

# Tema 1. Introducción a la Programación

Vicente Benjumea García

Introducción a la Programación  
Departamento de Lenguajes y Ciencias de la Computación.  
E.T.S.I. Informática. Univ. de Málaga.

# Introducción a la Programación

## Tema 1. Introducción a la programación

- La informática y el papel de la programación de computadores.
- El computador: una máquina que procesa información.
- Algoritmos y resolución de problemas.
  - ¿Que se quiere hacer? Concepto de Algoritmo.
  - ¿Cómo hay que hacerlo? Corrección.
  - ¿Qué se puede hacer? Calculabilidad y Complejidad.
- Lenguajes de programación.
  - Paradigmas de Programación.
  - Definición de Lenguajes. Gramáticas.
  - Traductores, Compiladores e Intérpretes.
- Visión general de un sistema informático.
- Estructura funcional de los computadores.
- Codificación de la información.
  - Representacional posicional de los números.
  - Códigos de entrada y salida de datos.

Esta obra se encuentra bajo una licencia Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) de Creative Commons.



# La Informática y la Programación de Computadores

- **INFORMÁTICA:**

- Procesamiento de la **INFOR**mación de forma auto**MÁTICA**.

- **CIENCIA:**

- Conjunto de **conocimientos científicos** sobre modelos de computación, la representación y el procesamiento automático de la información.

- **INGENIERÍA:**

- Utilización, diseño, y creación de **técnicas y herramientas prácticas** para el procesamiento de la información de forma automática y resolver problemas reales por medio de computadores electrónicos.

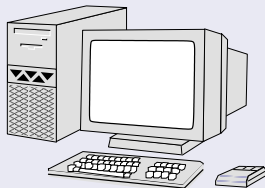
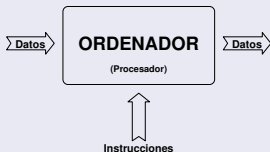
- La programación de computadores es una actividad fundamental y transversal asociada a cualquier área de la informática:

- Ingeniería de los Computadores
- Ingeniería del Software
- Sistemas de Información
- Inteligencia Artificial
- Etc.

# El Computador: una Máquina que Procesa Información

- **COMPUTADOR (ORDENADOR):**

- Entidad capaz de **procesar** información automáticamente.
  - **Hardware:** es la parte física del computador (circuitos electrónicos, parte mecánica).
  - **Software:** es la parte lógica del computador (programas/instrucciones).
- El procesamiento de información consiste en:
  - El computador recibe información de **entrada** (en forma de **datos**).
  - El **programa** (secuencia de instrucciones) le indica al computador como debe manipular y transformar los datos.
  - Como **resultado** de la computación, se produce información de **salida** (en forma de **datos**).



# El Computador: una Máquina que Procesa Información

## • INSTRUCCIONES:

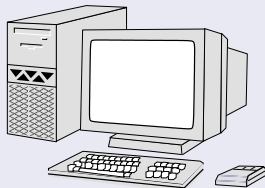
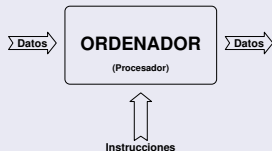
- Conjunto de símbolos que representan **órdenes de operación** que indican al computador como manipular y transformar los **datos**.

## • PROGRAMA:

- Secuencia de **instrucciones** que ejecutadas por el computador, resuelve un determinado problema.

## • LENGUAJE DE PROGRAMACIÓN:

- Establece los símbolos y reglas para codificar **programas**.



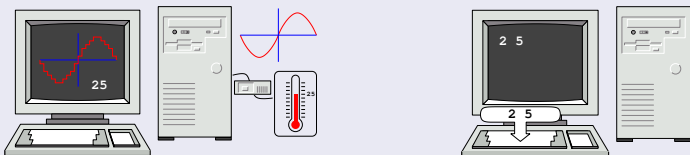
# El Computador: una Máquina que Procesa Información

- DATOS:

- Conjunto de símbolos utilizados para **representar la información**: entidades, objetos, magnitudes, hechos, conceptos, ideas, etc.
- **Codificados** en la forma adecuada para su procesamiento automático.

- Es necesaria una codificación de los datos:

- Codificación adecuada para representar *información de entrada*:
  - Datos captados directamente (convertidores analógico/digital, etc.).
  - Datos captados mediante **caracteres**.
- Codificación adecuada para su *representación interna y procesamiento*.
- Codificación adecuada para representar *información de salida*:
  - Resultados proporcionados directamente (convertidores digital/analógico, etc.).
  - Resultados proporcionados mediante **caracteres**.



## Programación

**Objetivo:** definir una **secuencia de acciones**, que tras ser ejecutadas por un **procesador**, resuelva un determinado **problema**.

## Fases para Desarrollar un Programa

- 1 Análisis del problema.
- 2 Estudio de su solución.
- 3 Diseño del **algoritmo**.
- 4 Codificación del programa en un determinado lenguaje de programación.
- 5 Depuración y prueba.

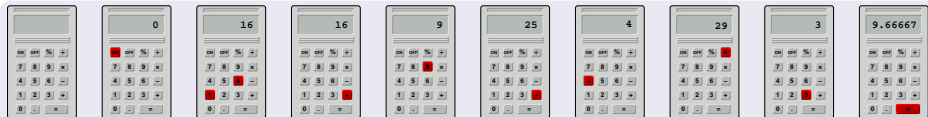
# Concepto de Algoritmo

- Ejemplos intuitivos de algoritmos:

- Problema: calcular la media aritmética de tres números cualesquiera utilizando una calculadora básica.

- Fase 1: Análisis del problema.
- Fase 2: Estudio de la solución.

Por ejemplo, la media de los números 16, 9, 4, dará como resultado 9.66667.



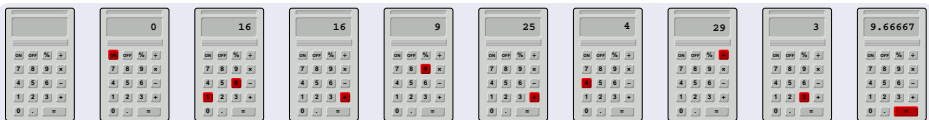


# Concepto de Algoritmo

- Fase 3: Diseño del **algoritmo** (en lenguaje natural).

