) + (S1 * B1) + (S0 * B0)

N = (1 * 103) + (2 * 102) + (5 * 101) + (3 * 100)

N = (1 * 1000) + (2 * 100) + (5 * 10) + (3 * 1)

N = 1000 + 200 + 50 + 3

N = 1253

O así:

S		B			S*B
1	X	1000	=	+	1000
2	X	100	=		200
5	X	10	=		50
3	X	1	=		3
					1253

En el caso de los números fraccionarios, los pesos son potencias negativas de la base "10" que comienza con 10-1 = 0.1 y aumentan de izquierda a derecha.

Figura 2.
Fuente: Propia.

Sistema Binario

Es otra manera de representar magnitudes, emplea dos símbolos para representar cualquier cantidad posible.

Base: 2

Símbolos: 1-0

Ya que solo utiliza dos símbolos, requiere de muchas cifras para representar una cantidad grande, lo que resulta engorroso para las personas. Ssin embargo para los computadores esto no es un problema, resulta ser más una ventaja, pues los circuitos electrónicos tiene fundamentalmente dos estados (Encendido/Apagado).

La posición de cada dígito indica la magnitud de la cantidad que representa y al cual es posible fijarle un peso, el peso son las potencias de la base "2" que comienza con 20 = 1 y aumentan de derecha a izquierda. En el caso de los números fraccionarios, los pesos son potencias negativas de la base "2" que comienza con 2-1 = 0.5 y aumentan de izquierda a derecha.

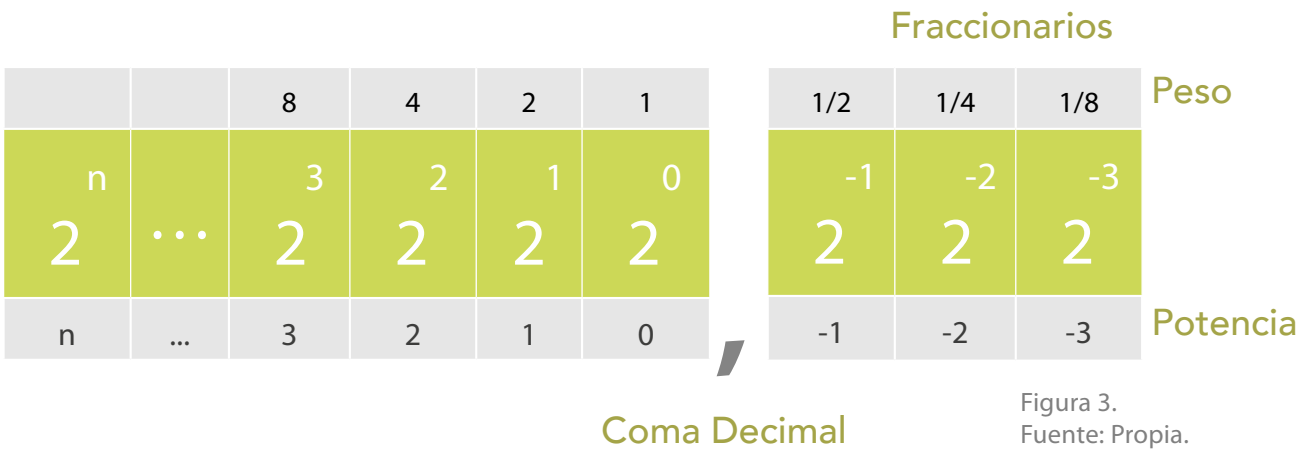


Figura 3.
Fuente: Propia.

En el sistema binario, cada dígito recibe el nombre de bit, una agrupación de 4 bits se denomina *nibble,* una agrupación de 8 bits se denomina *byte* y una agrupación de 16 bits, se denomina *palabra.*

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binario	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Tabla 1.
Fuente: Propia.

Conversión de Decimal a Binario

Se divide sucesivamente el número decimal entre 2 y cada cociente obtenido se divide nuevamente entre 2 hasta que se obtiene un cociente cuya parte entera es cero. Los restos generados de cada división forman el número binario, el primer resto es el bit menos significativo y el último resto es el bit más significativo.

Ejemplo: convertir 1253 a binario.

1253	/	2	=	626	Resto	1
626	/	2	=	313	Resto	0
313	/	2	=	156	Resto	1
156	/	2	=	78	Resto	0
78	/	2	=	39	Resto	0

Ejemplo: convertir 1253 a binario.

Imagen 1.
Fuente: Propia.

Conversión de Decimal a Binario.

Para realizar la conversión acudimos nuevamente a la fórmula 1 así:

Ahora la base B = 2

N = (Sn * Bn) + (Sn-1 * Bn-1) + … + (S1 * B1) + (S0 * B0)

Ejemplo: Convertir el número 100111001012 a decimal

$N = (1*2^{10})+(0*2^{9})+(0*2^{8})+(1*2^{7})+(1*2^{6})+(1*2^{5})+(0*2^{4})+(0*2^{3})+(1*2^{2})+(0*2^{1})+(1*2^{0})$

N = (1*1024)+0+0+(1*128)+(1*64)+(1*32)+0+0+(1*4)+0+(1*1)

N = 1024+128+64+32+4+1

$N = 1253_{10}$

Sistema Octal

Este sistema tiene una base de 8 símbolos, existe una gran facilidad para convertir entre el sistema binario y el sistema octal y permite expresar los números binarios en un formato más compacto, pues cada dígito del sistema octal equivale a tres dígitos del sistema binario.

Base: 8

Símbolos: 1-2-3-4-5-6-7-0

A continuación veremos las equivalencias de los primeros 8 números octales:

Octal	Binario	Decimal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Tabla 2.
Fuente: Propia.

Conversión de Octal a Binario

Para convertir un número en sistema octal a su equivalente en sistema binario basta con reemplazar cada una de las cifras.

Ejemplo: convertir a binario el numero 23458:

Para realizar la conversión, recurrimos a la tabla 2:

2 = 010

3 = 011

4 = 100

5 = 101

Quedando así:

$23458 = 010\ 011\ 100\ 101 = 10011100101_2$

Conversión de Binario a Octal

Se realiza el proceso inverso agrupando de a tres los binarios comenzando de derecha a izquierda:

Ejemplo: convertir a Octal el número 10011100101_2:

Para realizar la conversión, recurrimos a la tabla 2:

10011100101 = **01**0 011 100 101 // En caso de ser necesario agregamos cero a la izquierda para completar la terna.

Tenemos:

010 = 2

011 = 3

100 = 4

101 = 5

Quedando así:

100111001012 = 2 3 4 5 = 23458

Conversión Octal a Decimal

Para realizar la conversión acudimos nuevamente a la fórmula 1 así:

Ahora la base B = 8

$N = (Sn * Bn) + (Sn-1 * Bn-1) + \ldots + (S1 * B1) + (S0 * B0)$

Ejemplo: convertir el número 23458 a decimal.

$N = (2 * 8^3) + (3 * 8^2) + (4 * 8^1) + (5 * 8^0)$

$N = (2 * 512) + (3 * 64) + (4 * 8) + (5 * 1)$

$N = (1024) + (192) + (32) + (5)$

$N = 1253$

Conversión de Decimal a Octal.

Se divide sucesivamente el número decimal entre 8 y cada cociente obtenido se divide nuevamente entre 8, hasta que se obtiene un cociente cuya parte entera es cero. Los restos generados de cada división forman el número octal, el primer resto es el dígito menos significativo y el último resto es el digito más significativo.

Ejemplo: convertir 1253 a Octal.

Imagen 2.
Fuente: Propia.

Sistema Hexadecimal

De la misma manera que el sistema octal, el sistema hexadecimal ofrece una manera de expresar de forma más compacta los números binarios.

Base: 16

Símbolos: 0-1-2-3-4-5-6-7-8-9-A-B-C-D-E-F

A continuación veremos las equivalencias de los primeros 16 números Hexadecimales:

Hexa	Binario	Octal	Decimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
A	1010	12	10
B	1011	13	11
C	1100	14	12
D	1101	15	13
E	1110	16	14
F	1111	17	15

Tabla 3.
Fuente: Propia.

Conversión de Hexadecimal a Binario

Para convertir un número en sistema hexadecimal a su equivalente en sistema binario basta con reemplazar cada una de las cifras.

Ejemplo: convertir a binario el número $4E5_{16}$:

Para realizar la conversión, recurrimos a la tabla 3:

4 = 0100

E = 1110

5 = 0101

Quedando así:

$4E5_{16} = 0100\ 1110\ 0101 = 10011100101_2$

Conversión de Binario a Hexadecimal

Se realiza el proceso inverso agrupando de a cuatro los binarios comenzando de derecha a izquierda:

Ejemplo: convertir a Hexadecimal el número 100111001012:

Para realizar la conversión, recurrimos a la tabla 3:

10011100101 = **01**00 1110 0101 //En caso de ser necesario agregamos cero a la izquierda para completar los cuartetos.

Tenemos:

0100 = 4

1110 = E

0101 = 5

Quedando así:

$10011100101_2 = 4\ E\ 5 = 4E5_{16}$

Conversión Hexadecimal a Decimal

Para realizar la conversión acudimos nuevamente a la fórmula 1 así:

Ahora la base B = 16

$N = (Sn * Bn) + (Sn-1 * Bn-1) + \ldots + (S1 * B1) + (S0 * B0)$

Ejemplo: convertir el número 4E516 a decimal.

$N = (4 * 16^2) + (E * 16^1) + (5 * 16^0)$

$N = (4 * 256) + (14 * 16) + (5 * 1)$

$N = (1024) + (224) + (5)$

$N = 1253$

Conversión de Decimal a Hexadecimal

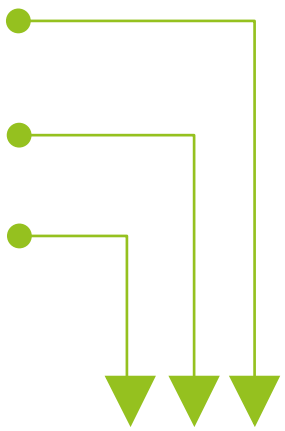
Se divide sucesivamente el número decimal entre 16 y cada cociente obtenido se divide nuevamente entre 16, hasta que se obtiene un cociente cuya parte entera es cero. Los restos generados de cada división forman el número Hexadecimal, el primer resto es el dígito menos significativo y el último resto es el dígito más significativo.

Ejemplo: convertir 1253 a Hexadecimal.

1253 / 16 = 78 Resto 5

78 / 16 = 4 Resto 14

4 / 16 = 0 Resto 4



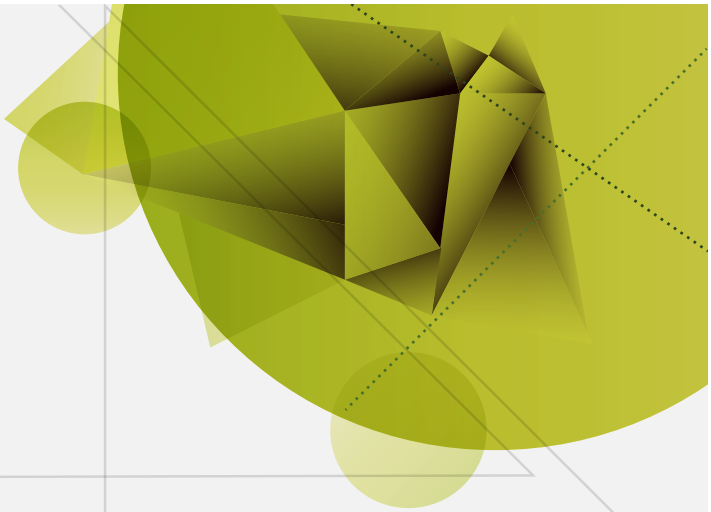
El hexadecimal resultante sería: **$4\ E\ 5_{16}$**

Imagen 3.
Fuente: Propia.

1

Unidad 1

Evolución del hardware y esquema del computador



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

En la presente unidad veremos con más detalle la evolución del hardware desde sus inicios en la antigüedad hasta nuestros días; cómo se fueron logrando avances tecnológicos verbo redundante revolucionarios que marcaron hitos en su época. También revisaremos cada una de las partes que conforman la estructura de un computador actual.

U1

Metodología

Se invita al estudiante a leer completamente la guía, de manera que logre alcanzar gran solidez en cuanto a evolución del hardware y estructura fundamental del computador. Se le recomienda revisar los contenidos adicionales a fin de lograr claridad y comprensión, estar atento de la realización de las actividades, participar en los foros y formular todas las dudas que pueda tener en relación al tema tratado.

U1 Desarrollo temático

Reseña histórica del desarrollo de Hardware de computador

La curiosidad e ingenio del hombre lo han llevado a entender y aprovechar el mundo en el que vive: la creación de la lectura, los números, el desarrollo de grandes obras de ingeniería, la declaración y demostración de hipótesis; son grandes logros de la humanidad que han ido acrecentándose a lo largo de la historia. Sin embargo, en la actualidad, la producción de conocimiento se ha incrementado exponencialmente gracias a las tecnologías de la comunicación.

Pues bien, vamos a revisar cómo ha sido la evolución del hardware de computador y su esquema estructural hoy en día.

Dispositivos de la antigüedad

Aunque algunos historiadores creen que antes del año 500 a.C. ya existían mecanismos básicos para el procesamiento de la información, partiremos de uno de los más conocidos, el ábaco. Se cree que tuvo su origen en la antigua Mesopotamia, pero fue ampliamente utilizado en Europa y Asia, permitía llevar cuentas básicas de aritmética. El ábaco consiste en un marco con rieles o barras sobre las que se deslizan esferas.

Durante esta época temprana del desarrollo de nuestra sociedad, surgieron paulatinamente algunos dispositivos más complejos, como el planisferio, el mecanismo de Anticitera, el astrolabio, entre otros.

Figura 1. Ábaco.
Fuente: antecedenteshistoricosdelcalculocecyt.blogspot.com

Primeras calculadoras

En esta categoría están la regla de cálculo, el ábaco Neperiano, entre los más representativos tenemos la Pascalina, inventada en el año 1642; una máquina que funciona con base en engranajes y permite llevar cuentas. En su interior, se disponían unas ruedas dentadas conectadas entre sí, formando una cadena de transmisión, de modo que, cuando una rueda giraba completamente sobre su eje, hacía avanzar un grado a la siguiente.

Las ruedas representaban el «sistema decimal de numeración». Cada rueda constaba de diez pasos, para lo cual estaba convenientemente marcada con números del 9 al 0. El número total de ruedas era ocho (seis ruedas para representar los números enteros y dos ruedas más, en el extremo izquierdo, para los decimales). Con esta disposición «se podían obtener números entre 0'01 y 999.999'99».

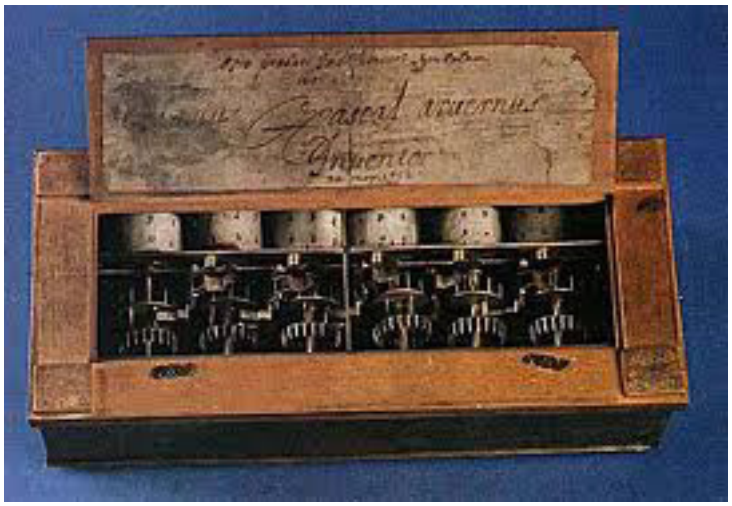


Figura 2. Pascalina.
Fuente: timerime.com

Tarjeta perforada

En 1801, Joseph Marie Jacquard desarrolló un telar en el que el patrón que era tejido era controlado por tarjetas perforadas. La serie de tarjetas podría ser cambiada sin cambiar el diseño mecánico del telar. Esto era un hito en programación.

En 1833, Charles Babbage avanzó, desde desarrollar su máquina diferencial a desarrollar un diseño más completo; la máquina analítica, que, para su programación, tomaría prestada directamente las tarjetas perforadas del telar Jacquar.

Tarjeta perforada

En 1801, Joseph Marie Jacquard desarrolló un telar en el que el patrón que era tejido era controlado por tarjetas perforadas. La serie de tarjetas podría ser cambiada sin cambiar el diseño mecánico del telar. Esto era un hito en programación.

En 1833, Charles Babbage avanzó, desde desarrollar su máquina diferencial a desarrollar un diseño más completo; la máquina analítica, que, para su programación, tomaría prestada directamente las tarjetas perforadas del telar Jacquar.

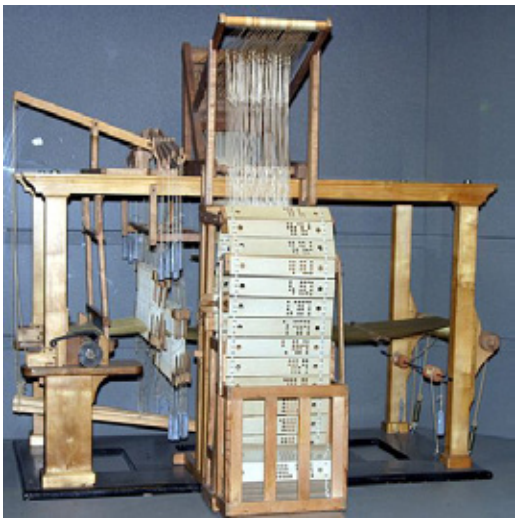


Figura 3. Telar con patrón de tarjeta perforada.
Fuente: es.wiki2.org

Computador análogo

Antes de la Segunda Guerra Mundial, las computadoras analógicas mecánicas y eléctricas eran consideradas el "estado del arte", y muchos pensaban que eran el futuro de la computación.

Las computadoras analógicas toman ventaja de las fuertes similitudes entre las matemáticas de propiedades de pequeña escala -- la posición y el movimiento de ruedas o el

voltaje y la corriente de componentes electrónicos -- y las matemáticas de otros fenómenos físicos, como: trayectorias balísticas, inercia, resonancia, transferencia de energía, momento, etc.

Modelar los fenómenos físicos con las propiedades eléctricas rinde una gran ventaja sobre usar modelos físicos: los componentes eléctricos son más pequeños y baratos; son más fácilmente construidos y ejercitados.

Aunque de otra manera similar, los fenómenos eléctricos pueden hacer que ocurran en marcos de tiempo convenientemente cortos.

Estos sistemas trabajan creando análogos eléctricos de otros sistemas, permitiendo a los usuarios predecir el comportamiento de los sistemas de interés, al observar los análogos eléctricos. La más útil de las analogías fue la manera en que el comportamiento en pequeña escala podía ser representado con ecuaciones diferenciales e integrales y, por lo tanto, podía ser usado para solucionar esas ecuaciones.

A diferencia de las computadoras digitales modernas, las computadoras analógicas no eran muy flexibles, y necesitaban ser reconfiguradas (es decir reprogramadas) manualmente para cambiar de un problema a otro. Las computadoras analógicas tenían una ventaja sobre los primeros computadores digitales en que podían ser usadas para solucionar problemas complejos usando comportamientos análogos, mientras que las primeras tentativas con computadoras digitales fueron muy limitadas. Pero a medida que las computadoras digitales han venido siendo más rápidas y usan memorias más grandes (ejemplo, RAM o almacenamiento interno), han desplazado casi completamente a las computadoras analógicas, y la programación de computadores o codificación, ha surgido como otra profesión humana.

El arte de la computación analógica alcanzó su cenit con el Analizador diferencial, inventado en 1876 por James Thomson y construido por H. W. Nieman y Vannevar Bush en el MIT comenzando en 1927. Fueron construidos menos de una docena de estos dispositivos; el más poderoso fue construido en la Escuela Moore de Ingeniería Eléctrica en la Universidad de Pensilvania, donde también fue construido el ENIAC. Las computadoras electrónicas digitales, como el ENIAC, significaron el fin para la mayoría de las máquinas analógicas de computación, pero las computadoras analógicas híbridas, controladas por electrónica digital, permanecieron en uso en los años 1950 y 1960, y siguieron más adelante en aplicaciones especializadas.

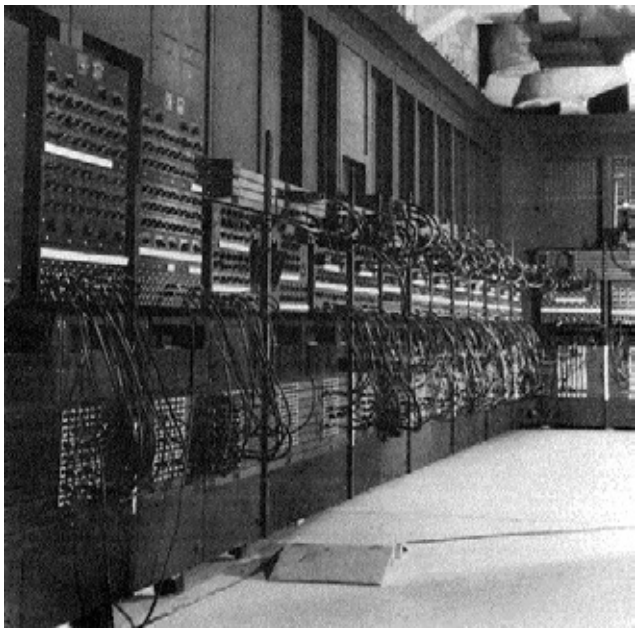


Figura 4. ENIAC.
Fuente: www.taringa.net

Computador digital

La era del computador moderno comenzó con un explosivo desarrollo antes y durante la Segunda Guerra Mundial, a medida que los circuitos electrónicos, los relés, los condensadores, y los tubos de vacío reemplazaron los equivalentes mecánicos y los cálculos digitales reemplazaron los cálculos análogos. Las máquinas como el Atanasoff–Berry Computer, Z3, Colossus y el ENIAC fueron construidas a mano usando circuitos que contenían relés o válvulas (tubos de vacío), y a menudo usaron tarjetas perforadas o cintas perforadas para la entrada y como el medio de almacenamiento principal (no volátil).

En esta era, un número de diferentes máquinas fueron producidas con capacidades que constantemente avanzaban. Al principio de este período, no existió nada que se asemejara remotamente a una computadora moderna, excepto en los planos perdidos por largo tiempo de Charles Babbage y las visiones matemáticas de Alan Turing y otros. Al final de la era, habían sido construidos dispositivos como el EDSAC, considerados universalmente como computadores digitales. Definir un solo punto en la serie, como la "primera computadora", pierde muchos sutiles detalles.

El texto escrito por Alan Turing en 1936, probó ser enormemente influyente en la computación y ciencias de la computación de dos maneras. Su principal propósito era probar que había problemas (nombrados el problema de la parada) que no podían ser solucionados por ningún proceso secuencial. Al hacer eso, Turing proporcionó una definición de una computadora universal, una construcción que vino a ser llamada máquina de Turing, un dispositivo puramente teórico que formaliza el concepto de ejecución de algoritmo, reemplazando el engorroso lenguaje universal basado en aritmética de Kurt Godel. Excepto por las limitaciones impuestas por sus almacenamientos de memoria finitos, se dice que las computadoras modernas son Turing completo, que es como decir que tienen la capacidad de ejecución de algoritmo equivalente a una máquina universal de Turing. Este tipo limitado de la cualidad de Turing completo es algunas veces visto como la capacidad umbral que separa las computadoras de uso general de sus precursores de propósito específico.

