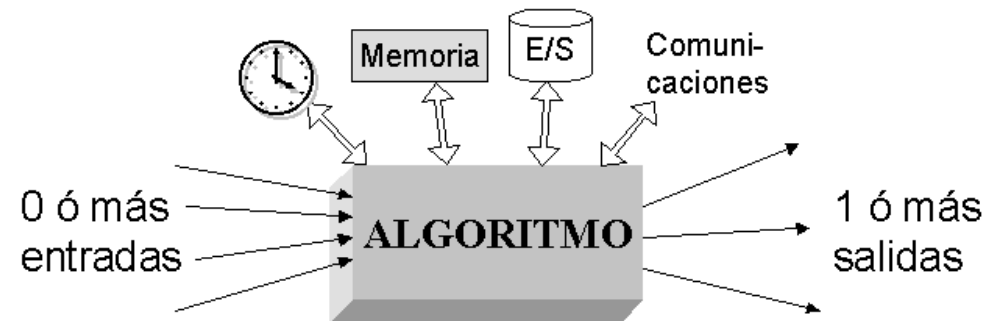




# Complejidad Computacional

## ¿Que es un algoritmo?

- Es un conjunto ordenado de instrucciones bien definidas, no ambiguas y finitas que permite resolver un determinado problema computacional. Su ejecución requiere ciertos recursos.

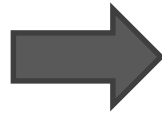


- Corolario: Un problema computacional puede ser resuelto por diversos (infinitos) algoritmos.

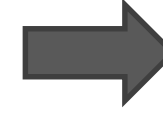
# ¿Qué algoritmo es el mejor?



Tenemos muchos  
algoritmos que dan  
solución a un problema



¿Cómo sabemos  
cual más eficiente?



Necesitamos una escala  
de medida  
común para  
poder  
compararlos

# La eficiencia de un algoritmo



Un algoritmo es eficiente cuantos menos recursos consume.

- ¿Cuáles son esos recursos que se consumen?
  - Tiempo de ejecución
  - Espacio (memoria)
  - Cantidad de procesadores (en el caso de algoritmos paralelos)
  - Utilización de la red de comunicaciones (para algoritmos paralelos)
- Otros criterios de interés para la Ingeniería de Software y que no hay que descuidar son:
  - Claridad de la solución
  - Facilidad de codificación
- Dependiendo de cómo balanceemos la importancia de cada uno de los criterios, podremos decir que un algoritmo es mejor que otro.  
En estas ppt no nos vamos a ocupar de ese balance, sino de la eficiencia en cuanto al tiempo.

# Tiempo de ejecución de un algoritmo

---

El tiempo de ejecución de un programa depende de factores como

- Los datos de entrada
- La calidad del código generado por el compilador utilizado para crear el programa
- La naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa y
- La complejidad de tiempo del algoritmo base del programa

El hecho que el tiempo de ejecución dependa de la entrada indica que el tiempo de ejecución de un programa debe definirse como una función de la entrada

# Complejidad de Tiempo de ejecución de un algoritmo

---

La complejidad computacional representa la cantidad de **recursos** temporales que necesita un algoritmo para resolver un problema.

- Permite determinar la **eficiencia** de dicho algoritmo.
- Los criterios que se emplean para evaluar la complejidad computacional **no proporcionan medidas absolutas**, sino medidas relativas al tamaño del problema.
- Es decir, el tiempo de Ejecución de un programa se mide en función de  $n$ , lo que designaremos como  $T(n)$

# Complejidad de Tiempo de ejecución de un algoritmo



Distintas entradas, aunque tengan el mismo tamaño, pueden hacer que el algoritmo se comporte de maneras muy diferentes, y por lo tanto, tomar distinto tiempo, y/o requerir distinta cantidad memoria.