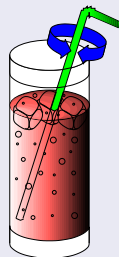
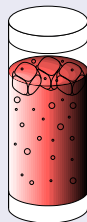
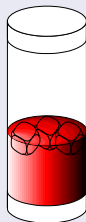
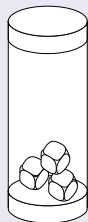
## Algoritmo: calcular media de tres números con una calculadora

- 1 Pulsar la tecla "ON"
- 2 Teclear los dígitos del primer número
- 3 Pulsar la tecla "+"
- 4 Teclear los dígitos del segundo número
- 5 Pulsar la tecla "+"
- 6 Teclear los dígitos del tercer número
- 7 Pulsar la tecla "÷"
- 8 Pulsar la tecla "3"
- 9 Pulsar la tecla "="
- 10 La media de los tres números aparece en la pantalla
- 11 Pulsar la tecla "OFF"



# Concepto de Algoritmo

- Ejemplos intuitivos de algoritmos:
  - Problema: Preparación de un **“Tinto de Verano”**
    - Fase 1: Análisis del problema.
    - Fase 2: Estudio de la solución.

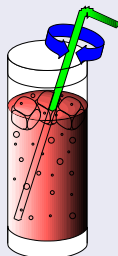
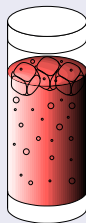
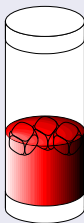
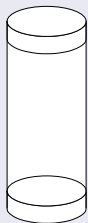


# Concepto de Algoritmo

- Fase 3: Diseño del **algoritmo** (en lenguaje natural).

## Algoritmo: preparación de un “Tinto de Verano”

- 1 Tomar un vaso.
- 2 Colocar algunos cubitos de hielo en el vaso.
- 3 Echar vino tinto en el vaso.
- 4 Añadir gaseosa al contenido del vaso.
- 5 Agitar el contenido.

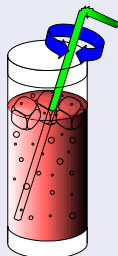
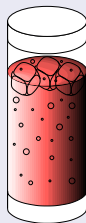
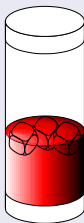
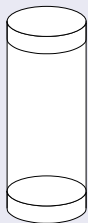


# Concepto de Algoritmo

- Fase 3: Diseño del **algoritmo** (en lenguaje natural).

## Algoritmo: preparación de un “Tinto de Verano”

- 1 Tomar un vaso **limpio vacío**.
- 2 Colocar **tres** cubitos de hielo en el vaso.
- 3 Echar vino tinto **hasta la mitad** del vaso.
- 4 Añadir gaseosa **hasta llenar** del vaso.
- 5 Agitar **tres segundos** el contenido.



## Definiciones

- **Procesador**: entidad capaz de **entender** una secuencia finita de acciones y **ejecutarlas** en la forma en que se especifican.
- **Entorno**: conjunto de **condiciones** necesarias para la ejecución de un algoritmo.
- **Acciones primitivas**: son acciones que el procesador es capaz de **entender y ejecutar directamente**.
- **Secuencialidad**: cada acción se ejecuta en orden, cuando la anterior ha terminado.
- **Paralelismo**: es posible ejecutar varias acciones simultáneamente.
- **Algoritmo**:
  - Dado un procesador y un entorno bien definido, es el enunciado de una **secuencia finita** de **acciones primitivas** que **resuelven** un determinado problema.
  - Hay que considerar 3 aspectos a la hora de establecer un algoritmo:
    - Lenguaje simbólico a utilizar
    - Acciones Primitivas
    - Representación de los datos.
- **Programa**: es la implementación de un algoritmo, codificado en un lenguaje de programación concreto, adecuado para ser ejecutado en una computadora.

# Concepto de Algoritmo

## Refinamiento de Acciones Primitivas

- Un algoritmo es el enunciado de una secuencia finita de acciones primitivas que resuelve un problema.
- En caso de que tengamos acciones que no sean primitivas, estas acciones no primitivas se tendrán que **refinar**.
  - Serán como un *subproblema* dentro del *problema general*.
  - Tendremos que especificar la secuencia de acciones primitivas necesarias para resolver este *subproblema*.
  - Repetiremos este proceso hasta que finalmente el algoritmo este expresado completamente en base a acciones primitivas.

## Refinamiento de Acciones Primitivas. Ejemplo

Algoritmo: preparación de un “Tinto de Verano”.

- 1 Tomar un vaso **limpio vacío**.
- 2 Colocar **tres** cubitos de hielo en el vaso.
- 3 Echar vino tinto **hasta la mitad** del vaso.
- 4 Añadir gaseosa **hasta llenar** del vaso.
- 5 Agitar **tres segundos** el contenido.

## Refinamiento de Acciones Primitivas

Algoritmo: preparación de un “Tinto de Verano”.

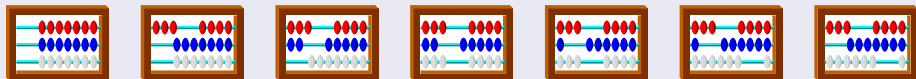
- ① Tomar un vaso **limpio vacío**.
- ② Colocar **tres** cubitos de hielo en el vaso.
  - ① Sacar la cubitera del congelador.
  - ② Rociar la parte inferior con agua.
  - ③ **REPETIR**
    - ① Extraer un cubito de la cubitera.
    - ② Echarlo al vaso.
  - ④ **HASTA QUE** el nº de cubitos sea 3.
  - ⑤ Rellenar los huecos de la cubitera con agua.
  - ⑥ Meter de nuevo la cubitera en el congelador.
- ③ Echar vino tinto **hasta la mitad** del vaso.
- ④ Añadir gaseosa **hasta llenar** del vaso.
- ⑤ Agitar **tres segundos** el contenido.

# Concepto de Algoritmo

## Problema: producto de dos números cualesquiera $X$ e $Y$

- **Entorno:** ábaco de tres filas.
- **Procesador:** persona que sabe contar y desplazar bolas. Sólo puede recordar la última cuenta.
- **Representación de los datos:** número de bolas desplazadas a la izquierda.

- Fase 1: Análisis del problema.
- Fase 2: Estudio de la solución.



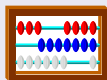
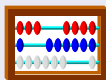
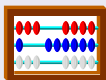
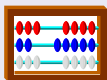
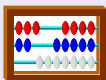
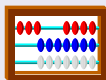
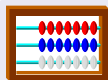


# Concepto de Algoritmo

- Fase 3: Diseño del **algoritmo** (en lenguaje natural).

## Algoritmo: producto de dos números cualesquiera $X$ e $Y$

- 1 **Desplazar** todas las bolas a la derecha.
- 2 **Desplazar** tantas bolas rojas a la izquierda como el valor del primer número.
- 3 **Desplazar** tantas bolas azules a la izq. como el valor del segundo número.
- 4 **MIENTRAS** haya bolas azules a la izquierda **REPETIR**
  - 1 **Desplazar** tantas bolas blancas a la izq. como bolas rojas haya a la izq.
  - 2 **Desplazar** una bola azul a la derecha.
- 5 **HASTA-AQUI**
- 6 El resultado es el número de bolas blancas a la izquierda.