Para que una máquina de computación sea una computadora de propósito general práctica, debe haber algún mecanismo de lectura/escritura conveniente, como por ejemplo la cinta perforada. Para la completa versatilidad, la arquitectura de von Neumann usa la misma memoria para almacenar tanto los programas como los datos; virtualmente todas las computadoras contemporáneas usan esta arquitectura (o alguna variante). Mientras que es teóricamente posible implementar una computadora completa mecánicamente (como demostró el diseño de Babbage), la electrónica hizo posible la velocidad y más adelante la miniaturización que caracterizan las computadoras modernas.

En la era de la Segunda Guerra Mundial había tres corrientes paralelas en el desarrollo de la computadora, y dos fueron ignoradas en gran parte o deliberadamente mantenidas en secreto. La primera fue el trabajo alemán de Konrad Zuse. La segunda fue el desarrollo secreto de la computadora Colossus en el Reino Unido. Ninguna de estas tuvo mucha influencia en los varios proyec-

tos de computación en los Estados Unidos. La tercera corriente de desarrollo de la computadora, el ENIAC y el EDVAC de Eckert y Mauchly, fue publicada extensamente.

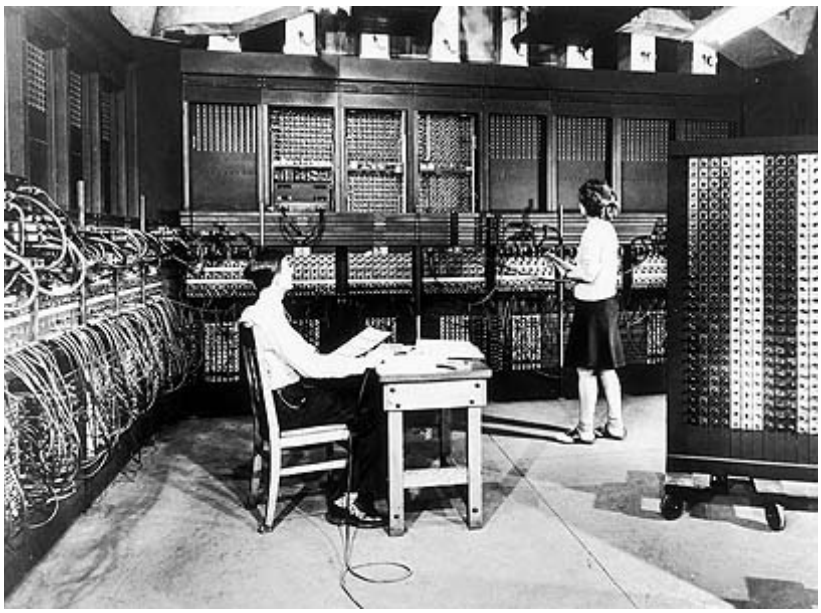


Figura 5. Colosuss.
Fuente: samacacagime.wordpress.com

Generaciones

Primera generación (1946-1958)

En esta época las computadoras funcionaban con válvulas, usaban tarjetas perforadas para entrar los datos y los programas, utilizaban cilindros magnéticos para almacenar información e instrucciones internas y se utilizaban exclusivamente en el ámbito científico o militar. La programación implicaba la modificación directa de los cartuchos y eran sumamente grandes, utilizaban gran cantidad de electricidad, generaban gran cantidad de calor y eran sumamente lentas.

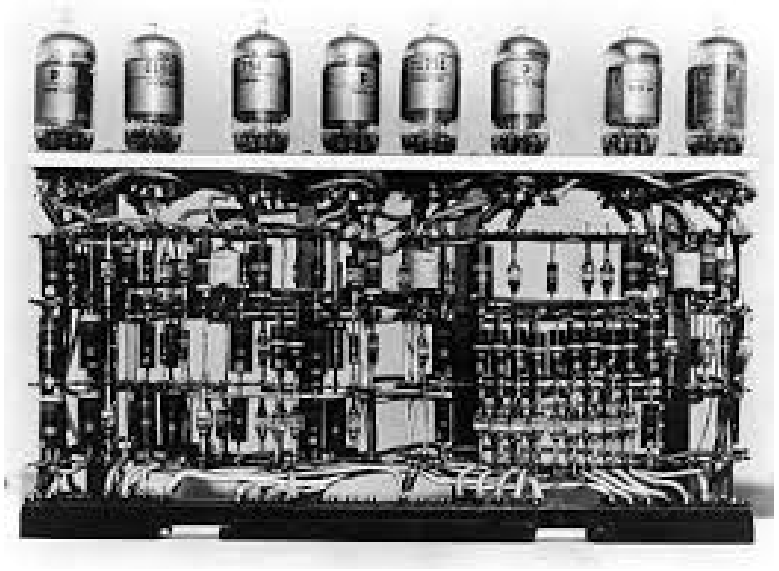


Figura 6. Primera generación.
Fuente: portalinformatico.wikispaces.com

Segunda generación (1958-1964)

Características de ésta generación: usaban transistores para procesar información. Los transistores eran más rápidos, pequeños y más confiables que los tubos al vacío. 200 transistores podían acomodarse en la misma cantidad de espacio que un tubo al vacío. Usaban pequeños anillos magnéticos para almacenar información e instrucciones. Producían gran cantidad de calor y eran sumamente lentas. Se mejoraron los programas de computadoras que fueron desarrollados durante la primera generación.

Se desarrollaron nuevos lenguajes de programación como COBOL y FORTRAN, los cuales eran comercialmente accesibles. Se usaban en aplicaciones de sistemas de reservaciones de líneas aéreas, control del tráfico aéreo y simulaciones de propósito general. La marina de los Estados Unidos desarrolló

el primer simulador de vuelo, Computadora Whirlwind. Se comenzó a disminuir el tamaño de las computadoras.

Aparecieron muchas compañías y las computadoras eran bastante avanzadas para su época, como la serie 5000 de Burroughs y la ATLAS de la Universidad de Mánchester. Algunas computadoras se programaban con cintas perforadas y otras por medio de cableado en un tablero.

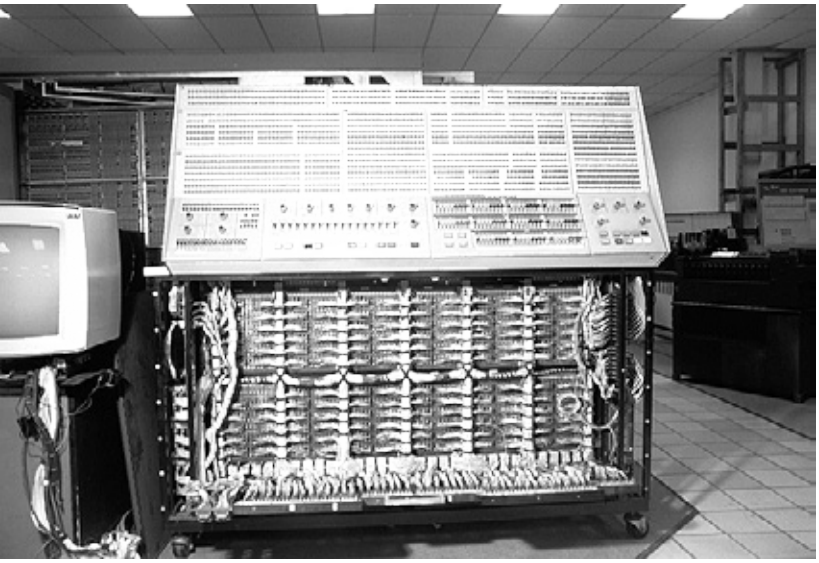


Figura 7. Transistores.
Fuente: www.arqhys.com

Tercera generación (1964-1971)

Comienza a utilizarse los circuitos integrados, lo cual permitió abaratar costos al tiempo que se aumentaba la capacidad de procesamiento y se reducía el tamaño de las máquinas. La tercera generación de computadoras emergió con el desarrollo de circuitos integrados (pastillas de silicio) en las que se colocan miles de componentes electrónicos en una integración en miniatura. El PDP-8 de la Digital Equipment Corporation fue el primer miniordenador.

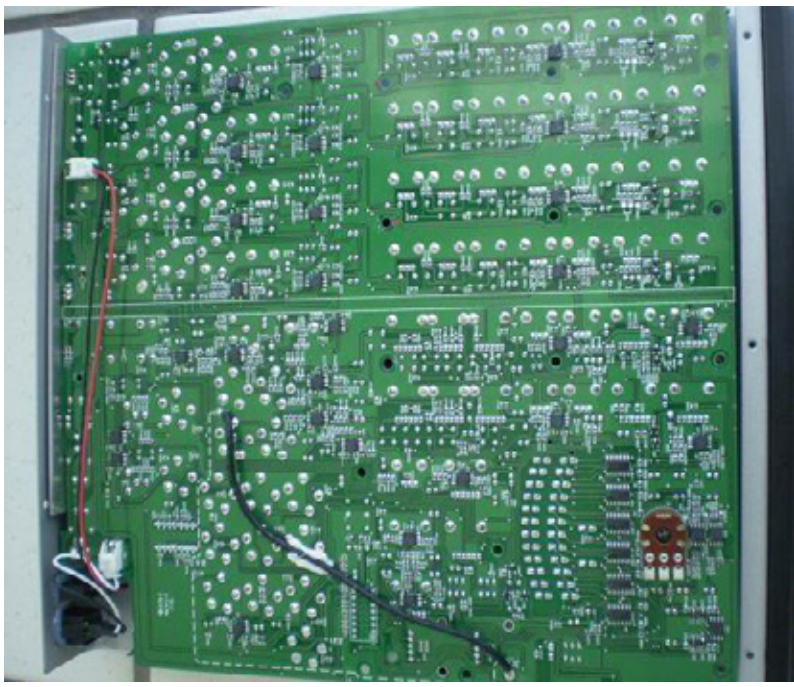


Figura 8. Circuitos integrados.
Fuente: www.somaovivo.org

Cuarta generación (1971-1983)

Fase caracterizada por la integración sobre los componentes electrónicos, lo que propició la aparición del microprocesador, es decir, un único circuito integrado en el que se reúnen los elementos básicos de la máquina. Se desarrolló el microprocesador. Se colocaron más circuitos dentro de un "chip". "LSI - Large Scale Integration circuit" "VLSI - Very Large Scale Integration circuit". Cada "chip" ya puede hacer diferentes tareas. Un "chip" sencillo actualmente contiene la unidad de control y la unidad de aritmética/lógica. El tercer componente, la memoria primaria, es operado por otros "chips". Se reemplaza la memoria de anillos magnéticos por la memoria de "chips" de silicio. Se desarrollan las microcomputadoras, o sea, computadoras personales o PC. Se desarrollan las supercomputadoras.

Figura 9. Tecnología VLSI.
Fuente: www.dicyt.com

Quinta generación (1984 -1999)

Surge la PC tal cual como la conocemos en la actualidad. IBM presenta su primera computadora personal y revoluciona el sector informativo. En vista de la acelerada marcha de la microelectrónica, la sociedad industrial se ha dado a la tarea de poner también a esa altura el desarrollo del software y los sistemas con que se manejan las computadoras.

Figura 10. PC.
Fuente: p8ariadnaguadalupe423marilu.blogspot.com

Sexta generación (2000 - Actualidad)

Los computadores cuentan con arquitecturas combinadas paralelo vectorial, con cientos de microprocesadores vectoriales trabajando al mismo tiempo, se han creado computadores capaces de realizar más de un millón de millones de operaciones aritméticas de coma flotante por segundo, las redes de área mundial siguen creciendo desorbitadamente, utilizando medios de comunicación a través de fibra óptica y satélites.

Figura 11. Dispositivos.
Fuente: hostmonsterz.info

Esquema estructural de un computador

Un computador se divide fundamentalmente en dos partes: el Hardware y el Software. El hardware es la parte física del computador, la parte tangible; es decir aquello que podemos tocar del computador. El software es la parte lógica del computador, es decir el conjunto de instrucciones que le ordenan al hardware que tarea debe realizar.

Unidad Central de Procesamiento (CPU)

Unidad central de proceso o CPU (mejor conocida por sus siglas en inglés, CPU), es un circuito microscópico que interpreta y ejecuta instrucciones. La CPU se ocupa del control y del proceso de datos en las computadoras. Generalmente, la CPU es un microprocesador fabricado en un chip, un único trozo de silicio que contiene millones de componentes electrónicos. El microprocesador de la CPU está formado por una unidad aritmético-lógica que realiza cálculos y comparaciones y toma decisiones lógicas (determinando si una afirmación es cierta o falsa mediante las reglas del álgebra de Boole. Para aceptar órdenes del usuario, acceder a los datos y presentar los resultados, la CPU se comunica a través de un conjunto de circuitos o conexiones llamado *bus*. El bus conecta la CPU a los dispositivos de almacenamiento (por ejemplo, un disco duro), los dispositivos de entrada (por ejemplo, un teclado o un mouse) y los dispositivos de salida (por ejemplo, un monitor o una impresora).

Cuando se ejecuta un programa, el registro de la CPU, llamado *contador* de programa, lleva la cuenta de la siguiente instrucción del programa, para garantizar que las instrucciones se ejecuten en la secuencia adecuada. La unidad de control de la CPU coordina y temporiza las funciones de la CPU, tras lo cual recupera la siguiente instrucción desde la memoria. En una secuencia típica, la CPU localiza la instrucción en el dispositivo de almacenamiento correspondiente. La instrucción viaja por el bus desde la memoria hasta la CPU, donde se almacena en el registro de instrucción. Entretanto, el contador de programa se incrementa en uno para prepararse para la siguiente instrucción. A continuación, la instrucción actual es analizada por un descodificador, que determina lo que hará la instrucción. Cualquier dato requerido por la instrucción es recuperado desde el dispositivo de almacenamiento correspondiente y se almacena en el registro de datos de la CPU. A continuación, la CPU ejecuta la instrucción, y los resultados se almacenan en otro registro o se copian en una dirección de memoria determinada.

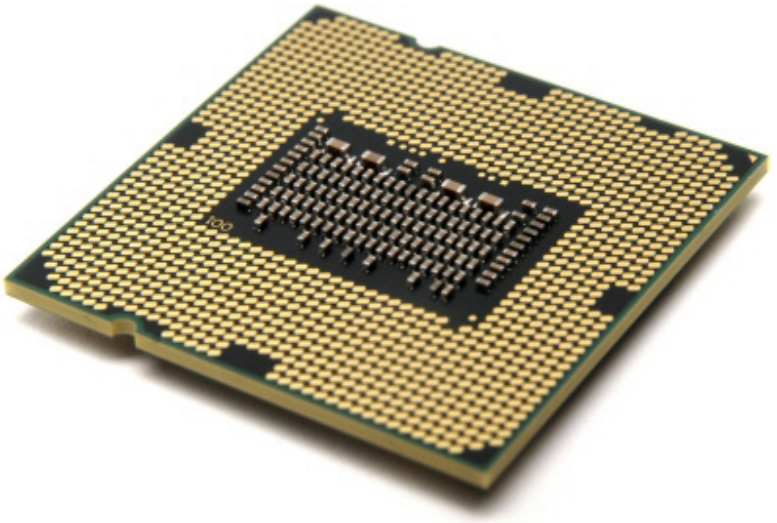


Figura 12. Microprocesador.
Fuente: www.pcworld.idg.com.au

La Memoria

Es la encargada de almacenar toda la información que el computador está usando. Las hay de tres tipos y son:

Memoria RAM

Del inglés *Random Access Memory,* es la memoria principal. Se caracteriza y diferencia de la memoria ROM porque una vez apagado el sistema operativo, toda la información almacenada en la memoria RAM es automáticamente borrada.

Figura 13. Memoria RAM.
Fuente: culturacion.com

Memoria ROM

Del inglés *Read Only Memory*, es permanente, ya que lo que permanece en la ROM no se pierde aunque el computador se apague.

Su función principal es guardar información inicial que el computador necesita para colocarse en marcha una vez que se enciende. Solo sirve para leer. Se puede leer la información desde esta memoria y no recibir información.

Memoria Caché

Es aquella que se usa como puente entre el CPU y la memoria RAM para evitar demoras en el procesamiento de los datos. Existen varios núcleos de esta memoria (denominados con la letra L y un número, por ejemplo L1). Cuanto menor el número más rápida es la memoria.

La memoria Caché es, por proximidad a la CPU, mucho más rápida que la memoria RAM. También es mucho más pequeña. La velocidad de la caché con respecto a la memoria RAM convencional es del orden de 5 a 10 veces superior.

El bus de Datos o cables de datos

Es el conjunto de interconexiones entre las distintas partes del computador que permiten la comunicación entre todos los dispositivos del mismo. También un bus es un conjunto cableado que sirve para que los dispositivos hardware puedan comunicarse entre sí. Son rutas compartidas por todos los dispositivos y les permiten transmitir información de unos a otros, son, en definitiva, las autopistas de la información interna, las que permiten las transferencias de toda la información manejada por el sistema.

Dispositivos de almacenamiento

Son todos aquellos que permiten almacenar los datos en el computador. Entre estos encontramos: el disquete, el disco duro (interno y externo), el CD, el DVD, la memoria USB, etc.

Dispositivos de entrada

Son todos aquellos que permiten la entrada de datos a un computador. Entre estos encontramos: el teclado, el ratón, el escáner, el micrófono, la cámara web, lápices ópticos, memorias USB, CD, DVD.

Figura 14. Dispositivos de entrada.
Fuente: www.informatica-hoy.com.ar

Dispositivos de Salida

Son todos aquellos que permiten mostrar la información procesada por el computador. Entre estos encontramos: la pantalla, la impresora, los altavoces, etc.

Figura 15. Dispositivos de salida.
Fuente: unpaseoporsexto.blogspot.com

Dispositivos de Comunicación

Son todos aquellos que permiten la comunicación entre computadores. Entre estos encontramos: el módem, la tarjeta de red y el enrutador (router).

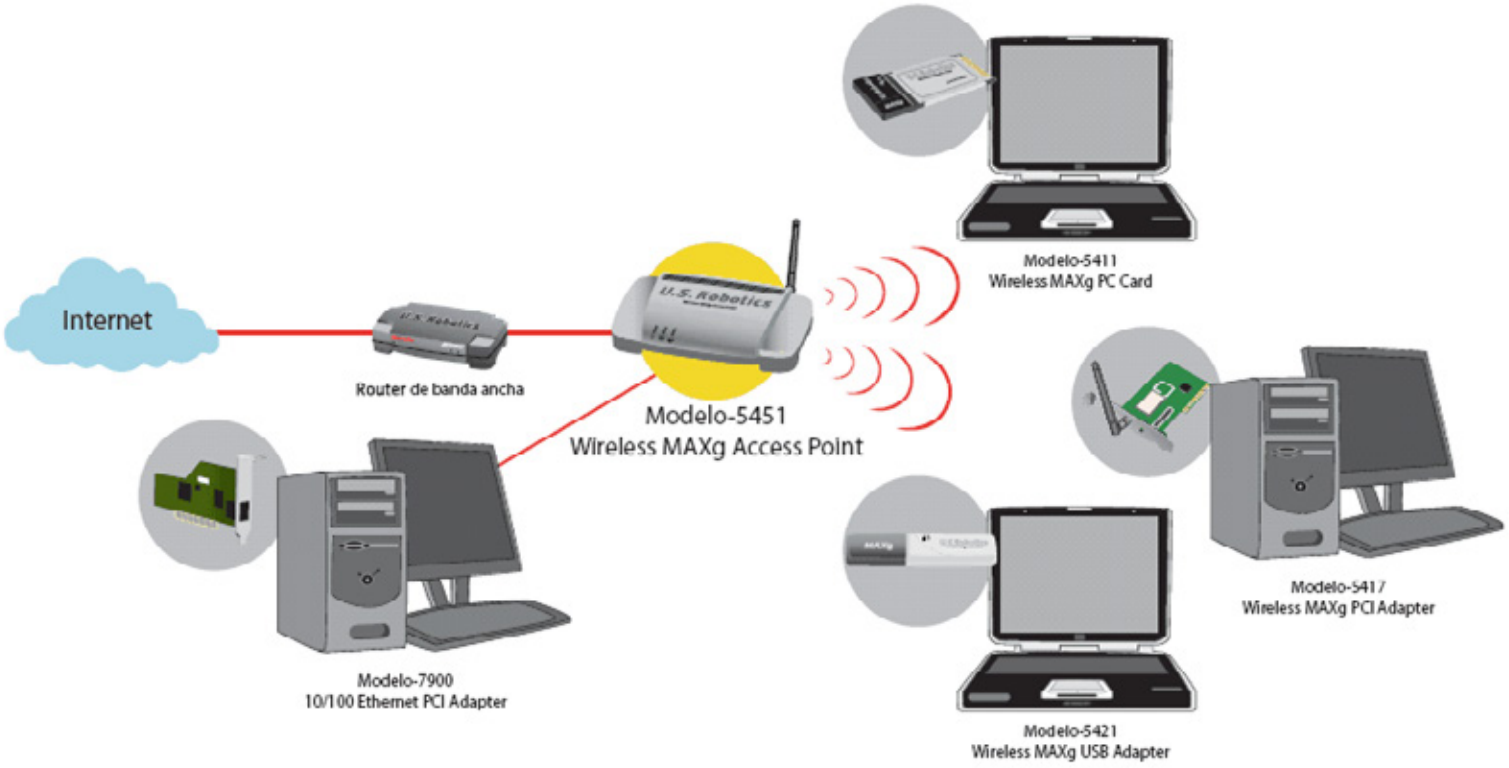


Figura 16. Dispositivos de comunicación.
Fuente: maria4b.blogspot.com

Dispositivos de Almacenamiento

Los dispositivos o unidades de almacenamiento de datos son componentes que leen o escriben datos en medios o soportes de almacenamiento, y juntos conforman la memoria o almacenamiento secundario de la computadora.

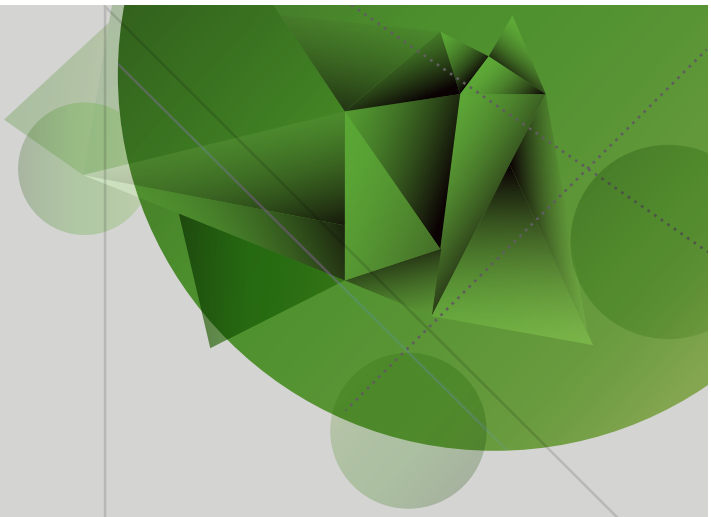
Estos dispositivos realizan las operaciones de lectura o escritura de los medios y soportes donde se almacenan o guardan, lógica y físicamente, los archivos de un sistema informático. Como: Disco Duro, Disquetera, Memorias USB, discos y cintas magnéticas.

Figura 17. Dispositivos de almacenamiento.
Fuente: usc2014informatica.wordpress.com

2

Unidad 2

Algoritmos



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Más que acumular conocimientos, el estudiante debe aprender a resolver problemas, debe desarrollar habilidades para establecer un marco de trabajo que le sea útil en la resolución de los mismos. Este marco comprende la capacidad de entender plenamente el enunciado, saber cómo organizarse para buscar información e interpretar los resultados.

Los algoritmos son una excelente herramienta con la que podemos visualizar de manera completa, lógica y fluida, el comportamiento de una solución propuesta a un problema, de manera que es fácil detectar errores y presentar mejoras.

U2

Metodología

Para las personas con habilidades en las matemáticas, resulta más fácil aprender y desarrollar algoritmos gracias al dominio de conceptos como variables, funciones, ecuaciones y lógica. Ya que un algoritmo es la solución a un problema, se debe leer completamente el enunciado y entender tanto la información suministrada, como la solicitada para poder plantear un procedimiento de resolución.

