Ministerio de Producción

Secretaría de Emprendedores y de la Pequeña y Mediana Empresa

Dirección Nacional de Servicios Basados en el Conocimiento

Programadores

Apunte Teórico del Módulo 2

Pensamiento Lógico

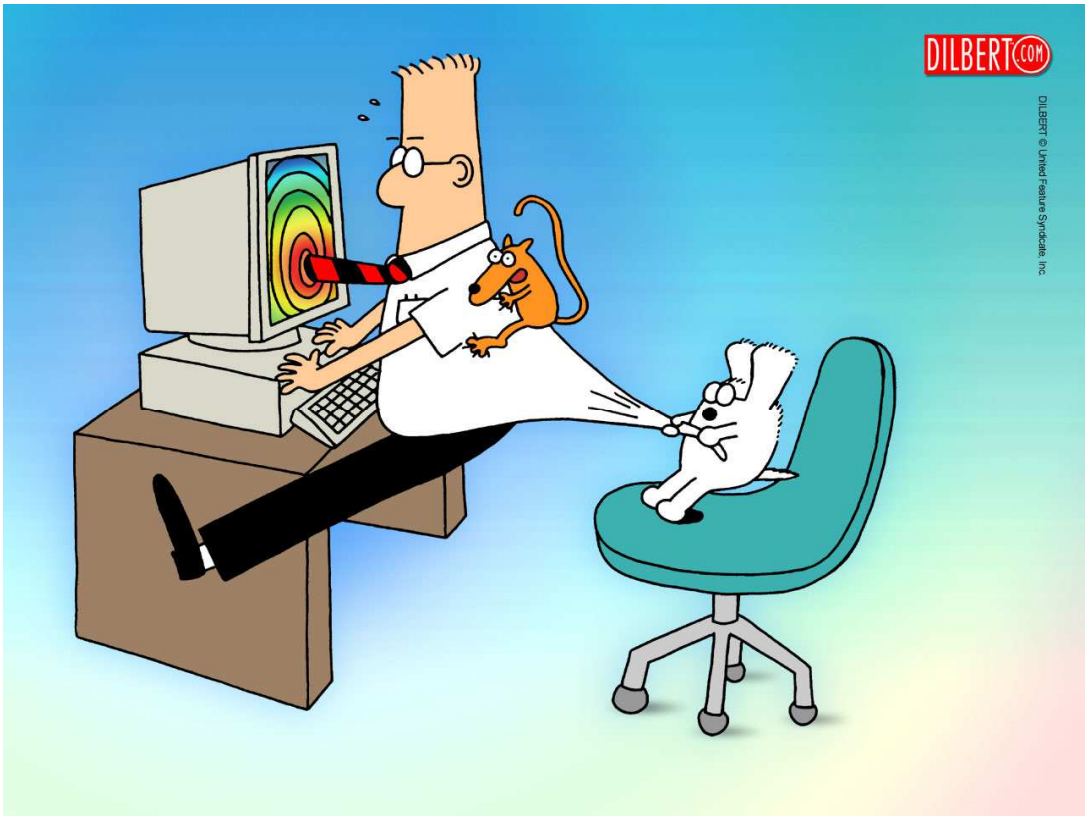


Tabla de Contenido

DEFINICIÓN DEL MÓDULO 5

PRESENTACIÓN 5

INTRODUCCIÓN 7

INTRODUCCIÓN A LA LÓGICA 8

¿QUÉ ES LA LÓGICA? 9
PROPOSICIONES LÓGICAS 9
TIPOS DE PROPOSICIONES 10
VALORES DE VERDAD 11
OPERADORES LÓGICOS 12
EJEMPLO 1 *NEGANDO PROPOSICIONES* 16
SOLUCIÓN 16
EJEMPLO 2 *CONJUNCIÓN* 16
SOLUCIÓN 16
***EJEMPLO 3 DISYUNCIÓN* 17**
SOLUCIÓN 17

ELEMENTOS INFORMÁTICOS 18

EL SOFTWARE Y SUS CARACTERÍSTICAS 18
CARACTERÍSTICAS DEL SOFTWARE: 18
ESTRUCTURA INTERNA DE UNA COMPUTADORA 18
REDES DE COMPUTADORAS 21
PROGRAMACIÓN Y CONSTRUCCIÓN DE SOFTWARE 22
LOS SISTEMAS Y SU ENFOQUE 24
¿POR QUÉ HABLAMOS DE *SISTEMAS*? 24
¿QUÉ ES UN SISTEMA? 24
CARACTERÍSTICAS DE LOS SISTEMAS 25
INTERCAMBIO ENTRE SISTEMAS 26
SISTEMAS TECNOLÓGICOS 27
¿CÓMO SE CONSTRUYE EL SOFTWARE? 27

DISEÑO DE ALGORITMOS 29

ALGORITMOS 30
CONCEPTO 30
CARACTERÍSTICAS DE LOS ALGORITMOS 30
HERRAMIENTAS PARA LA REPRESENTACIÓN GRÁFICA DE LOS ALGORITMOS 32
DIAGRAMAS DE FLUJO 32
PSEUDOCÓDIGO 32

LENGUAJES DE PROGRAMACIÓN 33
CONCEPTO 33
TIPOS DE LENGUAJES DE PROGRAMACIÓN 33
EL LENGUAJE MÁQUINA 33
EL LENGUAJE DE BAJO NIVEL 34
LENGUAJES DE ALTO NIVEL 34
¿QUÉ ES UN PROGRAMA? 34
PASOS PARA LA CONSTRUCCIÓN DE UN PROGRAMA 34
ELEMENTOS DE UN PROGRAMA 35

DESARROLLO DE PROGRAMAS 43

ESTRUCTURAS DE PROGRAMACIÓN 43
ESTRUCTURA SECUENCIAL 43
ESTRUCTURA ALTERNATIVA 43
ESTRUCTURA REPETITIVA O ITERATIVA 47
RECURSIVIDAD 50
ESTRUCTURAS DE DATOS: PILAS, COLAS Y LISTAS 51
LISTAS 51
PILAS 53
COLAS 54
ÁRBOLES 55
ÁRBOLES BINARIOS 57
ALGORITMOS FUNDAMENTALES 58
ALGORITMOS DE ORDENACIÓN 58
ORDENAMIENTO POR INSERCIÓN 58
ALGORITMO DE LA BURBUJA 59
ORDENAMIENTO POR SELECCIÓN 61
ALGORITMO QUICK-SORT 62
ALGORITMOS DE BÚSQUEDA 62
BÚSQUEDA SECUENCIAL 63
BÚSQUEDA BINARIA 63
ALGORITMOS DE RECORRIDO 65

FUENTES DE INFORMACIÓN 67

Definición del Módulo

Denominación de Módulo: **Pensamiento Lógico**

Presentación

El módulo Pensamiento lógico tiene, como propósito general, la formación en la construcción de las capacidades técnicas en torno a la lógica de programación. Se constituye, así, en un espacio de aproximación y desarrollo de saberes fundamentales para la figura profesional de "Programador".

Se parte conceptualmente de introducción a la lógica para luego analizar problemas de base informática, con el fin de que los estudiantes adquieran los conceptos y las técnicas de resolución de problemas, cuya lógica se utiliza y amplía en el módulo específico de Programación Orientada a Objetos.

En este contexto se entiende por Programación al método de resolución de problemas que utiliza algoritmos y un lenguaje de programación, respetando los principios del desarrollo de software con calidad, utilizados en el campo profesional de actuación de este técnico.

La calidad en el desarrollo del software implica la modularización, la utilización de estructuras de datos adecuados al problema, las normas de estilo de programación y las pruebas de funcionamiento. Se considera que un programa está modularizado cuando está compuesto por subrutinas con fines específicos, comunicadas a través de parámetros. Las subrutinas son unidades lógicamente independientes que se invocan desde otras subrutinas o desde un programa principal.

Este módulo se orienta al desarrollo de las siguientes **capacidades profesionales referidas al perfil profesional en su conjunto**:

1. Escribir código de programación de acuerdo a especificaciones.

El Programador interpreta las especificaciones de diseño y de requisitos de las asignaciones a programar, comprendiendo en su contexto inmediato, cuál es el problema a resolver, determinando el alcance del problema y convalidando su interpretación a fin de identificar aspectos faltantes.

Desarrolla algoritmos que den soluciones a los problemas asignados o derivados de los mismos, procurando tener un código eficiente, documentado, fácil de entender y mantener. Efectúa pruebas de unidad al código construido para asegurar que cumpla con las especificaciones recibidas.

2. Interpretar especificaciones de diseño que le permitan construir el código en el contexto del desarrollo de software en el que participa.

El Programador recibe las especificaciones y analiza el problema a resolver; interpreta el material recibido y clarifica eventuales malas interpretaciones o desacuerdos convalidando su interpretación con los miembros del equipo de proyecto que correspondan.

Debe ser analítico y tener capacidad de abstracción, para poder comprender las especificaciones, observando reglas de los lenguajes de modelado en la que estas especificaciones están expresados. También deberá

describir en sus propios términos el problema, identificar puntos ambiguos, aspectos faltantes o eventuales contradicciones entre distintos requisitos a cumplir o inconsistencias entre estos y otros aspectos conocidos del producto.

5. Analizar Errores de código

El Programador relaciona resultados incorrectos con los datos o porciones de código que los originaron, analiza estos datos y/o partes del código que causaron el mal funcionamiento y determina el tipo de corrección o reemplazo requeridos; verifica que la corrección y/o reemplazo solucionen el mal funcionamiento o la salida de resultados erróneos.

8. Integrar un equipo en el contexto de un Proyecto de Desarrollo de Software.

El desarrollo de software es una actividad social, que se desarrolla principalmente en equipo, en consecuencia, el Programador debe poder integrarse en un equipo de trabajo, sea este un contexto de proyecto de gestión tradicional o de gestión ágil.

Debe poder manejar su entorno personal y el entorno laboral donde se insertará.

Los contenidos del módulo se resumen seguidamente:

- Introducción a la Lógica
- Lógica proposicional, Lógica de predicados y operadores lógicos
- Premisas lógicas, Tablas de Verdad (and, or, xor, not), silogismos (tipos válidos y falacias).
- Elementos informáticos.
- Desarrollo de algoritmos
- Desarrollo de programas.
- Estructuras de Datos
- Estructuras de Control
- Estructuras de Programación
- Algoritmos fundamentales de búsqueda, recorrido y ordenamiento

Introducción

Cada uno de los tres últimos siglos ha estado dominado por una nueva tecnología. El siglo XVIII fue la época de los grandes sistemas mecánicos que dieron paso a la Revolución Industrial. El siglo XIX fue la era de la máquina de vapor. Durante el siglo xx, la tecnología clave fue la recopilación, procesamiento y distribución de información. Entre otros desarrollos vimos la instalación de las redes telefónicas a nivel mundial, la invención de la radio y la televisión, el nacimiento y crecimiento sin precedentes de la industria de la computación, el lanzamiento de satélites de comunicaciones y, desde luego, Internet.

En 1977 Ken Olsen era presidente de Digital Equipment Corporation, en ese entonces la segunda empresa distribuidora de computadoras más importante del mundo (después de IBM). Cuando se le preguntó por qué Digital no iba a incursionar a lo grande en el mercado de las computadoras personales, dijo: “No hay motivos para que una persona tenga una computadora en su hogar”. La historia demostró lo contrario y Digital desapareció. En un principio, las personas compraban computadoras para el procesamiento de palabras y para juegos. En los últimos años, probablemente la razón más importante sea acceder a Internet.

En la actualidad disponemos de un abanico de dispositivos tecnológicos que facilitan las tareas cotidianas, desde electrodomésticos que simplifican las tareas del hogar hasta computadoras, notebooks, smartphones y tablets que nos permiten trabajar, comunicarnos a través de Internet, esparcirnos y mucho más. Cada uno de ellos ha pasado por un proceso de ideación, planificación, desarrollo, manufactura y logística que permitieron que el producto terminado esté disponible para nosotros.

En 1965, Gordon Moore (Cofundador de Intel) afirmó que la tecnología tenía futuro, que el número de transistores en los circuitos integrados, uno de los componentes esenciales en el procesador de una computadora, se duplicaba cada año y que la tendencia continuaría durante las siguientes dos décadas. Aunque luego disminuyó este periodo a 2 años, esta ley empírica se ha cumplido y se traduce en que tengamos cada día dispositivos más pequeños, más veloces y a un costo más bajo. Tanto es así, que empresas de electrónica ya se encuentran desarrollando electrodomésticos con programas inteligentes que pueden conectarse a través de Internet, de forma tal de que podamos consultar el estado del trabajo de nuestro lavarropas desde la oficina o en viaje.

A pesar de que la industria de la computación es joven si se la compara con otras (como la automotriz y la de transporte aéreo), las computadoras han progresado de manera espectacular en un periodo muy corto. Durante las primeras dos décadas de su existencia, estos sistemas estaban altamente centralizados y por lo general se encontraban dentro de un salón grande e independiente. Era común que este salón tuviera paredes de vidrio, a través de las cuales los visitantes podían mirar boquiabiertos la gran maravilla electrónica que había en su interior. Una empresa o universidad de tamaño mediano apenas lograba tener una o dos computadoras, mientras que las instituciones muy grandes tenían, cuando mucho, unas cuantas docenas. La idea de que en un lapso de 40 años se produjeran en masa miles de millones de computadoras mucho más poderosas y del tamaño de una estampilla postal era en ese entonces mera ciencia ficción.

Introducción a la Lógica

Cuando requerimos juicios confiables, el recurso en el que más correctamente nos apoyamos es la razón. Sabemos que comúnmente se utilizan recursos no racionales, como hábitos, intuición o cosas similares. Pero cuando enfrentamos circunstancias difíciles, cuando nuestras decisiones pueden repercutir seriamente en nosotros o en otras personas, cuando por emitir un juicio ponemos muchas cosas en riesgo, *razonamos* el asunto lo mejor que podemos, porque ése es el curso de acción más **lógico**.

Existen métodos racionales, métodos probados y confirmados para determinar lo que es verdad. Existen técnicas establecidas, técnicas racionales, para extraer derivaciones (inferencias) nuevas a partir de lo que ya sabemos que es verdad. Debido a nuestra ignorancia, a menudo nos vemos obligados a recurrir a una autoridad para establecer un juicio pero incluso entonces no podemos escapar a la necesidad de emplear el *razonamiento*, porque tenemos que decidir con la mayor confianza qué autoridades merecen nuestro respeto y por qué. En toda actividad intelectual seria, confiamos en última instancia en el razonamiento porque no existe nada que pueda reemplazarlo satisfactoriamente.

Por naturaleza, los seres humanos fuimos dotados con las habilidades de razonamiento. Tal vez por mucho tiempo nos hemos dejado conducir por principios sólidos que comprendemos sólo de manera parcial. Si nos esforzamos lo suficiente, podemos sacar esos principios a la superficie, formularlos y aprender a aplicarlos a problemas que se pueden solucionar por medio de la razón. Con el estudio de la lógica aprendemos a reconocer nuestras capacidades innatas y luego a fortalecerlas mediante el ejercicio. El estudio de la lógica nos ayuda a razonar de forma adecuada porque ilumina los principios del razonamiento correcto.

Las Ciencias Informáticas en su conjunto no escapan a esta regla general. En efecto, desde la programación elemental, hasta los estudios más avanzados de sistemas operativos o inteligencia artificial, la necesidad de estructurar nuestro pensamiento de una manera lógica (es decir, que nuestros métodos de inferencia sean claros y válidos) se presenta como indispensable ya que, de otra forma, resultaría imposible establecer claramente las características de un problema informático determinado y desarrollar e implementar correctamente su posible solución.

De esta forma existe una relación esencial entre la lógica matemática y la informática, de forma tal que resulta necesario conocer los principios que sostienen al razonamiento, para poder hacer una aplicación en el campo de cualquier disciplina de la Informática.

La lógica constituye la herramienta formal de razonamiento de la mayor parte de las asignaturas de la carrera de informática, sobre todo de las que están más relacionadas con las matemáticas y la programación, tales como Programación y Bases de Datos entre otras. En cuanto a la inteligencia artificial, la lógica es el fundamento de todos los métodos de representación del conocimiento y del razonamiento como el procesamiento del lenguaje natural (entre los ejemplos que podemos mencionar están: los asistentes personales como Alexa, Siri o Google Home), razonamiento espacial y temporal, visión artificial, robótica, etc.

¿Qué es la lógica?

Definimos a la lógica como el estudio de los principios y métodos utilizados para distinguir el razonamiento correcto del incorrecto.

Constantemente pensamos. Pensar es un complejo proceso que se inicia con la creación de imágenes mentales en nuestro cerebro. Estas imágenes las integramos, emparejamos, proyectamos o asociamos con nuestros conceptos o esquemas que tenemos memorizados; representándonos las situaciones del mundo y de nosotros mismos en un proceso simbólico que necesitamos estructurar en secuencias organizadas, es decir, lógicamente.

A partir de esto podemos prever lo que sucederá, evaluar las consecuencias de nuestros actos, anticiparnos para evitar episodios desfavorables y promocionar los que más nos beneficien.