## Objetivos en el Desarrollo de Software

- Construir un sistema software que sea **correcto**:
  - Que se comporte según las especificaciones del problema a resolver.
  - Que cumpla los requisitos y restricciones especificados.
  - Que cuando sea ejecutado, resuelva el problema correctamente, para todos los valores posibles, y siempre que sea ejecutado.
- Se debe seguir una metodología que facilite la detección y corrección de errores.

## Un programa puede presentar varios tipos de errores:

- **Errores de compilación**:
  - Lexicográficos, sintácticos y semánticos.
  - No son importantes, ya que el compilador avisa de ellos, y son fáciles de corregir.
- **Errores lógicos (de comportamiento)**:
  - Son difíciles de detectar y de corregir.
  - Hacen que el comportamiento del programa no sea adecuado. Es decir, hacen que la ejecución del **programa no resuelva** adecuadamente el problema.
  - Se introducen durante las fases de especificación del problema, el estudio de la solución, el diseño del algoritmo o la codificación del programa.
  - **Errores en tiempo de ejecución**: son errores lógicos que hacen que el programa aborte la ejecución.

## ¿ Como comprobar si un programa es correcto ?

- **Prueba experimental:** se ejecuta el programa múltiples veces para conjuntos de datos adecuadamente seleccionados, y se comprueba si los resultados son los esperados.
  - Si durante alguna prueba se produce algún error entonces el programa es **incorrecto**.
  - Sin embargo, aunque las pruebas sí hayan producido los resultados esperados, **NO** se puede concluir que el programa sea correcto.
  - Es muy importante la selección de los datos para los casos de prueba.
  - **La prueba experimental puede ser empleada para mostrar la presencia de errores, pero nunca su ausencia.**

**LA PRUEBA EXPERIMENTAL  
PUEDE MOSTRAR QUE UN PROGRAMA SEA INCORRECTO,  
PERO NUNCA PUEDE MOSTRAR QUE UN PROGRAMA SEA CORRECTO.**

- La única forma de demostrar la corrección de un programa es mediante la **Verificación Formal** de programas.
  - Es un método matemático basado en técnicas de demostración de teoremas.
  - Es un método complejo y difícil de llevar a la práctica.

## Metodología de Programación y Ciclo de Vida del Software

- Cuando desarrollamos programas, es posible que cometamos **errores**, que podrían ser detectados durante el desarrollo, o bien en el futuro durante su utilización.
- Una vez que un programa es desarrollado, será **utilizado** durante un largo periodo de tiempo.
  - Durante su periodo de utilización, el software deberá ir evolucionando para adaptarse a **nuevos requisitos**.
  - Además, cuando se detecten errores, deberán ser **depurados** y corregidos adecuadamente.
  - Estos cambios en el software podrán ser desarrollados por el propio programador, por otros programadores, o por un equipo de ellos.
- Debemos seguir **metodologías de programación y estrategias adecuadas** para desarrollar **software de calidad**:
  - Que facilite la modificación y evolución del software.
  - Que minimice las posibilidades de introducir errores en el software,
  - Que facilite la depuración (detección y corrección) de los errores existentes.
  - Que facilite la cooperación entre diversos desarrolladores a lo largo del tiempo.

- A principios del siglo XX existió la creencia de que no existía ningún problema que no pudiera resolverse algorítmicamente.
- David Hilbert (1862-1943): desarrolló un **sistema matemático formal** con el objetivo de resolver de forma algorítmica **cualquier problema** que se plantease.
- A partir de estos estudios, muchos autores (Church, Kleene, Post, Turing, etc.) encontraron problemas **NO COMPUTABLES**.
  - Si un problema es **NO COMPUTABLE**, entonces significa que **NO EXISTE, Y NUNCA EXISTIRÁ**, ningún algoritmo que lo resuelva.

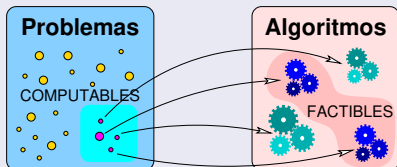
# Calculabilidad y Complejidad

La teoría de la **calculabilidad** intenta identificar qué **problemas son computables** (existe una solución algorítmica) y cuales no.

- Ejemplo de problema no computable: *el problema de la parada*.

La teoría de la **complejidad** se encarga de estudiar la cantidad de **recursos computacionales** (en tiempo y espacio) que necesita un determinado algoritmo para ejecutarse.

- Para ello, se contabilizan el número de **operaciones básicas** que un algoritmo realiza, en relación con el **tamaño del problema** a resolver.
- Sólo aquellos algoritmos que utilizan una **cantidad factible de recursos** son útiles en la práctica.
- Ejemplo de algoritmo no factible: *el problema del viajante de comercio*.



T.CALCULABILIDAD

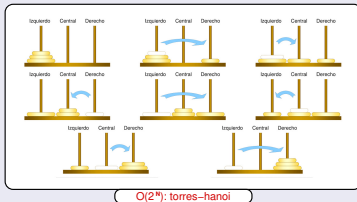
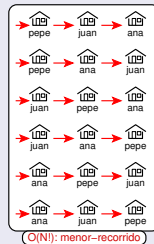
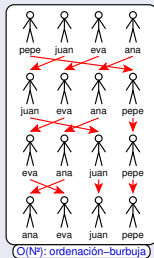
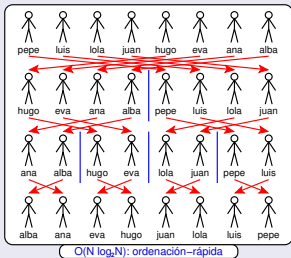
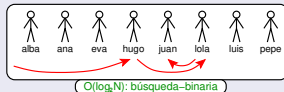
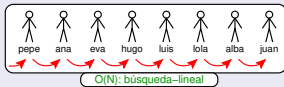
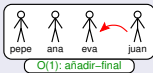
T.COMPLEJIDAD

## Orden de Complejidad Respecto al Tamaño del Problema a Resolver

El orden de complejidad ( $O_N$ ) representa el número simplificado de operaciones básicas realizadas por un algoritmo, al resolver un problema dado de tamaño  $N$ .