Se invita al estudiante a leer completamente la guía de manera que entienda los contenidos y conceptos desarrollados. Así mismo, se le recomienda encarecidamente analizar los ejemplos planteados, siguiendo paso a paso el desarrollo a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participar en los foros y formular todas las dudas que pueda tener en relación al tema tratado.

U2 Desarrollo temático

Algoritmo

Ya que el término aún no ha sido estandarizado o formalizado, podemos encontrar gran cantidad de definiciones para un algoritmo, de las cuales, la mayoría son válidas, entre algunas tenemos:

- Un algoritmo es la expresión de una secuencia precisa de operaciones que conduce a la resolución de un problema.
- Sistema de reglas que permiten obtener una salida específica a partir de una entrada específica. Cada paso debe estar definido exactamente, de forma que pueda traducirse a lenguaje de computadora.
- Un algoritmo es el método para expresar los pasos requeridos para resolver un problema de forma finita, precisa, definida y legible.

Sin embargo, todos plantean la misma noción; resolver un problema de manera sistemática que podamos repetir obteniendo siempre los mismos resultados.

Ahora bien, si repasamos la definición de un problema dentro del contexto de la programación, podríamos decir que se trata de un asunto, cuestión o situación presentada, que se intenta resolver de manera lógica.

Aparte de la teoría propia relacionada con el argumento de cada problema -es decir, si trabajamos con áreas, debemos saber que es el área, cómo calcularla, entender que existen diferentes fórmulas dependiendo del tipo de figura geométrica-, un asunto crucial para poder resolverlo es entender el problema. Entender el problema significa comprender lo que el enunciado nos ofrece, las suposiciones que se pueden realizar, los resultados que se esperan, la forma de relacionar y vincular los procedimientos y el análisis de la mejor forma de abordarlo. Esta habilidad se desarrolla con la práctica, de aquí la verdad de que la programación se aprende programando, así mismo ocurre con la formulación de Algoritmos.

Afortunadamente existen un par de herramientas que nos ayudan a estructurar y a diseñar los algoritmos, haciendo más corta la curva de aprendizaje, estos son: los diagramas de flujo y los pseudocódigos, que veremos más adelante en esta cartilla.

Propiedades de un algoritmo

Un algoritmo debe ser:

- Finito: el algoritmo debe tener un número limitado de pasos.
- Preciso: claro con la indicación y secuencia de los pasos a seguir.
- Definido: si se sigue el algoritmo n veces, siempre se debe obtener el mismo resultado.

- Legible: la manera de expresarlo debe ser clara, de forma que sea fácil de leer y de entender.

Partes de un algoritmo

La estructura básica de un algoritmo corresponde a la estructura de un sistema, que se divide en 3 partes: Inicio, Proceso y Fin:

Inicio:

Es la información que necesita el algoritmo para poder comenzar a ejecutarse, se incluyen valores iniciales, constantes, condiciones.

Proceso:

Son todos los pasos requeridos para obtener la solución al problema, tenemos formulas, asignaciones, comparaciones, ciclos.

Fin:

Son los resultados entregados por el algoritmo como solución al problema.

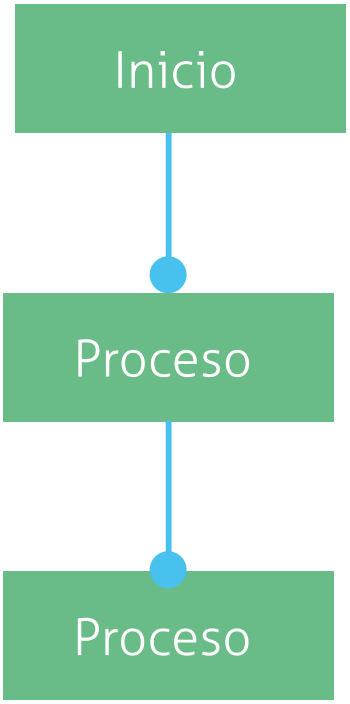


Imagen 1.
Fuente: Propia.

Principios de formulación de algoritmos

Las herramientas mayormente empleadas para el diseño de algoritmos son:

- Diagrama de flujo.
- Pseudocódigo.

Diagrama de flujo

Un diagrama de flujo es la representación gráfica de un algoritmo, se basa en la utilización de símbolos para representar operaciones, la secuencia de cada operación se establece mediante líneas que las interconectan.

La simbología para el diseño de diagramas no ha sido estandarizada, depende de cada área particular de implementación, sin embargo existen algunos esquemas que son empleados con más regularidad.

Simbología

	Terminal : representa el inicio y fin de un algoritmo.
	Entrada/salida: cualquier tipo de dato introducido u obtenido.
	Proceso: cualquiera operación que pueda cambiar un valor, formato o posición.
	Decisión: es una operación lógica que con base en una comparación, determina el camino a seguir.
	Conector sirve para enlazar dos partes de un algoritmo.
	Línea de flujo: indica el sentido de ejecución de las operaciones.
	Conector: sirve para enlazar dos partes de un algoritmo ubicadas en diferentes pagina.
	Llamada a subrutina es un modulo independiente del programa, incluso podría ser otro algoritmo.
	Pantalla: se utiliza para indicar operaciones de entrada/salida de datos.
	Impresora: se utiliza para indicar operaciones de entrada/-salida de datos.
	Impresora: se utiliza para indicar operaciones de entrada/-salida de datos.

Tabla 1. Simbología.
Fuente: Propia.

Recomendaciones para el diseño de Diagramas de flujo:

- Ser lo más ordenado posible.
- Siempre debe existir el Inicio y el Fin del algoritmo.
- Se deben usar solamente líneas de flujo horizontales y/o verticales.
- Se debe evitar el cruce de líneas utilizando los conectores.
- Se deben usar conectores sólo cuando sea necesario.
- No deben quedar líneas de flujo sin conectar.
- Se deben trazar los símbolos de manera que se puedan leer de arriba hacia abajo y de izquierda a derecha.
- Todo texto escrito dentro de un símbolo deberá ser escrito claramente, evitando el uso de muchas palabras.

Ventajas

- Facilita el entendimiento del algoritmo.
- Favorece la detección de opciones de mejora y error en el planteamiento de la solución.
- Permite analizar relaciones y transacciones entre operaciones.

Desventajas

- Pueden resultar laboriosos de construir.
- En diagramas de flujo grandes se hace difícil realizar el seguimiento.
- No existen normas fijas para la simbología y el diseño.

Pseudocódigo

Se trata de un lenguaje para la especificación de algoritmos previo a la codificación en términos de programación, muy fácil de entender y manipular.

Se puede considerar como un medio para representar estructuras de control que surgió como un lenguaje muy similar al de los humanos que puede ser convertido rápidamente a lenguaje de programación.

Su estructura cumple con las partes del algoritmo Inicio-Proceso-Fin, el proceso se establece como acciones que describen los pasos que se deben realizar, se ejecutan en el orden de cada línea de arriba hacia abajo. Así:

Inicio
    Acción 1
    Acción 2
    …
    Acción n
Fin

Una acción puede contener a su vez mas acciones:

Inicio
    Acción 1
        Acción 1.1
        Acción 1.2
        …
        Acción 1.m
    Acción 2
        Acción 2.1
        Acción 2.2
            …
        Acción 2.m

…
Acción n
Acción n.1
Acción n.2
…
Acción n.m

Fin

Ventajas

- Ocupa menos espacio que un diagrama.
- Se pueden expresar muy fácilmente operaciones complejas.
- Fáciles de traducir a un lenguaje de programación.

Desventajas

- Se requiere manejo de lenguaje técnico.
- Es complejo de entender para la persona común.
- No existen normas para la sintáctica y la semántica.

Ejemplos

Ejemplo 1

Dibujar un diagrama de flujo para determinar si un número N es par o impar.

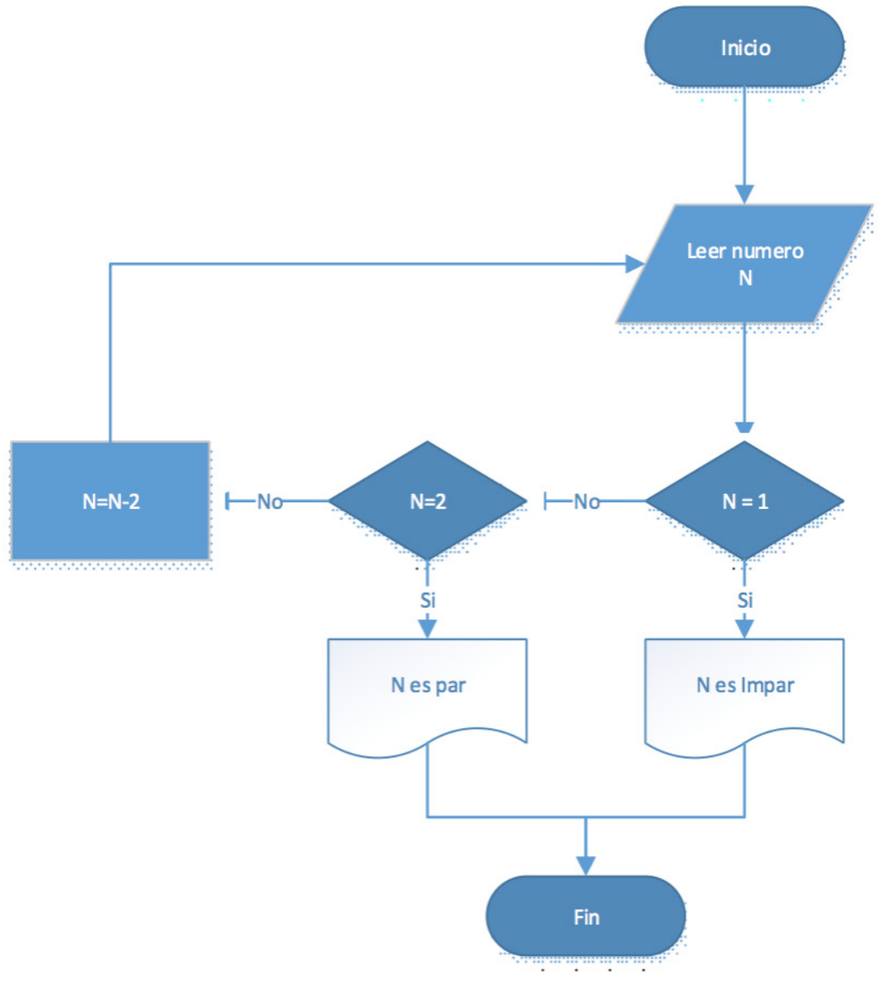


Imagen 2.
Fuente: Propia.

Ejemplo 2

Hacer un Pseudocódigo que lea dos números (N1 y N2) y determine cuál es el mayor.
Inicio:
Leer N1 y N2
Si N1 = N2 entonces
Escribir: "N1 es igual a N2"
Si N1>N2 entonces
Escribir: "N1 es mayor que N2"
Sino entonces
Escribir: "N2 es mayor que N1"
Fin

Ejemplo 3

Dibujar un diagrama de flujo y escribir un Pseudocódigo para calcular la suma, resta, multiplicación y división de cantidades dadas (C1 y C2).

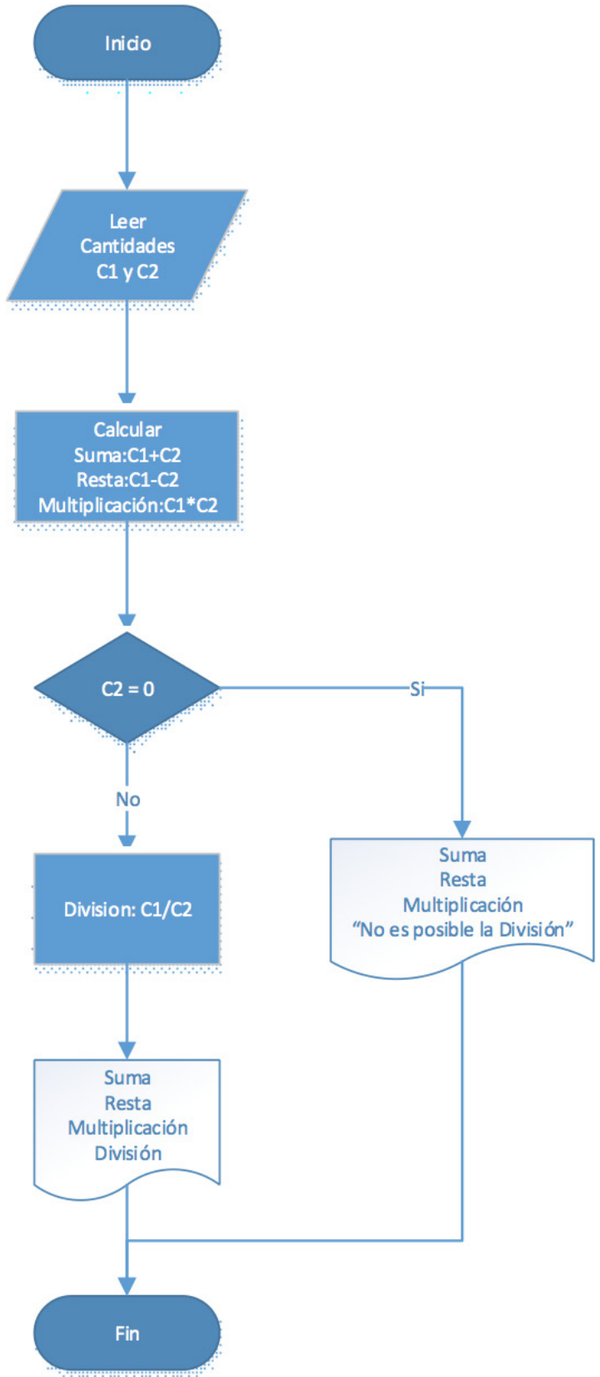


Imagen 3.
Fuente: Propia.

Inicio:

Leer C1 y C2

Si C2=0 entonces

Calcular

Suma= C1+C2

Resta= C1-C2

Multiplicación= C1*C2

Escribir: Suma, Resta, Multiplicación, "No es posible la división por cero".

Sino entonces

Calcular

Suma= C1+C2

Resta= C1-C2

Multiplicación= C1*C2

División= C1/C2

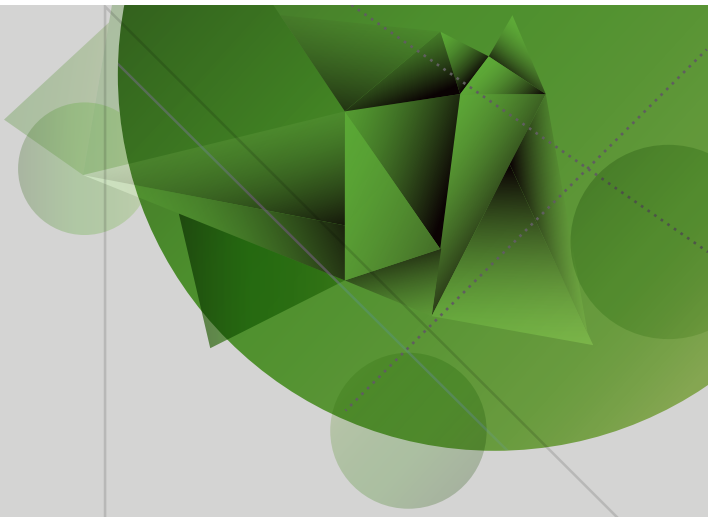
Escribir: Suma, Resta, Multiplicación, División.

Fin

2

Unidad 2

Problemas



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

En la presente carilla estudiaremos el concepto de problemas algorítmicos, y las herramientas y métodos que tenemos para resolverlos, nuevamente reiterando la importantísima relación que existe entre entender el problema y solucionarlo.

Para los humanos, la manera natural de abordar los problemas de tipo algorítmico es conociendo el tipo de problema y el tipo de respuesta esperada, cuando organizamos y categorizamos los diferentes tipos de problemas que se pueden presentar, podemos establecer si es posible resolverlo y de qué manera.

U2

Metodología

El fortalecimiento de las capacidades matemáticas y lógicas le presentará grandes ventajas al estudiante tanto para la comprensión, como para la creación de algoritmos, por tal motivo se recomienda tratar de mejorar e incrementar tales habilidades, repasando conceptos matemáticos y resolviendo a manera de lúdica ejercicio relacionados con la lógica y el razonamiento.

Se invita al estudiante a leer completamente la guía de manera que entienda los contenidos y conceptos desarrollados. Así mismo, se le recomienda insistentemente analizar los ejemplos planteados, siguiendo paso a paso el desarrollo a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participar en los foros y formular todas las dudas que pueda tener en relación al tema tratado.

U2

Desarrollo temático

Problemas algorítmicos

¿Qué es un problema?

Un problema es un escenario, en el que se buscar alcanzar un objetivo empleando medios y estrategias, se caracteriza por tener:

- Un estado inicial.
- Un objetivo, meta o estado al que se espera llegar.
- Un grupo de recursos de lo que puede disponer para resolverlo.
- Un dominio que se refiere al grupo de conocimientos de quien lo resolverá.

Así mismo, existen muchas estrategias para abordar y resolver problemas:

- Ensayo y error: Este método consiste en un intento de fuerza bruta en la que se intenta y se intenta resolver el problema, tratando de emplear todas las opciones posibles hasta que alguna permita alcanzar la meta, no se garantiza resolver el problema.
- Heurística: Se aprovecha la experiencia para determinar el mejor camino para resolver el problema.
- Algoritmos: Es la expresión de una secuencia precisa de operaciones que conduce a la resolución de un problema.
- Lluvia de ideas: Se puede hacer individual o grupalmente, consiste en postular un grupo de soluciones viables y luego desarrollarlas o complementarlas entre sí.

El método por excelencia para resolver problemas por medio de computadores es el algoritmo, se debe tener siempre en mente buscar la posibilidad de reducir el coste que representa en términos de consumo de tiempo, memoria y procesamiento la ejecución del programa codificado a partir del algoritmo, aunque esta parte será analizada con detalle más adelante.

El método Pólya

Fue propuesto por el húngaro George Pólya en el que se generalizaban cuatro pasos en el que debemos plantear y resolver una serie de cuestionamientos para resolver un problema, se debe tener en cuenta, que podemos pasar de un paso a otro indistintamente:

1. Entender el problema
   - ¿Cuál es la incógnita?
   - ¿Cuáles son los datos?
   - ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

2. Trazar un plan
   - ¿Te has encontrado con un problema semejante?
   - ¿Conoces algún teorema que te pueda ser útil?
   - ¿Puedes enunciar al problema de otra forma?
   - Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general?
   - ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?
   - Escoger las operaciones que se deben efectuar.
   - Descomponer el problema en otros más pequeños.
   - Realizar diagrama de flujo o pseudocódigo.
3. Ejecutar el plan:
   - Al ejecutar tu plan de la solución, comprueba cada uno de los pasos.
   - ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?
   - Ejecutar en detalle cada operación.
   - Simplificar antes de calcular.
4. Revisar:

- ¿Puedes verificar el resultado?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?
- Dar una respuesta completa.
- Hallar el resultado de otra manera o realizar el cálculo con otros valores.
- Hacer seguimiento al procedimiento y verificar los resultados.

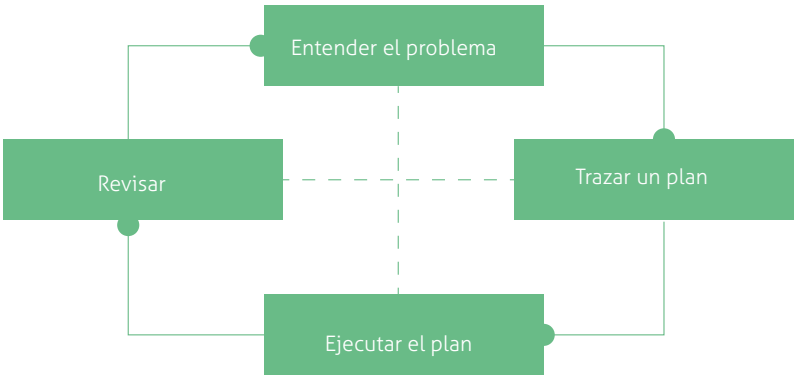


Figura 1. Método Pólya.
Fuente: Propia.

Clasificación de los problemas

Naturaleza.

Tratabilidad.

Tipo de respuesta.

Clasificación por naturaleza

Los problemas algorítmicos son clasificados de acuerdo a su naturaleza intrínseca respecto a la posibilidad y dificultad de resolverlos. Podemos encontrar problemas indecidibles y problemas decidibles.

Problemas indecidibles: aquellos problemas los cuales no se pueden solucionar a través de un algoritmo.

Problemas decidibles: aquellos problemas que cuentan al menos con un algoritmo para su cómputo o solución.

Sin embargo, que un problema sea decidible no implica que pueda ser resuelto, existe gran cantidad de problemas que cuentan con algoritmos para su resolución, pero son inabordables para un computador por el gran número de operaciones que hay que realizar para hallar su solución. Esto permite separar los problemas decidibles en dos:

Intratables: aquellos para los que no es factible obtener su solución. A los problemas intratables se le conoce también como problemas NP (de orden no determinístico polinominal). Ejemplo: determine las variables de espacio–tiempo que influyeron en la generación del Big Bang que dio origen al sistema solar.

Tratables: aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable. Ejemplo: calcule la rentabilidad económica de las acciones de una empresa X en la bolsa internacional de valores del último semestre.

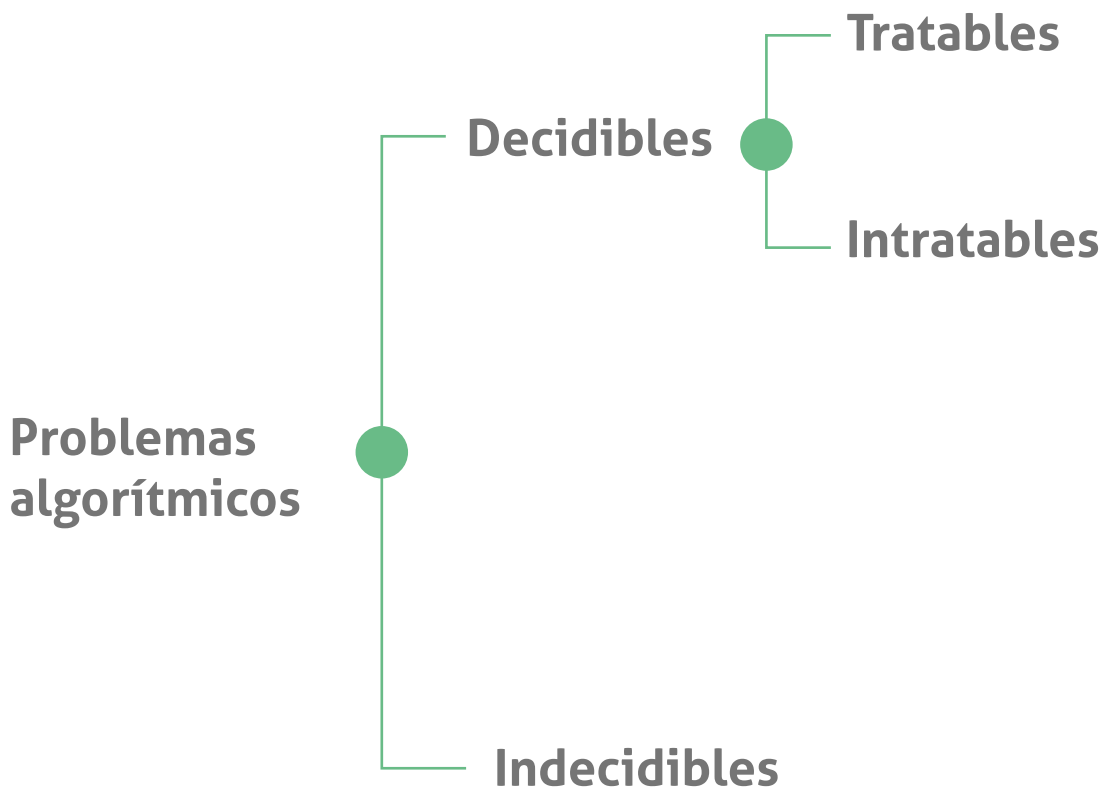


Figura 2.
Fuente: Propia.