Esto significa que construimos secuencias temporalizadas de imágenes o conceptos que representan simbólicamente cosas o eventos y que podemos poner en movimiento para producir (simbólicamente) lo que aún no ha acontecido. *Ese poner en movimiento, que necesita naturalmente no sólo una memoria en funcionamiento, sino también una conciencia de lo que estamos pensando, es a lo que podemos denominar* ***razonamiento****.*

De esta manera, razonar consiste en producir juicios. Un juicio tiene la forma de una proposición, es decir, de una oración. Por ejemplo 'esta mesa es verde' es un juicio. En ese juicio, están contenidos los conceptos: 'mesa', 'lo verde'; también hay imágenes que distinguen nuestros objetos o que asociamos con los conceptos y hay una estructura lógica, sintáctica, que nos permite en una secuencia expresar un estado de cosas del mundo.

Proposiciones Lógicas

Las proposiciones son el material de nuestro razonamiento. *Una* ***proposición*** *afirma que algo es o no es por lo tanto cualquier proposición puede ser afirmada o negada*. Es posible que la verdad (o falsedad) de algunas proposiciones, como por ejemplo “Existe vida en algún otro planeta de nuestra galaxia”, no se conozca nunca. Pero esa proposición, como cualquier otra, tiene que ser verdadera o falsa de forma independiente de que podamos o no verificarla.

Así, las proposiciones difieren de las preguntas, de las órdenes y de las exclamaciones. Ninguna de las anteriores se puede afirmar o negar. La verdad y la falsedad siempre se aplican a las proposiciones, pero no se aplican a las preguntas, ni a las órdenes ni a las exclamaciones.

También se tiene que distinguir a las proposiciones de las oraciones a través de lo que cada una asevera. Dos oraciones distintas constituidas por diferentes palabras, arregladas de diferente manera, pueden tener el mismo significado y utilizarse para aseverar la misma proposición. Por ejemplo, “María ganó la elección” y “La elección fue ganada por María”, claramente son dos oraciones distintas que afirman lo mismo.

Proposición es el término empleado para referirnos a aquello para lo que las oraciones declarativas se utilizan normalmente para aseverar. Las oraciones son partes de una lengua, pero las proposiciones no están atadas a ninguna lengua dada.

Las oraciones tienen el mismo valor de verdad sin importar el idioma en el cual estén enunciadas, por lo tanto las siguientes oraciones son equivalentes por más de que utilicen palabras muy distintas para su confección:

lt is raining. (Inglés)

Está lloviendo. (Español)

Il pleut. (Francés)

Es regnet. - (Alemán)

Aristóteles (384 aC — 332 aC) desarrolló el primer tratado sistemático de las leyes de pensamiento para la adquisición de conocimiento en el Órganon, como el primer intento serio para fundar la lógica como ciencia. Fue el primero en dar el concepto de proposición como "*un discurso enunciativo perfecto que se expresaba en un juicio que significaba falso o verdadero*".

Algunos autores diferencian el concepto de **enunciado** del concepto de **proposición**, pero en nuestro caso para su uso en el ámbito de la lógica, los trataremos como equivalentes.

Tipos de proposiciones

En adelante cuando hablemos de proposiciones, éstas serán lógicas. Para operar con las proposiciones las clasificamos en dos tipos: Simples y Compuestas, dependiendo de cómo están conformadas.

Proposiciones Simples son aquellas que no tienen oraciones componentes afectadas por negaciones ("no") o términos de enlace como conjunciones ("y"), disyunciones ("o") o implicaciones ("si . . . entonces"). Pueden aparecer términos de enlace en el sujeto o en el predicado, pero no entre oraciones.

Proposiciones Compuestas: Una proposición será **compuesta** si no es simple. Es decir, si está afectada por negaciones o términos de enlace entre oraciones componentes.

Veamos algunos ejemplos:

Jorge Luis Borges es un escritor	Simple
Jorge Luis Borges es un escritor argentino que publicó el libro Ficciones	Compuesta
Sen(x) no es un número mayor que 1	Compuesta
Los números 3 y 7 son números primos	Simple
3 es número primo y 7 también es número primo	Compuesta

De dos proposiciones contradictorias una y sólo una de ellas debe ser verdadera. En este contexto analicemos las siguiente proposiciones:

1. Jorge Luis Borges es un escritor
2. Jorge Luis Borges no es un escritor

De forma intuitiva podemos decir que no existe la posibilidad de que al mismo tiempo Borges *sea* y *no sea* un escritor, o haríamos que el autor entre en un dilema existencial muy propio de sus narraciones. Estas definiciones son simples y fáciles de comprender pero a partir de ellas podemos identificar dos principios fundamentales de la lógica clásica, que podemos enunciar como:

Principio de Contradicción: Dadas dos proposiciones contradictorias entre sí, no pueden ser ambas verdaderas.

Principio de Tercero Excluido: Dadas dos proposiciones contradictorias entre sí, no pueden ser ambas falsas.

Denotaremos a las proposiciones con letras minúsculas del alfabeto (generalmente comenzando con la p, q, r, etc.) con la siguiente notación:

p: Jorge Luis Borges es un escritor

Cuando hagamos referencia a las letras p, q, r… sin aclarar explícitamente a qué proposición se refieren estamos hablando de variables proposicionales, como representantes o contenedores de cualquier proposición y no alguna en particular. Esto nos permitirá establecer un grado de abstracción para representar de forma más conveniente las relaciones entre distintas proposiciones.

Por otro lado denotaremos con letras mayúsculas P, Q, R… a las proposiciones compuestas, considerando por ejemplo:

P = P(p, q, r..., s)

Para expresar que la proposición compuesta P está formada por una combinación de proposiciones simples p, q, r..., s. El valor de verdad de una proposición compuesta depende directamente de los valores de verdad de las proposiciones simples que la conforman y de cómo se encuentran conectadas. Veremos más sobre cómo conectar proposiciones simples a continuación.

Valores de verdad

Entramos en el estudio semántico cuando hacemos referencia al carácter de verdad o falsedad que pueda tener una proposición. Al hacer referencia al posible valor de verdad o falsedad que pueda tener una fórmula estamos admitiendo un principio, el principio de *bivalencia*: todo enunciado es o verdadero o falso, pero no ambas cosas a la vez.

El principio de bivalencia puede aplicarse tanto a las proposiciones simples como a las compuestas. Si una proposición es verdadera, se dirá que tiene valor de verdad positivo; si es falsa, negativo. El criterio que se adopta para atribuir valor de verdad o falsedad a una proposición simple no es un problema de análisis lógico, sino un problema de experiencia. Si lo enunciado en una proposición simple está conforme con los hechos la proposición es verdadera, de lo contrario es falsa.

Un segundo principio de la lógica bivalente es aquel que mantiene que el valor de verdad de las proposiciones compuestas depende del valor de verdad de las proposiciones simples que la forman. En este sentido, podemos decir que las fórmulas compuestas son también funciones de verdad o funciones veritativas, ya que los valores que adoptan son valores de verdad.

Para determinar el valor de verdad de una proposición compuesta, independientemente de los valores de sus componentes, existe un procedimiento mecánico: las **tablas de verdad**.

Para construir las tablas de verdad hemos de tener en cuenta el número de filas con valor de verdad V y de falsedad F, de los que ha de constar la tabla. Así, para una sola variable proposicional p, la tabla tendría 2 filas con la siguiente representación:

p
1
0

Operadores Lógicos

Podemos formar nuevas proposiciones de otras en varias formas utilizando operadores. Por ejemplo, comenzando con p: "soy de Córdoba," que pueden constituir la negación de p: "no soy de Córdoba". Representamos la negación de p por ~p, que se lee "no p." Lo que queremos decir con esto es que, si p es verdadera, entonces ~p es falsa, y viceversa. Podemos demostrar esto en la forma de una tabla de verdad:

p	~p
V	F
F	V

A la izquierda son los dos valores de verdad posibles de p, con los correspondientes valores de verdad de ~p de la derecha. El símbolo ~ es nuestro primer ejemplo de un operador lógico.

Lo que sigue es una definición formal.

Negación

La negación de p es la proposición ~p, que se lee "no p. "Su valor de verdad se define por la siguiente tabla de verdad.

p	~p
V	F
F	V

El símbolo de la negación "~" es un ejemplo de operador **unario** lógico. El término "unario" indica que el operador actúa en una única proposición y no sobre una combinación de ellas.

Es importante destacar que ~p es falsa ya que p es verdadera. Sin embargo, ~q es verdadera ya que q es falsa. Una declaración de la forma ~q también puede ser verdadera y es un error común pensar que debe ser falsa. La negación no es el polo opuesto, pero cualquiera que puede negar la verdad de la declaración original.

Conjunción

Veamos otra forma en la que podemos formar nuevas proposiciones desde otras.

A partir de p: "Hace calor" y q: "Hoy es Viernes", se puede formar la proposición "Hace calor y hoy es viernes".

Indicamos esta nueva proposición por $p \wedge q$, y se lee "p y q". Para que sea verdadera $p \wedge q$ ambas p y q deben ser verdaderas. Así, por ejemplo, si hace calor pero hoy es Lunes, entonces $p \wedge q$ es falsa.

El símbolo $\wedge$ es otro operador lógico. La declaración $p \wedge q$ se llama la **conjunción de p y q**. Su valor de verdad se define por el texto de la siguiente tabla de verdad.

Ejemplo ➔	Hace calor	Hoy es viernes	Hace calor **y** hoy es viernes
	p	q	$p \wedge q$
	V	V	V
	V	F	F
	F	V	F

F	F	F

En las columnas p y q se enumeran las cuatro posibles combinaciones de valores de verdad para p y q, y en la columna de p∧q se encuentra el valor de verdad asociada por p∧q. Por ejemplo, las entradas de la tercera fila nos dicen que, si p es falsa y q es verdadera, entonces p∧q es falsa. Casualmente la única manera de obtener una V en el columna de p∧q es que sean verdaderas p y q como se indica en la tabla de verdad.

El símbolo de conjugación "∧" es un ejemplo de un operador lógico **binario.** La palabra "binario" indica que el operador actúa en un par de proposiciones.

En el lenguaje de la lógica proposicional los significados de las expresiones deben comunicarse de forma precisa y sin ambigüedades para poder usarlas para sacar conclusiones sólidas. En el lenguaje natural un enunciado se puede interpretar de formas distintas. Las ambigüedades se resuelven usando el contexto o quedan sin dilucidar. En este contexto los paréntesis nos permiten agrupar expresiones e indicar el orden exacto en el que se deben evaluar.

Podemos usar la tabla de verdad para calcular los posibles valores de verdad de proposiciones compuestas. Para hacerlo agregamos columnas adicionales con proposiciones compuestas que dependen únicamente de las proposiciones a su izquierda. En los casos más sencillos aplicamos solamente una conectiva lógica a las proposiciones simples. Por ejemplo, si tenemos las proposiciones p y q y les aplicamos una conjunción y negación de la forma p ^ ~q, la tabla de verdad resultante será:

Ejemplo ➔	Hace Calor	Hoy es viernes	Hoy no es viernes	Hace calor **y** hoy no es viernes
	p	q	~q	p∧~q
	V	V	F	F
	V	F	V	V
	F	V	F	F
	F	F	V	F

Para crear la tabla de verdad de una proposición más compleja debemos:

1. Separar la proposición en proposiciones cada vez más sencillas. Para hacer esto debemos analizar la proposición considerando la precedencia de los operadores y su agrupación si hay paréntesis.

2. Agregar una columna en la tabla de verdad por cada proposición más sencilla. Las columnas se deben organizar de forma que las proposiciones correspondientes sólo dependan de las proposiciones simples y de las proposiciones más sencillas que se encuentran a su izquierda.
3. Calcular los valores de verdad para cada una de las proposiciones más sencillas hasta llegar a la proposición original.

Disyunción

Ahora veremos un tercer operador lógico. A partir una vez más con p: "Hace calor", y q: "Hoy es Viernes", se puede formar la declaración "Hace calor u hoy es Viernes", que se puede escribir simbólicamente como p∨q, que se lee "p o q".

En español la palabra "o" tiene varios significados posibles por lo que es necesario acordar qué significado le daremos en este contexto, veamos algunos ejemplos:

P: José Hernandez es escritor o es jardinero

En caso de usar el operador o de forma inclusiva, diremos que la proposición P es verdadera en caso de que José Hernandez sea efectivamente escritor pero no jardinero, aunque también puede ser verdadera si es escritor y tiene una afición hacia la jardinería. En cambio si consideramos el operador de forma exclusiva, el valor de P verdadero será sólo cuando exactamente una de las condiciones sea verdadera, es decir que sea escritor y no jardinero o viceversa.

Matemáticos se han asentado sobre el uso del “o” inclusivo: p∨q significa que p es verdadera o q es verdadera, o que ambas son verdaderas.

Entonces si quisiéramos explicitar el uso del “o” inclusivo podríamos expresar con p y q como antes, "Hace calor u hoy es Viernes, o ambos". Llamamos p∨q la disyuntiva de p y q.

La disyunción de p y q es la declaración p∨q, que se lee "p o q" y su valor de verdad se define por la siguiente tabla de verdad:

Ejemplo ➔	Hace calor	Hoy es viernes	Hace calor **u** hoy es viernes
	p	q	p∨q
	V	V	V
	V	F	V
	F	V	V
	F	F	F

Este es el inclusivo o, por lo que p∨q es verdadera q cuando p es verdadera, q es verdadera o las dos son verdaderas. Es importante destacar que la única forma para que la disyunción sea falsa es que ambas proposiciones lo sean. El símbolo de disyunción "∨" es nuestro segundo ejemplo de un operador lógico binario.

Ejercicios

Ejemplo 1 *Negando proposiciones*

- Exprese las negaciones de las siguientes proposiciones.
    - p: "2+2 = 4"
    - q: "1 = 0"
    - r: "Los diamantes son el mejor amigo de una perla."
    - s: "Todos los políticos en esta ciudad son ladrones. "

Solución

- **(a)** ~p es la proposición "no es cierto que 2+2 = 4," o más sencillamente,
- ~p: "2+2 ≠ 4."
- **(b)** ~q: "1 ≠ 0."
- **(c)** ~r: "Los diamantes no son son el mejor amigo de una perla".
- **(d)** ~s: "Los políticos en esta ciudad no son todos ladrones." [Pero puede ser algunos...]

Ejemplo 2 *Conjunción*

- Si p: "Esta galaxia, en última instancia, terminará en un agujero negro" y q: "2+2 = 4," ¿entonces qué significa p∧q?

Solución

- p∧q: "Esta galaxia, en última instancia, desaparece en un agujero negro y 2+2=4," o la más sorprendente declaración: "¡No sólo desaparece en última instancia esta galaxia desaparece en un agujero negro, pero 2+2 = 4!"

Ejemplo 3 Disyunción

- Sean p: "El mayordomo lo hizo", q: "El cocinero lo hizo", y r: "El abogado lo hizo".
  - **(a)** ¿Qué significa p∨q?
  - **(b)** ¿Qué significa (p∨q) ∧(~r)?

Solución

- **(a)** p∨q: "o bien el mayordomo o el cocinero lo hizo".
- (Recuerde que esto no excluye la posibilidad de que el mayordomo y el cocinero lo hicieron ambos o que fueron en realidad la misma persona! La única forma en que p∨q podría ser falsa, si ni el mayordomo, ni el cocinero lo hicieron).
- **(b)** (p∨q)∧(~r) dice que "el mayordomo o el cocinero lo hizo, pero no el abogado".

Elementos informáticos

El Software y sus características

El software en sus comienzos era la parte insignificante del hardware, lo que venía como añadidura, casi como regalo. Al poco tiempo adquirió una entidad propia.
En la actualizad, el software es la tecnología individual más importante en el mundo. Nadie en la década de 1950 podría haber predicho que el software se convertiría en una tecnología indispensable en los negocios, la ciencia, la ingeniería; tampoco podría preverse que una compañía de software podría volverse más grande e influyente que la mayoría de las compañías de la era industrial; que una red construida con software, llamada Internet cubriría y cambiaría todo, desde la investigación bibliográfica hasta las compras de los consumidores y los hábitos de las personas. Nadie podría haber imaginado que estaría relacionado con sistemas de todo tipo: transporte, medicina, militares, industriales, entretenimiento, automatización de hogares.

Una definición formal de software según la IEEE (Instituto de Ingeniería Eléctrica y Electrónica) es la siguiente:

> Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación.

El software puede definirse como “el alma y cerebro de la computadora, la corporización de las funciones de un sistema, el conocimiento capturado acerca de un área de aplicación, la colección de los programas, y los datos necesarios para convertir a una computadora en una máquina de propósito especial diseñada para una aplicación particular, y toda la información producida durante el desarrollo de un producto de software”. El software viabiliza el producto más importante de nuestro tiempo: la información.

Características del software:

1. El software es intangible, es decir, que se trata de un concepto abstracto.
2. Tiene alto contenido intelectual.
3. Su proceso de desarrollo es humano intensivo, es decir que la materia prima principal radica en la mente de quienes lo crean.
4. El software no exhibe una separación real entre investigación y producción.
5. El software puede ser potencialmente modificado, infinitamente.
6. El software no se desgasta
7. La mayoría del software, en su mayoría, aún se construye a medida.
8. El software no se desarrolla en forma masiva, debido a que es único.

Estructura Interna de una Computadora

Una computadora moderna consta de uno o más procesadores, una memoria principal, discos, impresoras, un teclado, un ratón, una pantalla o monitor, interfaces de red y otros dispositivos de entrada/salida. En general es un sistema complejo. Si todos los programadores de aplicaciones tuvieran que comprender el funcionamiento de todas estas partes, no escribirían código alguno. Es más: el trabajo de administrar todos

estos componentes y utilizarlos de manera óptima es una tarea muy desafiante. Por esta razón, las computadoras están equipadas con una capa de software llamada sistema operativo, cuyo trabajo es proporcionar a los programas de usuario un modelo de computadora mejor, más simple y pulcro, así como encargarse de la administración de todos los recursos antes mencionados.