- Por lo tanto se deben considerar tres casos para el análisis de un mismo algoritmo:
  - **caso peor** ->  $T_{\max}(n)$ : Representa la complejidad temporal en el peor de los casos.
  - **caso mejor** ->  $T_{\min}(n)$ : Representa la complejidad en el mejor de los casos posibles.
  - **caso medio** ->  $T_{\text{prom}}(n)$ : Expresa la complejidad temporal en el caso promedio. Para su cálculo se suponen que todas las entradas son equiprobables.
- Ejemplo para un algoritmo de ordenamiento de un array de  $N$  elementos sería:
  - peor caso -> en orden inverso
  - Mejor caso -> ordenado
  - Caso medio -> elementos en orden aleatorio

# Complejidad computacional



El análisis de la complejidad temporal de un algoritmo se puede hacer de forma:

- **Empírica o experimental**

Medir el tiempo de ejecución para una determinada entrada en una computadora concreta.


- Usando un cronómetro, o analizando el consumo de recursos de la computadora (tiempo de CPU).
- Desventaja: Hay que codificar el algoritmo.

- **Teórica**

Medida teórica del comportamiento de un algoritmo



# Análisis Teórico

- 
- Número de Operaciones Básicas
  - Análisis asintótico

# Número de Operaciones Básicas

---

Una manera objetiva de determinar que tan "bueno" es un algoritmo es por medio del número de **operaciones básicas** que este debe realizar para resolver un problema cuyo tamaño tiene una entrada ( $n$ ).

## ¿Qué es una operación básica?

- Operaciones básicas serán aquellas que el procesador realiza en tiempo acotado por una constante (que no depende del tamaño de la entrada).
- Consideraremos operaciones básicas a las operaciones aritméticas elementales, comparaciones lógicas, transferencias de control, asignaciones a variables de tipos básicos, etc.
- En el etc. está el problema. Por eso es importante definir bien el modelo de cómputo, y cuáles son las operaciones elementales.

## Ejemplo: Algoritmo de Ordenamiento por Selección

```
private static void ordenarPorSeleccion(int [] datos){  
    int n = datos.Length;  
    for(int i=0; i<(n-1); i++){  
        int menor = i;  
        for(int j=i+1; j<n; j++){  
            if(datos[j] < datos[menor])  
                menor = j;  
            int swap = datos[i];  
            datos[i] = datos[menor];  
            datos[menor] = swap;  
        }  
    }  
}
```

# Ejemplo: Algoritmo de Ordenamiento por Selección



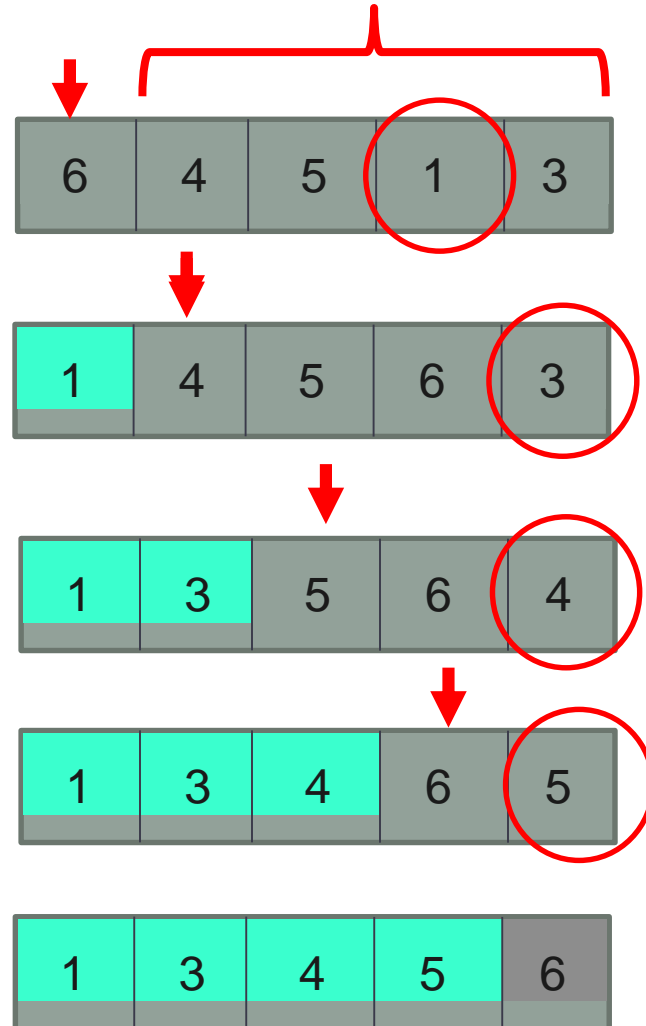
Este algoritmo realiza los siguientes pasos:

- Iteración 1  
Selecciona la posición del elemento más chico del vector y lo intercambia con el de la primera. El elemento de la primera posición está en su posición definitiva
- Iteración 2  
Se considera el vector a partir de la segunda posición, se selecciona la posición del elemento menor y se lo intercambia con el segundo elemento.
- Iteración siguientes  
Se continua este proceso hasta que en la iteración  $n-1$  se ordenan los 2 últimos elementos

# Ejemplo: Calculo de la cantidad de operaciones básicas

Tamaño de la entrada =  $n$

La cantidad de iteraciones es  $n-1$  ya que en la última iteración se ordenan simultáneamente los dos elementos



| Iteración | Cantidad de comparaciones |
|-----------|---------------------------|
| 1         | $4=n-1$                   |
| 2         | 3                         |
| 3         | 2                         |
| $4=n-1$   | 1                         |

Cantidad de comparaciones =  $\sum_{i=1}^n i = \frac{n(n-1)}{2}$

# Análisis asintótico



- El orden de la función  $T(n)$  expresa el comportamiento dominante para los datos de gran tamaño.
- Medidas del comportamiento asintótico de la complejidad:
  - $O$  ( $O$  grande) cota superior.
  - $\Omega$  (omega) cota inferior. Permite representar el límite inferior del tiempo de ejecución de un algoritmo.
  - $\Theta$  (theta) orden exacto de la función.