Clasificación por tratabilidad

Los problemas que admiten solución son clasificados de acuerdo a la complejidad que presentan los algoritmos para resolverlos.

Complejidad	P	NP	NP-completo
	(Polinómico)	(No-determinista polinómico)	(No-determinista polinómico completo)
Conceptos			
Definición	Son los problemas tratables, es decir que suelen ser abordables en la práctica, pueden contener muchos problemas naturales.	Son los problemas en los cuales se aplica un algoritmo polinómico para comprobar si una posible solución es viable o no.	Son aquellos problemas los cuales están en la frontera externa de la clase NP, son los peores problemas posibles de la clase NP.
Diferencias	En P están los problemas que se pueden resolver en tiempo polinómico.	Sus mejores algoritmos conocidos son no deterministas.	Imposible encontrar un algoritmo eficiente para una solución óptima.
Ejemplos	Búsqueda binaria, secuencial, factorial.	Torres de Hanói, Ordenación Shell.	Vendedor viajero, camino largo, ciclo Hamiltoniano.

Tabla 1:
Fuente: Propia.

La Clase P está constituida por todos los problemas comprobadamente tratables, esto es, problemas que pueden ser resueltos por algoritmos de complejidad polinominal.

Otros ejemplos de la Clase P:

- Resolución de Sistemas de Ecuaciones Lineales.
- Contabilidad (registrar y/o modificar transacciones).
- Ordenar números, buscar palabras en un texto.
- Juntar Archivos.
- En general los sistemas operacionales (facturación, control de almacenes, planillas, ventas, etc.).
- Cualquier problema de la Programación Lineal.

- Sistemas de transacciones bancarias.En general los sistemas de información gerencial.

La Clase NP está constituida por todos los problemas que pueden ser resueltos por algoritmos enumerativos, cuya búsqueda en el espacio de soluciones es realizada en un árbol con profundidad limitada por una función polinominal respecto al tamaño de la instancia del problema y con ancho eventualmente exponencial.

Otros ejemplos de la clase NP:

- Clique.
- Cobertura de Vértices y Aristas.
- Coloración de Grafos.
- Mochila Lineal y Cuadrática.
- Optimización de Desperdicios.

Algoritmos Enumerativos

Estos algoritmos obtienen la solución con base a enumerar, implícita o explícitamente, todas las soluciones posibles y escogiendo la mejor de todas ellas.

Enumeración explícita: Se calculan todas las posibles soluciones y se escoge la mejor de ellas.

Ejemplo:

En programación entera (PE) 0-1 el número de posibles soluciones es 2nºvar.enteras

4 variables: $2^4 = 16$ soluciones o nodos

10 variables: $2^{10} = 1.024$ nodos

20 variables $2^{20} = 1.048.276$ nodos

Este método tiene un grave inconveniente, en problemas complejos, un ordenador no sería capaz de enumerar todas las posibles soluciones.

Enumeración implícita: En este enfoque se trata de resolver el problema de tiempo de la enumeración explicita para ello se aplican un conjunto de reglas para evitar enumerar soluciones no factibles o peores que la mejor solución factible que se haya localizado hasta el momento.

Clasificación por tipo de respuesta

Los problemas son clasificados de acuerdo a la respuesta requerida por el problema así:

- Problemas de decisión.
- Problemas de localización
- Problemas de optimización.

Problemas de Decisión: responder SI o NO a una determinada indagación.

Ejemplo:

Determinar el importe a pagar de un almacén de cadena que ha determinado realizar descuentos a sus clientes de acuerdo al medio de pago, para lo cual ha determinado los siguientes criterios:

15% de descuento si paga con tarjeta de crédito.

10% de descuento si paga con tarjeta de debito.

5% adicional de descuento si la tarjeta es Visa.

Solución:

El problema nos está pidiendo calcular el importe.

Tenemos 3 condiciones, por lo tanto tenemos 23 = 8 diferentes casos.

No hay restricciones.

Realizaremos la tabla de decisión:

Caso	1	2	3	4	5	6	7	8
Tarjeta Crédito	Si	Si	Si	Si	No	No	No	No
Tarjeta Débito	Si	Si	No	No	Si	Si	No	No
Tarjeta Visa	Si	No	Si	No	Si	No	Si	No
Inviable	X	X					X	
Aplicar Descuento (15%) Crédito			X	X				
Aplicar Descuento (10%) Débito					X	X		
Aplicar Descuento (5%) Adicional (Visa)			X		X			
Descuento Total			20%	15%	15%	15%	10%	0%
Calcular Importe	No	No	Si	Si	Si	SI	No	Si

Tabla 1. Descuentos por medio de pago.
Fuente: Propia.

Los casos 1, 2 y 7 no son casos posibles, ya que en los casos 1 y 2, una tarjeta no puede ser débito y crédito simultáneamente (lo vamos a suponer para el ejemplo) y para el caso 7 tampoco puede ser visa sin ser crédito o débito, los demás casos si arrojan resultados lógicos.

Problemas de Localización: encontrar, en caso de existir, una estructura que verifique las restricciones del problema, dicha estructura es denominada de solución viable.

Ejemplo:

Un centro de suministro envía a sus centros de demanda, mercancía todos los días, se adjuntan los puntos y las distancias entre ellos, determinar una ruta de manera que todos los centros de demanda sean visitados.

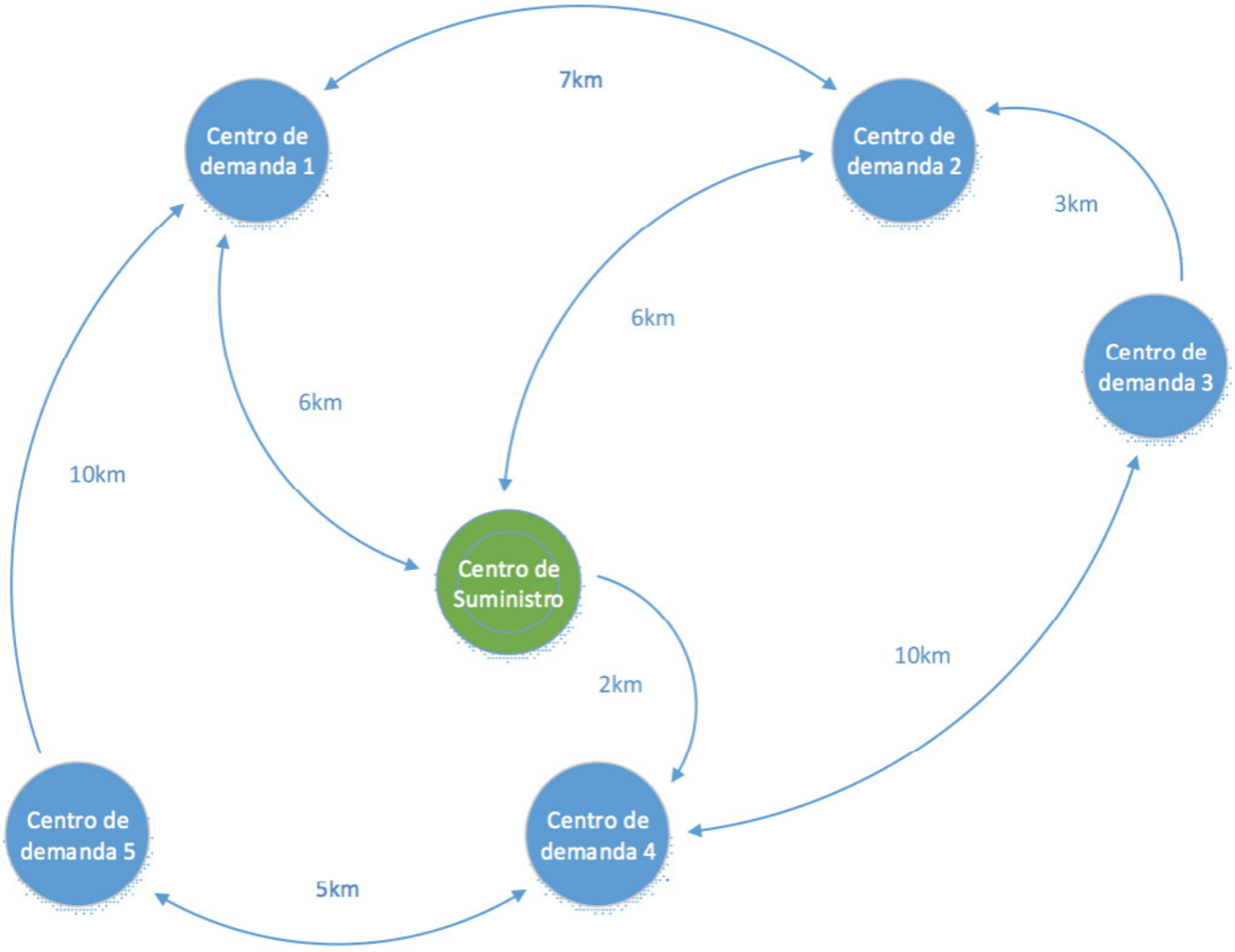


Figura 2. Rutas de suministros.
Fuente: Propia.

Solución:

El problema nos está pidiendo determinar una ruta.

Tenemos 6 nodos con interconexiones entre sí.

Las restricciones están determinadas por las conexiones entre nodos y la dirección de la comunicación, algunas son en ambos sentidos y otros no.

El ejercicio no indica que los centros no se pueden repetir en la ruta y tampoco dice que debe ser la ruta más corta

Entonces debemos determinar una manera de calcular la secuencia de centros que manera que los recorramos a todos.

De esta manera podríamos encontrar un número casi ilimitado de posibilidades de rutas para un recorrido por todos los centros.

Problemas de Optimización: encontrar una estructura que verifique las restricciones del problema y optimice un criterio predefinido. Esto es, encontrar una solución viable que optimice un criterio pre-determinado.

Ejemplo:

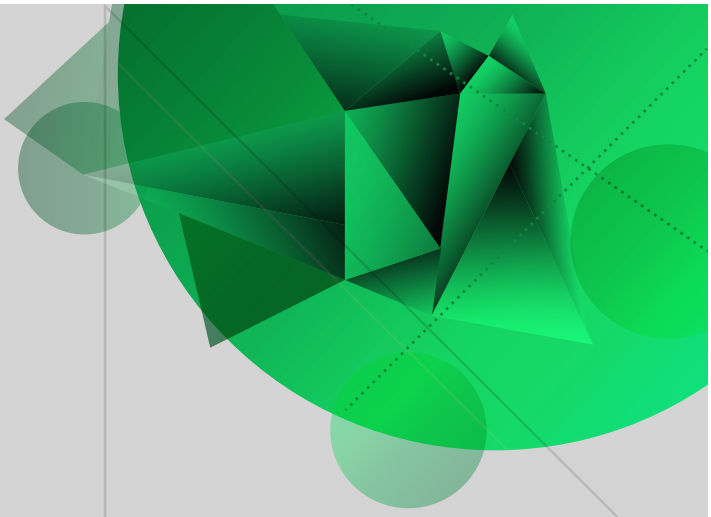
El planteamiento del ejemplo anterior agregando además la condición que debe ser la ruta más corta posible.

Se desarrolla de la misma manera para determinar todas las posibles rutas, agregando además un mecanismo que permita encontrar el recorrido que menos kilómetros genera al incluir todos los centros.

3

Unidad 3

Estructura de programas, datos, variables, operadores



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Antes de crear un programa informático en un lenguaje determinado hay que conocer una serie de conceptos básicos comunes a todos estos lenguajes, debemos entender las partes que lo conforman y la función que cumplen dentro del mismo. En primer lugar, se estudia la estructura general. Dicha estructura está dividida en bibliotecas, cabecera y lista de instrucciones o cuerpo del programa. Para cada una de estas partes, existe una caracterización sintáctica que depende de cada lenguaje en particular.

En segundo lugar, estudiaremos los tipos de datos, que definen el modo en que se usa el espacio (memoria) en los programas. Especificando un tipo de datos, se está indicando al compilador como crear un espacio de almacenamiento en particular, y también como manipular este espacio.

Un tipo de dato puede ser tan complejo como se quiera. Puede necesitar un byte para almacenar cualquier valor de su dominio, o requerir de muchos bytes. Cada lenguaje ofrece una colección de tipos de datos. También ofrece herramientas para crear tipos de dato distintos, más complejos y más acordes con el tipo de problema que se aborde en cada momento.

U3 Metodología

Se invita al estudiante a leer completamente la guía de manera que logre obtener una sólida y consistente fundamentación teórica, de igual forma, se le recomienda analizar los ejemplos planteados, siguiendo paso a paso el desarrollo a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participación en los foros y formular todas las dudas que pueda tener en relación al tema.

U3 Desarrollo temático

Estructura básica de un programa

Un programa es básicamente un sistema, que se define como un conjunto de elementos que interactúan entre sí para la consecución de un objetivo, en el que cada uno de sus elementos tiene funciones muy claras y definidas:

- Bibliotecas
- Cabecera
- Cuerpo
  - Declaración de variables
  - Flujos de entrada
  - Proceso o algoritmo
  - Flujos de salida
  - Anidamiento
  - Comentarios

Bibliotecas

Una biblioteca es una colección o conjunto de subprogramas, procedimientos y funciones agrupados en un archivo con el fin de ser aprovechadas por otros programas y usados también para desarrollar software. En general, las bibliotecas no son ejecutables, pero sí pueden ser usadas por ejecutables que las necesitan para poder funcionar correctamente.

Cabecera

La cabecera contiene, normalmente, una declaración directa de clases, subrutinas, variables, u otros identificadores. Aquellos programadores que desean declarar identificadores estándares en más de un archivo fuente pueden colocar esos identificadores en un único archivo de cabecera, que se incluirá cuando el código que contiene sea requerido por otros archivos.

Para la programación tenemos una cabecera principal y cabeceras de función, la cabecera principal indica el comienzo del programa, las cabeceras de funciones contienen rutinas escritas para desarrollar un tipo de actividad.

Cuerpo

Declaración de variables

Para que un programa pueda hacer uso de una o más variables, éstas deben ser declaradas previamente. Todas las variables de un programa se declaran de la misma forma, indicando de cada una de ellas: el tipo de dato que puede almacenar (mediante un identificador) y su nombre (mediante otro identificador).

Flujos de entrada

La constituyen todas las instrucciones que toman los datos de entrada desde un dispo-

sitivo externo y los almacena en la memoria principal para que puedan ser procesados.

Proceso o algoritmo

Está formado por las instrucciones que modifican los objetos a partir de su estado inicial (flujos de entrada) hasta el estado final (resultados) dejando los objetos que lo contiene disponibles en la memoria principal.

Flujos de salida

Conjunto de instrucciones que toman los datos finales (resultado) de la memoria principal y los envían a los dispositivos externos.

Anidamiento

El cuerpo de cualquier estructura puede ser instrucciones simples u otras estructuras, que a su vez pueden contener a otras.

Comentarios

Los comentarios se emplean para facilitar la tarea de los programadores humanos ya que no realizan ningún papel activo en la generación del código. Los comentarios se pueden introducir en el código fuente de un programa. Así, el compilador ignora todo lo que se incluya entre la secuencia de caracteres // y el final de la línea. Por ejemplo:

```
// Este es un comentario, llega al final de la línea.
```

La pareja de caracteres / hay que escribirla sin dejar ningún espacio en blanco entre ellos. El segundo tipo de comentario es el que se utiliza también en el lenguaje de programación C: el compilador también ignora todo lo que se incluya entre las secuencias de caracteres /* y */. Por ejemplo:

```
/* Este es otro comentario, el final
lo indica la marca */
```

El comentario con // es más fácil de teclear si sólo ocupa una línea, pero si ocupa varias, entonces el más sencillo es el de /* ... */. Los comentarios pueden incluir cualquier carácter válido en Unicode y no pueden anidarse.

Estos dos formatos de comentario se emplean para los denominados comentarios de implementación. Los comentarios de implementación se utilizan en el programa fuente para resaltar código o para aclarar un desarrollo en particular.

Ejemplo, tenemos el siguiente programa desarrollado en Java para calcular el cuadrado de un número dado:

```
import java.util.Scanner;
Import java.math;
// Programa para calcular el cuadrado de un numero
public class Cuadrado {
        public static void main(String[] args) {
        Scanner Sca = new Scanner(System.in);
        int n1=0;
        int Resultado:0;
        System.out.print("Ingresar el número a elevar al cuadrado: ");
        n1 = Sca.nextInt();
        Resultado = n1 * n1;
        System.out.print( n1 + " elevado al cuadrado es: " + Resultado);
        }

}
```

Ahora vamos a ver cada una de las partes que componen el programa:

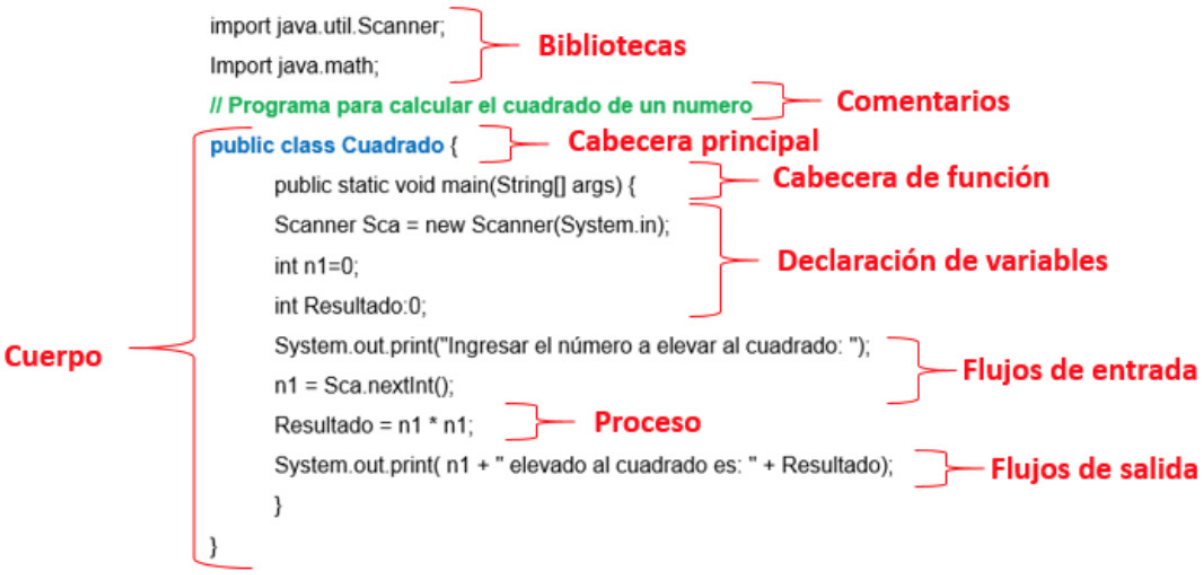


Figura 1. Partes de un programa.
Fuente: Propia.

Variables

En programación, las variables son espacios reservados de memoria en los que se almacena un dato cuyo valor puede cambiar durante el proceso. Toda variable debe tener un nombre que la identifique.

Tipos de variables

Las variables se pueden clasificar según el alcance que tiene dentro del programa, según el tipo de dato que contienen o según su función:

- Alcance:
  - Variable del entorno.
  - Variable local.
  - Variable global.
  - Variable de instancia.
- Tipo de dato:
  - Numérica.
  - Alfanumérica.
- Función:
  - Contador.
  - Acumulador.
  - Banderas.
  - Auxiliares.

Clasificación de las variables según el alcance

Variables del entorno

Una variable del entorno es un valor dinámico cargado en la memoria, que puede ser utilizado por varios procesos que funcionan simultáneamente. En la mayoría de los sistemas operativos, la ubicación de algunas bibliotecas o de los archivos ejecutables del sistema más importantes puede variar según la instalación.

Por eso es posible, para un programa dado, remitirse a una ubicación basada en las variables del entorno que definen estos datos.

Variable local

Una variable local es, en informática, la variable a la que se le otorga un ámbito local. Tales variables sólo pueden accederse desde la función o bloque de instrucciones en donde se declaran.

Variable global

Una variable global es, en informática, una variable accesible en todos los ámbitos de un programa informático. Los mecanismos de interacción con variables globales se denominan mecanismos de entorno global.

El uso de este tipo de variables suele considerarse como una mala práctica, por el riesgo que conlleva esa deslocalización: una variable global puede ser modificada en cualquier parte del programa (a menos que resida en una sección de memoria protegida) y cualquier parte del programa depende de ella. Es por ello que una variable global tiene un potencial ilimitado para crear dependencias, factor que aumenta la complejidad. Sin embargo, en algunas ocasiones, las variables globales resultan muy útiles. Por ejemplo, se pueden usar para evitar tener que pasar variables usadas muy frecuentemente de forma continua entre diferentes subrutinas.

Variable de instancia

En programación orientada a objetos, una variable de instancia o miembro de dato es una variable que se relaciona con una única instancia de una clase.

Cada vez que se crea un objeto, el sistema crea una copia de todas las variables que están vinculadas con dicha clase, haciéndolas propias de esa instancia. Solamente se puede acceder a ellas a través del identificador del objeto.

Estas variables son declaradas fuera del cuerpo de los métodos y dentro de la clase por lo tanto son de tipo global. Y por lo tanto, pueden ser utilizadas por cualquier método no estático de dicha clase.

Clasificación de las variables según el tipo de dato

Numéricas

Solo contienen números

Ejemplo:

Estatura: 1.65

Edad: 31

X: -38.2

Alfanuméricas

Pueden contener letras, números.

Ejemplo:

Nombre: Alberto Fuentes.

Dirección: Calle 1 N 23 - 45

Ciudad: Bogotá.

Clasificación de las variables según su función

Existen ciertos tipos de variables que cumplen una función específica dentro del proceso y se utiliza teniendo en cuenta ciertas reglas que se deben respetar.

Los tipos de esta clase de variables más comunes son:

Contador

Es una variable a la que se suma constantes y se utiliza para realizar un conteo. Siempre debe ser inicializado (generalmente el valor inicial del contador es cero). También se puede utilizar para realizar una cuenta regresiva, restándole la constante al contador.

Sintaxis: Contador = Contador + Constante

Acumulador

Es una variable a la que se le suma otra variable y se utiliza para realizar una sucesión de sumas. Al igual que el contador, el acumulador siempre debe ser inicializado.

Sintaxis: Acumulador = Acumulador + Variable

Banderas

Es una variable booleana (es decir que solo admite dos estados posibles: 0 o 1), Si la bandera es igual a 0, significa que está inactiva o apagado, Si la bandera es igual a 1, significa que está activa, encendida o levantada.

Los usos más comunes de la bandera son:

- Saber si el programa ha pasado o no por un punto determinado.
- Terminar un ciclo cuando se produce determinada condición que modifica el estado de la bandera
- Ejecutar procesos diferentes según el estado de la bandera.

Sintaxis: Bandera = Valor Booleano

Auxiliar

Es una variable que se utiliza para almacenar el contenido de otra variable. El auxiliar debe ser del mismo tipo que la otra variable.