La mayoría de las computadoras, grandes o pequeñas, están organizadas como se muestra en la siguiente figura. Constan fundamentalmente de tres componentes principales: Unidad Central de Proceso (UCP) o procesador, la memoria principal o central.

Si a los componentes anteriores se les añaden los dispositivos para comunicación con la computadora, aparece la estructura típica de un sistema de computadora: dispositivos de entrada, dispositivos de salida, memoria externa y el procesador/memoria central.

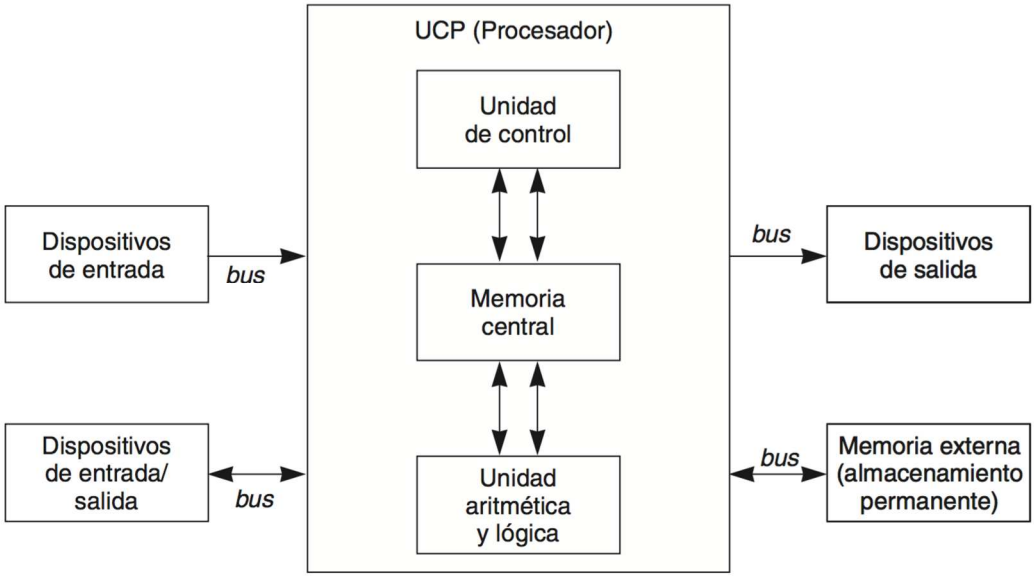


Figura 1: Estructura interna de un sistema de computadora

Los dispositivos de Entrada/Salida (E/S) (en inglés, Input/Output I/O) o periféricos permiten la comunicación entre la computadora y el usuario.

Los dispositivos de entrada, como su nombre indica, sirven para introducir datos en la computadora para su proceso. Los datos se leen de los dispositivos de entrada y se almacenan en la memoria central o interna. Los dispositivos de entrada convierten la información de entrada en señales eléctricas que se almacenan en la memoria central. Dispositivos de entrada típicos son teclados, lápices ópticos, joysticks, lectores de códigos de barras, escáneres, micrófonos, lectores de tarjetas digitales, lectores RFID (tarjetas de identificación por radio frecuencia), etc. Hoy, tal vez el dispositivo de entrada más popular es el ratón (mouse) que mueve un puntero gráfico (electrónico) sobre la pantalla, o más recientemente las pantallas táctiles, que facilitan la interacción usuario-máquina.

Los dispositivos de salida permiten representar los resultados (salida) del proceso. El dispositivo de salida típico es la pantalla o monitor. Otros dispositivos de salida son: impresoras (imprimen resultados en papel), trazadores gráficos (plotters), reconocedores (sintetizadores) de voz, parlantes, entre otros.

Los dispositivos de entrada/salida y dispositivos de almacenamiento masivo o auxiliar (memoria externa) son: unidad de discos (disquetes, CD-ROM, DVD, discos duros, etc.), videocámaras, memorias flash, USB, etc.

La memoria central o simplemente memoria (interna o principal) se utiliza para almacenar información (RAM, del inglés Random Access Memory). En general, la información almacenada en la memoria puede ser de dos tipos: instrucciones de un programa y datos con los que operan las instrucciones. Por ejemplo, para que un programa se pueda ejecutar (correr, funcionar..., en inglés, run), debe ser situado en la memoria central, en una operación denominada carga (load) del programa. Después, cuando se ejecuta el programa, cualquier dato a procesar se debe llevar a la memoria mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta a fin de poder funcionar.

La memoria central de una computadora es una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de celdas de memoria (estas celdas o posiciones de memoria se denominan también palabras, aunque no guardan analogía con las palabras del lenguaje). El número de celdas de memoria de la memoria central, depende del tipo y modelo de computadora; hoy día el número suele ser millones (512, 1.024, etc.). Cada celda de memoria consta de un cierto número de bits (normalmente 8, un byte).

La unidad elemental de memoria se llama byte. Un byte tiene la capacidad de almacenar un carácter de información, y está formado por un conjunto de unidades más pequeñas de almacenamiento denominadas bits, que son dígitos binarios que pueden asumir como valor un 0 o un 1.

Siempre que se almacena una nueva información en una posición, se destruye (desaparece) cualquier información que en ella hubiera y no se puede recuperar. La dirección es permanente y única, el contenido puede cambiar mientras se ejecuta un programa.

La memoria central de una computadora puede tener desde unos centenares de miles de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos de potencia de 2 para definir el tamaño de la memoria central: Kilobyte (KB o Kb) igual a 1.024 bytes (2^{10}) —prácticamente se consideran 1.000—; Megabyte (MB o Mb) igual a 1.024 × 1.024 bytes = = 1.048.576 (2^{20}) —prácticamente se consideran 1.000.000; Gigabyte (GB o Gb) igual a 1.024 MB (2^{30}), 1.073.741.824 = prácticamente se consideran 1.000 millones de MB.

Byte	**Byte (B)**	*equivale a*	8 bits
Kilobyte	**Kbyte (KB)**	*equivale a*	1.024 bytes
Megabyte	**Mbyte (MB)**	*equivale a*	1.024 Kbytes
Gigabyte	**Gbyte (GB)**	*equivale a*	1.024 Mbytes
Terabyte	**Tbyte (TB)**	*equivale a*	1.024 Gbytes
1 **Tb** = 1.024 **Gb** = 1.024 × 1.024 **Mb** = 1.048.576 **Kb** = 1.073.741.824 **B**			

Tabla 1: Unidades de medida para el almacenamiento en la memoria

La Unidad Central de Proceso UCP, o procesador, dirige y controla el proceso de información realizado por la computadora. La UCP procesa o manipula la información almacenada en memoria; puede recuperar información desde memoria (esta información son datos o instrucciones de programas) y también puede almacenar los resultados de estos procesos en memoria para su uso posterior.

Más adelante veremos en profundidad cómo los programas hacen uso de la memoria para almacenar o leer datos a fin de utilizarlos para el desarrollo de sus funciones.

Redes de Computadoras

La fusión de las computadoras y las comunicaciones ha tenido una profunda influencia en cuanto a la manera en que se organizan los sistemas de cómputo. El concepto una vez dominante del "centro de cómputo" como un salón con una gran computadora a la que los usuarios llevaban su trabajo para procesarlo es ahora totalmente obsoleto, (aunque los centros de datos que contienen miles de servidores de Internet se están volviendo comunes). El viejo modelo de una sola computadora para atender todas las necesidades computacionales de la organización se ha reemplazado por uno en el que un gran número de computadoras separadas pero interconectadas realizan el trabajo. A estos sistemas se les conoce como redes de computadoras.

Se dice que dos computadoras están interconectadas si pueden intercambiar información. La conexión no necesita ser a través de un cable de cobre; también se puede utilizar fibra óptica, microondas, infrarrojos y satélites de comunicaciones. Las redes pueden ser de muchos tamaños, figuras y formas, como veremos más adelante. Por lo general se conectan entre sí para formar redes más grandes, en donde Internet es el ejemplo más popular de una red de redes.

Imaginemos el sistema de información de una empresa como si estuviera constituido por una o más bases de datos con información de la empresa y cierto número de empleados que necesitan acceder a esos datos en forma remota. En este modelo, los datos se almacenan en poderosas computadoras denominadas servidores. A menudo estos servidores están alojados en una ubicación central y un administrador de sistemas se encarga de su mantenimiento. Por el contrario, los empleados tienen en sus escritorios máquinas más simples conocidas como clientes, con las cuales acceden a los datos remotos, por ejemplo, para incluirlos en las hojas de cálculo que desarrollan (algunas veces nos referiremos al usuario humano del equipo cliente como el "cliente", aunque el contexto debe dejar en claro si nos referimos a la computadora o a su usuario). Las máquinas cliente y servidor se conectan mediante una red, como se muestra en la figura 2.

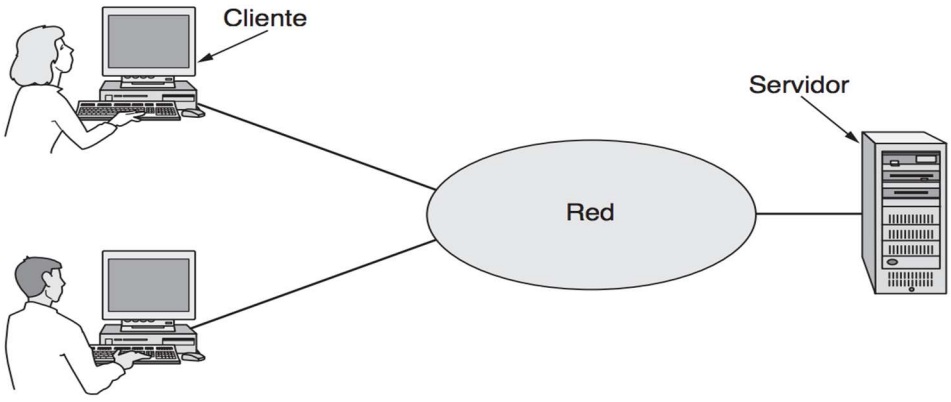


Figura 2: Esquema de una red de computadoras

A esta disposición se le conoce como modelo cliente-servidor. Es un modelo ampliamente utilizado y forma la base de muchas redes. La realización más popular es la de una aplicación web, en la cual el servidor genera

páginas web basadas en su base de datos en respuesta a las solicitudes de los clientes que pueden actualizarla. El modelo cliente-servidor es aplicable cuando el cliente y el servidor se encuentran en el mismo edificio (y pertenecen a la misma empresa), pero también cuando están muy alejados. Por ejemplo, cuando una persona accede desde su hogar a una página en Internet se emplea el mismo modelo, en donde el servidor web remoto representa al servidor y la computadora personal del usuario representa al cliente. En la mayoría de las situaciones un servidor puede atender un gran número (cientos o miles) de clientes simultáneamente.

La evolución de las comunicaciones y los dispositivos personales, así como las necesidades emergentes de compartir información en tiempo real han posibilitado la expansión de Internet a todos los rincones del mundo. De esta forma cualquier persona puede acceder a sus archivos, compartir datos, comunicarse o buscar información en cualquier momento a través de su computadora, notebook, teléfonos celulares entre otros, tal como se muestra en la siguiente figura.

Figura 3: Integración de tecnología a través de Internet

Programación y construcción de Software

El único tipo de instrucciones que una computadora puede entender es el lenguaje de máquina, o lenguaje de bajo nivel, donde diferentes tipos de procesadores pueden tener distintos lenguajes de máquina. El lenguaje máquina está compuesto de ceros y unos lo que hace que programar en lenguaje máquina sea un proceso tedioso y sujeto a errores.

Una alternativa a utilizar lenguaje de máquina es el lenguaje Assembly, Assembler o ensamblador, que es también un lenguaje de bajo nivel que utiliza mnemonics (o abreviaturas) y es más fácil de entender que ceros y unos. Sin embargo, el único lenguaje que una computadora puede entender directamente es el lenguaje máquina, ¿entonces cómo es posible que entienda lenguajes como Assembler? La respuesta es que el lenguaje Assembler es convertido o traducido a lenguaje de máquina mediante un programa llamado *ensamblador*. Es importante destacar que hay una correspondencia directa entre el lenguaje Assembler y el

lenguaje máquina, lo que significa que para cada instrucción de lenguaje assembler existe una instrucción de lenguaje máquina, lo que hace la traducción un proceso directo.

Sin embargo, más allá de que el lenguaje Assembler es más sencillo que el lenguaje máquina, distintos tipos de procesadores tienen diferentes conjuntos de instrucciones lo que se traduce en distintos dialectos de Assembler de una computadora a otra.

La solución para hacer la tarea de programación más sencilla y posibilitar a los programas funcionar en distintos tipos de computadoras es utilizar lenguajes de alto nivel, que son más similares al lenguaje natural que utilizamos para comunicarnos diariamente y por motivos históricos estos lenguajes utilizan palabras del idioma inglés. Uno de los primeros lenguajes de programación de alto nivel fue FORTRAN (del inglés FORmula TRANslation, o traducción de fórmulas) que fue desarrollado en los comienzos de los años 50 para ayudar a resolver problemas matemáticos. Desde ese entonces, una gran cantidad de lenguajes de programación de alto nivel han sido creados para abordar distintos tipos de problemas y solucionar las necesidades de distintos tipos de usuarios. Algunos de ellos incluyen a COBOL, también desarrollado en los 50 para abordar aplicaciones empresariales y de negocios; BASIC en los 60 para programadores recién iniciados, Pascal en los 70 para problemas científicos, C, C++ y muchos otros. En este material nos centraremos en el lenguaje Java, también de alto nivel y de propósito general: es decir que puede usarse para una gran variedad de problemas y rubros.

Los sistemas y su enfoque

¿Por qué hablamos de *sistemas*?

En la primera mitad del siglo XX, surgió la necesidad de diseñar métodos de investigación y estudio de los fenómenos complejos a causa de una acumulación de problemáticas en las que los métodos de investigación de las ciencias particulares se mostraban insuficientes. Por un lado, los *nuevos sistemas de producción* que incluían varias automatizaciones, el manejo de grandes cantidades de energía (termoeléctrica, nuclear…) que requería de especialistas de variadas ramas, el desarrollo y organización de transporte terrestre, marítimo y aéreo y otros fenómenos. Por otro, los *grandes desarrollos científicos* en la física (relatividad, estructura atómica, mecánica cuántica), biología (genética, evolución, estudio de poblaciones), química (teoría del enlace de Lewis, tabla periódica, estructura cristalina), matemática (álgebra de Boole, desarrollo del cálculo, problemas de Hilbert). Estas grandes revoluciones en el hacer y el pensar hicieron necesario el desarrollo de un enfoque complejo para la investigación de fenómenos complejos. Así nació el *enfoque sistémico*, sustentado por la Teoría General de los Sistemas (TGS) formulada por Ludwig von Bertalanffy a mediados del siglo XX.

Bertalanffy se dedicó especialmente a los organismos como sistemas biológicos, pero luego generalizó su estudio a todo tipo de sistemas. De tal manera que hoy se utiliza el término sistema en todas las áreas del conocimiento humano.

¿Qué es un Sistema?

Llamamos **sistema** a todo conjunto de elementos relacionados entre sí –puede ser por una finalidad en común-, que tienen un cierto orden u organización y que cumplen una función.

Los sistemas tienen **composición** (los elementos que lo forman), una **estructura** interna dada por el conjunto de relaciones entre sus componentes. Y también tienen un **entorno o ambiente** que es el conjunto de cosas que no pertenecen al sistema pero que actúan sobre él o sobre las que él actúa intercambiando **materia, energía** e **información** (MEI).

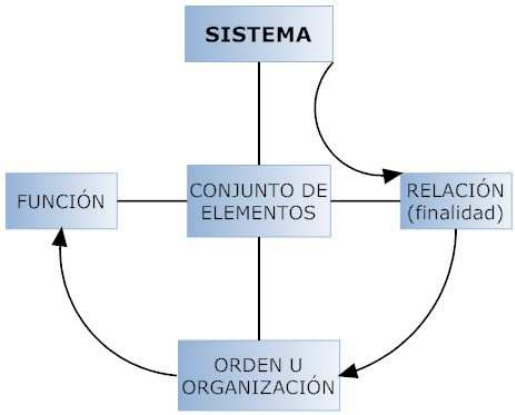


Figura 4: Elementos de un sistema

Los sistemas están inmersos en un **entorno o ambiente,** que es el conjunto de elementos que está fuera del sistema, es decir que no pertenecen al sistema pero que actúan sobre él o sobre las que el sistema actúa intercambiando **materia, energía** e **información** (MEI).