## La notación "Big Oh"

---

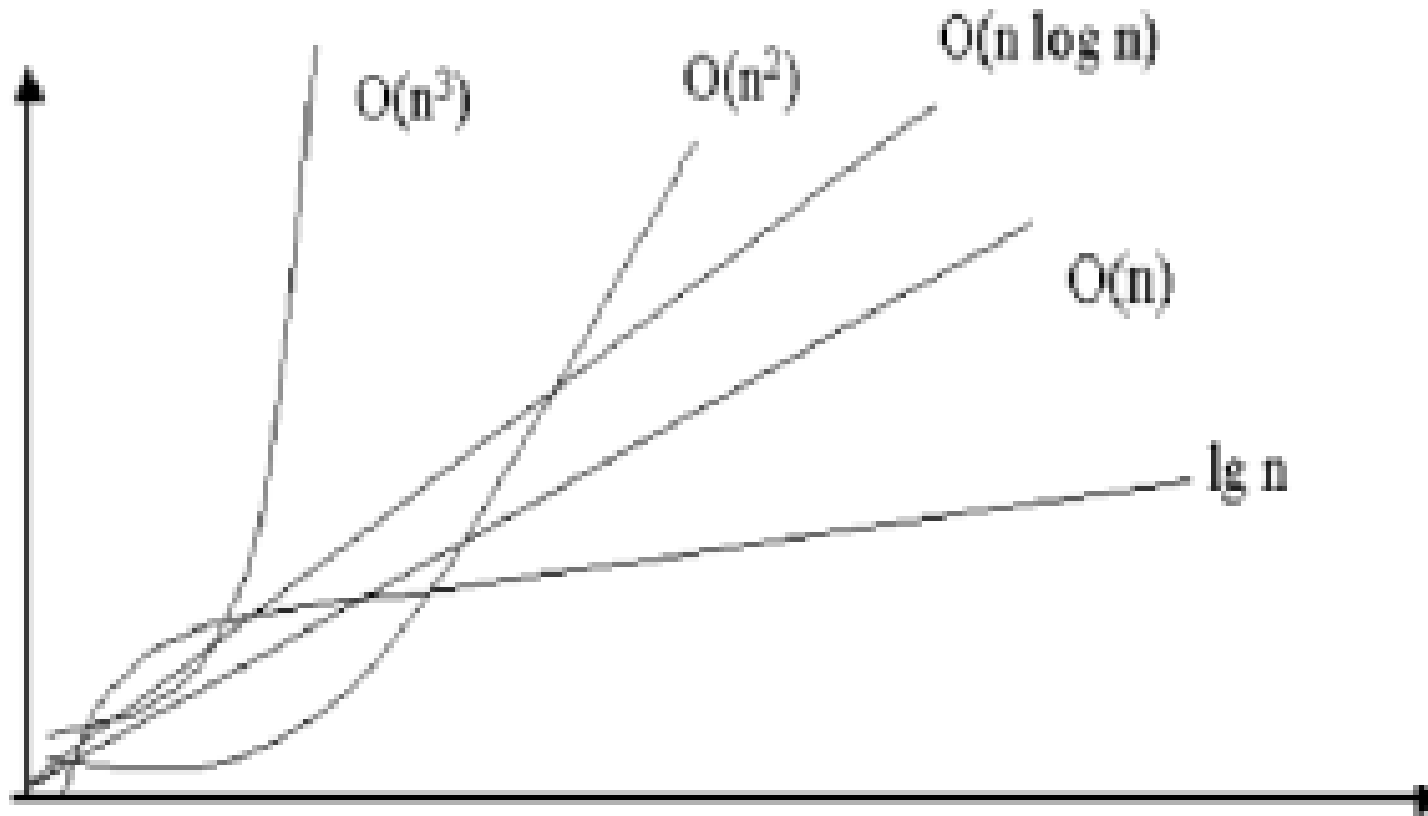
Se dice que  $T(n)$  es  $O(f(n))$  si existen constantes positivas  $c$  y  $n_0$  tales que

$$T(n) \leq c f(n) \text{ cuando } n \geq n_0$$

Cuando el tiempo de ejecución de un programa es  $O(f(n))$ , se dice que tiene velocidad de crecimiento  $f(n)$ .

Esto significa que  $T(n)$  no crece más deprisa que  $f(n)$ . De esta forma acotamos superiormente el comportamiento asintótico de la función salvo constantes.

# Medidas de comportamiento asintótico. Ejemplos





# Funciones de complejidad temporal comunes

Ordenar de menor a mayor según su crecimiento

- $O(1)$  Complejidad constante. Es independiente de los datos de entrada. Cuando las instrucciones se ejecutan una vez
- $O(\lg n)$  Complejidad logarítmica. (p.e., búsqueda binaria). Todos los logaritmos, sea cual sea su base, son del mismo orden, por lo que se representan en cualquier base.  $O(\log_a n) = O(\log_b n)$
- $O(n)$  Complejidad lineal. Suele aparecer en bucles simples cuando la complejidad de las operaciones internas es constante.
- $O(n \log n)$ : Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación.
- $O(n^2)$ : Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si  $n$  se duplica, el tiempo de ejecución aumenta cuatro veces.
- $O(n^3)$ : Complejidad cúbica. Suele darse en bucles con triple anidación. Si  $n$  se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de  $n$  empieza a crecer dramáticamente.
- $O(n^a)$ : Complejidad polinómica ( $a > 3$ ).
- $O(2^n)$ : Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas.
- $O(n!)$  factorial Se prueban todas las combinaciones posibles.