Se utiliza para:

- Conservar el valor que tiene una variable en determinado momento, debido a que un proceso modificara ese valor.
- Intercambiar el contenido de dos variables

Sintaxis: Auxiliar = Variable

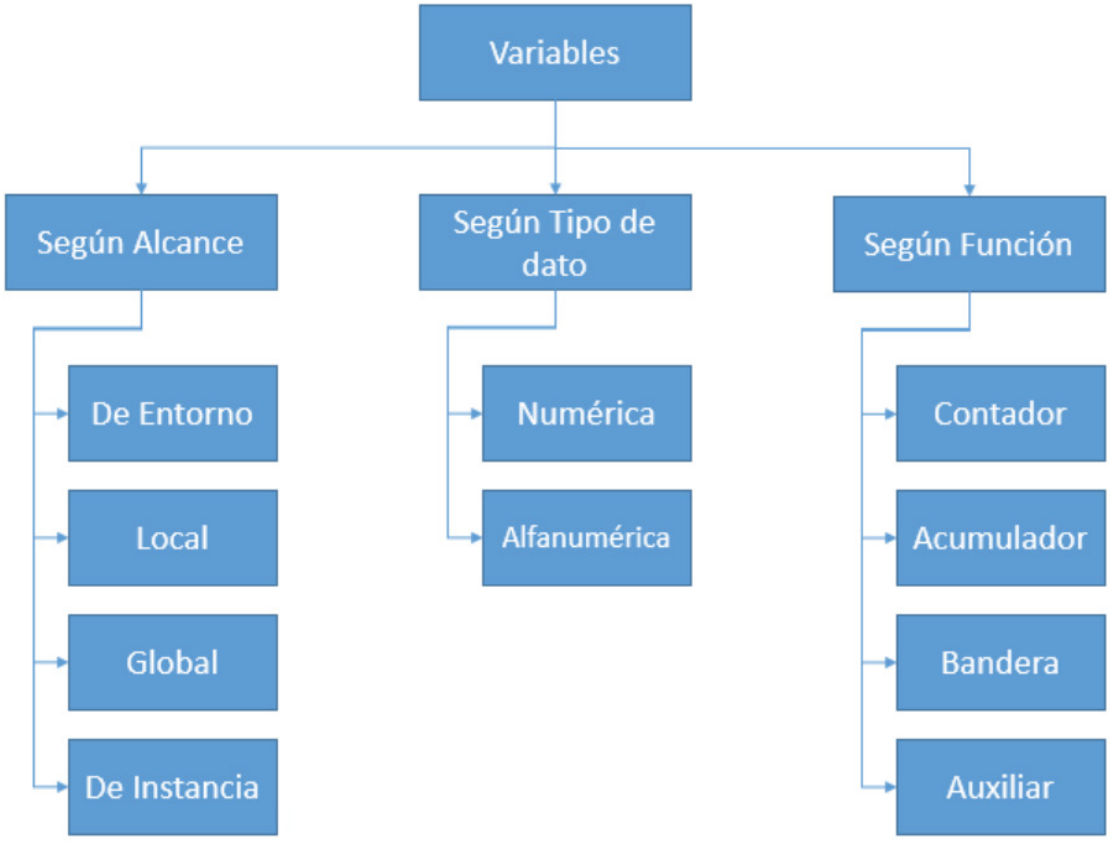


Figura 2. Variables.
Fuente: Propia.

Reglas para nombrar y definir variables

Los nombres de las variables son identificadores y tradicionalmente los identificadores se han formado con caracteres alfanuméricos (esto es, letras y números) incluyendo el carácter de subrayado y excluyendo los blancos y caracteres no anglosajones (como la 'ñ' y las vocales acentuadas).

Otra característica de los identificadores es que no pueden empezar por número, sólo por letra (o subrayado). En algunas notaciones (y lenguajes de programación), las mayúsculas se consideran como caracteres diferentes de las minúsculas mientras que en otros no se hace así; para simplificar las cosas nuestra notación será insensible a mayúsculas.

Reglas de construcción de los identificadores

- Un identificador debe comenzar por una letra
- La primera letra puede ir seguida de:
  - Letras.
  - Dígitos numéricos.
  - Carácter '_'.
- No se permiten espacios en blanco

Ejemplos de identificadores válidos:

v

aceleracion

K

v1

b_n

Pot

Ejemplos de identificadores no válidos:

1n (empieza por número).

año (incluye un carácter no válido, la 'ñ')

aceleración (incluye un carácter no válido, la 'ó').

p v (incluye un espacio en blanco).

En la programación con Java, todas las variables deben ser definidas antes de su uso. La sintaxis para definir una variable es la que sigue:

int i; **// en donde "int" se refiere al tipo de dato e "i" se refiere al nombre de la variable.**

Posterior a su definición, se debe inicializar su valor:

i = 0;

También se puede hacer la inicialización en la misma línea de la definición:

int i = 0;

Además se pueden definir varias variables del mismo tipo en una sola línea:

int i, j, k;

Constante

En programación, las constantes son espacios reservados de memoria en los que se almacena un dato cuyo valor permanece invariable durante todo el proceso. Podría entenderse como un tipo especial de variable en la que no cambiamos su valor. Toda constante debe tener un nombre que la identifique.

Existen dos tipos de constante:

- Numéricas: solo contiene números.

Ejemplos:

PI = 3,14149

Meses = 12

- Alfanuméricas: pueden contener letras, números y caracteres especiales.

Ejemplos:

Ciudad = "Cartagena"

Pais = "Colombia"

Datos

El dato es una representación simbólica (numérica, alfabética, algorítmica, entre otros) de un atributo o característica de una entidad. Los datos describen hechos empíricos, sucesos y entidades. En programación, un dato es la expresión general que describe las características de las entidades sobre las cuales opera un algoritmo.

Tipos de datos

En un sentido amplio, un tipo de dato define un conjunto de valores y las operaciones sobre estos valores

Tipo de dato Entero

Un tipo de dato entero en computación es un tipo de dato que puede representar un subconjunto finito de los números enteros. El número mayor que puede representar depende del tamaño del espacio usado por el dato y la posibilidad (o no) de representar números negativos. Los tipos de dato entero disponibles y su tamaño dependen del lenguaje de programación usado así como la arquitectura en cuestión.

Por ejemplo, si para almacenar un número entero disponemos de 4 bytes de memoria tememos que:

4 Bytes = 4x8 = 32 bits

Con 32 bits se pueden representar 232=4294967296 valores:

Sólo positivos (enteros sin signo): del 0 al 4294967295

Positivos y negativos (enteros con signo): del -2147483648 al 2147483647

Operaciones con enteros

Las típicas operaciones aritméticas: suma, resta, multiplicación y división, se pueden realizar con datos de tipo entero. En el caso de la división, el resultado podría ser un valor real, en ese caso, si el resultado se ha de almacenar como entero la parte decimal del resultado deberá ser eliminada, en principio hay dos métodos para hacerlo:

- El redondeo: Aproximar el valor real al entero más cercano (Ej.: 3,8-->4 / 3,2-->3)
- El truncamiento: Eliminar del valor real la parte decimal (Ej.: 3,8-->3 / 3,2-->3)

Otra operación importante que se puede realizar con número enteros es la operación de módulo o resto de la división entera, es decir:

184 dividido 3 = 61 **(resto 1)** --> 184 módulo 3 = 1

En general la operación módulo cumple que:

a mod **b** = c

- **c** ≥ 0
- **c** < b
- si **c** es igual a 0, entonces **a** es múltiplo de **b**
- si **c** es igual a 0 y **b** es igual a 2, entonces **a** (Es Par)

Los tipos de datos enteros son:

Tipo	Tamaño (Bytes)	Rango
Byte	1	-128 a 127
Short	2	-32768 a 32767
Int	4	-2147483648 a 2147483647
Long	8	-9223372036854775808 a 9223372036854775807

Tabla 1.
Fuente:Propia.

Tipo	Tamaño (Bytes)	Rango
Float	4	1,18E-38 a 3,4E+38
Double	8	2,23E-308 a 1,79E+308

Tabla 2.
Fuente:Propia.

Cuando la precisión que admite un valor real es rebasada el valor de este trunca o se redondea. Por ejemplo si el máximo número de dígitos decimales que puede albergar un tipo real es 10 la siguiente operación:

a = 123,123456789 / 100

Debería dar como resultado que a es igual a 1,23123456789, pero este valor tiene 11 decimales, por lo que el valor de a será uno de estos:

- Truncando: **a** = 1,2312345678
- Redondeando: **a** = 1,2312345679

Tipo de dato Carácter

En terminología informática y de telecomunicaciones, un caracter es una unidad de información que corresponde aproximadamente con un grafema o símbolo, como los de un alfabeto o silabario, de la forma escrita de un lenguaje natural.

Un ejemplo de carácter es una letra, un número o un signo de puntuación. El concepto también abarca a los caracteres de control, que no se corresponden con símbolos del lenguaje natural sino con otros fragmentos de información usados para procesar textos, tales como el retorno de carro y el tabulador.

La única operación que se pueden realizar con los caracteres es la concatenación así: se concatenan dos caracteres **"a"** y **"X",** se obtendría la cadena "aX".

Cadena de caracteres

Con el tipo de dato Carácter es posible tener la representación de una letra, un número, etc. Es decir, se pueden usar variables o constantes que almacenan un valor alfanumérico, pero únicamente es posible almacenar un carácter.

Las denominadas cadenas de caracteres o (String) permite contener en una sola variable más de un carácter, así, con un String podemos almacenar un nombre, un texto, etc. Generalmente se debe incluir dentro de comillas dobles el texto, depende del lenguaje, ejemplo.

Definir una variable para almacenar un nombre:

String Nombre = "Gabriel García Márquez";

Tipo de dato lógico o Boolean

El tipo de dato lógico o booleano es en computación aquel que puede representar valores de lógica binaria, esto es 2 valores, valores que normalmente representan falso o verdadero. Se utiliza normalmente en la programación, estadística, electrónica, matemáticas (Álgebra booleana), etc.

Para generar un dato o valor lógico a partir de otros tipos de datos, típicamente, se emplean los operadores relacionales (u operadores de relación), por ejemplo: 0 es igual a falso y 1 es igual a verdadero

- (3>2)= 1 = verdadero
- (7>9)= 0 = falso

Una vez se dispone de uno o varios datos de tipo booleano, estos se pueden combinar en expresiones lógicas mediante los operadores lógicos (AND, OR, NOT,...). Un ejemplo de este tipo de expresiones sería:

- verdadero **AND** falso --> falso
- falso **OR** verdadero --> verdadero
- **NOT** verdadero --> falso

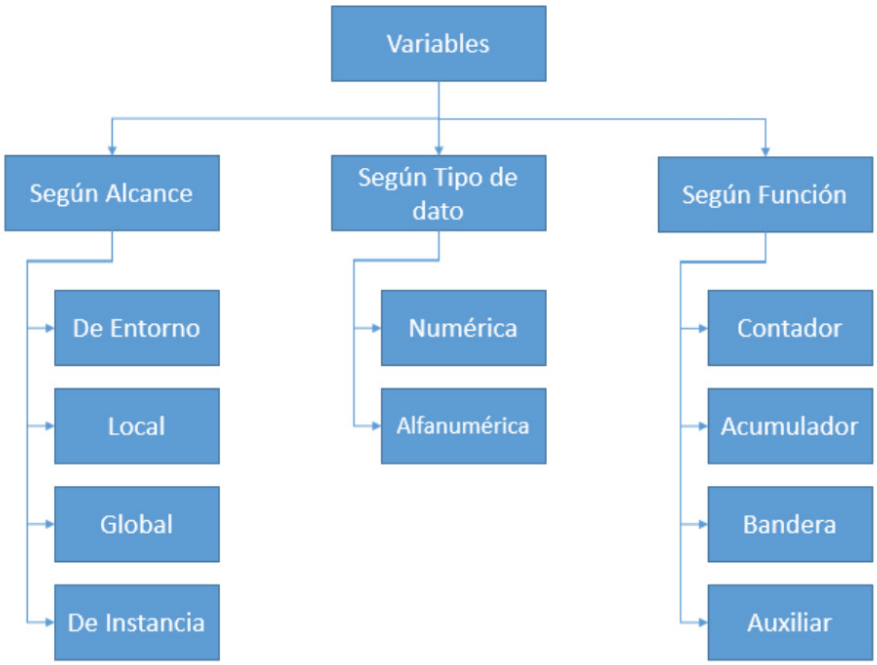


Figura 3. Tipos de datos.
Fuente: Propia.

Expresiones y Operadores

Expresiones

Una expresión es una combinación de variables, constantes y literales de tipos compatibles entre sí, estos elementos están combinados mediante operadores válidos; la expresión más simple está formada por un único elemento (variable, literal o constante) sin ningún operador. Es posible tener expresiones aritméticas (combinando expresiones de tipo entero y/o real), expresiones lógicas (combinando expresiones de tipo lógico) y expresiones de tipo caracter. A continuación se muestran algunas expresiones típicas:

2 * pi * r

v * t

(a>5) and (a<10)

"Sub" + "cadena"

Toda expresión tiene asociada un tipo que se corresponde con el tipo del valor que devuelve la expresión cuando se evalúa, por lo que habrá tantos tipos de expresiones como tipos de datos. Habrá expresiones numéricas y lógicas.

Operadores

Son símbolos que se utilizan para definir las operaciones que se realicen entre los datos, pueden ser:

- Aritméticos.
- Relacionales.
- Lógicos
- Alfanuméricos.

Aritméticos

Se utilizan para realizar operaciones con datos numéricos, formando expresiones aritméticas, tenemos:

+ (Suma)

- (Resta)

* (Multiplicación)

/ (División)

^ (Potenciación)

() (Agrupación de operaciones)

El orden de prioridad en la evolución de los operadores aritméticos es el siguiente:

1. Paréntesis.
2. Potenciación.
3. Multiplicación y división.
4. Suma y resta.

Ejemplo: Si a = 9; b =2; c =5; d = 3

Expresión Aritmética	Resultado
a + b * c	19
(a + b) * c	55
((a + c – b) * b) / (c + d)	3
(c + d) ^ b	64

Tabla 3.
Fuente:Propia.

Relacionales

Se utiliza para relacionar expresiones que al ser evaluadas producen un valor booleano: Verdadero o Falso.

< Menor que

> Mayor que

= Igual a

<= Menor o igual que

>= Mayor o igual que

<> Distinto de

No existen prioridades en el orden de evaluación de los operadores relacionales. Si es necesario establecer alguna prioridad en el orden de evaluación, se deben utilizar los paréntesis.

Ejemplo: si a = 9; b = 2; c = 5; d = 3

Expresión relacional resultado

c < a	Verdadero
a >= c	Verdadero
a – b < d	Falso

d <= c – b Verdadero

d < c – b Falso

También se puede comprar datos alfanuméricos. En este caso se compran uno a uno los caracteres, comenzando desde la izquierda.

Si los datos tienen diferentes longitudes, pero son exactamente iguales hasta el último carácter del más corto, entonces se considera que el más largo es el mayor. El único caso en que son caracteres, en el mismo orden. La misma longitud y los mismos caracteres, en el mismo orden. Las letras minúsculas son mayores que las mayúsculas.

Ejemplo:

Expresión relacional resultado

"E" < "F" Verdadero

"e" < "F" Falso

"NADAR" = "NADAR" Verdadero

"2" < "12" Falso

"B" > "ASTRO" Verdadero

Lógicos

Permiten realizar operaciones con expresiones relacionales, efectuando combinación de condiciones que generan un resultado booleano: verdadero o falso.

Signos	Significado
AND	Producto lógico (y)
OR	Suma lógica (0)
NOT	Negación (NO)

Tabla 4.
Fuente:Propia.

Orden de prioridad en la evaluación de los operadores lógicos:

1. NOT
2. AND

3. 3OR

AND: Es el operador lógico de conjunción. La expresión que se evalúa de cómo resultado verdadero si y solo si todos las condiciones son verdaderas. Dicho de otra manera, es suficiente que unas de las condiciones sea falsa para que el resultado sea falso.

Condición 1 A	Condición 2 B	Resultado A AND B
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

Tabla 5.
Fuente:Propia.

Ejemplo: si a = 9; b = 2; c =5; d = 3

Expresión Lógica	Resultado
(a > b) AND (b<= c) {V} {V}	Verdadero
(C > d) AND (c = a) {V} {F}	Falso
b = (c – d) AND (a + b) > = (c * d) {V} {V}	Verdadero
(c = 5) AND (c >b) AND (a <= d) {F} {F} {F}	Falso
(C = 5) AND (b < c) AND (a >= d) {V} {V} {V}	Verdadero

Tabla 6.
Fuente:Propia.

OR: es el operador lógico de disyunción. La expresión que se evalúa da como resultado falso sí y solo sí todas las condiciones son falsas. Dicho de otra manera, es suficiente que una de las condiciones sea verdadera para que el resultado sea verdadero.

Condición1 A	Condición 2 B	Resultado A OR B
Falso Falso Verdadero Verdadero	Falso Verdadero Falso Verdadero	Falso Verdadero Verdadero Verdadero

Tabla 7.
Fuente:Propia.

Ejemplo: si a = 9; b = 2; c =5; d = 3

Expresión Lógica	Resultado
(a > b) AND (b<= c) {V} {V}	**Verdadero**
(C > d) AND (c = a) {V} {F}	**Falso**
b = (c – d) AND (a + b) > = (c * d) {V} {V}	**Verdadero**
(c = 5) AND (c >b) AND (a <= d) {F} {F} {F}	**Falso**
(C = 5) AND (b < c) AND (a >= d) {V} {V} {V}	**Verdadero**

Tabla 8.
Fuente:Propia.

NOT: es el operador lógico de negación. Afecta a una sola expresión, a cambio su estado lógico: si era falso se convierte en verdadero y viceversa.

Condición 1 A	Resultado Not A
Falso Verdadero	Verdadero Falso

Tabla 9..
Fuente:Propia

Ejemplos: A =9, B: 2, C: 5, D: 3.

Expresión lógica	Resultado
NOT (A < B) {F}	VERDADERO
NOT ((B + D) > C {F}	FALSO
NOT (C = D) {V}	VERDADERO

Tabla 10..
Fuente:Propia

Alfanuméricos

Se utilizan para unir datos alfanuméricos.

Signo	Signicado
+	Concatenacion

Tabla 11..
Fuente:Propia

Ejemplos: si nombre = "JUAN" Y APELLIDO = "PEREZ"

Expresion alfanumerica	Resultado
NOMBRE + APELLIDO	"JUAN PEREZ"
"NOMBRE: + " " + APELLIDO	"JUAN PEREZ"
"5 " + "17"	"517"

Tabla 12..
Fuente:Propia

Orden de evaluación (Reglas de prioridad)

El orden de evaluación de los operadores en una expresión es el siguiente:

1. Paréntesis (comenzando por lo más internos).
2. Potencias.
3. Productos y divisiones.
4. Sumas y restas.
5. Concatenación.

6. Relacionales.
7. NOT.
8. AND.
9. OR.

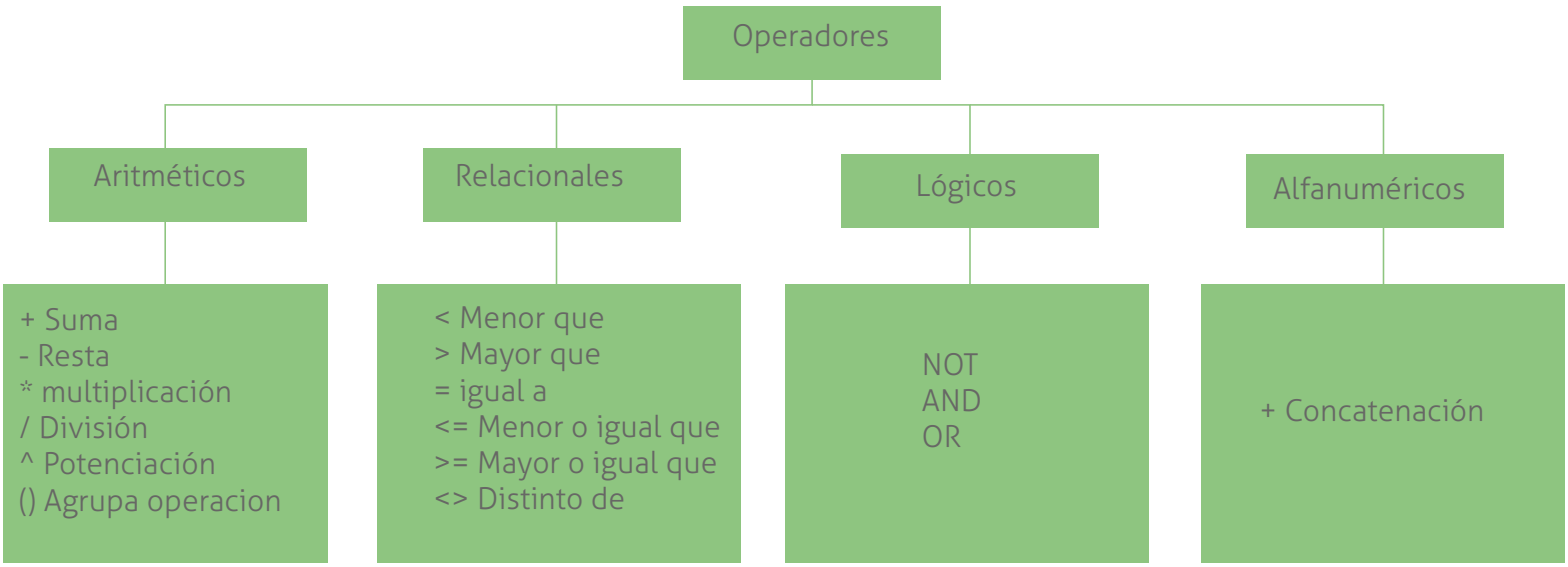
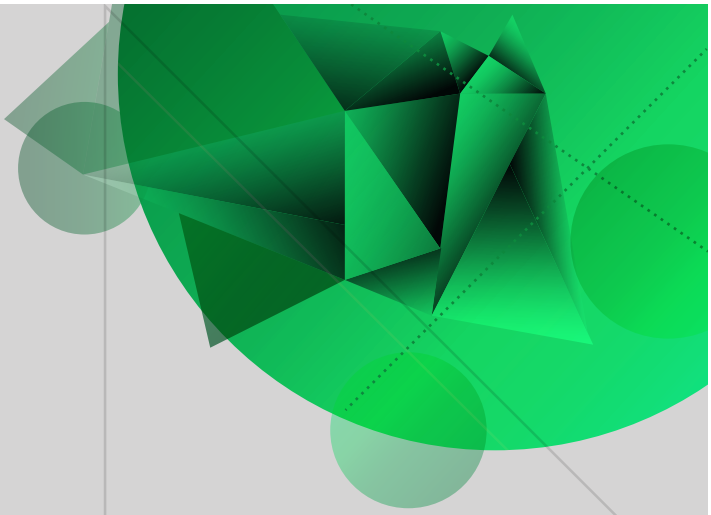


Figura 4. Operadores.
Fuente: Propia.

3

Unidad 3

Ciclos y condicionales



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

En la programación tradicional, las sentencias se ejecutan sucesivamente una tras otra. Esto define un camino o dirección según la cual se va desarrollado el programa. Sin embargo, habrá momentos en que el programa deba ejecutar determinadas partes dependiendo del estado en el que se halle el programa o de las variables externas. Esto permitir modificar el orden de la ejecución para adaptarse al estado del programa y bifurcar hacia nuevas subrutinas cuando se cumplan ciertas condiciones, que el programador fijar de antemano.

Para realizar este tipo de programas son necesarias las estructuras de control de flujo, que son instrucciones del tipo "si se cumple esta condición, hazlo esto; si no se cumple, haz esto otro". También existen instrucciones del tipo "repite esto mientras se cumpla esta condición".

Si se utilizan estructuras de control de flujo, los programas dejan de ser una sucesión lineal de instrucciones para convertirse en programas inteligentes que pueden tomar decisiones en función del valor de las variables.

U3 Metodología

Se invita al estudiante a leer completamente la guía de manera que logre obtener una sólida y consistente fundamentación teórica, de igual forma, se le recomienda analizar los ejemplos planteados, siguiendo paso a paso el desarrollo a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participación en los foros y formular todas las dudas que pueda tener en relación al tema.

U3

Desarrollo temático

Sentencias de control de flujo

¿Qué es controlar el flujo?

Controlar el flujo es determinar el orden en el que se ejecutarán las instrucciones en nuestros programas. Si no existiesen las sentencias de control entonces los programas se ejecutarían de forma secuencial, empezarían por la primera instrucción e irían una a una hasta llegar a la última.