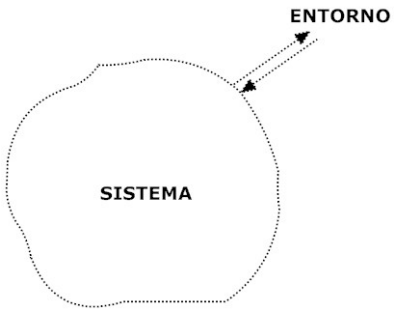


Figura 5: Relación del sistema con su entorno

Características de los sistemas

La característica principal de los sistemas es que poseen una **propiedad emergente** que no poseen sus componentes particulares. Por ejemplo, *la vida* es la propiedad emergente de un sistema compuesto por huesos, órganos, etc.; *marchar* es la propiedad emergente del sistema automóvil compuesto por chapas, motor, luces, etc. Este hecho se suele enunciar con la siguiente afirmación

EL TODO ES MÁS QUE LA SUMA DE LAS PARTES

Otras características de los sistemas son:

a. **Límite o frontera**: Son demarcaciones que permiten establecer qué elementos pertenecen o no al sistema. Los límites pueden ser:
   - *Concretos*: los que tienen existencia material (ríos que separan países, paredes que definen aulas, etc.)
   - *Simbólicos*: los que no tienen existencia material y vienen dados por acuerdos, reglas o normas (un alumno pertenece a un curso porque lo establece la escuela, más allá de que pueda hallarse en otro salón o fuera de la misma)
b. **Depósitos o almacenamientos**: son lugares donde se almacena materia, energía o información (MEI). Los depósitos pueden ser:
   - *Permanentes*: aquellos en que están diseñados para que su contenido no se altere (CD-ROM, libros, carteles fijos, etc.)
   - *Transitorios*: aquellos diseñados para que su contenido sufra modificaciones (pizarrón, cartuchera, tanques de agua, etc.)
c. **Canales**: Son lugares o conductos por donde circula materia, energía o información (MEI). Los canales pueden comunicar dos sistemas entre sí o partes de un mismo sistema (las calles pueden ser canales de materia, los cables pueden ser canales de energía si llevan corriente o de información si son telefónicos o de redes, etc.)
d. **Subsistemas**: los sistemas complejos (muchos componentes y relaciones entre ellos) pueden dividirse para su estudio en subsistemas. Esto permite diferentes niveles de estudio de los mismos. Se llama *nivel cero* al análisis del sistema en su totalidad y su intercambio con el entorno. A partir de allí se define el nivel 1, nivel 2, etc.

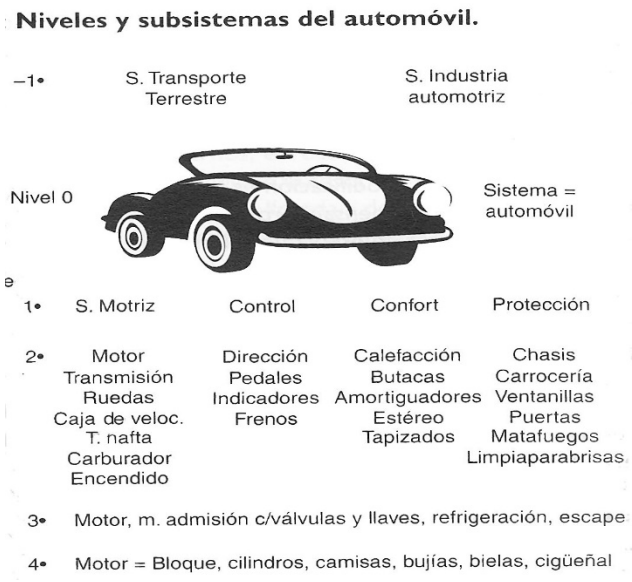


Figura 6: Sistemas y Subsistemas

Intercambio entre sistemas

Los sistemas intercambian entre sí materia, energía e información (MEI). Para que se dé este intercambio es necesario que MEI atraviese los límites del sistema hacia (o desde) el entorno. Si el sistema intercambia con el medio se dice que es *abierto*, de lo contrario se considera *cerrado*.

En *sistemas cerrados* cualquier estado final está determinado por sus condiciones iniciales, ya que no hay modo de que el entorno actúe sobre él. Si un sistema cerrado tampoco intercambia energía se dice que es aislado. En realidad, el único sistema que se considera absolutamente aislado es el universo. De igual modo, muchos sistemas mecánicos e informáticos pueden considerarse razonablemente cerrados.

Los *sistemas abiertos*, en cambio, pueden, crecer, cambiar, adaptarse al ambiente, incluso algunos reproducirse. Si un sistema posee la organización necesaria para controlar su propio desarrollo, asegurando la continuidad de su composición y estructura (*homeostasis*) y la de los flujos y transformaciones con que funciona (*homeorresis*) –mientras las perturbaciones producidas desde su entorno no superen cierto grado–, entonces el sistema es *autopoyético*. Los seres vivos, los ecosistemas y organizaciones sociales pueden considerarse sistemas abiertos.

Estos flujos de MEI se pueden representar en diagramas como el siguiente

Figura 7: Entradas y Salidas

Para clarificar, las líneas de los diferentes flujos pueden representarse por diferentes colores o trazos.

Este es el *nivel cero* de representación de un sistema, con las entradas y salidas de MEI que atraviesan sus límites. Este tipo de representaciones se denomina *diagrama de entrada y salida* (E/S o U/O) o *diagrama de caja negra*, ya que no interesa mostrar qué sucede dentro del sistema.

Sistemas tecnológicos

Los **sistemas tecnológicos**, son aquellos diseñados por los seres humanos para que cumplan con una finalidad específica. Por eso se dice que son *sistemas teleológicos artificiales* (del griego *telos* = fin). La orientación para al fin que se busca suele definir la propiedad emergente del sistema tecnológico. En el ejemplo del automóvil, la propiedad emergente de marchar también se busca como finalidad o propósito del sistema.

Es conveniente aclarar que *los sistemas son recortes de la realidad* que alguien se propone estudiar o considerar; a ese recorte se le llama ***Abstracción***. En algunos sistemas tecnológicos como un automóvil es sencillo identificar este recorte. Sin embargo, en la red de generación y distribución de energía eléctrica del país no resulta tan sencillo.

Algunos sistemas tecnológicos se caracterizan por procesar materia: son los **sistemas de procesamiento de materia** (SM). Estos están diseñados para producir, procesar, generar, transformar o distribuir materiales. Las industrias, las huertas, las licuadoras, etc. pueden considerarse SM.

Otros se caracterizan por procesar energía, los **sistemas de procesamiento de energía** (SE). Estos están diseñados para generar, transformar, distribuir energía. Los ventiladores, automóviles, represas hidroeléctricas, explosivos, etc. pueden considerarse SE.

Los que se caracterizan por procesar información se llaman **sistemas de información** (SI). Están diseñados con el fin de generar, trasformar y distribuir información entre otras tareas. Los sistemas que controlan los automóviles, las redes sociales, los sistemas de punto de venta, el comercio electrónico, por mencionar algunos, son ejemplos de SI.

Desde la aparición del software, los SI han incorporado el software para hacer más eficiente su funcionamiento a un grado tal que se los denomina **Sistemas Informáticos,** acoplando la palabra “automático” a la palabra “información”.

¿Cómo se construye el Software?

El software, como cualquier otro producto, se construye aplicando un proceso que conduzca a un resultado de calidad, que satisfaga las necesidades de quienes lo utilizan. Un proceso de desarrollo de software es una secuencia estructurada de actividades que conduce a la obtención de un producto de software. En definitiva, un proceso define quién está haciendo qué, cuándo y cómo alcanzar un determinado objetivo. En este caso el objetivo es construir un producto de software nuevo o mejorar uno existente.

Figura 9: Proceso de Construcción del Software

Pueden identificarse cuatro actividades fundamentales que son comunes a todos los procesos de software:

1. **Especificación del software**: donde clientes y profesionales definen el software que se construirá, sus características y las restricciones para su uso.
2. **Desarrollo del software**, donde se diseña y programa el software.
3. **Validación del software**, donde se controla que el software satisfaga lo que el cliente quiere.
4. **Evolución del software**, donde se incorporan mejoras y nuevas características que permitirán a ese producto adaptarse a las necesidades cambiantes del cliente y el mercado.

Si consideramos las características del software que se explicaron anteriormente, determinamos como conclusión que el software no se obtiene por medio de un proceso de manifactura en serie o como líneas de producción, sino que para obtenerlo usamos un **proyecto**, que se lo puede definir como un esfuerzo planificado, temporal y único, realizado para crear productos o servicios únicos que agreguen valor. Estos proyectos utilizan **procesos** que definen que tareas deben realizar las personas que trabajan en el proyecto, para obtener los resultados deseados, utilizando como apoyo herramientas que facilitarán su trabajo. En este caso el resultado deseado en el **Producto de Software.**

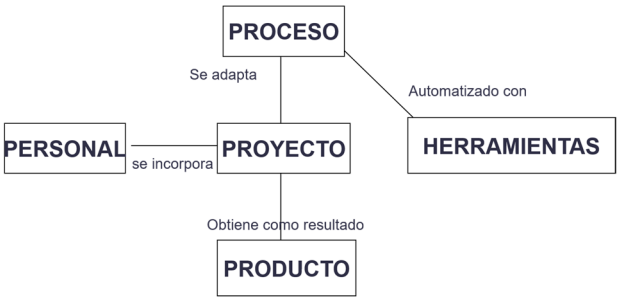


Figura 10: Relación entre Proceso, Proyecto y Producto en el desarrollo de Software

Diseño de Algoritmos

El hombre, en el día a día, se enfrenta constantemente a diferentes problemas que debe solucionar y para lograr solucionarlos hace uso de herramientas que le facilitan la tarea. Así, podemos pensar el uso de una calculadora para poder sumar el precio de los productos en un local y así cobrarle al cliente.

Al igual que la calculadora, la computadora también sirve para resolver problemas, pero la diferencia está en la capacidad de procesamiento de las computadoras, que hace que se puedan resolver problemas de gran complejidad, que, si los quisiéramos resolver manualmente, nos llevaría mucho tiempo o ni siquiera podríamos llegar a resolverlos.

Un programador es antes que nada una persona que resuelve problemas; el programador procede a resolver un problema, a partir de la definición de un algoritmo y de la traducción de dicho algoritmo a un programa que ejecutará la computadora.

En la oración anterior se nombran algunos conceptos que debemos profundizar:

Algoritmo: un algoritmo es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema. Así, como en la vida real, cuando tenemos que resolver un problema, o lograr un objetivo, por ejemplo: “Tengo que atarme los cordones”, para alcanzar la solución de ese problema, realizamos un conjunto de pasos, de manera ordenada y secuencial. Es decir, podríamos definir un algoritmo para atarnos los cordones de la siguiente forma:

1. Ponerme las zapatillas.
2. Agarrar los cordones con ambas manos.
3. Hacer el primer nudo.
4. Hacer un bucle con cada uno de los cordones.
5. Cruzar los dos bucles y ajustar.
6. Corroborar que al caminar los cordones no se sueltan y la zapatilla se encuenta correctamente atada.

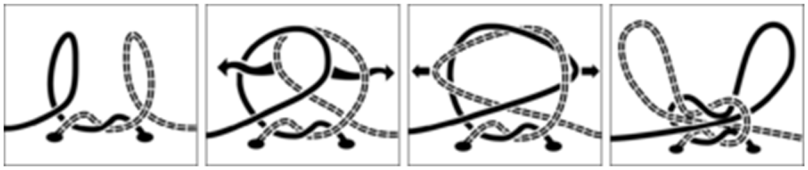


Figura 11: Algoritmo gráfico para atarse los cordones

El concepto de algoritmo es fundamental en el proceso de programación de una computadora, pero si nos detenemos a observar a nuestro alrededor, así como el ejemplo anterior podemos descubrir muchos otros: nos están dando un algoritmo cuando nos indican la forma de llegar a una dirección dada, seguimos

algoritmos cuando conducimos un automóvil o cualquier tipo de vehículo. Todos los procesos de cálculo matemático que normalmente realiza una persona en sus tareas cotidianas, como sumar, restar, multiplicar o dividir, están basados en algoritmos que fueron aprendidos en la escuela primaria. Como se ve, la ejecución de algoritmos forma parte de la vida moderna.

Por otro lado, la complejidad de los distintos problemas que podamos abordar puede variar desde muy sencilla a muy compleja, dependiendo de la situación y la cantidad de elementos que intervienen. En casos de mayor complejidad suele ser una buena solución dividir al problema en diferentes subproblemas que puedan ser resueltos de manera independiente. De esta forma la solución final al problema inicial será determinada por las distintas soluciones de los problemas más pequeños cuya resolución es más sencilla.

Programa: luego de haber definido el algoritmo necesario, se debe traducir dicho algoritmo en un conjunto de instrucciones, entendibles por la computadora, que le indican a la misma lo que debe hacer; este conjunto de instrucciones conforma lo que se denomina, un programa.

Para escribir un programa se utilizan lenguajes de programación, que son lenguajes que pueden ser entendidos y procesados por la computadora. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Algoritmos

Concepto

Es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados, tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

• Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.

• Un algoritmo debe estar específicamente definido. Es decir, si se ejecuta un mismo algoritmo dos veces, con los mismos datos de entada, se debe obtener el mismo resultado cada vez.

• Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos. Debe tener un inicio y un final.

• Un algoritmo debe ser correcto: el resultado del algoritmo debe ser el resultado esperado.

• Un algoritmo es independiente tanto del lenguaje de programación en el que se expresa como de la computadora que lo ejecuta.

Como vimos anteriormente, el programador debe constantemente resolver problemas de manera algorítmica, lo que significa plantear el problema de forma tal que queden indicados los pasos necesarios para obtener los resultados pedidos, a partir de los datos conocidos. Lo anterior implica que un algoritmo básicamente consta de tres elementos: Datos de Entrada, Procesos y la Información de Salida.

Figura 12: Estructura de un programa, datos de entrada y salida

Cuando dicho algoritmo se transforma en un programa de computadora:

- Las entradas se darán por medio de un dispositivo de entrada (como los vistos en el bloque anterior), como pueden ser el teclado, disco duro, teléfono, etc. Este proceso se lo conoce como entrada de datos, operación de lectura o acción de leer.
- Las salidas de datos se presentan en dispositivos periféricos de salida, que pueden ser pantalla, impresora, discos, etc. Este proceso se lo conoce como salida de datos, operación de escritura o acción de escribir.

Dado un problema, para plantear un algoritmo que permita resolverlo, es conveniente entender correctamente la situación problemática y su contexto, tratando de deducir del mismo los elementos ya indicados (entradas, procesos y salida). En este sentido entonces, para crear un algoritmo:

1. Comenzar identificando los resultados esperados, porque así quedan claros los objetivos a cumplir.

2. Luego, individualizar los datos con que se cuenta y determinar si con estos datos es suficiente para llegar a los resultados esperados. Es decir, definir los datos de entrada con los que se va a trabajar para lograr el resultado.

3. Finalmente si los datos son completos y los objetivos claros, se intentan plantear los procesos necesarios para pasar de los datos de entrada a los datos de salida.

Para comprender esto, veamos un ejemplo:

Problema:

Obtención del área de un rectángulo:

Altura: 5 cm

Base: 10 cm

1. Resultado esperado: área del rectángulo. Salida: área
Fórmula del área: base x altura.
2. Los datos con los que se dispone, es decir las entradas de datos son:
    Dato de Entrada 1: altura: 5 cm
    Dato de Entrada 2: base: 10 cm
3. El proceso para obtener el área del rectángulo es:
    área = base * altura ⇨ área = 50

Herramientas para la representación gráfica de los algoritmos

Como se especificó anteriormente, un algoritmo es independiente del lenguaje de programación que se utilice. Es por esto, que existen distintas técnicas de representación de un algoritmo que permiten está diferenciación con el lenguaje de programación elegido. De esta forma el algoritmo puede ser representado en cualquier lenguaje. Existen diversas herramientas para representar gráficamente un algoritmo. En este material presentaremos dos:

1. Diagrama de flujo.

2. Lenguaje de especificación de algoritmos: pseudocódigo.

Diagramas de Flujo

Un diagrama de flujo hace uso de símbolos estándar que unidos por flechas, indican la secuencia en que se deben ejecutar.

Estos símbolos son, por ejemplo:

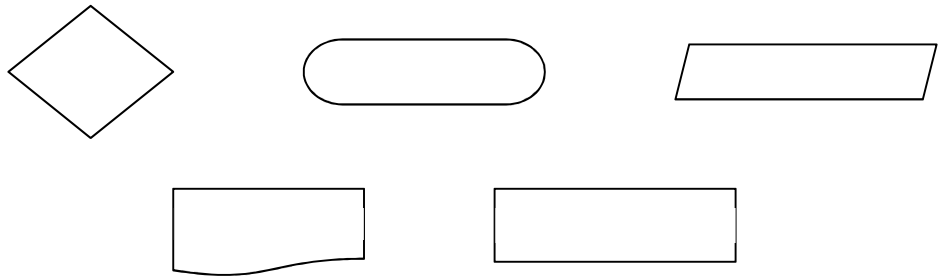


Figura 13: Símbolos utilizados en Diagramas de Flujo

Más adelante, a medida que profundizamos en los temas, retomaremos el uso de esta herramienta y mostraremos algunos ejemplos de su uso.

Pseudocódigo

Conocido como lenguaje de especificación de algoritmos, el pseudocódigo tiene una estructura muy similar al lenguaje natural y sirve para poder expresar algoritmos y programas de forma independiente del lenguaje de programación. Además, es muy utilizado para comunicar y representar ideas que puedan ser entendidas por programadores que conozcan distintos lenguajes. El pseudocódigo luego se traduce a un lenguaje de programación específico ya que la computadora no puede ejecutar el pseudocódigo. Su uso tiene ventajas porque permite al programador una mejor concentración de la lógica y estructuras de control y no preocuparse de las reglas de un lenguaje de programación específico.