# Los tamaños máximos de entrada

La velocidad de crecimiento es quien determina el tamaño de problema que se puede resolver en una computadora

La tabla muestra los tamaños máximos de la entrada, para que el algoritmo de una complejidad computacional dada se pueda ejecutar en un tiempo razonable.

| C.C.                  | N máximo (aprox.) | Orden      |
|-----------------------|-------------------|------------|
| $\log_2(N)$           | $2^{100.000.000}$ | $2^{10^8}$ |
| $\sqrt{N}$            | $10^{16}$         | $10^{16}$  |
| N                     | 100.000.000       | $10^8$     |
| $N \cdot \log_2(N)$   | 4.500.000         | $10^6$     |
| $N^2$                 | 10.000            | $10^4$     |
| $N^2 \cdot \log_2(N)$ | 3.000             | $10^3$     |
| $N^3$                 | 500               | $10^2$     |
| $N^4$                 | 100               | $10^2$     |
| $2^N$                 | 27                | $10^1$     |
| $3^N$                 | 17                | $10^1$     |
| $N!$                  | 11                | $10^1$     |

## Regla de la suma

---

Si  $T1(n)$  y  $T2(n)$  son las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa, y se acotan de forma que se tiene:

$$T1(n) \text{ es } O(f1(n)) \text{ y } T2(n) \text{ es } O(f2(n))$$

Se puede decir que:

$$T1(n) + T2(n) = O(\max(f1(n), f2(n)))$$

Ejemplo: El tiempo de ejecución para una instrucción condicional de tipo *IF-THEN-ELSE* resulta de evaluar la condición, más el máximo valor del conjunto de instrucciones de las ramas *THEN* y *ELSE*.

$$T(\text{IF-THEN-ELSE}) = T(\text{condición}) + \max(T(\text{rama THEN}), T(\text{rama ELSE}))$$

Aplicando la regla de la suma:

$$O(T(\text{IF-THEN-ELSE})) = \max(O(T(\text{condición})), \max(O(T(\text{rama THEN})), O(T(\text{rama ELSE})))$$

# Regla del producto

---

Si  $T1(n)$  y  $T2(n)$  son las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa, y se acotan de forma que se tiene:

$$T1(n) \text{ es } O(f1(n)) \text{ y } T2(n) \text{ es } O(f2(n))$$

Se puede decir que:

$$T1(n) T2(n) = O(f1(n) f2(n))$$

Ejemplos:

- El tiempo de ejecución de un bucle *FOR* es el producto del número de iteraciones por la complejidad de las instrucciones del cuerpo del mismo bucle.
- Para los ciclos del tipo *WHILE-DO* y *DO-WHILE* se sigue la regla anterior, pero se considera la evaluación del número de iteraciones para el peor caso posible.

## Ventajas del enfoque teórico:

---

- El análisis se puede hacer a priori, aún antes de escribir una línea de código
- Vale para todas las instancias del problema
- Es independiente del lenguaje de programación utilizado para codificarlo
- Es independiente de la máquina en la que se ejecuta
- Es independiente de la pericia del programador

# BIBLIOGRAFIA

---

- ALFRED V. AHO, JOHN E. HOPCROFT, JEFREY D. ULLMAN. Estructura de datos y algoritmos. México, DF : Addison-Wesley. Iberoamericana: Sistemas Técnicos de Edición, 1988. ISBN 968-6048-19-7
- CEBALLOS SIERRA, FCO JAVIER, Curso de Programación en C++. Madrid, Ed. Ra-Ma
- JOYANES AGUILAR, Luis. Programación en C++ algoritmos, Estructuras de Datos y Objetos. España, Ed. McGraw-Hill. 2000.
- JOYANES AGUILAR, Luis. Fundamentos de Programación, Algoritmos y Estructuras de datos. España, Seg. Edición McGraw-Hill. 1996