Pero, obviamente este panorama sería muy malo para el programador. Por un lado, en sus programas no existiría la posibilidad de elegir uno de entre varios caminos en función de ciertas condiciones. Y por el otro, no podrían ejecutar algo repetidas veces, sin tener que escribir el código para cada una.

Para estos problemas tenemos tres soluciones: las sentencias de control condicionales, los bucles y las sentencias de salto. Estos conjuntos de sentencias forman el grupo de las sentencias estructuradas. Y se les llama estructuradas porque a diferencia de las simples pueden contener en su cuerpo otras sentencias.

Las sentencias de control permiten seleccionar uno de entre varios caminos por donde seguirá la ejecución del programa. En algunos casos esta selección viene determinada por la evaluación de una expresión lógica.

Este tipo de sentencias se dividen en:

La sentencia IF, ELSE y ELSE IF

La sentencia SWITCH

A los bucles se les conoce también como sentencias iterativas ya que permiten realizar algo varias veces (repetir, iterar). Dentro de ellas distinguimos tres:

La sentencia FOR

La sentencia WHILE

La sentencia DO WHILE

Las sentencias de salto nos permiten interrumpir la ejecución del programa y pasar "Saltar" a otra parte del mismo en cualquier momento en que se cumpla determinada condición, para este caso tenemos:

La sentencia GOTO

La sentencia BREAK

Condicionales

Las estructuras condicionales surgen en casos donde necesitamos que la computadora tome una decisión.

Con la sentencia **IF,** podemos tomar una decisión.

Con la sentencia **ELSE,** hacemos algo si no ocurre el IF.

Con la sentencia **ELSE IF,** podemos elegir una opción entre varias.

Condicional IF

Es una estructura simple que realizar una acción si y solo si se cumple una condición determinada.

Estructura de programación:

```
If (Comparación Booleana) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;
  }
```

El siguiente ejemplo determina si una persona es mayor de edad o no, para determinarlo, solicita ingresad la edad y la compara con 18 años, si el valor ingresado es mayor o igual a 18 indica que la persona es mayor de edad, en caso contrario no hace nada.

```
public class Ejemplo1 {
      public static void main(String[] arg) {
         teclado=new Scanner(System.in);
         int edad;
         System.out.print("Ingrese el edad:");
         edad=teclado.readline();
         if (edad>=18) {
              System.out.println("Esta persona es mayor de edad");
         }
   }
}
```

Condicional ELSE

Se trata de una estructura compuesta que realiza una comparación inicial, si la primera condición no se cumple, automáticamente se ejecuta la segunda secuencia de instrucciones.

Estructura de programación:

```
If (Comparación Booleana) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;

  } else {
        Instruccion1';
        Instruccion2';
        …
        InstruccionN';

  }
```

El siguiente ejemplo compara 2 números enteros ingresados por teclado e indica cual es mayor, nótese que únicamente es necesaria una comparación, pues en caso de no cumplirse la primera, resulta que la segunda es correcta.

```
public class Ejemplo2 {
   public static void main(String[] arg) {
         teclado=new Scanner(System.in);
         int num1,num2;
         System.out.print("Ingrese primer valor:");
         num1=teclado.readline();
         System.out.print("Ingrese segundo valor:");
```

```
        num2=teclado.readline();
        if (num1>num2) {
                System.out.print("el número mayor es: "+num1);
        } else {
                System.out.print("el número mayor es: "+num2);
        }
    }
}
```

Condicional ELSE IF

Se trata de una estructura compuesta que realiza una comparación inicial, si la primera condición no se cumple, realiza una segunda comparación y continua así sucesivamente hasta verificar todas las comparaciones programadas hasta que alguna se cumpla.

Estructura de programación:

```
If (Comparación Booleana1) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;
   } else if (Comparación Booleana2) {
        Instruccion1';
        Instruccion2';
        …
        InstruccionN';
   }
} else if (Comparación Booleana3) {
        Instruccion1";
        Instruccion2";
        …
        InstruccionN";
   }
…
} else if (Comparación BooleanaN) {
```

Instruccion1‴;
Instruccion2‴;
…
InstruccionN‴;

}

El siguiente ejemplo representa una estructura condicional compuesta para realizar la operación matematice indicada por la variable char, para este ejemplo, la operación seleccionada se ingresa por teclado.

```
public class Ejemplo3 {
  public static void main(String[] arg){
    teclado=new Scanner(System.in);
    int a = 1;
    int b = 1;
    char op = teclado.readline();
    if ( op == '+' ) {
       System.out.println( a + b);
    } else if ( op == '-') {
       System.out.println( a - b);
    } else if ( op == '*') {
       System.out.println( a * b);
    } else if ( op == '/') {
       System.out.println( a / b);
    }
  }
}
```

Repaso sobre los operadores de igualdad y relacionales

Los operadores de igualdad y relacionales determinan si un operando es mayor, menor, igual a o distinto de otro. Probablemente la mayoría de estos operadores también le serán familiares. Tenga en mente que debe utilizar «==», no «=» cuando compruebe si dos valores primitivos son iguales.

== igual a

!= distinto de

> mayor que

>= mayor que o igual a

< menor que

<= menor que o igual a

Repaso sobre los operadores condicionales

Los operadores && y || realizan las operaciones *AND-Condicional* y OR-Condicional sobre dos expresiones booleanas. Estos operadores muestran un comportamiento de «cortocircuito», lo que significa que el segundo operando solamente se evalúa si es necesario (Estos operadores se pueden emplear con IF, ELSE o ELSE IF).

&& AND-Condicional

|| OR-Condicional

Estructura de programación para AND:

If (Comparación Booleana1) && (Comparación Booleana1) {

Instruccion1;
Instruccion2;
…
InstruccionN;

}

Estructura de programación para OR:

```
If (Comparación Booleana1) || (Comparación Booleana1) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;
  }
```

Sentencia SWITCH

La estructura SWITCH es una estructura de control empleada en programación, a similar a los condicionales IF, ELSE o ELSE IF, SWITCH no es tan popular como el IF, pero se utiliza con regularidad en la programación. En principio la funcionalidad de un SWITCH también se puede implementar con múltiples IF anidados. En el caso de que haya muchas acciones dependientes de muchos valores iniciales, es recomendable su uso. El SWITCH favorece la legibilidad y rapidez en la programación.

Estructura de programación:

```
switch (variable){
 case valor1: accion1;
 case valor2: accion2;
 …
 case valorN: accionN;

 default: accionD;
}
```

El siguiente ejemplo muestra el nombre correspondiente al mes ingresado numéricamente por teclado.

```
public class Ejemplo4 {
   public static void main(String[] arg) {
      teclado=new Scanner(System.in);
      int month = tecaldo.readline();
      switch (month) {
         case 1:  System.out.println("Enero"); break;
         case 2:  System.out.println("Febrero"); break;
         case 3:  System.out.println("Marzo"); break;
         case 4:  System.out.println("Abril"); break;
```

```
        case 5:  System.out.println("Mayo"); break;
        case 6:  System.out.println("Junio"); break;
        case 7:  System.out.println("Julio"); break;
        case 8:  System.out.println("Agosto"); break;
        case 9:  System.out.println("Septiembre"); break;
        case 10: System.out.println("Octubre"); break;
        case 11: System.out.println("Noviembre"); break;
        case 12: System.out.println("Diciembre"); break;
         default: System.out.println("El mes no existe");
      }
   }
}
```

Bucles

Ciclo FOR

Se trata de una secuencia de repetición de un determinado grupo de instrucciones lo que facilita enormemente los cálculos y operaciones iterativas en la programación.

Estructura de programación:

```
for (inicialización; condición; incremento) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;
  }
```

El siguiente ejemplo imprime los primeros 1000 números naturales

```
public class Ejemplo5 {
   public static void main(String[] arg) {
      int i;
      for (i=0;i<1000;i++) {
           System.out.println(i);
      }
```

Ciclo WHILE

Se diferencia del FOR en que este es más universal en el sentido que solo se preocupa por evaluar una expresión y mientras dicha expresión sea verdadera se encarga de repetir el bloque delimitado en el WHILE. La expresión pude ser un valor lógico, una función que retorne un valor lógico, una expresión relacional o una expresión lógica.

Estructura de programación:

```
while (Comparación Booleana1) {
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;

  }
```

El siguiente ejemplo imprime los primeros 1000 números naturales

```
public class Ejemplo6 {
   public static void main(String[] arg) {
      int i=0;
      while (i<1000) {
           System.out.println(i);
         i = i + 1;

      }
```

Ciclo DO WHILE

Es similar a WHILE, solo que este primero ejecuta y luego comprueba la condición.

Estructura de programación:

```
do
{
        Instruccion1;
        Instruccion2;
        …
        InstruccionN;

  } while (Comparación Booleana1)
```

El siguiente ejemplo imprime los primeros 1000 números naturales

```
public class Ejemplo7 {
   public static void main(String[] arg) {
      int i=0;
      do{
           System.out.println(i);
         i = i + 1;
      } while (i<1000)
```

Sentencias de salto

Sentencia GOTO

Esta función nos permite saltar a una línea del código en la que haya una etiqueta. Las etiquetas se definen con el nombre y dos puntos :

Vamos a ver cómo funciona.

Estructura de programación:

```
If (Comparación Booleana1)
{
        Goto (cabecera o función)

}
```

En el siguiente ejemplo vemos que al pulsar la letra "r", el programa volverá al inicio de su ejecución:

```
public class Ejemplo8 {
   public static void main(String[] arg) {
      char letra;
      printf("\nPulsa r para reiniciar: ");
      scanf("%c", &letra);
      if (letra != 'r')
        goto Ejemplo6;
}
```

Sentencia BREAK

La instrucción break nos sirve para salir de cualquier bloque de código, ya sea un if, un for, o lo que sea:

Estructura de programación:

"Condicional o bucle" (Comparación Booleana1)

{

Instruccion1;

Instruccion2;

…

InstruccionN;

…

break; // Puede ir en cualquier lugar del programa

}

El siguiente ejemplo muestra un bucle infinito, del que únicamente se puede pulsando la tecla "s".

```
public class Ejemplo9 {
   public static void main(String[] arg) {
      int i=0;
      while(1) //Esto provoca un bucle infinito.
      {
        char a;
        scanf("%c", &a);
        if(a == 's')
        {
          break;
        }
        else
        {
          print("Para salir del bucle pulsa la tecla s\n");
        }
   }
   printf("Pulsaste la tecla s\n");

}
```

Funciones

Supongamos que necesitamos escribir un programa que calcule el número combinatorio C(m,n), definido como:

C(m,n)=m!(m−n)!n!,

donde n! (el factorial de n) es el producto de los números enteros desde 1 hasta n:

n!=1·2 ·····((n−1)·n=∏i=1ni

El código para calcular el factorial de un número entero n es sencillo:

```
f = 1
for (i = 1, i = n+1; i++):

f = f * i
```

Sin embargo, para calcular el número combinatorio, hay que hacer lo mismo tres veces:

```
comb = 1
// multiplicar por m!
f = 1
for (i=1,i= m + 1;i++):
   f = f * i
comb = comb * f
// dividir por (m - n)!
f = 1
for i in
for (i = 1, i = m-n+1; i++):
   f = f * i
comb = comb / f
// dividir por n!
f = 1
for (i = 1, i = n+1; i++):
   f = f * i

comb = comb / f
```

La única diferencia entre los tres cálculos de factoriales es el valor de término de cada ciclo for (m, m - n y n, respectivamente). Escribir el mismo código varias veces es tedioso y propenso a errores. Además, el código resultante es mucho más difícil de entender, pues no es evidente a simple vista qué es lo que hace.

Lo ideal sería que existiera una función llamada factorial que hiciera el trabajo sucio, y que pudiéramos usar de la siguiente manera:

factorial(m) / (factorial(m - n) * factorial(n))

Funciones

En programación, una función es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:

- Los parámetros, que son los valores que recibe la función como entrada.
- El código de la función, que son las operaciones que hace la función.
- El resultado (o valor de retorno), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

En el ejemplo del factorial, el parámetro es el entero al que queremos calcularle el factorial, el código es el ciclo que hace las multiplicaciones, y el resultado es el valor calculado.

Definición de funciones

En Java las funciones son creadas con la siguiente estructura:

```
tipo_resultado nombre_función (tipo_parámetro nombre_parámetro, ... ) {
  sentencia1;
  sentencia2;
  ...
  sentenciaN;
}
```

Los parámetros son variables en las que quedan almacenados los valores de entrada.

La función contiene código igual al de cualquier programa. La diferencia es que, al terminar, debe entregar su resultado usando la sentencia return.

Por ejemplo, la función para calcular el factorial puede ser definida de la siguiente manera:

```
int factorial(n) {
   f = 1
   for (i = 1, i = n+1; i++):
      f = f * i;
   return f
}
```

En este ejemplo, el resultado que entrega una llamada a la función es el valor que tiene la variable f al llegar a la última línea de la función.

Una vez creada, la función puede ser usada como cualquier otra, todas las veces que sea necesario:

```
>>> factorial(0)
1
>>> factorial(12) + factorial(10)
482630400
>>> factorial(factorial(3))

720
```

Las variables que son creadas dentro de la función (incluyendo los parámetros y el resultado) se llaman variables locales, y sólo son visibles dentro de la función, no desde el resto del programa.

Por otra parte, las variables creadas fuera de alguna función se llaman variables globales, y son visibles desde cualquier parte del programa. Sin embargo, su valor no puede ser modificado, ya que una asignación crearía una variable local del mismo nombre.

Ejemplo, la siguiente función recibe una cantidad de segundos, y retorna el equivalente en horas, minutos y segundos:

```
float convertir_segundos(segundos) {
   horas = segundos / (60 * 60);
   minutos = (segundos / 60) % 60;
   segundos = segundos % 60;
   return (horas, minutos, segundos);

}
```

Al llamar la función, se puede asignar un nombre a cada uno de los valores retornados:

```
>>> h, m, s = convertir_segundos(9814)

2 h, 43m, 34s
```

Técnicamente, la función está retornando una tupla de valores, un tipo de datos que veremos más adelante:

```
>>> convertir_segundos(9814)

(2, 43, 34)
```

Funciones que no retornan nada

Una función puede realizar acciones sin entregar necesariamente un resultado, por ejemplo, si un programa necesita imprimir cierta información muchas veces, conviene encapsular esta acción en una función que haga los print

```
void imprimir_datos(nombre, apellido, rol, dia, mes, anno) {
   print 'Nombre completo:', nombre, apellido;
   print 'Rol:', rol;
   print 'Fecha de nacimiento:', dia, '/', mes, '/', anno;
}
imprimir_datos('Perico', 'Palotes', '201101001-1',  3, 1, 1993)
imprimir_datos('Juan', 'Vinagre',  '201101002-2', 10, 9, 1992)

imprimir_datos('Fulano', 'De Tal', '201101003-3', 14, 5, 1990)
```

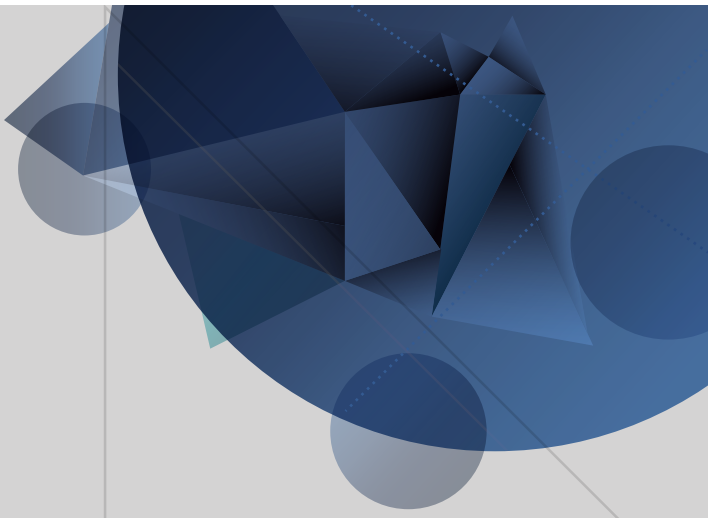
En este caso, cada llamada a la función imprimir_datos muestra los datos en la pantalla, pero no entrega un resultado. Este tipo de funciones son conocidas en programación como procedimientos o subrutinas.

Técnicamente, todas las funciones retornan valores. En el caso de las funciones que no tienen una sentencia return, el valor de retorno siempre es void. Pero como la llamada a la función no aparece en una asignación, el valor se pierde, y no tiene ningún efecto en el programa.

4

Unidad 4

Diseño de algoritmos



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

El desarrollo de algoritmos es un tema fundamental en el diseño de programas, por lo cual el estudiante debe tener buenas bases que le sirvan para poder desarrollar de manera fácil y rápida sus programas.

Estos apuntes servirán de apoyo a los estudiantes y les facilitará desarrollar su capacidad analítica y creadora para, de esta manera, mejorar su destreza en la elaboración de algoritmos que sirven como principio para la codificación de los diferentes programas que tendrá que desarrollar a lo largo de su carrera.

El diseño de soluciones a la medida de nuestros problemas, requiere como en otras disciplinas una metodología que nos enseñe de manera gradual, la forma de llegar a estas soluciones.

Para poder realizar programas, además de conocer la metodología mencionada, también debemos conocer, de manera específica, las funciones que puede realizar el computador y las formas en que se pueden manejar los elementos que hay en el mismo.

U4 Metodología

Se invita al estudiante a leer completamente la guía, de manera que logre alcanzar gran solidez en cuanto al tema de diseño estructurado de algoritmos. dDe igual forma, se le recomienda analizar los ejemplos planteados, siguiendo paso a paso el desarrollo, a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participar en los foros y formular todas las dudas que pueda tener en relación al tema tratado.

U4 Desarrollo temático

Diseño estructurado de un algoritmo

El estilo de programación está definido particularmente por cada programador. Sin embargo, es muy importante tener en cuenta que un programa legible, ordenado y comprensible es más fácil de corregir, entender y mantener.

Pautas para el diseño de algoritmos

1. Sangrado: en cada estructura, y alineando las instrucciones (sentencias) dentro de cada una de ellas y dentro de todo el algoritmo (Comienzo, Fin).
2. Líneas en blanco: dejarlas entre partes importantes o que estén lógicamente separadas. Recomendable luego de cada estructura (Repetitiva o Selectiva), luego de declaración de variables.
3. Comentarios: parte importante de la documentación de un programa que permite mayor comprensión del mismo. Para comentarios en Pseudocódigos, usaremos { }.
4. Nombres significativos de identificadores: que representen aquello que estamos tratando. Si son palabras compuestas usar guion común. Todos deben comenzar con una letra.
5. Cada sentencia en una línea distinta: al colocar una nueva sentencia comenzar una nueva línea, incluso las palabras claves de las estructuras en líneas separadas.
6. Espacios entre elementos de una sentencia: lo hace más legible. Por ejemplo: suma ← suma + numero

Ejemplos

1) Suponga que un individuo desea invertir su capital en un banco y desea saber cuánto dinero ganará después de un mes, si el banco paga a razón de 2% mensual.

```
Comienzo
  Leer ( capital )
  ganancia ←capital * 0.02
  Mostrar ( ganancia )

Fin
```

2) Una tienda ofrece un descuento del 15% sobre el total de la compra y un cliente desea saber cuánto deberá pagar finalmente por su compra.

```
Comienzo
  Leer ( total-compra)
  descuento← total-compra * 0.15
  total-a-pagar← total-compra - descuento
  Mostrar ( total-a-pagar )

Fin
```

3) Un maestro desea saber qué porcentaje

de hombres y qué porcentaje de mujeres hay en un grupo de estudiantes.

```
Comienzo
  Leer ( nun-hombres )
  Leer ( num-mujeres )
  total-estudiantes ←nun-hombres + num-mujeres
  porcen-hombres← nun-hombres * 100 / total-estudiantes
  porcen-mujeres  ← num-mujeres * 100 / total-estudiantes
  Mostrar ( porcen-hombres )
 Mostrar (porcen-mujeres )

Fin
```

Uso de estructuras selectivas

Razonemos a partir de un ejemplo. Realizar un algoritmo que permita ingresar un número e informe "Número mayor a cero" o "Número menor o igual a cero" según corresponda.

Análisis del problema. El algoritmo deberá:

1. Leer el número que ingresa el usuario
2. Seleccionar mediante una condición si ese número es o no mayor a cero
3. Informar con una de las leyendas enunciadas

Desarrollo del algoritmo en pseudocódigo

```
COMIENZO
        {Declaro las variables a utilizar}
        NUMERO
        {Ingreso de datos}
        MOSTRAR ("Ingrese un número:")
        LEER (NUMERO)
        {Proceso de selección}
        SI ( NUMERO > 0)
                ENTONCES
                        MOSTRAR("Número mayor a cero")
                FIN-ENTONCES
                SINO
                        MOSTRAR("Número menor o igual a cero")
        FIN-SI

FIN
```

Ejemplos

1) Determinar si un estudiante aprueba o reprueba un curso, sabiendo que aprobará si su promedio de tres calificaciones es mayor o igual a 7; reprueba en caso contrario.

```
Comienzo
 Leer ( nota1 )
 Leer ( nota2 )
 Leer ( nota3 )

  prom ← nota1 +nota2 + nota3 )/3
  Si ( prom >= 7 )
entonces
                Mostrar ( "estudiante aprobado" )
        Fin-entonces
si no
                Mostrar ( "estudiante reprobado" )
  Fin-si

Fin
```

2) En un almacén se hace un 20% de descuento a los clientes cuya compra supere los $1000 ¿Cuál será la cantidad que pagará una persona por su compra?

```
Comienzo
  Leer ( compra )
  Si ( compra > 1000 )
entonces
                desc  ← compra * 0.20
        fin-entonces
si no
                desc  ← 0
  fin-si
  tot_pag ←compra - desc
  Mostrar ( tot_pag )

fin.
```

3) Un obrero necesita calcular su salario semanal, el cual se obtiene de la siguiente manera:

Si trabaja 40 horas o menos, se le paga $16 por hora.

Si trabaja más de 40 horas, se le paga $16 por cada una de las primeras 40 horas y $20 por cada hora extra.

```
Comienzo
  Leer (horas-trab )
  Si ( horas-trab > 40 )
entonces
        horas-extras ← horas-trab - 40
        salario-sem ← horas-extras * 20 + 40 * 16
fin-entonces
si no
                salario-sem ← horas-trab * 16
  Fin-si
  Mostrar ( salario-sem )
Fin
```

4) Hacer un algoritmo que lea dos números y los imprima en forma ascendente.

```
Comienzo
Leer ( num1 )
Leer ( num2 )
 Si ( num1 < num2 )
    entonces
        Mostrar ( num1, num2 )
      Fin-entonces
      si no
            Mostrar ( num2, num1 )
   fin-si
fin
```

5) Hacer un algoritmo que calcule el total a pagar por la compra de camisas. Si se compran tres camisas o más, se aplica un descuento del 20% sobre el total de la compra y si son menos de tres camisas, un descuento del 10%.

```
Comienzo
  Leer ( num_camisas )
  Leer ( prec )
```

```
        tot_comp ←num_camisas * prec
        Si ( num_camisas > = 3 )
entonces
                    tot_pag ← tot_comp - tot_comp * 0.20
            fin-entonces
si no
                    tot_pag←tot_comp - tot_comp * 0.10
        fin-si
        Mostrar ( tot_pag )

    fin
```

6) Leer 2 números; si son iguales que los multiplique, si el primero es mayor que el segundo que los reste y si no, que los sume.

```
    Comienzo
      Leer ( num1 )
      Leer ( num2 )
      si ( num1 = num2 )
      entonces
                resul ←num1 * num2
      fin-entonces
     si no
          si ( num1 > num2 )
                entonces
                      resul← num1 - num2
                fin-entonces
                si no
                      resul ←num1 + num2
                fin-si
      fin-si
    fin
```

7) Leer tres números diferentes e imprimir el número mayor de los tres.

```
    Comienzo
      Leer ( num1 )
       Leer ( num2 )
```

```
        Leer ( num3 )

        Si ( (num1 > num2) and (num1 > num3) )
entonces
                mayor ← num1
            fin-entonces
si no
                Si ( (num2 > num1) and (num2 > num3) )
entonces
                mayor ← num2
            fin-entonces
si no
                mayor ←num3
                fin-si
        fin-si
        Mostrar ( mayor )
      fin
```

Uso de estructuras repetitivas

Realizar un algoritmo que permita realizar 3 veces lo siguiente: ingresar 2 números, calcular su suma e informar el resultado.