Un ejemplo básico de pseudocódigo, considerando el ejemplo utilizado anteriormente, es el siguiente:

```
BEGIN FUNCION CALCULAR_AREA
  DEFINIR BASE: 5
  DEFINIR ALTURA: 10
  DEFINIR AREA: BASE * ALTURA
  DEVOLVER AREA
END
```

Lenguajes de Programación

Concepto

Los lenguajes de programación son lenguajes que pueden ser entendidos y procesados por la computadora.

Tipos de Lenguajes de Programación

Los principales tipos de lenguajes utilizados en la actualidad son tres:

- Lenguajes máquina
- Lenguaje de bajo nivel (ensamblador)
- Lenguajes de alto nivel

La elección del lenguaje de programación a utilizar depende mucho del objetivo del software. Por ejemplo, para desarrollar aplicaciones que deben responder en tiempo real como por ejemplo, el control de la velocidad crucero en un sistema de navegación de un auto[1], debemos tener mayor control del hardware disponible, por lo que privilegiaremos lenguajes de más bajo nivel que nos permitan hacer un uso más eficiente de los recursos. En cambio, para aplicaciones de escritorio como sistemas de gestión de productos, calendarios, correo electrónico, entre otras privilegiaremos la elección de lenguajes de más alto nivel que nos permitan ser más eficientes en cuanto a la codificación ya que, en términos generales, es necesario escribir menos líneas de código en los lenguajes de alto nivel, que para su equivalente en bajo nivel.

El lenguaje máquina

Los lenguajes máquina son aquellos que están escritos en lenguajes cuyas instrucciones son cadenas binarias (cadenas o series de caracteres -dígitos- O y 1) que especifican una operación, y las posiciones (dirección) de memoria implicadas en la operación se denominan instrucciones de máquina o código máquina. El código máquina es el conocido código binario.

En los primeros tiempos del desarrollo de los ordenadores era necesario programarlos directamente de esta forma, sin embargo, eran máquinas extraordinariamente limitadas, con muy pocas instrucciones por lo que aún era posible; en la actualidad esto es completamente irrealizable por lo que es necesario utilizar lenguajes más fácilmente comprensibles para los humanos que deben ser traducidos a código máquina para su ejecución.

[1] **Control de velocidad crucero**: El control de velocidad, también conocido como regulador de velocidad o autocrucero, es un sistema que controla de forma automática el factor de movimiento de un vehículo de motor. El conductor configura la velocidad y el sistema controlará la válvula de aceleración del vehículo para mantener la velocidad de forma continua.

Ejemplo de una instrucción:
1110 0010 0010 0001 0000 0000 0010 0000

El lenguaje de bajo nivel

Los lenguajes de bajo nivel son más fáciles de utilizar que los lenguajes máquina, pero, al igual, que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el ensamblador o assembler. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como mnemotécnicas (mnemonics). Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés, ADD, SUB, DIV, etc.; en español, SUM, para sumar, RES, para restar, DIV, para dividir etc.

Lenguajes de alto nivel

Los lenguajes de alto nivel son los más utilizados por los programadores. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensambladores. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la máquina; esto es, las instrucciones del programa de la computadora no dependen del diseño del hardware o de una computadora en particular. En consecuencia, los programas escritos en lenguaje de alto nivel son portables o transportables, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras; al contrario que los programas en lenguaje máquina o ensamblador, que sólo se pueden ejecutar en un determinado tipo de computadora. Esto es posible porque los lenguajes de alto nivel son traducidos a lenguaje máquina por un tipo de programa especial denominado “compilador”. Un compilador toma como entrada un algoritmo escrito en un lenguaje de alto nivel y lo convierte a instrucciones inteligibles por el ordenador; los compiladores deben estar adaptados a cada tipo de ordenador pues deben generar código máquina específico para el mismo.

Ejemplos de Lenguajes de Alto Nivel:

C, C++, Java, Python, VisualBasic, C#, JavaScript

¿Qué es un Programa?

Es un algoritmo escrito en algún lenguaje de programación de computadoras.

Pasos para la construcción de un programa

DEFINICIÓN DEL PROBLEMA

En este paso se determina la información inicial para la elaboración del programa. Es donde se determina qué es lo que debe resolverse con el computador, el cual requiere una definición clara y precisa.

Es importante que se conozca lo que se desea que realice la computadora; mientras la definición del problema no se conozca del todo, no tiene mucho caso continuar con la siguiente etapa.

ANÁLISIS DEL PROBLEMA

Una vez que se ha comprendido lo que se desea de la computadora, es necesario definir:

- Los datos de entrada.
- Los datos de salida
- Los métodos y fórmulas que se necesitan para procesar los datos.

Una recomendación muy práctica es la de colocarse en el lugar de la computadora y analizar qué es lo que se necesita que se ordene y en qué secuencia para producir los resultados esperados.

DISEÑO DEL ALGORITMO

Se puede utilizar algunas de las herramientas de representación de algoritmos mencionadas anteriormente. Este proceso consiste en definir la secuencia de pasos que se deben llevar a cabo para conseguir la salida identificada en el paso anterior.

CODIFICACIÓN

La codificación es la operación de escribir la solución del problema (de acuerdo a la lógica del diagrama de flujo o pseudocódigo), en una serie de instrucciones detalladas, en un código reconocible por la computadora. La serie de instrucciones detalladas se conoce como código fuente, el cual se escribe en un lenguaje de programación o lenguaje de alto nivel.

PRUEBA Y DEPURACIÓN

Se denomina prueba de escritorio a la comprobación que se hace de un algoritmo para saber si está bien realizado. Esta prueba consiste en tomar datos específicos como entrada y seguir la secuencia indicada en el algoritmo hasta obtener un resultado, el análisis de estos resultados indicará si el algoritmo está correcto o si por el contrario hay necesidad de corregirlo o hacerle ajustes.

Elementos de un Programa

Variables y Constantes

A la hora de elaborar un programa es necesario usar datos; en el caso del ejemplo del cálculo del área del rectángulo, para poder obtener el área del mismo, necesitamos almacenar en la memoria de la computadora el valor de la base y de la altura, para luego poder multiplicar sus valores.

Recordemos que no es lo mismo grabar los datos en memoria que grabarlos en el disco duro. Cuando decimos grabar en memoria nos estaremos refiriendo a grabar esos datos en la RAM. Ahora bien, para grabar esos

datos en la RAM podemos hacerlo utilizando dos elementos, llamados: variables y constantes. Los dos elementos funcionan como fuentes de almacenamiento de datos, la gran diferencia entre los dos es que en el caso de las constantes su valor dado no varía en el transcurso de todo el programa.

Podría decirse que tanto las variables como las constantes, son direcciones de memoria con un valor, ya sea un número, una letra, o valor nulo (cuando no tiene valor alguno, se denomina valor nulo). Estos elementos permiten almacenar temporalmente datos en la computadora para luego poder realizar cálculos y operaciones con los mismos. Al almacenarlos en memoria, podemos nombrarlos en cualquier parte de nuestro programa y obtener el valor del dato almacenado, se dice que la variable nos devuelve el valor almacenado.

A continuación, se muestra un esquema, que representa la memoria RAM, como un conjunto de filas, donde, en este caso, cada fila representa un byte y tiene una dirección asociada.

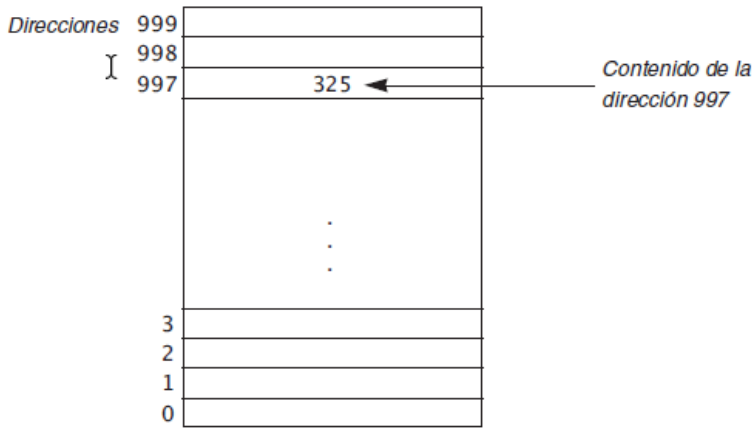


Figura 14: Representación de la memoria RAM

Variables

Son elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato, que puede variar en el desarrollo del programa. Una variable es un grupo de bytes asociado a un nombre o identificador, y a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a esa variable.

En una variable se puede almacenar distintos tipos de datos. De acuerdo al tipo de dato, definido para cada lenguaje de programación, será la cantidad de bytes que ocupa dicha variable en la memoria.

Type	Size	Range
byte	1 byte	−128 to 127
short	2 bytes	−32,768 to 32,767
int	4 bytes	−2,147,483,648 to 2,147,483,647
long	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	$-3.40282347 \times 10^{38}$ to 3.4028347×10^{38}
double	8 bytes	$-1.79769313486231570 \times 10^{308}$ to $1.79769313486231570 \times 10^{308}$
char	2 bytes	one character
String	2 or more bytes	one or more characters

Tabla 1: Tabla de equivalencias de las unidades de memoria

Lo más importante de la definición de las variables y la elección del tipo de datos asociados es el significado de la variable, o su semántica: ya que en base al tipo de datos seleccionado serán las operaciones que podamos realizar con esa variable, por ejemplo: si tenemos la variable edad deberíamos seleccionar un tipo de datos como integer (número entero) ya que las operaciones relacionadas serán de comparación, sumas o restas y no es necesario tener una profundidad de decimales.

A continuación, se listan algunos de los tipos de datos más comunes y sus posibles usos:

Tipo de Datos	Significado	Ejemplos de uso
Byte	Número entero de 8 bits. Con signo	Temperatura de una habitación en grados Celsius
Short	Número entero de 16 bits. Con signo	Edad de una Persona
Int	Número entero de 32 bits. Con signo	Distancia entre localidades medida en metros
Long	Número entero de 64 bits. Con signo	Producto entre dos distancias almacenadas en variables tipo int como la anterior
Float	Número Real de 32 bits.	Altura de algún objeto
Double	Número Real de 64 bits.	Proporción entre dos magnitudes
Boolean	Valor lógico: true (verdadero) o false (falso)	Almacenar si el usuario ha aprobado un examen o no

Tabla 2: Tipos de dato

Para algunos lenguajes de programación es posible especificar si las variables numéricas tendrán o no signo (es decir que puedan tomar valores negativos y positivos, o sólo positivos). La elección de tener o no signo (definidas con la palabra reservada unsigned) depende del significado de la variable ya que al no tenerlo

podremos almacenar el doble de valores, por ejemplo, una variable short sin signo posee valores desde el 0 al 65535.

Constantes

Elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato pero que no varía durante la ejecución del programa. Se podría pensar en un ejemplo de necesitar utilizar en el programa el número pi, como el mismo no varía, se puede definir una constante pi y asignarle el valor 3.14.

Operadores

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Los operadores son símbolos especiales que sirven para ejecutar una determinada operación, devolviendo el resultado de la misma.

Para comprender lo que es un operador, debemos primero introducir el concepto de Expresión. Una expresión es, normalmente, una ecuación matemática, tal como 3 + 5. En esta expresión, el símbolo más (+) es el operador de suma, y los números 3 y 5 se llaman operandos. En síntesis, una expresión es una secuencia de operaciones y operandos que especifica un cálculo.

Existen diferentes tipos de operadores:

Operador de asignación

Es el operador más simple que existe, se utiliza para asignar un valor a una variable o a una constante. El signo que representa la asignación es el = y este operador indica que el valor a la derecha del = será asignado a lo que está a la izquierda del mismo.

Ejemplo en pseudocódigo:

Entero edad = 20

Decimal precio = 25.45

Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (+), resta (-), multiplicación (*), división (/) y resto de la división entera (%), por ejemplo 50%8=2 porque necesitamos obtener un número entero que queda luego de determinar la cantidad de veces que 8 entra en 50.

Ejemplo:

Expresión	Operador	Operandos	Resultado arrojado
5 * 7	*	5 , 7	35
6 + 3	+	6 , 3	9
20 - 4	-	20 , 4	16
50 % 8	%	50 , 8	2
45/5	/	45,5	9

Tabla Resumen Operadores Aritméticos:

Operador	Significado
+	Suma
-	Resta
*	Producto
/	División
%	Resto de la división entera

Los operadores unitarios

Los operadores unitarios requieren sólo un operando; que llevan a cabo diversas operaciones, tales como incrementar/decrementar un valor de a uno, negar una expresión, o invertir el valor de un booleano.

Operador	**Descripción**	**Ejemplo**	**Resultado**
++	operador de incremento; incrementa un valor de a 1	int suma=20; suma++;	suma=21
--	operador de decremento; Reduce un valor de a 1	int resta=20; resta--;	resta=19
!	operador de complemento lógico; invierte el valor de un valor booleano	boolean a=true; boolean b= !a;	b=false

Operadores Condicionales

Son aquellos operadores que sirven para comparar valores. Siempre devuelven valores booleanos: TRUE O FALSE. Pueden ser Relacionales o Lógicos.

Operadores relacionales

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor.

Los operadores relacionales determinan si un operando es mayor que, menor que, igual a, o no igual a otro operando. La mayoría de estos operadores probablemente le resultará familiar. Tenga en cuenta que debe utilizar " == ", no " = ", al probar si dos valores primitivos son iguales.

Operador	Significado
==	Igual a
!=	No igual a
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que

Expresión	Operador	Resultado
a > b	>	true: si a es mayor que b false: si a es menor que b
a >= b	>=	true:si a es mayor o igual que b false: si a es menor que b
a < b	<	true: si a es menor que b false: si a es mayor que b
a <= b	<=	true: si a es menor o igual que b false: si a es mayor que b.

Expresión	Operador	Resultado
a == b	==	true: si a y b son iguales. false: si a y b son diferentes.
a != b	!=	true: si a y b son diferentes false: si a y b son iguales.

Operadores lógicos

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. Se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales.

Expresión	Nombre Operador	Operador	Resultado
a && b	AND	&&	true: si a y b son verdaderos. false: si a es falso, o si b es falso, o si a y b son falsos
a || b	OR	||	true: si a es verdadero, o si b es verdadero, o si a y b son verdaderos. false: si a y b son falsos.

Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser true y el primero es false ya se sabe que la condición de que ambos sean true no se va a cumplir).

Rutinas

Las rutinas son uno de los recursos más valiosos cuando se trabaja en programación ya que permiten que los programas sean más simples, debido a que el programa principal se compone de diferentes rutinas donde cada una de ellas realiza una tarea determinada.

Una rutina se define como un bloque, formado por un conjunto de instrucciones que realizan una tarea específica y a la cual se la puede llamar desde cualquier parte del programa principal. Además, una rutina puede opcionalmente tener un **valor de retorno** y **parámetros**. El valor de retorno puede entenderse como el resultado de las instrucciones llevadas a cabo por la rutina, por ejemplo si para una rutina llamada *sumar(a, b)* podríamos esperar que su valor de retorno sea la suma de los números a y b. En el caso anterior, a y b son los datos de entrada de la rutina necesarios para realizar los cálculos correspondientes. A estos datos de

entrada los denominamos **parámetros** y a las rutinas que reciben parámetros las denominamos **funciones, procedimientos o métodos**, dependiendo del lenguaje de programación.

Ejemplos de rutinas:

SumarPrecioProductos(precioProducto1, precioProducto2)

Rutina que realiza la suma de los precios de los productos comprados por un cliente y devuelve el monto total conseguido.

AplicarDescuento(montoTotal)

Rutina que a partir de un monto total aplica un descuento de 10% y devuelve el monto total con el descuento aplicado.

Entonces nuestro programa puede hacer uso de dichas rutinas cuando lo necesite. Por ejemplo cuando un cliente realice una compra determinada, podemos llamar a la rutina *sumarPrecioProductos* y cobrarle el monto devuelto por la misma. En el caso que el cliente abonara con un cupón de descuento, podemos entonces llamar a la rutina *aplicarDescuento* y así obtener el nuevo monto con el 10% aplicado.

Desarrollo de programas

Estructuras de programación

Un programa puede ser escrito utilizando tres tipos de estructuras de control:

a) secuenciales
b) selectivas o de decisión
c) repetitivas

Las Estructuras de Control determinan el orden en que deben ejecutarse las instrucciones de un algoritmo: si serán recorridas una luego de la otra, si habrá que tomar decisiones sobre si ejecutar o no alguna acción o si habrá repeticiones.

Estructura secuencial

Es la estructura en donde una acción (instrucción) sigue a otra de manera secuencial. Las tareas se dan de tal forma que la salida de una es la entrada de la que sigue y así en lo sucesivo hasta cumplir con todo el proceso. Esta estructura de control es la más simple, permite que las instrucciones que la constituyen se ejecuten una tras otra en el orden en que se listan. Por ejemplo, considérese el siguiente fragmento de un algoritmo:

En este fragmento se indica que se ejecute la operación 1 y a continuación la operación 2.

Figura 15: Diagrama de Flujo Secuencial

Estructura alternativa

Estas estructuras de control son de gran utilidad para cuando el algoritmo a desarrollar requiera una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existe un número de posibles alternativas que resultan de la evaluación de una determinada condición.