Orden	Ejemplo
$O(1)$	Añadir un alumno al final de una lista desordenada
$O(\log_2 N)$	Buscar un alumno en una lista ordenada (búsqueda-binaria)
$O(N)$	Buscar un alumno en una lista desordenada (búsqueda-lineal)
$O(N \log_2 N)$	Ordenar una lista de alumnos (ordenación-rápida)
$O(N^2)$	Ordenar una lista de alumnos (ordenación-burbuja)
$O(2^N)$	Movimientos de las "Torres de Hanoi" (movimientos de Ajedrez, etc)
$O(N!)$	Calcular el menor recorrido que pase por las casas de la lista de alumnos

# Complejidad



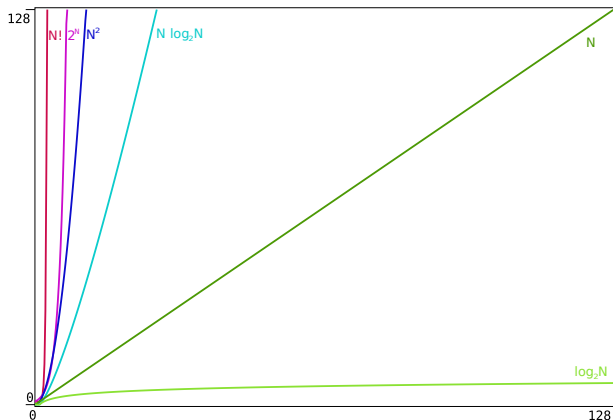


## Orden de Complejidad Respecto al Tamaño del Problema a Resolver

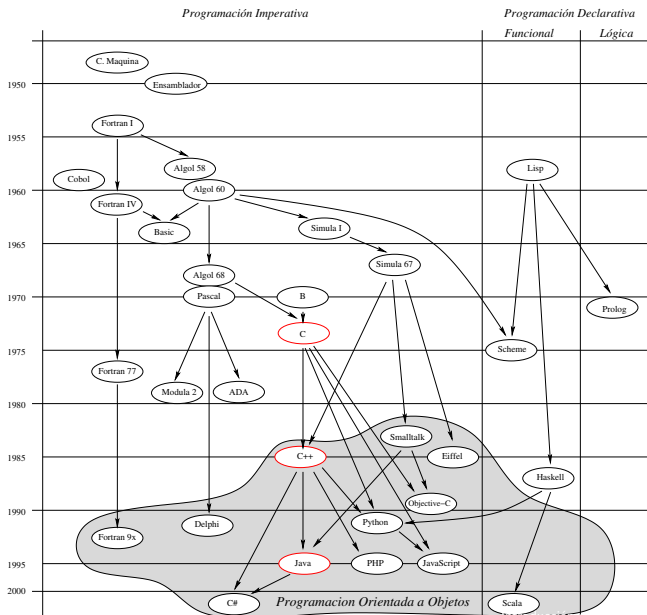
El orden de complejidad ( $O_N$ ) representa el número simplificado de operaciones básicas realizadas por un algoritmo, al resolver un problema dado de tamaño  $N$ .

Orden	N:32	N:64	N:128	N:256	N:512	N:1024
$O(1)$	1	1	1	1	1	1
$O(\log_2 N)$	5	6	7	8	9	10
$O(N)$	32	64	128	256	512	1024
$O(N \log_2 N)$	160	384	896	2048	4608	10240
$O(N^2)$	1024	4096	16384	65536	262144	1048576
$O(2^N)$	$4 \times 10^9$	$4 \times 10^{19}$	$3 \times 10^{38}$	$1 \times 10^{77}$	$1 \times 10^{154}$	$\infty$
$O(N!)$	$2 \times 10^{35}$	$1 \times 10^{89}$	$3 \times 10^{215}$	$\infty$	$\infty$	$\infty$

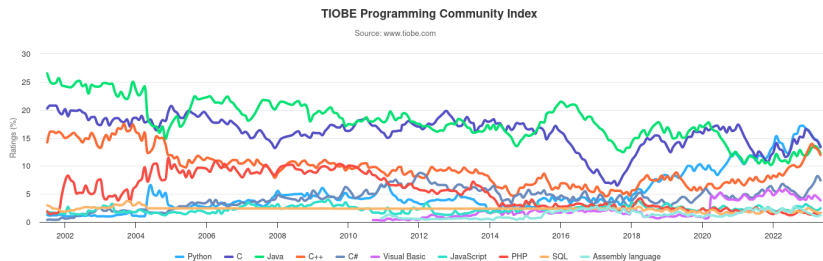
## Curvas de Eficiencia:



# Lenguajes de Programación



- Clasificación de los lenguajes más *populares* (2023) según *Tiobe* (<https://www.tiobe.com/tiobe-index/>):



- El **lenguaje de programación** establece los **símbolos** y las **reglas** para codificar programas en ese determinado lenguaje.
- Los lenguajes de programación se pueden clasificar según diferentes criterios, entre ellos:
  - Nivel de abstracción.
  - Propósito del lenguaje.
  - Paradigmas de programación soportados.

## Clasificación según el Nivel de Abstracción

- **Código Máquina:**
  - Código **binario** que puede **ejecutar directamente** el procesador.
- Lenguaje de Programación de **bajo nivel:**
  - Lenguaje para codificar programas en términos de la arquitectura y las **operaciones básicas** que realiza un procesador. Por ej.: Ensamblador, etc.
- Lenguaje de Programación de **alto nivel:**
  - Lenguaje para codificar programas en **términos abstractos** cercanos al problema a resolver. Por ej.: C++, Java, Python, etc.

## Clasificación según el Propósito

- **Científicos:** Algol, Fortran, ...
- **Ingeniería:** Ada, Dynamo, ...
- **Gestión:** Cobol, SQL, ...
- **Inteligencia Artificial:** Lisp, Prolog, ...
- **Aplicaciones Web:** PHP, JavaScript, ...
- **Propósito General:** Pascal, Modula-2, C, C++, Java, Python, ...

## Clasificación según el Paradigma de Programación Soportado

- Un **paradigma de programación** es un **modelo** que determina ciertas características **estructurales** y **metodológicas** de los lenguajes de programación. Existen lenguajes **multi-paradigma**.
- **Programación Imperativa**: el programa define una **secuencia de acciones** que se ejecutan en el orden especificado y manipulan el estado de las variables.
- **Programación Declarativa**: el programa especifica la lógica del problema, pero es el modelo el que define cómo se realiza la computación.
  - **Programación Funcional**: el programa define funciones. La computación consiste en la evaluación de las funciones especificadas, sobre datos inmutables, según el modelo de computación **lambda-cálculo**.
  - **Programación Lógica**: el programa define hechos y reglas lógicas. La computación consiste realizar una demostración, a partir de los hechos y reglas, mediante el principio de **resolución** como regla de inferencia.
- **Programación Orientada a Objetos**: el programa define un modelo del problema a resolver, basado en la definición de **objetos** como **abstracciones** de los datos, que especifican su *comportamiento* y manipulan su *estado interno*.