El algoritmo deberá:

1. Leer el número que ingresa el usuario
2. Efectuar la suma y almacenarla
3. 3. Mostrar el resultado almacenado

Para efectuar esto una sola vez bastaría con el siguiente algoritmo:

```
COMIENZO
    { Ingreso de datos}
    MOSTRAR ("Ingrese primer número:")
    LEER (NUMERO1)
    MOSTRAR ("Ingrese segundo número:")
    LEER (NUMERO2)
    { Proceso de suma}
    RESULTADO ←NUMERO1 + NUMERO2
```

{ Informe por pantalla}
MOSTRAR (“El resultado de la suma es:”)
MOSTRAR (RESULTADO)

FIN

Pero el algoritmo anterior solo efectúa una vez la suma de dos números, para que pueda hacerlo más veces deberíamos repetir las sentencias de entrada, proceso y salida de datos tantas veces como se solicite.

{Ingreso de datos}
MOSTRAR (“Ingrese primer número:”)
LEER (NUMERO1)
MOSTRAR (“Ingrese segundo número:”)
LEER (NUMERO2)
{Proceso de suma}
RESULTADO ← NUMERO1 + NUMERO2
{Informe por pantalla}
MOSTRAR (“El resultado de la suma es:”)
MOSTRAR (RESULTADO)

Para realizar esto utilizamos una estructura repetitiva que ejecute 3 iteraciones

COMIENZO

{Inicialización de variables}
CONTADOR ← 0

{La variable contador debe tener valor inicial para que sepa a partir de qué valor comienza a contar}

MIENTRAS (CONTADOR < 4) HACER

{Ingreso de datos}
MOSTRAR (“Ingrese primer número:”)
LEER (NUMERO1)
MOSTRAR (“Ingrese segundo número:”)
LEER (NUMERO2)
{Proceso de suma}
RESULTADO ← NUMERO1 + NUMERO2
{Informe por pantalla}
MOSTRAR (“El resultado de la suma es:”)

MOSTRAR (RESULTADO)

{Cuento la realización de las operaciones 1 vez más}

CONTADOR← CONTADOR + 1

FINMIENTRAS

FIN

Combinación de estructuras selectivas y repetitivas

Realizar un algoritmo que permita ingresar 5 números e informe cuantos números ingresados son mayores a cero y cuantos menores a cero.

El algoritmo deberá:

1. Leer el número que ingresa el usuario
2. Comparar el valor del número ingresado con cero, en caso de ser mayor deberá contar uno más de la cantidad que tenía
3. Comparar el valor del número ingresado con cero, en caso de ser menor deberá contar uno más de la cantidad que tenía
4. Informar con una de las leyendas enunciadas

Para realizar los pasos 2 y 3 se debe usar estructuras SELECTIVAS. La lectura de números como las distintas comparaciones se deberán ejecutar 5 veces, por lo tanto es necesaria una estructura REPETITIVA que abarque esas sentencias. El informe solo se realiza una vez.

COMIENZO

{Inicialización de variables}

CONTAR ← 0 {variable que contará hasta 5 para controlar la cantidad de números ingresados}

MAYOR ←0 {variable que contará la cantidad de números mayores a cero}

MENOR ← 0 {variable que contará la cantidad de números menores a cero}

{Ingreso de datos y proceso de selección}

MIENTRAS (CONTAR < 5) HACER

MOSTRAR (“Ingrese un número:”)

LEER (NUMERO)

{ Proceso de selección}

{Utilizamos 2 SI pues en caso de ingresarse un CERO no debemos contarlo}

SI (NUMERO > 0)

ENTONCES

```
                                MAYOR← MAYOR +1 {cuenta un número más mayor a cero}
        FINSI

        SI ( NUMERO < 0)
                ENTONCES
MENOR ← MENOR +1 {cuenta un número más menor a cero}
FINSI
                CONTAR ←CONTAR +1 {cuenta un número más ingresado}
        FINMIENTRAS
        {Informe final del proceso}
        MOSTRAR("Los números mayores a cero fueron:" )
        MOSTRAR( MAYOR)
        MOSTRAR("Los números menores a cero fueron:")
        MOSTRAR( MENOR)
FIN
```

Ciclos con un Número Determinado de Iteraciones (Para)

Son aquellos en que el número de iteraciones se conoce antes de ejecutarse el ciclo. La forma de esta estructura es la siguiente:

```
Para (variable ← desde  a hasta incremento) Hacer

      Accion1

      Accion2

      …

      AccionN

Fin-para
```

Dónde:

Variable: Variable de control del ciclo (un contador)

Desde: Límite inferir

Hasta: Límite superior

Incremento: indica de qué modo se incrementa la variable de control (de 1 en 1, de 2 en 2, etc.)

En este ciclo la variable de control toma el valor inicial del ciclo y el ciclo se repite hasta que la variable de control llegue al límite superior. La cantidad de repeticiones que tenga depende del límite superior y del incremento de la variable.

Ejemplo PARA (FOR)

1) Calcular el promedio de un estudiante que tiene 7 calificaciones en la materia de Diseño Estructurado de Algoritmos

```
Comienzo
  Sum←0
  Leer (Nom)
  Para (c ← 1 a 7 ) Hacer
        Leer calif
        Sum ← sum + calif
  Fin-para
  prom ←sum /7
  Mostrar (prom)

Fin.
```

2) Leer 10 números y obtener su cubo y su cuarta.

```
        Comienzo
        Para ( n← 1 a 10 ) hacer
              Leer ( num )
              cubo ←num * num * num
              cuarta← cubo * num
              Mostrar ( cubo )
              Mostrar ( cuarta )
          Fin-para

        Fin.
```

3) Leer 10 números e imprimir solamente los números positivos

```
        Comienzo
        Para ( n ← 1 a 10 ) Hacer
              Leer  ( num )
              Si ( num > 0 ) entonces
                    Mostrar  ( num )
              fin-si
          Fin-para
        Fin.
```

4) Leer 15 números negativos y convertirlos a positivos e imprimir dichos números.

```
Comienzo
 Para ( x ← 1 a 15 ) Hacer
        Leer ( num )
        pos ← num * -1
        Mostrar ( num, pos )
  Fin-para

Fin.
```

5) Suponga que se tiene un conjunto de calificaciones de un grupo de 40 estudiantes. Realizar un algoritmo para calcular la calificación media y la calificación más baja de todo el grupo.

```
Comienzo
  sum← 0
  baja ← 9999
 Para ( a← 1 a 40 ) Hacer
        Leer ( calif )
        sum ← sum + calif
        Si ( calif < baja ) entonces
                baja ← calif
        fin-si
  Fin-para
  media ←sum / 2
  Mostrar ( media, baja )
fin
```

6) Calcular e imprimir la tabla de multiplicar de un número cualquiera. Imprimir el multiplicando, el multiplicador y el producto.

```
Comienzo
  Leer ( num )
  Para ( X ← 1 a 10 ) Hacer
        resul ← num * x
        Mostrar ( num, " * ", X, " = ", resul )
  Fin-para
fin.
```

Algoritmos de ejemplo

Ejemplo de uso de contadores y acumuladores

Ingresar 10 números distintos de cero e informar el resultado de la suma de los positivos y el producto de los negativos

```
COMIENZO
    CONTADOR ← 0
    SUMA← 0
    PRODUCTO← 1

    BORRAR PANTALLA
    MIENTRAS(CONTADOR<10) HACER
        MOSTRAR("Ingrese número:")
        LEER(NUM)
        {Bucle mientras que valida que el número no sea cero}
        MIENTRAS( NUM = 0) HACER
         MOSTRAR("ERROR. Vuelva a ingresar")
        LEER(NUM)
        FINMIENTRAS

        SI(NUM>0)
        ENTONCES
          SUMA ←SUMA+NUM
FIN-ENTONCES
        SINO
          PRODUCTO←PRODUCTO*NUM
   FINSI

        CONTAR←CONTAR+1

        FINMIENTRAS
        MOSTRAR("Suma de positivos:", SUMA)
        MOSTRAR("Producto de negativos:", PRODUCTO)

FIN
```

Uso de distintas estructuras repetitivas

Ingresar 10 números distintos de cero. Sumar los positivos y obtener el producto de los negativos

1. Uso de la estructura MIENTRAS

```
COMIENZO
        canti←0
        suma←0
        prod←1

        MIENTRAS( canti<10) HACER
                MOSTRAR("Ingrese un número")
                LEER(num)
                MIENTRAS(num=0) HACER
                        MOSTRAR("Error. Vuelva a ingresar:")
                        LEER(num)
                FINMIENTRAS


                SI(num>0)
                        ENTONCES
                                Suma← suma+num
                        FIN-ENTONCES
                        SINO
                                Prod← prod*num
                FINSI
        canti←canti+1
        FINMIENTRAS
        MOSTRAR("La suma de los positivos es:",suma)
        MOSTRAR("El producto de los negativos es:",prod)
FIN
```

2. Uso de la estructura HACER...MIENTRAS

```
COMIENZO

        cant←0
        suma←0
        prod←1

        HACER
                MOSTRAR("Ingrese un número")
```

```
        LEER(num)
        MIENTRAS(num=0) HACER
            MOSTRAR("Error. Vuelva a ingresar:")
            LEER(num)
        FINMIENTRAS
        SI(num>0)
            ENTONCES
                Suma←suma+num
            FIN-ENTONCES
            SINO
                Prod← prod*num
        FINSI
    canti←canti+1
    MIENTRAS( canti<10)
    MOSTRAR("La suma de los positivos es:",suma)
    MOSTRAR("El producto de los negativos es:",prod)
FIN
```

3. Uso de la estructura PARA

```
COMIENZO

    suma←0
    prod←1

    PARA(canti←0; canti<10;canti←canti+1) HACER

        MOSTRAR("Ingrese un número")
        LEER(num)

        MIENTRAS(num=0) HACER
            MOSTRAR("Error. Vuelva a ingresar:")
            LEER(num)

        FINMIENTRAS

        SI(num>0)
            ENTONCES
                Suma← suma+num
            FIN-ENTONCES
```

```
                SINO
                        Prod←prod*num
        FINSI
    FINPARA

    MOSTRAR("La suma de los positivos es:",suma)
    MOSTRAR("El producto de los negativos es:",prod)
FIN
```

4. Ingresar números distintos de 0 (no se sabe cuántos). Sumar los positivos y multiplicar los negativos.

```
COMIENZO

    seguir← 'S'
    suma←0
    prod←1

    MIENTRAS( seguir = 'S' ) HACER

        MOSTRAR("Ingrese un número")

        LEER(num)

        MIENTRAS(num=0) HACER
            MOSTRAR("Error. Vuelva a ingresar:")
            LEER(num)
        FINMIENTRAS

        SI(num>0)
            ENTONCES
                Suma← suma+num
            FIN-ENTONCES
            SINO
                Prod← prod*num
        FINSI
    {Sentencias de validación de seguir}
    HACER

        MOSTRAR("Desea continuar(S/N):")
```

```
        LEER(seguir)
    MIENTRAS(seguir<>'S' AND seguir <> 'N')

    FINMIENTRAS

    MOSTRAR("La suma de los positivos es:",suma)
    MOSTRAR("El producto de los negativos es:",prod)
FIN
```

Obtención de máximos y mínimos

Se ingresa de 30 estudiantes su nombre y promedio. Se pide informar el nombre del mejor estudiante y su promedio

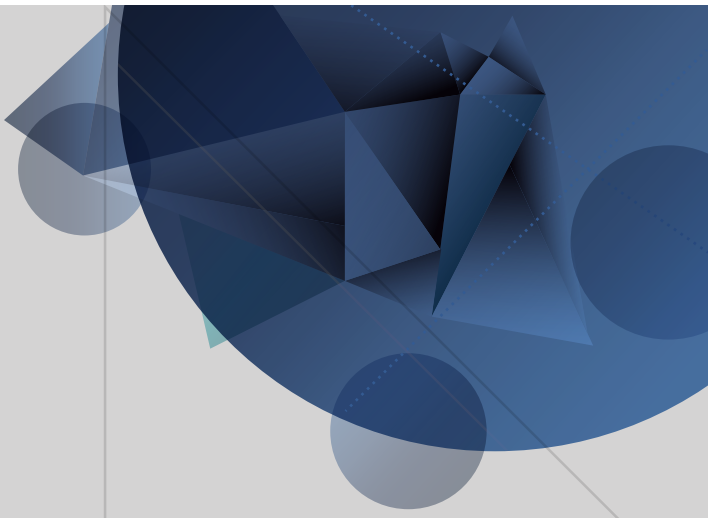
```
COMIENZO

MEJOR-PROM← 0   {variable que guarda el promedio mayor. Como se usa para comparar debe tener un  valor inicial}

MEJOR-NOMBRE    {variable que guarda el nombre del estudiante de mejor promedio}

CUENTA          {contador usado como variable de control del para}

{La carga se realiza dentro de una estructura para pues conozco el número de iteraciones que tendrá el bucle}
PARA (CUENTA←0; CUENTA < 30; CUENTA←CUENTA +1) HACER
    MOSTRAR("INGRESE NOMBRE DEL ESTUDIANTE:")
    LEER(NOMBRE)
    MOSTRAR("INGRESE PORMEDIO:")
    LEER(PROM)

    {Sentencias para validar que el promedio sea un valor correcto}
    MIENTRAS(PROM <1 O PROM>10) HACER
        MOSTRAR("ERROR. VUELVA A INGRESAR")
        LEER(PROM)
    FINMIENTRAS


{Compara el promedio ingresado con el mejor promedio almacenado hasta el momento}
    SI (PROM > MEJOR-PROM)
        ENTONCES
            MEJOR-PROM← PROM
```

```
                    MEJOR-NOMBRE←NOMBRE
            FINSI
FINPARA
{Escribe el informe de datos requeridos por el ejercicio}
MOSTRAR (“EL ESTUDIANTE DE MEJOR PROMEDIO ES:”)
MOSTRAR (MEJOR-NOMBRE)
MOSTRAR (“SU PORMEDIO ES :”))
MOSTRAR (MEJOR-PROM)
FIN
```

4

Unidad 4

Arreglos



lógica de programación

Autor: Camilo Augusto Cardona Patiño

AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO

Introducción

Los arreglos son estructuras de datos que permiten almacenar ordenadamente diferentes tipos de información; pueden ser de una única dimensión, como el caso de los vectores o tener múltiples dimensiones como las matrices. Para poder recorrer el arreglo y obtener el valor en alguna posición, es necesario tener un índice por cada dimensión que tenga.

Los arreglos se definen identificándolos por un nombre, un tipo de dato a almacenar y un tamaño; cada posición almacena un valor en una posición de memoria que queda referenciada para su posterior consulta.

U4 Metodología

Se invita al estudiante a leer completamente la guía de manera que logre obtener una sólida y consistente fundamentación teórica. De igual forma, se le recomienda analizar los ejemplos planteados, siguiendo paso a paso el desarrollo a fin de lograr claridad y comprensión en el procedimiento efectuado, estar atento de la realización de las actividades, participar en los foros y formular todas las dudas que pueda tener en relación al tema.

U4 Desarrollo temático

Arreglos

Un arreglo es una distribución organizada de datos, que hace referencia a un espacio reservado en la memoria en la que podemos contener un conjunto de elementos que comparten alguna característica. De manera gráfica, es conveniente representar un vector como la unión de una serie de celdas que tiene la capacidad de almacenar información, así:

0	1	2	3	4	5

La gráfica anterior simboliza un arreglo de una única dimensión que posee 6 celdas en las que podemos contener información. El número de casillas de un vector nos indica la dimensión del arreglo, para este ejemplo, concluimos que el vector tiene dimensión 6.

El índice en un arreglo cumple la función de identificar a cada una de las celdas que lo conforman, de manera que al consultar el arreglo indicando un índice, este debe entregar la información almacenada en dicha posición. En la figura anterior, debajo de cada casilla, aparece su índice. En lenguajes como C, C++ y java, la primera casilla del arreglo tiene índice 0, la segunda tiene índice 1, la tercera índice 2, y así sucesivamente.

Es muy importante tener presente que si el arreglo es de dimensión N, la última casilla tiene índice N-1.

Los lenguajes de programación, permiten que el programador declare arreglos de cualquier tipo y prácticamente de cualquier tamaño. En el seudolenguaje, un arreglo se declara usando el siguiente formato o plantilla:

<NOMBRE>: arreglo [<N>] de <TIPO>

La nomenclatura definida para escribir un arreglo es el nombre en mayúscula sostenida encerrada entre <>seguido por la dimensión y el tipo de dato a contener. Así por ejemplo, si se quiere declarar un arreglo con nombre letras, de dimensión 15 y que pueda almacenar datos de tipo caracter, se debe escribir la siguiente línea:

letras: arreglo [15] de caracter

El desarrollador debe nombrar el arreglo (identificándolo con un nombre explicativo de su función), debe establecer su dimensión, y también debe decir de qué tipo son los elementos que almacenará ese arreglo.

Enseguida se dan algunos ejemplos de declaraciones de arreglos.

- Si se necesita guardar las devoluciones diarias de un supermercado durante el último mes, se puede declarar el siguiente arreglo:

ventas: arreglo [7] de real

- Si se requiere almacenar las notas que ha obtenido un estudiante en los 4 talleres y en los 6 laboratorios del curso de Lógica de Programación se pueden declarar los siguientes arreglos:

talleres: arreglo [4] de real

laboratorios: arreglo [6] de real

- Si se quiere guardar el valor de las últimas 6 facturas de internet de una oficina, se puede declarar el siguiente arreglo:

facturas Internet: arreglo [6] de real

Para poder acceder o modificar a una posición especifica del arreglo, contamos con los índices de manera que podemos saltar directamente a la posición de memoria sin tener la necesidad de recorrer secuencialmente todas las posiciones del arreglo. Para referirse a una casilla particular de un arreglo se debe seguir el siguiente formato:

<NOMBRE>[<INDICE>]

Ejemplo:

cifras: arreglo [10] de entero

- La línea de código asigna y almacena el número 1000 en la primera celda del arreglo:

cifras[0]:= 1000

- La siguiente instrucción, almacena 1010 en las posiciones 5-6-7-8-9 del arreglo:

i:=5

MIENTRAS (i<10) HACER

cifras[i]:= 1010

i:=i+1

FIN-MIENTRAS

La siguiente figura muestra el arreglo cifras después de ejecutadas las instrucciones de los dos ejemplos anteriores. Las casillas vacías no tienen valores definidos.

1000					1010	1010	1010	1010	1010
0	1	2	3	4	5	6	7	8	9

Ejemplo completo

Un histograma para una colección de datos es una secuencia de parejas de la forma (d, f), donde d es un dato y f es su frecuencia en la colección. Por ejemplo, suponga que se le pide a 20 personas calificar con las letras a b c d y e el desempeño del gobierno actual, y que se obtienen las siguientes respuestas: c b c a b c d e e a b b d c a c c b d a.

El histograma para esta colección de datos se muestra enseguida en dos formas: con números y con asteriscos:

a: 4 a: ****

b: 5 b: *****

c: 6 c: ******

d: 3 d: ***

e: 2 e: **

Desarrollar un programa que permita construir un algoritmo que construya un histograma para una lista que puede contener hasta 100 valores, cada valor es un número entero incluso entre 1 y 5.

Las entradas (datos conocidos) para el algoritmo son:

- El número de datos de la colección.
- La colección misma.

La salida esperada (dato desconocido) es:

- El histograma de la colección.

Vemos en el ejemplo que los arreglos ayudan de manera que se puede almacenar el conjunto de datos y adicionalmente el histograma.

Tendríamos entonces que:

Tenemos unas situaciones iniciales "REQUIERE" que sirven de insumo para los resultados que se deben obtener "GARANTIZA" así:

Requiere:

Los límites para el conjunto de datos a recolectar deben ser superior a 0 e inferior a 100. (100≥totalDatos>0)

Cada uno de los elementos de la colección debe ser un número entre 1 y 5.

Garantiza:

Obtener el histograma asociado al conjunto. La frecuencia de cada dato aparece graficado como una secuencia de asteriscos.

Solución:

Inicio

Paso 1. Leer el número de elementos del conjunto de datos

Paso 2. Leer los elementos del conjunto de datos

Paso 3. Tabular el histograma

Paso 4. Imprimir histograma en pantalla

Fin

Los pasos 1 y 2 son capturas que hace el algoritmo de alguna fuente de información, se convierten en datos de entrada, para lo cual establecemos en este caso que la fuente de información es algún usuario que ingresa manualmente los números:

```
Procedimiento principal
Variables
        i,totalDatos: entero
        colección: arreglo [100] de entero
Inicio
escribir("Por favor digite el número de datos de la colección : (inferior o igual a 100)")
leer(totalDatos)
i:=0
```

mientras (i<totalDatos) hacer

escribir("Por favor digite el dato :")

escribir(i+1)

leer(colección[i])

i:= i+1

fin-mientras

Paso 3. Tabular el histograma

Paso 4. Imprimir histograma en pantalla

Fin

La solución al planteamiento presentado es el paso 3, que pide Tabular el histograma de una colección de datos tomada de alguna fuente de información. También sabemos que los únicos valores posibles son 1-2-3-4-5, por lo que lógicamente deberíamos emplear un arreglo de dimensión 5.

Histograma: arreglo[5] de entero

Por defecto indicamos que los valores iniciales en cada una de las celdas del vector histograma son cero, luego, debemos capturar el dato proporcionado por el usuario "Fuente de información" e ir incrementando la celda del vector en donde corresponda de acuerdo con la mencionada captura.

Para tabular el histograma:

i:=0

mientras (i<5) hacer // inicializa las frecuencias en 0. Paso 3.1

histograma[i]:= 0

i:= i+1

fin-mientras

i:=0

mientras (i<totalDatos) hacer // calcula las frecuencias. Paso 3.2

d:= colección[i]-1

histograma[d]:= histograma[d]+1

i:= i+1

fin-mientras

Una vez finalizada la captura de datos, tendremos automáticamente la tabulación del histograma, procediendo al paso 4.

```
i:=0
mientras (i<5) hacer // este ciclo recorre el histograma
        escribir(i+1)
        escribir(": ")
        f:= histograma[i] // f guarda la frecuencia de dato i+1
        j:= 0
mientras(j<f) hacer // este ciclo escribe f asteriscos
        escribir ('*')
        j:= j+1
        fin-mientras
        escribir(salto-de-linea)
        i:= i+1

fin-mientras
```

Solución completa:

```
Procedimiento principal
Constantes
N 5 / N es el tamaño máximo del histograma

MAXDATOS 100 / MAXDATOS es el tamaño máximo de la colección


Variables
i,j,totalDatos: entero
coleccion: arreglo [MAXDATOS] de entero

histograma: arreglo [N] de entero

Inicio
escribir("Por favor digite el número de datos de la colección : (inferior o igual a 100)")
leer(totalDatos)
i:=0
mientras (i<totalDatos) hacer
        escribir("Por favor digite el dato :")
        escribir(i+1)
        leer(colección[i])
        i:= i+1
fin-mientras
```

```
i:=0
mientras (i<N) hacer // inicializa las frecuencias en 0. Paso 3.1
        histograma[i]:= 0
        i:= i+1
fin-mientras
i:=0
mientras (i<totalDatos) hacer // este ciclo calcula las frecuencias. Paso 3.2
        d:= colección[i]-1
        histograma[d]:= histograma[d]+1
        i:= i+1
fin-mientras
i:=0
mientras (i<N) hacer // este ciclo recorre el histograma
        escribir(i+1)
        escribir(": ")
        f:= histograma[i] // f guarda la frecuencia de dato i+1
        j:= 0
        mientras(j<f) hacer // este ciclo escribe f asteriscos
                escribir ('*')
                j:= j+1
        fin-mientras
        escribir(salto-de-linea)
        i:= i+1
fin-mientras

Fin
```

Cadenas de caracteres

Las cadenas de caracteres pueden ser tratados como un arreglo, en los que cada símbolo que lo conforma ocupa una celda en el vector. Por ejemplo, "CH?*$A7!" y "soy cadena" son dos cadenas, la primera formada por 8 caracteres y la segunda por 10. En la memoria del computador una cadena se guarda en un arreglo de tipo caracter, de tal manera que cada símbolo de la cadena ocupa una casilla del arreglo. Sin embargo, se utiliza una casilla adicional del arreglo para guardar un carácter especial que se llama terminador de cadena. En Pseudocódigo este carácter especial es ´\0´. Como lo indica su nombre, la función de este carácter especial es indicar que la cadena termina. Las dos cadenas mencionadas arriba, se representan en la memoria del computador como lo indica la siguiente figura:

“C”	“H”	“?”	“*”	“$”	“A”	“7”	“!”	“\0”
0	1	2	3	4	5	6	7	8

“S”	“O”	“Y”	“ ”	“C”	“A”	“D”	“E”	“N”	“A”	“\0”
0	1	2	3	4	5	6	7	8	9	10

La longitud de una cadena es la cantidad de símbolos que la componen, sin contar el terminador de cadena.