Este tipo de estructuras son utilizadas para tomar decisiones lógicas, es por esto que también se denominan estructuras de decisión o selectivas.

En estas estructuras, se realiza una evaluación de una condición y de acuerdo al resultado, el algoritmo realiza una determinada acción. Las condiciones son especificadas utilizando expresiones lógicas.

Las estructuras selectivas/alternativas pueden ser:

- Simples
- Dobles
- Múltiples

Alternativa simple (si-entonces/if-then)

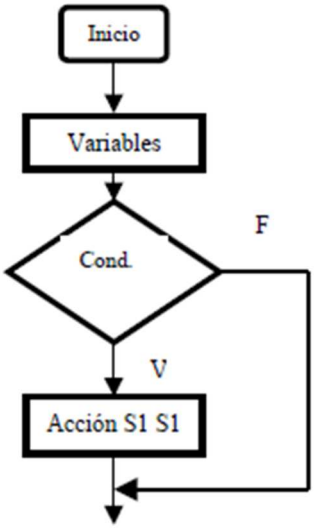


Figura 16: Diagrama de Flujo de alternativa simple

La estructura alternativa simple si-entonces (en inglés if-then) lleva a cabo una acción al cumplirse una determinada condición. La selección si-entonces evalúa la condición y:

- Si la condición es verdadera, ejecuta la acción S1
- Si la condición es falsa, no ejecuta nada.

En español:	En Inglés:
Si <condición> Entonces <acción S1> Fin_si	If <condición> Then <acción S1> End_if

Ejemplo:

```
BEGIN
  ENTERO edad = 18
  IF (edad > 18)
  THEN:
    puede manejar un auto
  END_IF
END
```

Alternativa Doble (si-entonces-sino/if-then-else)

Existen limitaciones en la estructura anterior, y se necesitará normalmente una estructura que permita elegir dos opciones o alternativas posibles, de acuerdo al cumplimiento o no de una determinada condición:

- Si la condición es verdadera, se ejecuta la acción S1
- Si la condición es falsa, se ejecuta la acción S2

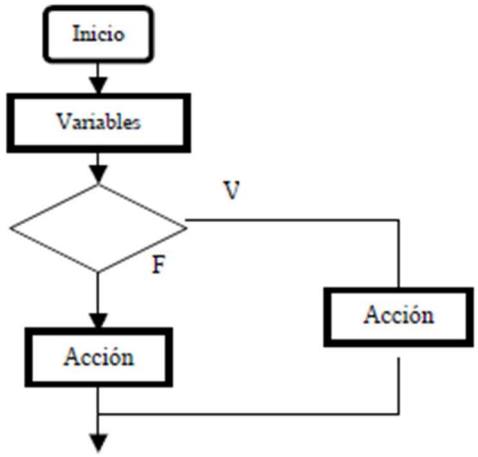


Figura 17: Diagrama de Flujo de alternativa doble

En español:

Si <condición>
entonces <acción S1>
sino <acción S2>
Fin_Si

En inglés:

If <condición>
then<acción>
else<acción S2>
End_if

Ejemplo:

```
BEGIN
  BOOLEAN afueraLlueve = verdadero
IF  (afueraLlueve es verdadero)
  THEN:
    me quedo viendo películas
  ELSE:
    salgo al parque a tomar mates
  END_IF
END
```

Alternativa de Decisión múltiple (según_sea, caso de/case)

Se utiliza cuando existen más de dos alternativas para elegir. Esto podría solucionarse por medio de estructuras alternativas simples o dobles, anidadas o en cascada. Sin embargo, se pueden plantear serios problemas de escritura del algoritmo, de comprensión y de legibilidad, si el número de alternativas es grande.

En esta estructura, se evalúa una condición o expresión que puede tomar n valores. Según el valor que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor.

PSEUDOCÓDIGO:

Según sea <expresión>

<Valor1>: <acción1>

<valor2>: <acción2>

.....

[<otro>: <acciones>]

fin según

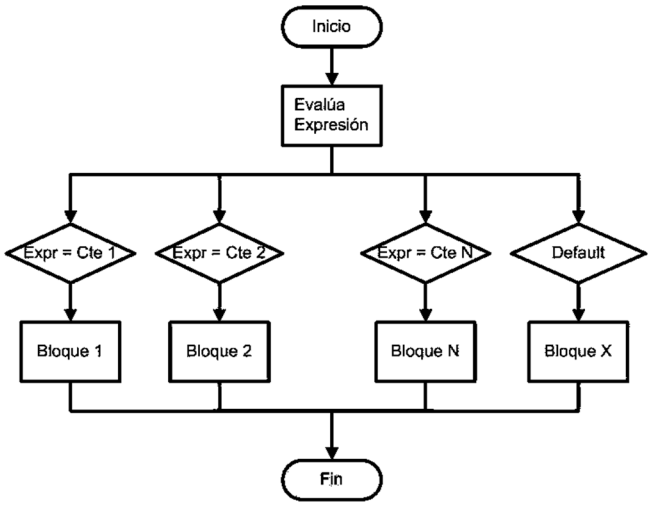


Figura 18: Diagrama de Flujo de Decisión Múltiple

Ejemplo en pseudocódigo:

```
INICIO
ENTERO posicionDeLlegada = 3
SEGUN SEA posicionDeLlegada
  1: entregar medalla de oro
  2: entregar medalla de plata
  3: entregar medalla de bronce
  otro: entregar mención especial
FIN
```

Es importante mencionar que la estructura anterior puede ser escrita usando los condicionales vistos anteriormente de la siguiente forma:

```
BEGIN
ENTERO posicionDeLlegada = 3
IF (posicionDeLlegada = 1)
 THEN:
  entregar medalla de oro
 ELSE:
  IF (posicionDeLlegada = 2)
  THEN:
   entregar medalla de plata
  ELSE:
   IF (posicionDeLlegada = 3)
   THEN:
    entregar medalla de bronce
   ELSE:
    entregar mención especial
   END_IF
  END_IF
 END_IF
END
```

Podemos ver que usar condiciones anidadas podemos resolver el mismo problema, pero la estructura resultante es mucho más compleja y difícil de modificar.

Estructura repetitiva o iterativa

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan BUCLES. Y cada repetición del bucle se llama iteración.

Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

Un bucle se denomina también lazo o loop. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalización del bucle no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores principiantes.

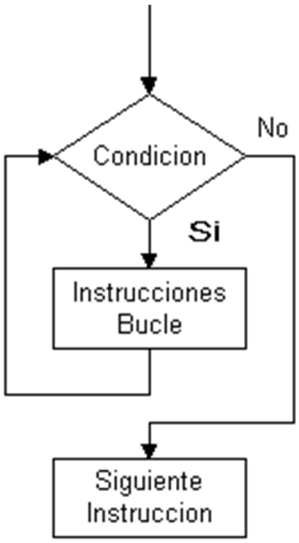
Hay distintos tipos de bucles:

- Mientras, en inglés: While
- Hacer Mientras, en inglés: Do While.
- Para, en inglés: For

Estructura mientras (while, en inglés)

Esta estructura repetitiva “mientras”, es en la que el cuerpo del bucle se repite siempre que se cumpla una determinada condición.

Diagrama de Flujo	Pseudocódigo:

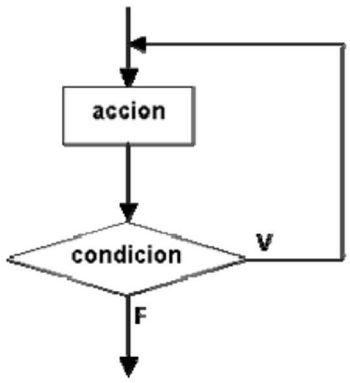
 *Figura 19: Diagrama de Flujo de la estructura “mientras”*	Español:
	Mientras <cond> hacer
	<acciones>
	fin mientras
	
	Inglés:
	While <cond> do
	<actions>
	End while

Ejemplo:

```
Begin
  BOOLEAN (tanqueLleno = falso
  WHILE (tanqueLleno == falso)
  DO:
    llenar tanque
  END_WHILE
  // el tanque ya está lleno :)
END
```

Estructura hacer-mientras (do while, en inglés)

Esta estructura es muy similar a la anterior, sólo que a diferencia del while el contenido del bucle se ejecuta siempre al menos una vez, ya que la evaluación de la condición se encuentra al final. De esta forma garantizamos que las acciones dentro de este bucle sean llevadas a cabo, aunque sea una vez independientemente del valor de la condición.

Diagrama de Flujo	Pseudocódigo:
 *Figura 20: Diagrama de flujo de "hacer-mientras"*	Español: Hacer <acciones> Mientras <cond> Fin mientras Inglés: Do <actions> While <cond> End_While

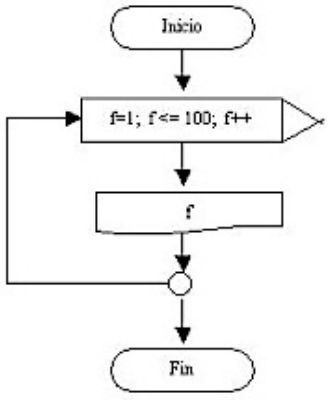
Ejemplo:

```
BEGIN
BOOLEAN llegadaColectivo=false;
  DO: esperar en la parada
  WHILE (llegadaColectivo == false)
  END_WHILE
END
```

Estructura para (for, en inglés)

La estructura for es un poco más compleja que las anteriores y nos permite ejecutar un conjunto de acciones para cada elemento de una lista, o para cada paso de un conjunto de elementos. Su implementación depende del lenguaje de programación, pero en términos generales podemos identificar tres componentes: la inicialización, la condición de corte y el incremento.

Diagrama de Flujo	Pseudocódigo:

 *Figura 21: Diagrama de Flujo de "para"*	Español:
	Para (inicialización; condición de corte; incremento)
	<acciones>
	Fin para
	
	Inglés:
	For (inicialización; condición de corte; incremento)
	<actions>
	End For

Ejemplo:

```
BEGIN
 FOR (ENTERO RUEDA = 1; RUEDA <= 4; RUEDA++)
  inflar_rueda (RUEDA)
 END_FOR
END
```

La ejecución del pseudocódigo anterior dará como resultado las siguientes llamadas a la función inflar_rueda():

1. inflar_rueda (1)

2. inflar_rueda (2)

3. inflar_rueda (3)

4. inflar_rueda (4)

Luego de esto podríamos suponer que hemos inflado las 4 cubiertas del auto y estamos listos para seguir viaje.

Recursividad

La recursividad es un elemento muy importante en la solución de algunos problemas de computación. Por definición, un algoritmo recursivo es aquel que utiliza una parte de sí mismo como solución al problema. La otra parte generalmente es la solución trivial, es decir, aquella cuya solución será siempre conocida, es muy fácil de calcular, o es parte de la definición del problema a resolver. Dicha solución sirve como referencia y además permite que el algoritmo tenga una cantidad finita de pasos.

La implementación de estos algoritmos se realiza generalmente en conjunto con una estructura de datos, la pila, en la cual se van almacenando los resultados parciales de cada recursión.

Un ejemplo es el cálculo de factorial de un número, se puede definir el factorial de un número entero positivo x como sigue:

x! = x * (x-1) * (x-2)... * 3 * 2 * 1

donde "!" indica la operación unaria de factorial. Por ejemplo, para calcular el factorial de 5, tenemos:

5! = 5 * 4 * 3 * 2 * 1 = 120

Definimos, además:

1! = 1 y 0! = 1

Sin embargo, podemos observar que la definición del factorial de un número x, puede expresarse, a su vez, a través del factorial de otro número:

x! = x * (x-1)!

Es decir, para conocer el factorial de x basta con conocer el factorial de x-1 y multiplicarlo por x. Para conocer el factorial de x-1 basta con conocer el factorial de x-2, y multiplicarlo por x-1. Este proceso se realiza recursivamente, hasta llegar a la solución trivial, donde necesitamos el factorial de 1, el cual es 1.

Lo importante a notar en la igualdad anterior es que expresa un proceso recursivo, donde definimos una operación en términos de sí misma.

El pseudocódigo queda así:

```
Factorial (x)
 IF (x == 1 O x == 0)
 THEN:
 DEVOLVER 1
 ELSE:
  DEVOLVER x * Factorial (x-1)
 END_IF
```

Ventajas

● Algunos problemas son esencialmente recursivos, por lo cual su implementación se facilita mediante un algoritmo de naturaleza recursiva, sin tener que cambiarlo a un método iterativo, por ejemplo.

● En algunas ocasiones el código de un algoritmo recursivo es muy pequeño.

Desventajas

● Puede llegar a utilizar grandes cantidades de memoria en un instante, pues implementa una pila cuyo tamaño crece linealmente con el número de recursiones necesarias en el algoritmo. Si los datos en cada paso son muy grandes, podemos requerir grandes cantidades de memoria lo que a veces puede agotar la memoria de la computadora donde está corriendo el programa.

Estructuras de Datos: Pilas, Colas y Listas

En esta sección veremos estructuras de datos que nos permiten coleccionar elementos. Para poder agregar u obtener elementos de estas estructuras de datos tenemos dos operaciones básicas:

- COLOCAR
- OBTENER.

Dependiendo de cómo se realicen las operaciones sobre los elementos de la colección es que definimos pilas, colas y listas.

Listas

Una lista (en inglés array) es una secuencia de datos. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3, etc. En una lista los datos deben ser del mismo tipo. Es importante destacar que la mayoría de las estructuras de datos en los lenguajes de programación son zero-based, es decir que el primer elemento siempre tendrá asignado el número de orden 0, lo que significa que la cantidad de elementos total

será igual al número del último elemento más 1. El tipo de elementos almacenados en la lista puede ser cualquier tipo de dato. Normalmente la lista se utiliza para almacenar tipos de datos, tales como cadenas de texto, números enteros o decimales.

Una lista puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de asientos que tiene un colectivo de larga distancia. Cada ítem de una lista se denomina elemento. Una lista tiene definida una longitud, que indica la cantidad de elementos que contiene la misma. Por ejemplo, si tenemos una lista que contiene los meses del año, entonces dicha lista tiene en total 12 elementos, donde el primer elemento “Enero” tiene el número de orden 0 y “Diciembre” el número 11.

Los elementos de una lista se enumeran consecutivamente 0, 1, 2, 3, 4, 5, etc. Estos números se denominan valores índices o subíndice de la lista.

Los índices o subíndices de una lista son números que sirven para identificar unívocamente la posición de cada elemento dentro de la lista. Entonces, si uno quiere acceder a un elemento determinado de la lista, conociendo su posición, es decir su índice, se puede obtener el elemento deseado fácilmente.

Podemos representar una lista de la siguiente forma:

Si consideramos una lista de longitud 6:

elemento 1	elemento 2	elemento 3	elemento 4	elemento 5	elemento 6
índice:0	índice:1	índice:2	índice:3	índice:4	índice:5

Ejemplo, volviendo al ejemplo planteado en el párrafo superior, se muestra a continuación una lista que contiene todos los meses del año:

Nombre de la Lista: **mesesDelAño**

Longitud de la Lista: **12**

Tipo de Datos: **String**

Enero	Febrero	Marzo	Abril	Mayo	Junio	Julio	Agosto	Septiembre	Octubre	Noviembre	Diciembre
0	1	2	3	4	5	6	7	8	9	10	11

Si quisiéramos acceder a un elemento determinado de la lista, simplemente debemos conocer cuál es la posición del elemento deseado.

Ejemplo:

```
BEGIN
  LISTA mesesDelAño

  OBTENER(mesesDelAño, 11) // esto nos devuelve “diciembre”
  OBTENER(mesesDelAño, 0)  // esto nos devuelve “enero”
  OBTENER(mesesDelAño, 7)  // esto nos devuelve “agosto”
  OBTENER(mesesDelAño, 12) // error: no existe el elemento 12
END
```

Si quisiéramos asignar un valor a una posición determinada de la lista, necesitamos conocer por un lado la posición que queremos asignar, y el elemento que vamos a asignar a dicha posición:

```
BEGIN
  LISTA mesesDelAño

  COLOCAR(mesesDelAño, 10, “noviembre”) // “noviembre”, en la posición 10
  COLOCAR(mesesDelAño, 0, “enero”) // esto asigna “enero”, en la posición 0
  COLOCAR(mesesDelAño, 3, “abril”) // esto asigna “abril”, en la posición 3
END
```

Pilas

Una pila puede imaginarse como un conjunto de platos colocados uno sobre otro, o como un tubo de papas fritas, o en donde al colocar elementos dentro de él, sólo podemos sacar los últimos colocados. Este tipo de estructuras de datos se denominan LIFO (Last In First Out, del inglés Primero en entrar último en salir). De esta forma los últimos elementos en ser extraídos serán los que estén ubicados en la parte superior de la estructura.

Las operaciones básicas de las pilas son dos:

- APILAR (PUSH, colocar un elemento al principio)
- DESAPILAR (POP, obtener el último elemento colocado)

Un ejemplo de pseudocódigo del uso de una pila sería:

```
BEGIN
  PILA sillaDelDormitorio

  PUSH (sillaDelDormitorio, “buzo”)
  PUSH (sillaDelDormitorio, “jeans”)
  PUSH (sillaDelDormitorio, “remera”)

  POP (sillaDelDormitorio) // esto nos devuelve “remera”
  POP (sillaDelDormitorio) // esto nos devuelve “jeans”
  POP (sillaDelDormitorio) // esto nos devuelve “buzo”
  POP (sillaDelDormitorio) // error: la pila está vacía
END
```

Colas

Esta estructura de datos se caracterizada por ser una secuencia de elementos en la que la operación de inserción:

- ENCOLAR (PUSH, agregar) se realiza por un extremo y,
- DESENCOLAR (POP, extraer) por el otro.