## Paradigma de Programación Imperativa

**Programación Imperativa:** el programa define una **secuencia de acciones** que se ejecutan en el orden especificado y manipulan el estado de las variables.

## Ejemplo. Cálculo del factorial de N en C++

```
int fact(int n)           // definición de la función factorial de N
{
    int fk;              // VARIABLE para almacenar los valores
    fk = 1;              //  $F_0$  y  $F_1$  son ambos igual a 1
    for (int k = 2; k <= n; ++k) { // iterar para los valores de  $K \in \{2 \dots N\}$ 
        fk = k * fk;      // ASIGNAR a  $F_k$  el nuevo valor  $K \times F_{k-1}$ 
    }                     //  $F_n = 1 \times 2 \times 3 \times \dots \times N$ 
    return fk;           // devolver el resultado  $F_n$  calculado
}
```

## Paradigma de Programación Funcional

**Programación Funcional:** el programa define funciones. La computación consiste en la evaluación de las funciones especificadas, sobre datos inmutables, según el modelo de computación **lambda-cálculo**.

## Ejemplo. Cálculo del factorial de N en Lisp

```
(defun fact (N) ; definición de la función factorial de N
  (if (= N 0) ; si N es igual a cero
      1 ; entonces el resultado es 1
      (* N (fact (- N 1))) ; sino, el resultado es N x fact(N-1)
  )
)
```

*; defun, fact, if, =, \* y - son FUNCIONES*

## Paradigma de Programación Lógica

**Programación Lógica:** el programa define hechos y reglas lógicas. La computación consiste realizar una demostración, a partir de los hechos y reglas, mediante el principio de **resolución** como regla de inferencia.

## Ejemplo. Cálculo de antecesores y descendientes en Prolog

```
antecesor(X,Y) :- padre(X, Y).           % Regla: X es ant. de Y si X es padre de Y
antecesor(X,Y) :- padre(Z, Y),         % Regla: X es ant. de Y si Z es padre de Y
                    antecesor(X, Z). % y además X es ant. de Z

padre("Juan", "María").                 % Hecho: Juan es padre de María
padre("Pepe", "Juan").                  % Hecho: Pepe es padre de Juan
padre("David", "Pepe").                 % Hecho: David es padre de Pepe

?- antecesor(X, "María");               % ¿ Antecesores de María ? Juan, Pepe y David

?- antecesor("David", X);               % ¿ Descendientes de David ? Pepe, Juan y María
```

## Paradigma de Programación Orientada a Objetos

**Programación Orientada a Objetos:** el programa define un modelo del problema a resolver, basado en la definición de **objetos** como **abstracciones** de los datos, que especifican su *comportamiento* y manipulan su *estado interno*.

## Ejemplo. Recuento de Votos en Java

```
public class Urna { // Abstracción sobre una urna de votación
    private int cntNo, cntSi; // Estado interno privado de cada objeto
    public Urna() { // Constructor del objeto
        cntSi = 0; // Inicializa la cuenta de positivos
        cntNo = 0; // Inicializa la cuenta de negativos
    }
    public void votar(boolean val) { // Añade un voto positivo o negativo
        if (val) {
            ++cntSi; // Incrementa la cuenta de positivos
        } else {
            ++cntNo; // Incrementa la cuenta de negativos
        }
    }
    public boolean resultado() { // devuelve el resultado de la votación
        return (cntSi > cntNo);
    }
}
```

- Un programa se codifica utilizando un **Lenguaje de Programación**.
- Un Lenguaje de Programación se define mediante una **Gramática**.
- Una **Gramática** establece los símbolos y las reglas de como se generan y analizan las frases de un determinado lenguaje.
- La gramática de un lenguaje se puede expresar de diferentes formas:
  - **Notación EBNF**.
  - **Diagramas Sintácticos de Conway**.

## Notación EBNF

- Los **símbolos no-terminales** se especifican entre  $\langle \dots \rangle$
- Los **símbolos terminales** se especifican *directamente*
- Una **regla** se especifica con el símbolo  $::=$ 
  - La **alternativa** se especifica con el símbolo  $|$
  - La **repetición** (cero o más veces) se especifica entre  $\{ \dots \}$
- Ejemplo que define, utilizando la notación EBNF, como se genera y analiza un **identificador** en un lenguaje de programación determinado.

(1)  $\langle \text{identificador} \rangle ::= \langle \text{letra} \rangle \{ \langle \text{letra} \rangle | \langle \text{digito} \rangle | \_ \}$

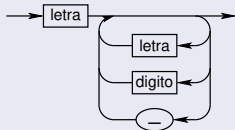
(2)  $\langle \text{letra} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m$   
 $| n | o | p | q | r | s | t | u | v | w | y | z$   
 $| A | B | C | D | E | F | G | H | I | J | K | L | M$   
 $| N | O | P | Q | R | S | T | U | V | W | Y | Z$

(3)  $\langle \text{digito} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

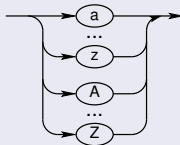
## Diagramas Sintácticos de Conway

- Los **símbolos no-terminales** se especifican en un rectángulo.
- Los **símbolos terminales** se especifican en una elipse.
- Una **regla** se especifica con un diagrama sintáctico.
  - La **alternativa** se especifica con caminos alternativos.
  - La **repetición** (cero o más veces) se especifica con bucles.
- Ejemplo que define, utilizando los diagramas sintácticos de Conway, como se genera y analiza un **identificador** en un lenguaje de programación determinado.

identificador :



letra :



digito :

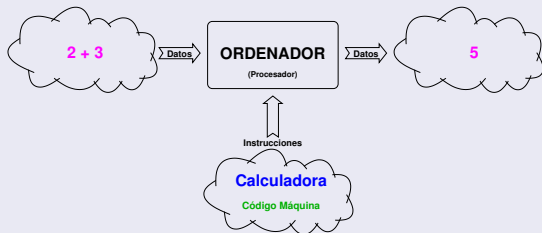
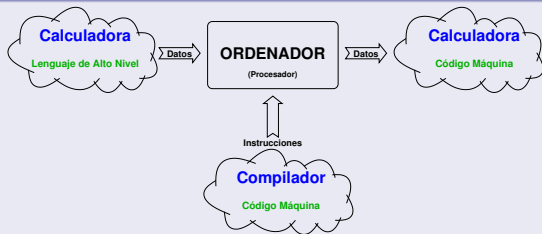