Algunas operaciones:

Operación	Descripción
leerCadena(cadena)	Guarda la cadena digitada por el usuario en el arreglo **cad.**
escribirCadena(cadena)	Escribe en la pantalla la cadena **cadena.**
longitudCadena(cadena)	Retorna la longitud de la cadena **cadena.**
copiarCadena(cadena1, cadena2)	Copia la cadena **cadena2** en la cadena **cadena1.**
concatenarCadena(cadena1, cadena2)	Retorna la concatenación de cadena1 con **cadena2,** en la cadena **cadena1.**
compararCadena(cadena1, cadena2)	Retorna menos uno (-1) si **cadena1** es menor que **cadena2,** cero (0) si son iguales y uno (1) si **cadena2** es menor que **cadena1.**

Tabla1. Operaciones de cadenas.
Fuente: Propia.

Ejemplo completo

Los palíndromos son expresiones que, al ser leídas de izquierda a derecha y de derecha a izquierda, se pronuncian igual. Ejemplo:

- Anilina.
- Amor a Roma.
- Dábale arroz a la zorra el abad.

- Reconocer.
- Anita lava la tina.
- Ala.

Considere el problema de construir un algoritmo que lea una PALABRA, de longitud máxima 30, y determine si es palíndromo o no.

Las entradas (datos conocidos) para el algoritmo son:

- La palabra a considerar

La salida esperada (dato desconocido) es:

- Un mensaje que indica si es palíndromo o no.

Requiere:

Capturar una frase, hasta 30 caracteres.

Garantiza:

Imprimir por pantalla si la frase ingresada es palíndromo.

Solución:

Inicio

Paso 1. Leer frase

Paso 2. Determinar si es palíndromo

Paso 3. Imprimir resultado por pantalla

Fin

Desarrollo:

Procedimiento principal
Variables
pal: arreglo [31] de caracter
respuesta: arreglo [20] de caracter
Inicio
escribir("Por favor digite una frase de máximo 30 letras")
leerCadena(pal)
Paso 2. Determinar si pal es palíndromo
Paso 3. Imprimir resultado por pantalla
Fin

Determinar si la frase es palíndromo

i:= 0 // i señala el primer caracter de la cadena

j:= longitudCadena(pal)-1 // j señala el último caracter de la cadena

siga:= verdadero //variable que indica cuándo parar el proceso

mientras (i<j y siga) hacer

    si (pal[i]=pal[j]) entonces

        i:= i+1

        j:= j-1

sino

    siga:= falso

fin-si

fin-mientras

si i<j entonces

    copiarCadena(respuesta,"no es palindromo")

sino

    copiarCadena(respuesta,"es palindromo")

fin-si

Solución completa:

Procedimiento principal
Constantes
N 30 / N es la máxima longitud de la palabra dada por el usuario
Variables
pal: arreglo [31] de caracter
respuesta: arreglo [20] de caracter
i,j: entero
siga: booleano

Inicio
escribir("Por favor digite una palabra de máximo 30 letras")
leerCadena(pal)
i:= 0 // i señala el primer caracter de la cadena
j:= longitudCadena(pal)-1 // j señala el último caracter de la cadena
siga:= verdadero //variable "bandera" que indica cuándo parar el proceso
mientras (i<j y siga) hacer
si (pal[i]=pal[j]) entonces
i:= i+1
j:= j-1
sino
siga:= falso
fin-si
fin-mientras

si i<j entonces
copiarCadena(respuesta,"no es palindromo")
sino
copiarCadena(respuesta,"es palindromo")
fin-si
escribirCadena(respuesta)

Fin

Matrices

Es también un tipo de arreglo, se diferencia de un vector en que la matriz posee 2 dimensiones: filas y columnas:

0	1	2	3	4	5
1					
2					
3					

En el ejemplo anterior vemos una matriz con cuatro filas seis columnas, la dimensión de la matriz se expresa como el producto de sus dimensiones, para el ejemplo, la presente matriz

tiene una dimensión de 4x6.

Cada posición en la matriz está identificada por una única combinación de índices de fila y columna.

Declaración de matrices

<NOMBRE> : matriz [<N>][<M>] de <TIPO>

La nomenclatura definida para escribir un arreglo es el nombre en mayúscula sostenida encerrada entre <>seguido por la dimensión y el tipo de dato a contener. Así por ejemplo, si se quiere declarar un arreglo con nombre letras, de dimensión 15x4 y que pueda almacenar datos de tipo carácter, se debe escribir la siguiente línea

mat : matriz [15][4] de caracter

El desarrollador debe nombrar el arreglo (identificándolo con un nombre explicativo de su función), debe establecer su dimensión, y también debe decir de qué tipo son los elementos que almacenará ese arreglo.

Enseguida se dan algunos ejemplos de declaraciones de arreglos.

- Si se necesita guardar la información relacionada con el tablero de un juego de *tic tac toe* (el tradicional triqui), se puede declarar la siguiente matriz:

tablero : matriz [3][3] de carácter

- Si se requiere guardar las notas que han sacado 35 estudiantes en los 5 talleres y en los 5 laboratorios del curso de Programación de Computadores se pueden declarar las siguientes matrices.

talleres : matriz [35][5] de real

laboratorios : matriz [35][5] de real

Para poder acceder o modificar una posición especifica del arreglo, contamos con los índices de manera que podemos saltar directamente a la posición de memoria sin tener la necesidad de recorrer secuencialmente todas las posiciones del arreglo. Para referirse a una casilla particular de un arreglo se debe seguir el siguiente formato:

<NOMBRE>[<INDICE-DE-FILA>][<INDICE-DE-COLUMNA>]

Ejemplo:

montos : matriz [6][10] de real

- La siguiente instrucción asigna o guarda el número 15.6 en la casilla de la esquina superior izquierda de esta matriz:

montos[0][0]:= 15.6

- La siguiente instrucción iterativa guarda 9.1 en cada una de las casillas de la última fila de esta matriz:

k:=0

MIENTRAS (k<10) HACER

montos[5][k]:= 9.1

k:=k+1

FIN-MIENTRAS

La siguiente figura muestra la matriz montos después de ejecutadas las instrucciones de los dos ejemplos anteriores. Las casillas vacías no tienen valores definidos.

0	1	2	3	4	5	6	7	8	9
1	**15.6**								
2									
3									
4									
5									
	9.1	9.1	9.1	9.1	9.1	9.1	9.1	9.1	9.1

Ejemplo completo

Se conoce como La matriz mágica a una matriz cuadrada en la que las sumas horizontales, verticales y diagonal dan lo mismo

Ejemplo:

2	7	6
9	5	1
4	3	8

En esta matriz las sumas son 15.

Se pide crear un algoritmo que permita verificar si una matriz de hasta 10x10 dada cumple con las propiedades características de una matriz mágica. Se debe guardar la suma de las filas, las columnas y las diagonales en un arreglo en el orden siguiente:

0	1	2		3	4	5		6	7
Fila 0	Fila 1	Fila 2	…	Columna 0	Columna 1	Columna 2	…	Diago-nal 1	Diago-nal 2

Información conocida:

- El tamaño "dimensión" de la matriz.
- Los datos contenidos por la matriz.

Información solicitada:

- Es mágica o no la matriz, en caso positivo, cual es el valor de la suma.

Requiere:

El tamaño máximo de la matriz es de 10x10.

Todos los números contenidos deben ser números enteros.

Garantiza

Determinar si la matriz es mágica, si lo es, indicar la suma.

Solución:

Inicio
Paso 1: calcular el tamaño de la matriz
Paso 2: Leer los números de la matriz
Paso 3: Determinar si la matriz es mágica o no
Paso 4: Si es mágica indicar la suma.

Fin

Los primeros dos pasos hacen referencia a la captura de información.

Procedimiento principal
variables
i, j, temp, tamano, suma: enteros //i=filas, j=columnas
mmagica: matriz [10][10] de enteros

```
Inicio
escribir("Digite el número de filas de la matriz (entre 2 y 10): ")
leer(tamano)
para (i=0 hasta tamano-1) hacer
        para(j=0 hasta tamano-1) hacer
                escribir("Digite el dato en la posición")
                escribir(i,j)
                leer(mmagica[i][j])
        fin_para
fin_para
Paso 3
Paso 4
```

Determinar si la matriz es mágica. Luego debemos definir un arreglo para almacenar la suma de filas, columnas y diagonales, empleando un vector que tendrá una dimensión de 2*tamano+2. La declaración del arreglo sumas es:

```
sumas: arreglo [22] de enteros
para(i=0 hasta 2*tamano+2) hacer
        sumas[i]:=0
fin_para
```

```
//Suma filas
para(i=0 hasta tamano-1) hacer
        para(j=0 hasta tamano-1) hacer
                sumas[i]=mmagica[i][j]+sumas[i]
        fin_para
fin_para
```

```
//Suma columnas
para(j=0 hasta tamano-1) hacer
        para(i=0 hasta tamano-1) hacer
                sumas[tamaño+j]=mmagica[i][j]+sumas[tamaño+j]
        fin_para

fin_para
```

//Suma diagonales

```
para(i=0 hasta tamano-1) hacer
        sumas[2*tamano]=mmagica[i][i]+sumas[2*tamano]
fin_para
para(i=0 hasta tamano-1) hacer
        sumas[2*tamano+1]=mmagica[i][(tamano-1)-i]+sumas[2*tamano+1];

fin_para
```

Paso 4: se debe verificar que la suma de todos los componentes del vector (sumas) son iguales, en caso afirmativo se indica que la matriz cumple con las propiedades de matriz mágica y se muestra el valor de la suma, en caso contrario se indica que no se tiene una matriz mágica.

```
int contador=0;
contador=sumas[0];
para(i=1 hasta 2*tamano+1) hacer
        si(contador<>sumas[i])
                escribir("la matriz no es mágica)
                i=2*tamano+3;
        fin_si
fin-para
si(i=2*tamano+2)
                escribir("la matriz es mágica y la suma es:")
                escribir(contador);

fin_si
```

Solución completa:

```
variables
i, j, temp, tamano, suma: entero //i señala las filas
//j señala las columnas
contador=0: entero
mmagica: matriz [10][10] de enteros

sumas: arreglo [22] de enteros


Inicio
escribir("Por favor digite el número de filas de la matriz (entre 2 y 10): ")
leer(tamano)
```

```
para (i=0 hasta tamano-1) hacer
        para(j=0 hasta tamano-1) hacer
                escribir("Por favor digite el dato en la posición")
                escribir(i,j)
                leer(mmagica[i][j])
        fin_para
fin_para
para(i=0 hasta 2*tamano+2) hacer
sumas[i]:=0
fin_para
//Sumas correspondientes a las filas
para(i=0 hasta tamano-1) hacer
        para(j=0 hasta tamano-1) hacer
                sumas[i]=mmagica[i][j]+sumas[i]
        fin_para
fin_para
//Sumas correspondientes a las columnas
para(j=0 hasta tamano-1) hacer
        para(i=0 hasta tamano-1) hacer
                sumas[j+tamano]=mmagica[i][j]+sumas[j+tamano]
        fin_para
fin_para

//Sumas correspondientes a las diagonales
para(i=0 hasta tamano-1) hacer
        sumas[2*tamano]=mmagica[i][i]+sumas[2*tamano]
fin_para
para(i=0 hasta tamano-1) hacer
        sumas[2*tamano+1]=mmagica[i][(tamano-1)-i]+sumas[2*tamano+1];
fin_para
contador=sumas[0];
para(i=1 hasta 2*tamano+1) hacer
si(contador<>sumas[i])
        escribir("la matriz no es mágica)
        i=2*tamano+3;
```

```
fin_si
fin-para
si(i=2*tamano+2)
        escribir("la matriz es mágica y la suma es:")
        escribir(contador);
fin_si

fin-procedimiento
```

Bibliografia

- Alcalá, R. (2008). Teoría de Algoritmos. Recuperado de http://sci2s.ugr.es/docencia/tasb/TA-Tema1-0809.pdf
- Alonso, J. (2012). El método de Pólya para resolver problemas. Recuperado de http://www.glc.us.es/~jalonso/vestigium/el-metodo-de-polya-para-resolver-problemas/
- Berzal, F. (2012). Análisis y diseño de algoritmos. Recuperado de http://elvex.ugr.es/decs-ai/algorithms/slides/1%20Introduction.pdf
- Castillo, V. (2003). Lenguajes Algorítmicos. Recuperado de http://docente.ucol.mx/abolio/tareas/Lenguaje.pdf
- Constantini, S. (2003). Sistemas Numéricos. Recuperado de http://jvmr2012.files.wordpress.com/2014/03/sistemasnumericos-1_tareaespecial.pdf
- Dornick. (2014). Historia del hardware. Recuperado de http://es.wikipedia.org/wiki/Historia_del_hardware
- Estévez, A. (2012). Estructura del computador. Recuperado de http://es.wikiversity.org/wiki/Estructura_del_computador
- García, A. & Arranz, J. M. (2009). Estructura de un programa en Java. Recuperado de http://ocw.upm.es/lenguajes-y-sistemas-informaticos/programacion-en-java-i/Contenidos/LecturaObligatoria/2-estructuradeunprogramaenjava.pdf
- Garrido, P. (2002). Sistemas de representación numérica. Recuperado de http://emp.usb.ve/mrivas/tema_2b.pdf
- González, L. (2013). Estructura básica de un programa. Recuperado de https://sites.google.com/site/algopromecanica/unidad-iii/3-2-estructura-basica-de-un-programa
- Hernández, M. (2010). Diseño Estructurado de Algoritmos. Recuperado de http://kesquivel.files.wordpress.com/2010/03/disenoestructuradoalgoritmos.pdf
- Islas, G. (2012). Estructura básica de un programa en C. Recuperado de http://gy94bfmv.blogspot.com/2012/10/estructura-basica-de-un-programa-en-c.html
- Lisit, T. (2012). Sistemas de numeración en informática. Recuperado de http://robotica.uv.es/pub/Libro/PDFs/CAPI4.pdf
- López, M. (2012). Generaciones del computador. Recuperado de http://camilalopez74.blogspot.com/2012/04/generaciones-del-computador-1-2-3-4-5-6.html
- Norma, M. (2011). Esquema estructural de una computadora. Recuperado de http://informatica1norma.blogspot.com/2011/04/esquema-estructural-de-una-computadora.html
- Perdomo, E. (2009). Estructura de un programa en C. Recuperadode http://eperdomo89.wordpress.com/2009/09/05/estructura-de-un-programa-en-c/
- Portella, J. (2011). Clasificación de problemas algorítmicos. Recuperado de http://es.scribd.com/doc/54470751/Clasificacion-de-problemas-algoritmicos

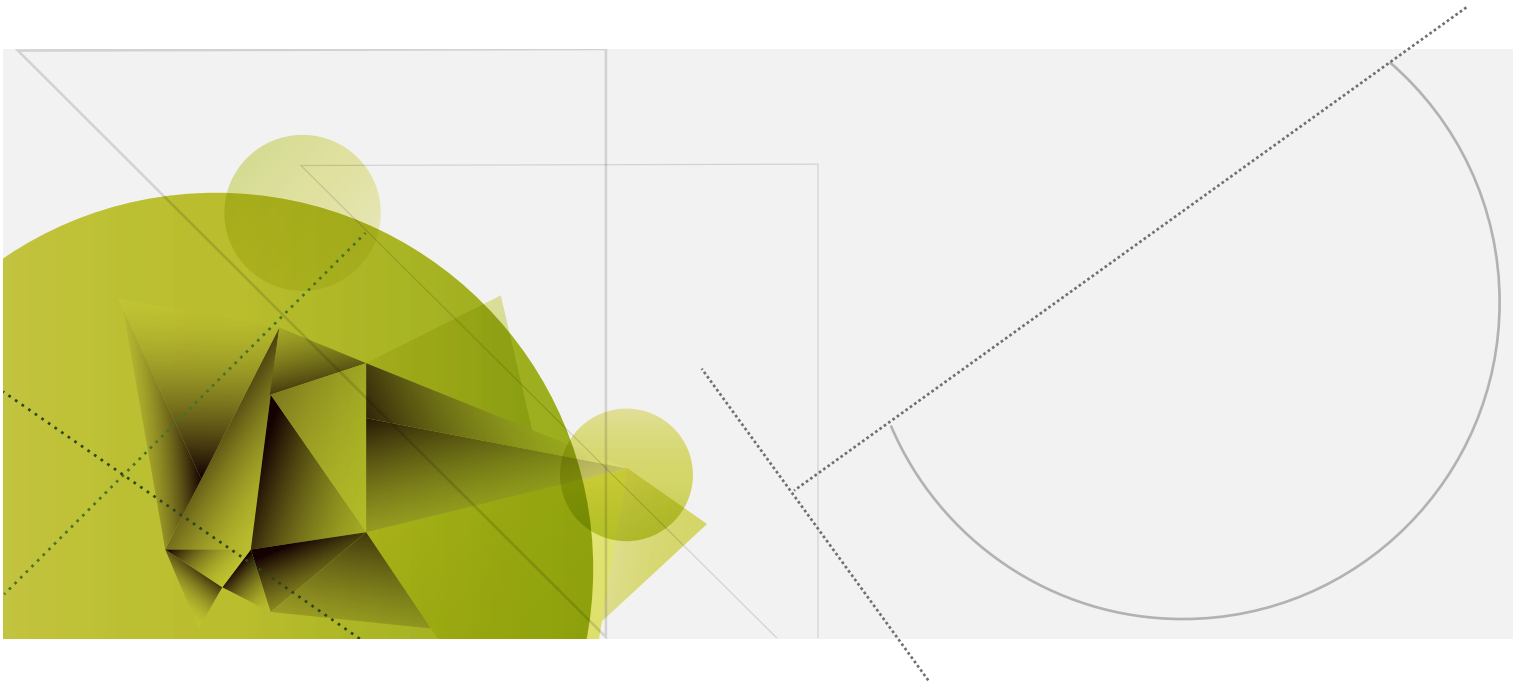
Bibliografia

- Silva, L. (2013). Arreglos. Recuperado de http://www2.elo.utfsm.cl/~lsb/pascal/clases/cap17.pdf
- Torres, H. Córdoba, L. Mesa, A. Torres, A. Mesa, M. & Salas, J. (2009). Generación de los computadores. Recuperado de http://www.slideshare.net/harryjeans/generaciones-de-los-computadores
- http://mimosa.pntic.mec.es/~flarrosa/pseudoco.pdf
- http://www.buenastareas.com/ensayos/Formulacion-De-Algoritmos/2768019.html
- http://www.cad.com.mx/generaciones_de_las_computadoras.htm
- http://www.ingelec.uns.edu.ar/icd2759/docs/apuntes/Unidad%201.pdf
- Alegsa, L. (2014). Definición de Función (programación). Recuperado de http://www.alegsa.com.ar/Dic/funcion.php
- Arias, K. (2012). Programación en Java. Recuperado de http://todoenjava.blogspot.com/2012/01/metodos-y-funciones.html
- Berzal, F. (2004). Datos y tipos de datos. Recuperado de http://elvex.ugr.es/decsai/java/pdf/2C-Datos.pdf
- Cepero, E. (2012). Problemas algorítmicos. Recuperado de http://slideplayer.es/slide/168746/
- Cristóbal, O. (2012). Técnicas para la formulación de algoritmos. Recuperado de http://unidadesdealgoritmos.blogspot.com/2012/12/unidad-v-tecnicas-para-la-formulacion.html
- EcuRed. (2013). Función (programación). Recuperado de http://www.ecured.cu/index.php/Funci%C3%B3n_(programaci%C3%B3n)
- García, A. & Arranz, J. M. (2010). Control de flujo del programa. Recuperado de http://www.uhu.es/04004/material/Transparencias5.pdf
- García, A. (2008). Expresiones y operadores. Recuperado de http://ocw.upm.es/ciencia-de-la-computacion-e-inteligencia-artificial/fundamentos-programacion/contenidosteoricos/ocwfundamentosprogramaciontema4.pdf
- Giraldo, G. & Niño, L. Técnicas para la formulación de algoritmos. Recuperado de http://www.virtual.unal.edu.co/cursos/ingenieria/2001839/modulo1/cap_02/leccion1022.htm
- Gutiérrez, T. (s.f.). Estructura básica de un programa. Recuperadode https://sites.google.com/site/tecnologicodetuxtlagutierrez/3-2-estructura-basica-de-un-programa
- Herrera, A. (2009). Ejemplos de uso de Arreglos. Recuperado el 31 de octubre de 2013 de http://www.herrera.unt.edu.ar/programacion/archivos/ejemplos%20de%20Arreglos.pdf
- Levine, G. (1990). Introducción a la Computación y a la Programación Estructurada Estados Unidos: Mc. Graw Hill.

Bibliografia

- López, J. (2010). Algoritmos y programación. Recuperado de http://www.eduteka.org/pdfdir/AlgoritmosProgramacion.pdf
- Mauricio, D. (2010). Complejidad de problemas. Recuperado de http://fisidds.galeon.com/s1.pdf
- Oñate, M. (2011). Características y propiedades de los Algoritmos. Recuperado de http://aulatec.wordpress.com/caracteristicas-y-propiedades-de-los-algoritmos/
- Pes, C. (2006). Declaración de variables (en programación). Recuperado de http://www.carlospes.com/curso_de_algoritmos/03_02_01_declaracion_de_variables.php
- Rancel, M. (2006). Declaración de variables. Recuperado de http://www.aprendera-programar.com/index.php?option=com_content&view=article&id=145:declaracion-de-variables-integer-single-boolean-visual-basic-dim-ejemplos-ejercicio-resuelto-cu00309a&catid=37:curso-qprogramacion-en-visual-basic-nivel-iq&Itemid=61
- Sumano, R (2012). Algoritmos. Recuperado de http://www.fba.unlp.edu.ar/tic/archivos/Q14.pdf
- Urbaez, W (2005). Técnicas para la formulación de algoritmos. Recuperado de http://www.desarrolloweb.com/articulos/2184.php
- http://www.uhu.es/javier.fernandez/Tema2.PDF
- http://www.buenastareas.com/ensayos/Matematicas-Para-Ingenieron/32185714.html
- http://www.uaeh.edu.mx/docencia/P_Presentaciones/icbi/asignatura/arreglos.pdf
- http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Funciones

Esta obra se terminó de editar en el mes de noviembre
Tipografá Myriad Pro 12 puntos
Bogotá D.C,-Colombia.



AREANDINA
Fundación Universitaria del Área Andina

MIEMBRO DE LA RED
ILUMNO