También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

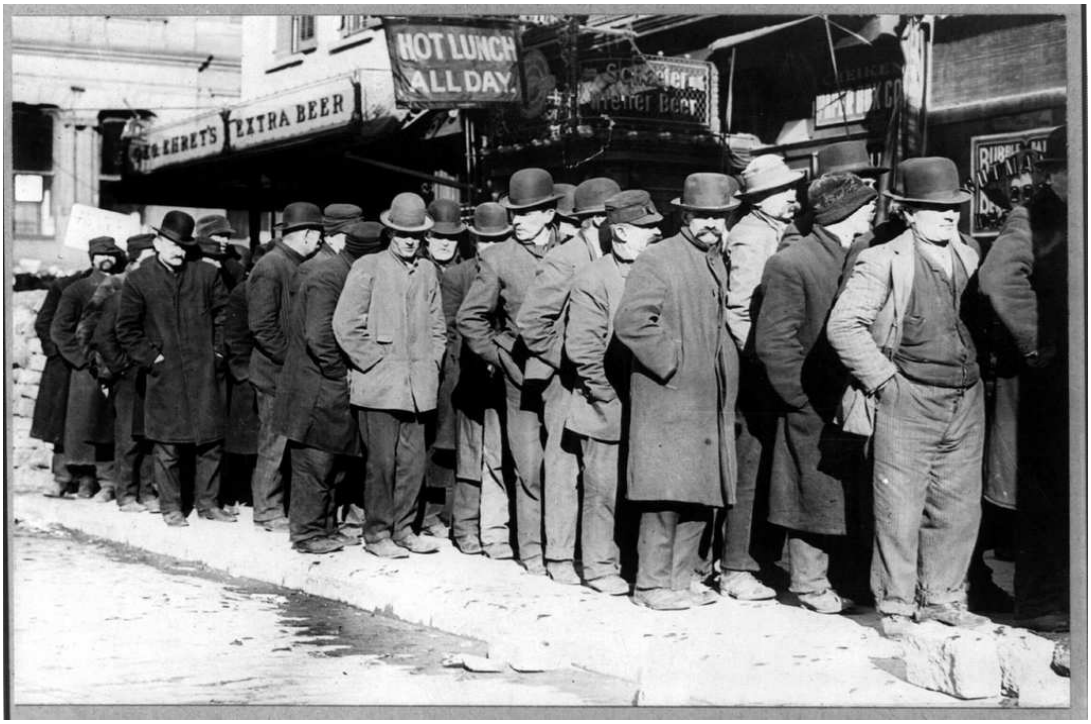


Figura 22: Ejemplo de una cola de espera

Esto representa la idea que tenemos de cola en la vida real. La cola para subir al colectivo está compuesta de elementos (personas), que dispone de dos extremos comienzo y fin. Por el comienzo se extraerá un elemento cuando haya comprado el pasaje para su viaje, y si llega una nueva persona con intención de viajar, tendrá que colocarse al final y esperar que todos los elementos situados antes que él abandonen la cola.

Existen otros tipos de colas más sofisticados como colas con prioridad en las que algunos elementos podrán abandonar la cola antes que otros independientemente de que hayan sido agregados luego, usando como factor de decisión cierto valor de prioridad. Un ejemplo puede darse en las colas para pagar los impuestos, en las que personas mayores pueden ser atendidos antes (abandonar la cola antes) debido a su condición prioritaria.

Un ejemplo de pseudocódigo del uso de una cola sin prioridad sería:

```
BEGIN
  COLA cajaDelSupermercado

  PUSH (cajaDelSupermercado, “Judith”)
  PUSH (cajaDelSupermercado, “Candelaria”)
  PUSH (cajaDelSupermercado, “Joaquin”)

  POP (cajaDelSupermercado) // esto nos devuelve “Judith”
  POP (cajaDelSupermercado) // esto nos devuelve “Candelaria”
  POP (cajaDelSupermercado) // esto nos devuelve “Joaquin”
  POP (cajaDelSupermercado) // error: la cola está vacía
END
```

Árboles

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras **no lineales** a diferencia de las listas, colas y pilas vistas anteriormente, que constituyen estructuras lineales.

Los árboles son muy utilizados en informática como un método eficiente para búsquedas grandes y complejas, listas dinámicas y aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de estas aplicaciones, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general.

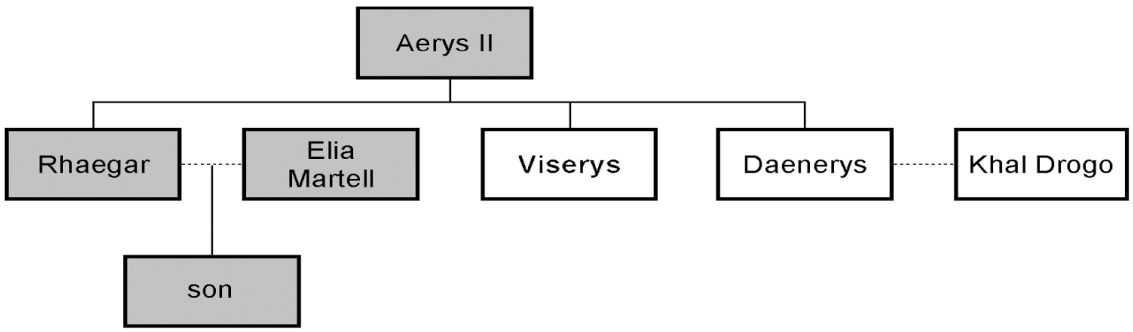


Figura 23: Un árbol genealógico como ejemplo de la estructura de datos

Un árbol consta de un conjunto finito de elementos, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo. Si un árbol no está vacío, entonces el primer nodo se llama **raíz**.

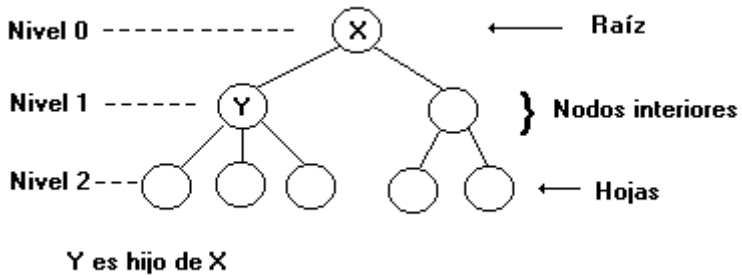


Figura 24: Componentes de un árbol

Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores, los cuales se llaman **hijos**. Dos o más nodos con el mismo padre se llaman **hermanos**. Los nodos sin hijos se denominan nodos **hoja**.

El **nivel** de un nodo es su distancia al raíz. El raíz tiene una distancia cero de sí misma, por lo que se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y hermanos. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos.

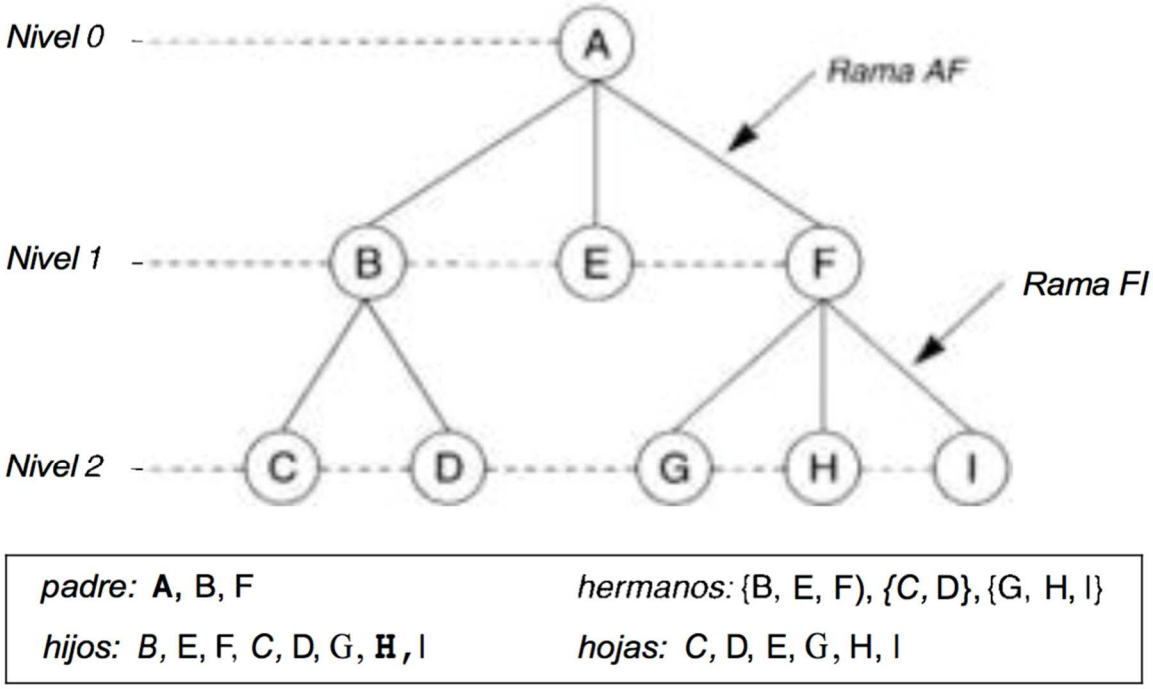


Figura 25: Niveles de profundidad de los árboles

Un árbol se divide en **subárboles**. Un subárbol es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes de ese nodo.

El primer nodo de un subárbol se conoce como el **raíz del subárbol** y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles.

Árboles binarios

Un **árbol binario** es un tipo particular de árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce al nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho.

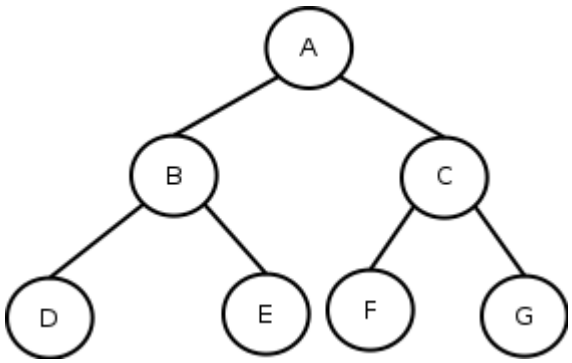


Figura 26: Un árbol binario como tipo particular de árbol

Algoritmos Fundamentales

Algoritmos de Ordenación

Los algoritmos de ordenación sirven para dar un orden determinado a los elementos de una lista. Este procedimiento de ordenación, mediante el cual se disponen los elementos del array en un orden especificado, tal como orden alfabético u orden numérico, es una tarea muy usual en la mayoría de los programas.

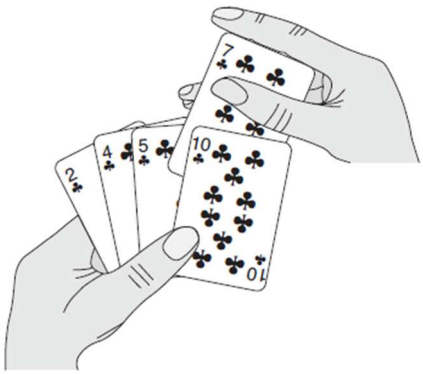
Un diccionario es un ejemplo de una lista ordenada alfabéticamente, y una agenda telefónica o lista de cuentas de un banco es un ejemplo de una lista ordenada numéricamente.

El orden de clasificación u ordenación puede ser ascendente (de menor a mayor) o descendente (de mayor a menor), por lo tanto, debe existir una función o característica de los elementos que determine su precedencia. Los ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Existen numerosos algoritmos de ordenación de listas: inserción, burbuja, selección, rápido (quick sort), fusión (merge), montículo (heap), shell, etc. Las diferencias entre estos algoritmos se basan en su eficiencia y en su **orden de complejidad** (es una medida de la dificultad computacional de resolver un problema). A continuación, describiremos algunos de ellos.

Ordenamiento por inserción

Este algoritmo es el más sencillo de comprender ya que es una representación natural de cómo aplicaríamos el orden a un conjunto de elementos. Supongamos que tenemos un mazo de cartas desordenadas, este algoritmo propone ir tomando las cartas de a una y luego ir colocándolas en la posición correcta con respecto a las anteriores ya ordenadas.



En términos generales, inicialmente se tiene un solo elemento, que por defecto es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

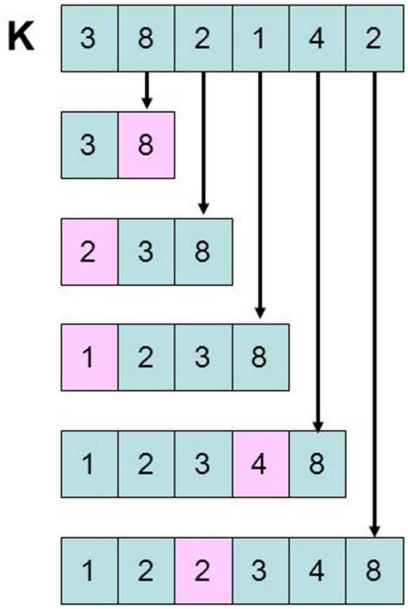


Figura 27: Ejemplo gráfico del algoritmo de ordenamiento por inserción

El pseudocódigo para este algoritmo es el siguiente:

```
BEGIN insercion (A: lista de elementos)
   FOR (ENTERO i = 1; i < longitud(A); i++) :
      ENTERO valor = A[i]
      ENTERO j = i-1

      WHILE (j >= 0 && A[j] > valor)
      DO:
         A[j+1] = A[j]
         j--
      END_WHILE

      A[j+1] = valor
   END_FOR
END
```

Algoritmo de la burbuja

La ordenación por burbuja es uno de los métodos más fáciles de ordenación, ya que el algoritmo de ordenación utilizado es muy simple.

Este algoritmo consiste en comparar cada elemento de la lista con el siguiente (por parejas), si no están en el orden correcto, se intercambian entre sí sus valores. El valor más pequeño flota hasta el principio de la lista como si fuera una burbuja en un vaso de gaseosa.

A continuación, se muestra un ejemplo gráfico de este algoritmo, considerando la siguiente lista inicial:

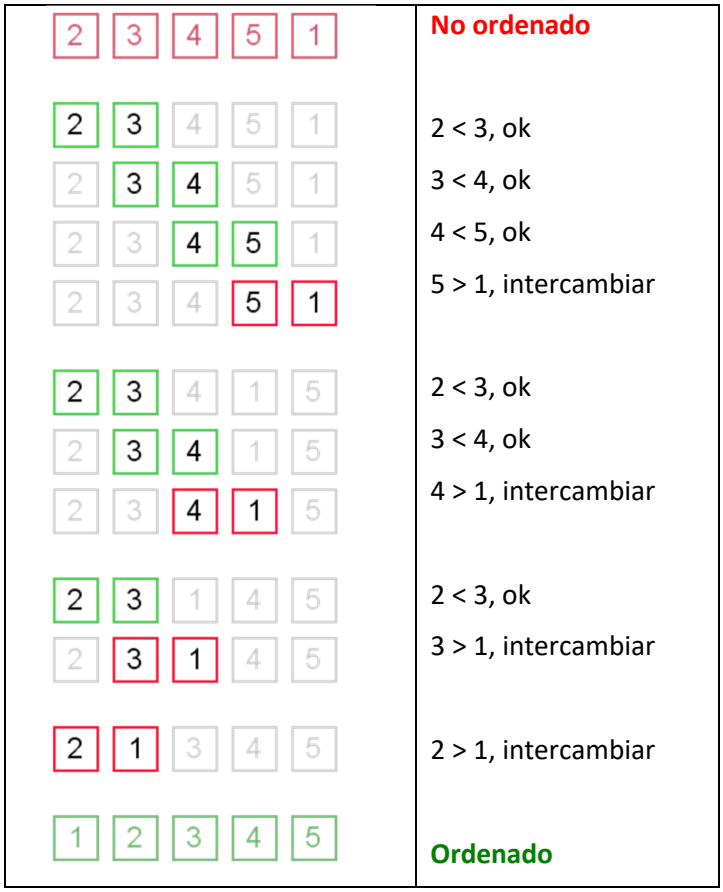


Figura 28: Pasos para ordenar una lista con el método de la burbuja

El pseudocódigo para este algoritmo es el siguiente:

```
BEGIN burbuja (A: lista de elementos)
  n = longitud(A)
  DO:
    intercambiado = falso

    FOR (ENTERO i = 1; i < n; i++)
      // si este par no está ordenado
      IF (A[i-1] > A[i]) THEN:
        // los intercambiamos y recordamos que algo ha cambiado
        ENTERO aux = A[i-1]
        A[i-1] = A[i]
        A[i] = aux

        intercambiado = verdadero
      END_IF
    END_FOR
  WHILE (intercambiado == verdadero)
END
```

Es importante notar que la recorrida completa de la lista (determinada por la sentencia FOR del pseudocódigo) será ejecutada hasta que intercambiado deje de ser verdadero, es decir que seguiremos recorriendo la lista e intercambiando elementos desordenados hasta que no encontremos ninguno más fuera de orden.