## Traducción entre Lenguajes

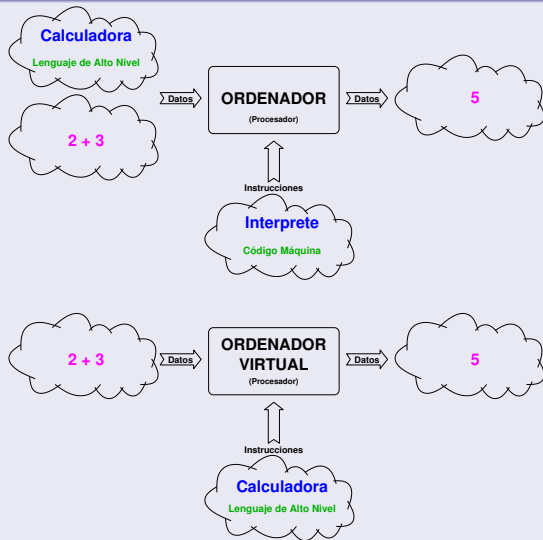
- El computador sólo puede **ejecutar** un programa codificado en **Código Máquina** binario, específico para ese computador.
- Sin embargo, es **necesario** que el procesador sea capaz de **ejecutar** un programa expresado en un lenguaje de **alto nivel**.
- Es necesario realizar una **traducción** entre lenguajes de programación.
  - **COMPILADOR**: **traduce** un programa escrito en un lenguaje de alto nivel a su equivalente en código máquina. Por ej. C++.
  - **INTERPRETE**: hace posible que el computador sea capaz de **ejecutar directamente** un programa escrito en un lenguaje de alto nivel. Por ej. Basic.
  - **Mixtas**: *compilación* a un código intermedio, y su *interpretación* mediante un emulador de una máquina abstracta. Por ej. Java, Python.



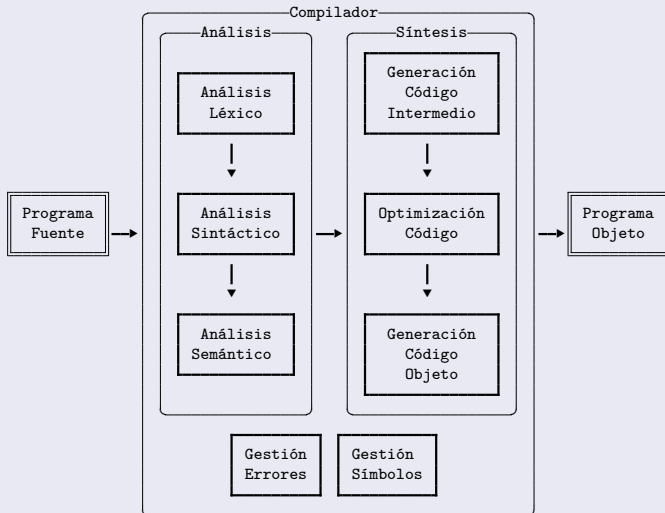
## Compilador



## Intérprete



## Esquema de un Compilador



## Análisis Léxico

- Agrupa los símbolos del programa en unidades léxicas, denominadas tokens.
- Elimina los comentarios, espacios en blanco, retornos de carro, tabuladores, etc.
- Añade los identificadores a la tabla de símbolos.
- Avisa de los errores léxicos que detecte.

## Análisis Sintáctico

- Crea un árbol sintáctico a partir de la secuencia de tokens del análisis léxico.
  - El árbol sintáctico sirve para el análisis semántico y la generación de código.
- Avisa de los errores sintácticos que detecte.

## Análisis Semántico

- Comprueba que una frase es semánticamente correcta.
  - Ejemplo asignación de valores de distintos tipos.
- Avisa de los errores semánticos que detecte.

## Generación de Código Intermedio

- Generación de código independiente de la máquina para la que se hace el compilador.
- Debe de ser fácil de producir a partir del análisis y fácil de traducir al código definitivo.

## Optimización de Código

- Genera un código más compacto y eficiente.
- Elimina las acciones redundantes e innecesarias.

## Generación de Código Objeto

- Genera el código objeto final a partir del código optimizado.

## Enlazado

- Enlazado de los diferentes códigos objeto que componen los módulos del programa para generar el código ejecutable.



# Visión General de un Sistema Informático

## Hardware

Conjunto de circuitos y dispositivos físicos de un computador.

## Software

Conjunto de programas que pueden ser ejecutados por un computador.

## Sistema Operativo

Software que controla el hardware y los recursos del computador (Unix, Linux, MacOSX, Windows, Android, IOS, etc.).

- Drivers, *firmware*, control de hardware y periféricos.
- Control de procesos, multitarea, comunicaciones, E/S.
- Gestión de memoria principal, memoria virtual.
- Sistema de archivos en memoria secundaria.
- Soporte al software de sistema y aplicación, librerías.
- Herramientas del sistema e interacción con el usuario.

# Visión General de un Sistema Informático

## Software de Aplicaciones

Software diseñado para proporcionar utilidades específicas a los usuarios.  
Navegadores, procesadores de texto, hojas de cálculo, gráficos, multimedia, juegos.

## Software de Sistema

Herramientas de administración, mantenimiento, entorno gráfico, desarrollo de software, editores, compiladores, librerías, emuladores, entornos de desarrollo (IDE).

## Entorno de Desarrollo Integrado (IDE)

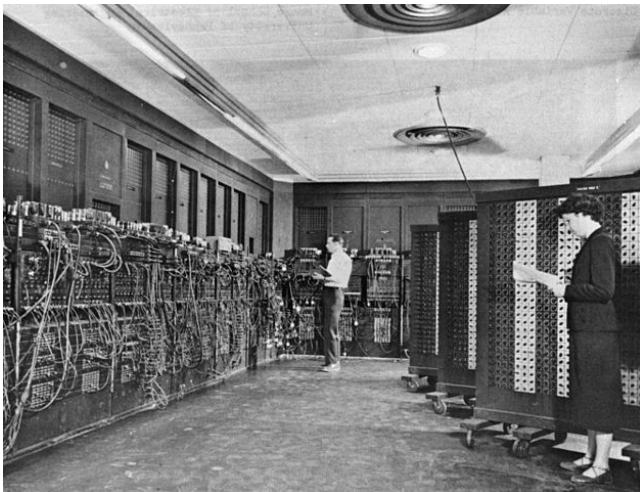
Es un programa informático compuesto por un conjunto de herramientas integradas para facilitar las tareas involucradas en el desarrollo de programas.

- Se encuentra compuesto por las siguientes herramientas (programas):
  - Editor de textos.
  - Compilador y Enlazador.
  - Depurador.
  - Control de versiones.
  - Herramientas auxiliares (análisis y generación de código, gestión de proyectos, importación, exportación, etc).



# Estructura Funcional de los Computadores

- En los inicios, la programación era **cableada**. El computador (ENIAC) ocupaba una sala.



# Estructura Funcional de los Computadores

- Placa base con procesador (bajo el ventilador), memoria principal, controladores y buses.



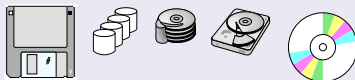
# Estructura Funcional de los Computadores

- System on a Chip (SoC): todo el sistema en un chip (Raspberry-Pi).

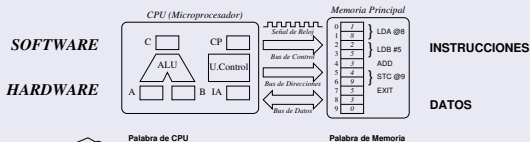


# Estructura Funcional de los Computadores

- La arquitectura *Von Neumann* se caracteriza porque tanto las **instrucciones** (que forman el programa) como los **datos** se encuentran almacenados en la **memoria principal**, codificados en *binario*.



Sistemas de Almacenamiento



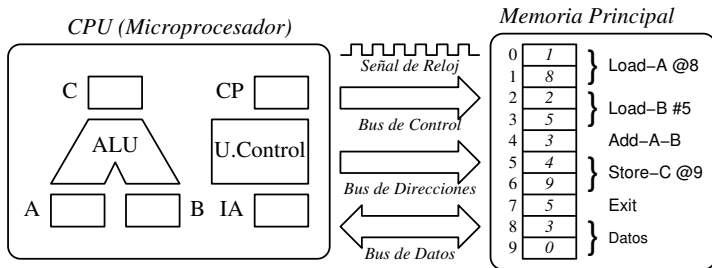
## Arquitectura **Von Neumann** (programa almacenado):

- **Procesador** (Unidad Central de Procesamiento – CPU): ejecuta las instrucciones.
  - **Unidad de Control** (CU): controla la ejecución de las instrucciones.
  - **Unidad Aritmético/Lógica** (ALU): realiza operaciones aritméticas y lógicas con operandos numéricos (binarios).
  - **Registros**: almacenan operandos numéricos, resultados de las operaciones, así como información adecuada para la ejecución del programa.
  - **Buses** (Control, Direcciones y Datos): son el medio para la transferencia de información entre el procesador, la memoria principal y el resto de dispositivos.
  - El tamaño de la palabra de CPU indica el número de bits con el que es capaz de trabajar el procesador.
  - La frecuencia del reloj y el tamaño de la palabra de CPU determinan la **potencia** del procesador.
- **Memoria Principal**: almacena tanto los programas como los datos (en forma de números codificados en binario). RAM, ROM.
- **Memoria Secundaria**: sistemas de almacenamiento persistente (no-volátil).
- **Dispositivos Periféricos**: transferencia de información con el exterior.

# Funcionamiento Interno de los Computadores

- Para **ejecutar** un programa codificado en código máquina (binario), el programa **cargador** debe *cargar* el programa en posiciones consecutivas de una zona de la memoria principal.
- A continuación comienza la ejecución del programa a partir de su dirección de inicio.
  - La *ejecución de un programa* se compone de la alternancia de **dos fases**:
    - **Captación de la Instrucción**: la Unidad de Control dirige la carga, desde la memoria principal, de la próxima instrucción a ejecutar (cuya dirección está indicada por el registro *contador de programa* – CP) a un registro interno de ejecución (IA).
      - ▶ Además, incrementa el valor del *registro contador de programa* (CP) para que apunte a la próxima instrucción.
    - **Ejecución de la Instrucción**: la Unidad de Control ejecuta, con la asistencia de la ALU, la instrucción actual (IA) (carga y almacenamiento de datos, aritmética, lógica, control), utilizando los registros internos (A, B, C) para el almacenamiento temporal de los datos.
      - ▶ En caso de *instrucción de salto*, se actualiza el *registro contador de programa* (CP) con la dirección de la nueva instrucción a ejecutar.
  - La transferencia de información entre el procesador y la memoria se realiza a través de los **buses**, sincronizada por la señal de reloj.

# Funcionamiento Interno de los Computadores



## CÓDIGO BINARIO

Los datos se representan internamente en el computador mediante el **CÓDIGO BINARIO**.

## BIT

El **BIT** es la **unidad elemental de información** y puede representar 2 estados diferentes.

- Usualmente se representa con los símbolos 0 y 1.
- Se representa de forma electrónica con dos voltajes diferentes: 0v y 5v (o 3v).
- Los bits se pueden agrupar para representar más estados diferentes:
  - **2 BITS** representan 4 ( $2^2$ ) estados: 00, 01, 10, 11.
  - **3 BITS** representan 8 ( $2^3$ ) estados: 000, 001, 010, 011, 100, 101, 110, 111.

## BYTE

El **BYTE** son 8 bits, representa 256 ( $2^8$ ) estados diferentes. Por ejemplo: 01101011. Es la **unidad mínima de trabajo**.



## Múltiplos del byte:

Sistema	Internacional	ISO/IEC	80000-13
kilobyte	1kB = $10^3$ B	kibibyte	1KiB = $2^{10}$ B
megabyte	1MB = $10^6$ B	mebibyte	1MiB = $2^{20}$ B
gigabyte	1GB = $10^9$ B	gibibyte	1GiB = $2^{30}$ B
terabyte	1TB = $10^{12}$ B	tebibyte	1TiB = $2^{40}$ B
petabyte	1PB = $10^{15}$ B	pebibyte	1PiB = $2^{50}$ B
exabyte	1EB = $10^{18}$ B	exbibyte	1EiB = $2^{60}$ B
zettabyte	1ZB = $10^{21}$ B	zebibyte	1ZiB = $2^{70}$ B
yottabyte	1YB = $10^{24}$ B	yobibyte	1YiB = $2^{80}$ B

# Representación Posicional de los Números

## Representación Posicional de los Números:

- Un sistema de numeración en base **B** utiliza un alfabeto compuesto por **B** símbolos o cifras para representar los números.
  - Sistema Binario (base 2): 0 1
  - Sistema Decimal (base 10): 0 1 2 3 4 5 6 7 8 9
  - Sistema Hexadecimal (base 16): 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Cada cifra contribuye al valor total representado por el número con un valor que depende de:
  - El valor asociado a la cifra en sí.
  - Un valor asociado a la posición (*i*) de la cifra dentro del número.