Ordenamiento por selección

El algoritmo de ordenamiento por selección es similar al método de la burbuja y funciona de la siguiente manera: inicialmente se recorre toda la lista buscando el menor de todos los elementos, una vez terminada la recorrida el menor elemento se coloca al inicio de la lista recorrida. En la siguiente iteración se recorre nuevamente la lista pero comenzando en el segundo elemento (ya que al haber insertado el menor encontrado al inicio ya lo consideramos ordenado). El procedimiento continúa hasta que el último elemento recorrido es el menor de su subconjunto.

Una desventaja de este algoritmo con respecto a los anteriores mencionados es que no mejora su rendimiento cuando los datos ya están ordenados o parcialmente ordenados debido a que necesariamente recorre la lista en busca del menor de los datos aun cuando el primero de ellos ya es el menor a encontrar.

El pseudocódigo de este algoritmo es muy similar al de la burbuja:

```
BEGIN selección (A : lista de elementos )
  n = longitud(A)
    FOR (ENTERO i = 0; i < n - 1; i++)
    ENTERO mínimo = i
    FOR (ENTERO j = i+1; j < n; j++)
      // si este par no está ordenado
      IF (A[j] < A[mínimo]) ENTONCES:
        // encontramos un nuevo mínimo
        mínimo = j
      END_IF
    END_FOR
    // intercambiamos el actual con el mínimo encontrado
    ENTERO aux = A[minimo]
    A[minimo] = A[i]
    A[i] = aux
  END_FOR
END
```

Algoritmo quick-sort

Esta es la técnica de ordenamiento más rápida conocida, desarrollada por C. Antony R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). Tiene la propiedad de trabajar mejor para elementos de entrada desordenados completamente que para elementos semiordenados. Esta situación es precisamente la opuesta al ordenamiento de burbuja o al de selección antes mencionados.

Este tipo de algoritmos se basa en la técnica "divide y vencerás", lo que supone que es más rápido y fácil ordenar dos arreglos o listas de datos pequeños, que un arreglo o lista más grande.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el **pivote** ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos **sub-listas**, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada **sub-lista** mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido, algunas alternativas son:

- Tomar un elemento cualquiera como pivote, tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo pivote. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

Algoritmos de Búsqueda

La búsqueda de un elemento dado en una lista es una aplicación muy usual en el desarrollo de programas. Dos algoritmos típicos que realizan esta tarea son la búsqueda **secuencial** o en serie y la búsqueda **binaria** o

dicotómica. La búsqueda secuencial es el método utilizado para listas no ordenadas, mientras que la búsqueda binaria se utiliza en listas que ya están ordenados.

Búsqueda secuencial

Este algoritmo busca el elemento dado, recorriendo secuencialmente la lista desde un elemento al siguiente, comenzando en la primera posición de la lista y se detiene cuando encuentra el elemento buscado o bien se alcanza el final de la lista sin haberlo encontrado.

Por consiguiente, el algoritmo debe comprobar primero el elemento almacenado en la primera posición de la lista, a continuación, el segundo elemento y así sucesivamente, hasta que se encuentra el elemento buscado o se termina el recorrido de la lista. Esta tarea repetitiva se realiza con bucles, en nuestro caso con el bucle Para (en inglés, for).

Consideremos que tenemos una lista de alumnos de un curso de Programación y queremos saber si el alumno 'Pedro Lopez', se encuentra cursando el mismo, entonces debemos recorrer toda la lista y buscar el nombre 'Pedro Lopez', e indicar si se encontró o no el alumno buscado.

```
BEGIN busquedaSecuencial (L: lista de alumnos, a: alumno buscado )
  ENTERO n = longitud(L)
  BOOLEAN seEncontró= falso;
  // recorremos la lista, revisando cada elemento de esta, para ver
  // si es el alumno a.

  FOR (ENTERO i = 1; i < n - 1; i++)
  // comparamos el alumno de la posición actual con el alumno buscado: a
     IF (L[i] == a) THEN:
       // encontramos el alumno buscado
       seEncontró = verdadero;
     END-IF
// si nunca se cumple L[i] == a, entonces la variable que indica si se
// encontró o no el alumno: seEncontró, quedará valiendo falso.
  END-FOR
END
```

Búsqueda Binaria

Este algoritmo se utiliza cuando disponemos de una lista ordenada, lo que nos permite facilitar la búsqueda, ya que podemos ir disminuyendo el espacio de búsqueda a segmentos menores a la lista original y completa.

La idea es no buscar en aquellos segmentos de la lista donde sabemos que el valor seguro que no puede estar, considerando que la lista esta ordenada.

Pensemos en el ejemplo anterior de la lista de alumnos del curso de Programación, si tenemos la lista ordenada alfabéticamente por Apellido, podemos comenzar la búsqueda considerando la lista completa y evaluar un valor central de la misma, es probable que ese valor central no sea el buscado, pero podemos ver si ese valor central es mayor o menor al alumno buscado. Si nuestro alumno buscado se llama: 'Lopez Pedro', y el alumno que corresponde a la posición central de la lista es: 'Martinez Sofia', entonces sabemos que como

la lista esta ordenada alfabéticamente por apellido, 'Lopez Pedro', efectivamente tiene que estar en el segmento de la lista que es la primera mitad de la misma, es decir que hemos reducido el espacio de búsqueda a la mitad, lo cual hace que encontremos el valor más rápido que si lo buscaríamos en toda la lista, recorriendo todos los elementos. Si proseguimos con este procedimiento y continuamos buscando el valor central de cada segmento obtenido, podemos ir reduciendo cada vez más el espacio de búsqueda hasta llegar al elemento buscado, si es que existe en la lista.

Para explicar el algoritmo de búsqueda binaria, consideremos que queremos ver si se encuentra en una lista el número 19, a partir de una lista que contiene 12 números ordenados de menor a mayor, como se muestra a continuación:

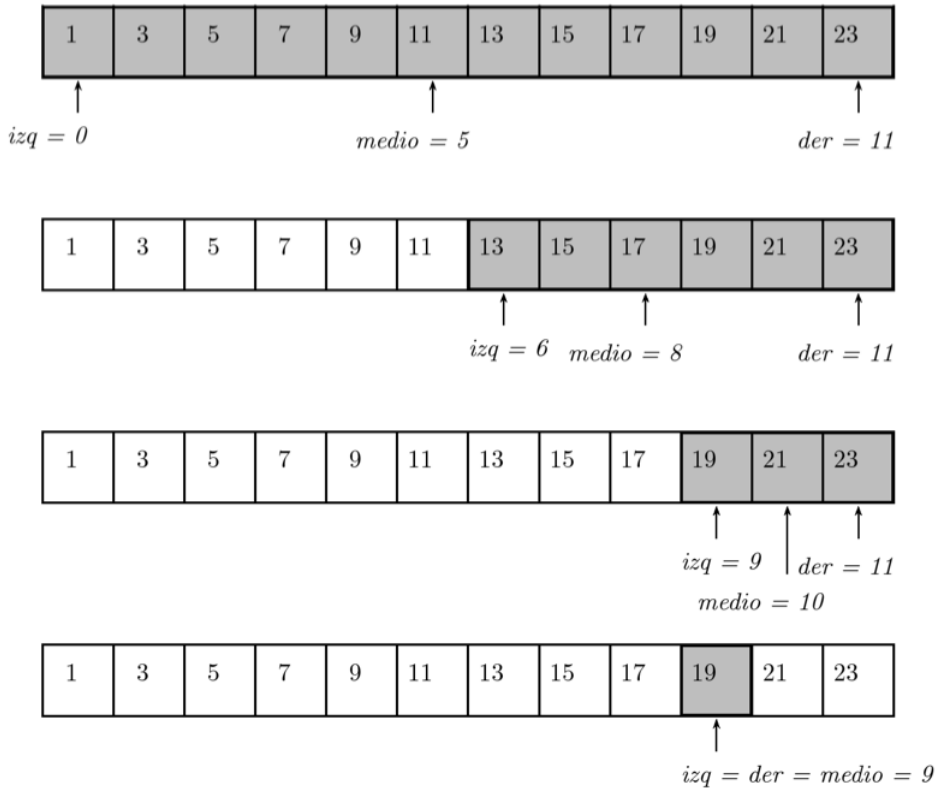


Figura 29: Búsqueda binaria sobre una lista de enteros

Para poder realizar la búsqueda binaria debemos:

- Primero, conocer cuál es el valor del índice izquierdo, derecho y del medio, de la siguiente forma:
  - Índice izq = 0; //sabemos que las listas comienzan con índices enumerados desde 0.
  - Índice der = Longitud de la lista inicial -1; // en este caso la longitud es 12, o sea, que el índice derecho valdrá 11.
  - Índice medio = (izq + der) / 2 // en este caso, (0 + 11)/2, considerando solo la parte entera de la división valdrá 5.
- A partir de la definición de estos índices, el siguiente paso es preguntar si en la posición del medio se encuentra el elemento buscado, es decir si Lista(medio)==19
- Si Lista(medio)==19, devuelve verdadero, entonces la búsqueda finaliza rápidamente.

- Si Lista(medio)==19, devuelve falso, entonces debemos preguntar si el valor de la lista en la posición medio es mayor o menor al valor buscado, para así saber si el segmento que nos interesa es del medio hacia la izquierda o del medio hacia la derecha. En este caso: Lista(medio) es menor a 19. Entonces el segmento que nos interesa de la lista es del medio (sin incluir, porque ya evaluamos y el medio no es igual a 19) hacia la derecha.
- El siguiente paso es volver a realizar el procedimiento descripto, pero considerando sólo el segmento que comienza en el medio hacia la derecha: son los mismos pasos pero para una nueva lista que es un segmento de la lista original. Entonces:
    - Índice izq = 6;
    - Índice der = 11
    - Índice medio = (izq + der) / 2 // en este caso, (6+ 11)/2, considerando solo la parte entera de la división valdrá 8.

Como se ve este algoritmo de búsqueda binaria, a diferencia del algoritmo de búsqueda secuencial, no recorre toda la lista, sino que acorta la lista en segmentos más pequeños sucesivamente, esto es muy ventajoso en el caso de tener listas con gran cantidad de elementos, es decir con millones de valores, en estos casos realizar una búsqueda secuencial lleva mucho tiempo y si la lista ya se encuentra ordenada es mucho más eficiente realizar una búsqueda binaria que secuencial.

Algoritmos de Recorrido

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario de las listas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el raíz viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

Un recorrido de un árbol binario requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, **profundidad** y **anchura**.

En el **recorrido en profundidad**, el proceso exige un camino desde el nodo raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo. Para saber cómo regresarnos, vamos guardando los nodos visitados en una estructura de **pila**. Es por esto que se acostumbra a programar esta búsqueda de forma recursiva, con lo que el manejo de la pila lo realiza el lenguaje de programación utilizado.

Haciendo un recorrido en profundidad recorreríamos los nodos en el siguiente orden:

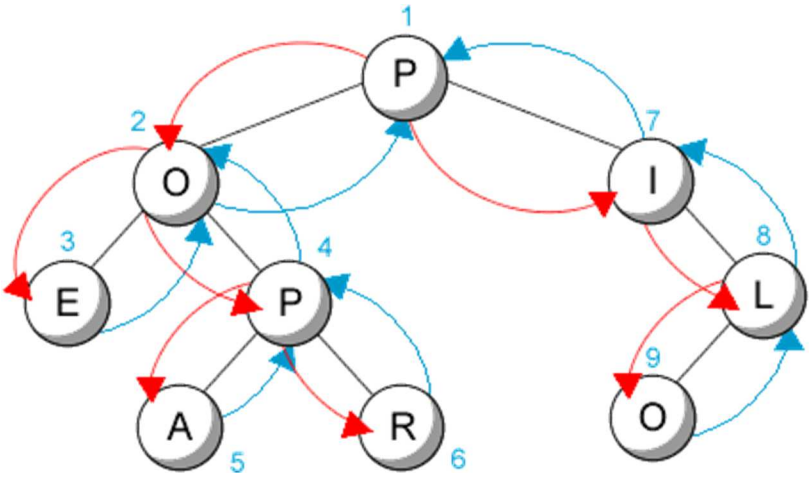


Figura 30: Recorrido en profundidad

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación, a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En otras palabras, en el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel. Para poder saber qué vértices visitar, utilizamos una **cola**.

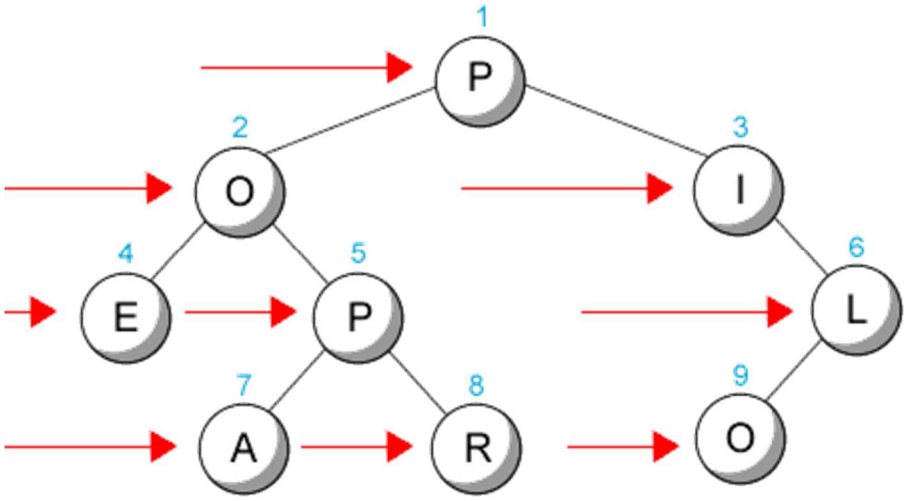


Figura 31: Recorrido en anchura

Fuentes de Información

- **Sommerville, Ian** - "INGENIERÍA DE SOFTWARE" 9na Edición (Editorial Addison-Wesley Año 2011).
- **Pressman Roger** - "Ingeniería de Software" 7ma. Edición - (Editorial Mc Graw Hill Año 2010).
- **Jacobson, Booch y Rumbaugh** - "EL PROCESO UNIFICADO DE DESARROLLO" (Editorial Addison-Wesley - Año 2000 1ª edición).
- **Guerequeta Rosa, Vallecillo Antonio** - TÉCNICAS DE DISEÑO DE ALGORITMOS (Servicio de Publicaciones de la Universidad de Málaga, Año 1998)
- **Hernandez, Pier Paolo Guillen - ALGORITMOS** http://pier.guillen.com.mx/algorithms/09-busqueda/09.1-introduccion.htm
- **Curso de Estructuras de Datos y Algoritmos / Algoritmos recursivos** https://es.wikiversity.org/wiki/Curso_de_Estructuras_de_Datos_y_Algoritmos_/_Algoritmos_recursivos
- **McConnell Steve** - CODE COMPLETE - Segunda edición (Editorial Microsoft Press, Año 2004)
- **Joyanes Aguilar Luis** PROGRAMACIÓN EN C++: ALGORITMOS, ESTRUCTURAS DE DATOS Y OBJETOS - Segunda edición (Editorial McGraw-Hill, Año 2006)
- **Joyanes Aguilar Luis, Rodriguez Baena Luis, Fernandez Azuela Matilde** - Fundamentos de Programación
- **Frittelli Valerio** - Algoritmos y Estructuras de Datos - Segunda edición (Editorial Científica Universitaria, Año 2004)
- **Streib James T., Soma Takako** - GUIDE TO JAVA - A CONCISE INTRODUCTION TO PROGRAMMING (Editorial Springer, Año 2014)
- **Guttag John V.** - INTRODUCTION TO COMPUTATION AND PROGRAMMING USING PYTHON (Editorial MIT Press, Año 2013)
- **Ley de Moore** https://es.wikipedia.org/wiki/Ley_de_Moore
- **Tanenbaum Andrew S., Wetherall David J.** - REDES DE COMPUTADORAS - Quinta edición (Editorial Pearson, Año 2012)
- **Tanenbaum Andrew S.**- SISTEMAS OPERATIVOS MODERNOS - Tercera edición (Editorial Pearson, Año 2009)
- **Cola (Informática)** https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)
- **Bubble Sort** https://en.wikipedia.org/wiki/Bubble_sort
- **Ordenamiento por Inserción** https://es.wikipedia.org/wiki/Ordenamiento_por_inserci%C3%B3n
- **Sorting algorithms/Insertion sort** http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort
- **Algoritmos de Ordenación** https://elbauldelprogramador.com/algoritmos-de-ordenacion/
- **Ordenamiento por Selección** https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n
- **Ordenamiento rápido (Quicksort)** http://www.mis-algoritmos.com/ordenamiento-rapido-quicksort
- **Quicksort** https://es.wikipedia.org/wiki/Quicksort
- **Variables y Constantes** http://aurea.es/assets/2-tiposdatoslenguajec.pdf
- http://ficus.pntic.mec.es/rdis0006/lecciones/logica_proposicional/lecciones/funciones%20veritativas.htm
- https://www.zweigmedia.com/MundoReal/logic/logic1.html
- https://logicaformalunah.files.wordpress.com/2017/01/irving_m-_copi_carl_cohen_introduccion_a_la_log.pdf
- http://www.cs.us.es/cursos/li-2003/li-g-2-3/libro-logica.pdf

- https://es.wikiversity.org/wiki/L%C3%B3gica_proposicional/Tablas_de_verdad
- https://es.wikiversity.org/wiki/L%C3%B3gica_proposicional/Proposiciones_compuestas