$$valor\_total = \sum_i valor\_cifra_i \times base^i$$

## Ejemplos:

- $3278.52_{(10)}$   
 $= 3 \times 10^3 + 2 \times 10^2 + 7 \times 10^1 + 8 \times 10^0 + 5 \times 10^{-1} + 2 \times 10^{-2} = 3278.52_{(10)}$
- $1101_{(2)}$   
 $= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{(10)}$
- $1A6F.B3_{(16)}$   
 $= 1 \times 16^3 + 10 \times 16^2 + 6 \times 16^1 + 15 \times 16^0 + 11 \times 16^{-1} + 3 \times 16^{-2} = 6767.6992_{(10)}$

# Representación Posicional de los Números

- Conversiones de **cualquier base a decimal**:

- $valor\_total = \sum_i valor\_cifra_i \times base^i$

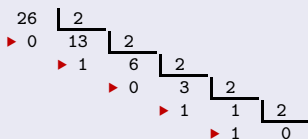
- $11010.0011_{(2)} = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$   
 $= 26.1875_{(10)}$

- Conversiones de **decimal a cualquier base**:

- **Parte Entera**: se **divide** la parte entera y cocientes entre la **base** hasta cero. Los restos de las divisiones forman las cifras del número en **orden inverso**.

- **Parte fraccionaria**: se **multiplica** la parte fraccionaria por la **base** tantas veces como cifras decimales queramos obtener. Las partes enteras resultados de las anteriores multiplicaciones forman las cifras decimales en **orden directo**.

- $26.1875_{(10)} = 11010.0011_{(2)}$



0.1875	0.3750	0.7500	0.5000
× 2	× 2	× 2	× 2
-----	-----	-----	-----
0.3750	0.7500	1.5000	1.0000
▼	▼	▼	▼
0	0	1	1

# Representación Posicional de los Números

## Sistemas de numeración:

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

## Conversión rápida entre bases

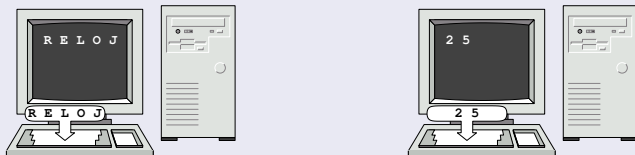
- Es posible realizar **conversión rápida** entre bases cuando una base es potencia de otra base ( $A = B^n$ ).
  - Binario  $\iff$  Hexadecimal ( $16 = 2^4$ )
  - Binario  $\iff$  Octal ( $8 = 2^3$ )
- Se realiza una **conversión directa** entre grupos de cifras de la longitud indicada por el exponente.
  - 4 cifras binarias se corresponden con 1 cifra hexadecimal y viceversa.
  - 3 cifras binarias se corresponden con 1 cifra octal y viceversa.
- Por ejemplo:
  - $010010111011111.1011101_{(2)} \iff 25DF.BA_{(16)}$
  - $10001101100.11010_{(2)} \iff 2154.64_{(8)}$

## Codificación y procesamiento de la información mediante números

- Hemos visto que el computador representa internamente la información mediante el **código binario**.
- Hemos visto que los sistemas de numeración posicionales permiten la representación de los números utilizando diversas bases:
  - Sistema decimal (base 10), sistema binario (base 2), hexadecimal (base 16), etc.
- Hemos visto que los números representados en una determinada base, también se pueden representar en otras bases, dando lugar a representaciones equivalentes.
- Por lo tanto, aunque el computador representa la información utilizando el código binario, nosotros consideraremos simplemente la **representación de la información mediante números**, sin importar la base que se utilice, ya que son representaciones equivalentes.

# Codigos de Entrada y Salida de Datos

- La representación de la **información** por medio de **caracteres** es uno de los medios más usuales para realizar la entrada y salida de datos (desde el teclado y hacia el monitor).
- **Caracteres:**
  - Letras Mayúsculas: **A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**
  - Letras Minúsculas: **a b c d e f g h i j k l m n o p q r s t u v w x y z**
  - Dígitos: **0 1 2 3 4 5 6 7 8 9**
  - Símbolos de puntuación: **, ; . : ? ! ( ) [ ] { }**
  - Otros símbolos: **= + - / \* > < \ @ # \$ % ^ & | \_ ~ ' "**
  - Caracteres de control: *tabulador, nueva-línea, retorno-de-carro, nueva-página, etc.*
  - Otros caracteres:
- Los caracteres se codifican **numéricamente** según una tabla de correspondencia: **ASCII** (7 bits), **ISO-8859** (8 bits), **Unicode** (32 bits).
  - A cada carácter se le asocia un número.
  - Para codificar **N** símbolos distintos se necesitan **x** bits, tal que:  $2^{x-1} < N \leq 2^x$ .
  - Unicode (32 bits) tiene diversas formas de codificación: UTF-8, UTF-16, UTF-32.



# Codigos de Entrada y Salida de Datos

- Tabla **ASCII** para la codificación de caracteres básicos (7 bits):

Cod	Car	Cod	Car	Cod	Car	Cod	Car	Cod	Car	Cod	Car
32	SP	48	0	64	@	80	P	96	'	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

- Los números del **0** hasta el **31** codifican caracteres de control: *retroceso, tabulación horizontal, nueva línea, tabulación vertical, nueva página, retorno de carro, escape, etc.*
- Los caracteres (ñ á é í ó ú ü Ñ Á É Í Ó Ú Û ÿ ¡) necesitan una codificación extendida para alfabetos internacionales (**Unicode**, UTF-8, UTF-16, UTF-32, etc.).



# Codigos de Entrada y Salida de Datos

- **Unicode:** es un estándar para la codificación y representación de texto que cubre la mayoría de los lenguajes internacionales, históricos y símbolos.
  - Utiliza la codificación ASCII para los caracteres básicos.
  - Utiliza codificaciones extendidas para lenguajes internacionales, históricos y símbolos.
    - Símbolos matemáticos, musicales, iconos, etc.



- Alfabetos europeos, griego, árabe, chino, cirílico, hebreo, fenicio, cuneiforme, etc.  
Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö × ø ù ú û ü ý þ ß µ ¶ ° - -  
Γ Δ Θ Λ Ξ Π Σ Φ Ψ Ω α β γ δ ε ε ζ η θ ϑ  
ι κ λ μ ν ξ π ρ ρ σ ς τ υ φ φ χ ψ ω
- La versión 11.0 de Unicode contiene un repertorio de 137439 caracteres.
- Unicode puede utilizar diferentes codificaciones (32 bits):
  - UTF-8 (Unix/Linux, MacOSX, Web): codificaciones con 1, 2, 3 y 4 bytes.
  - UTF-16 (Windows, Java): codificaciones con 2 y 4 bytes.
  - UTF-32: todas las codificaciones con 4 bytes.