Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 3

Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

- **Criterios de valoración.**
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

¿Cómo se puede saber qué algoritmo es el mejor?

Existen diferentes criterios:

- Legibilidad
- Usabilidad/interfaz
- Facilidad de mantenimiento
- Velocidad de ejecución
- Necesidad de memoria
- Tiempo de desarrollo
- Elegancia
- etc.

¿Cómo se puede saber qué algoritmo es el mejor?

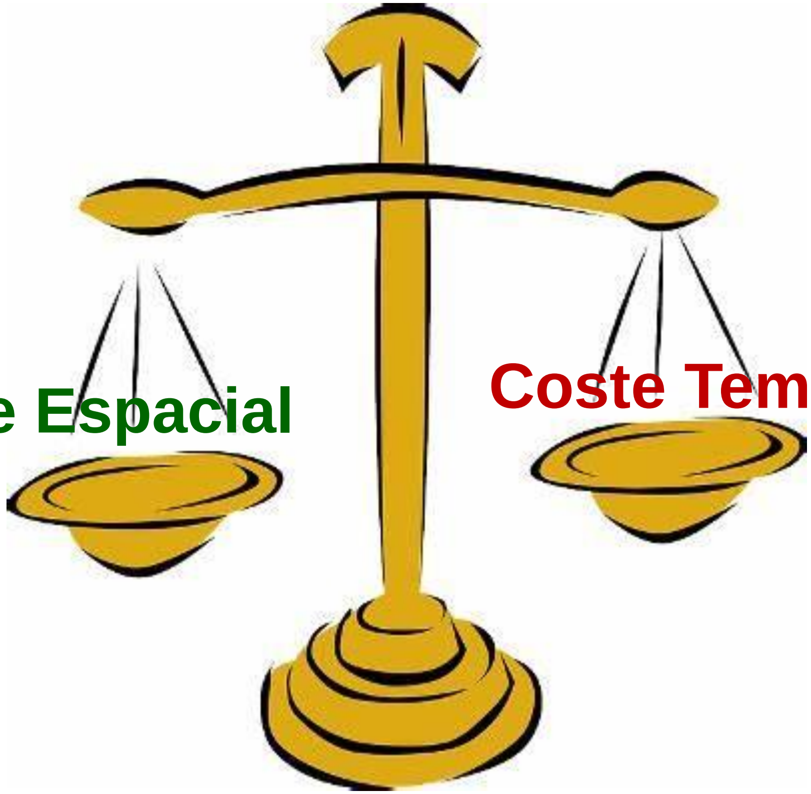
Existen diferentes criterios:

- Legibilidad
- Usabilidad/interfaz
- Facilidad de mantenimiento
- Velocidad de ejecución
- **Necesidad de memoria**
- **Tiempo de desarrollo**
- Elegancia
- etc.

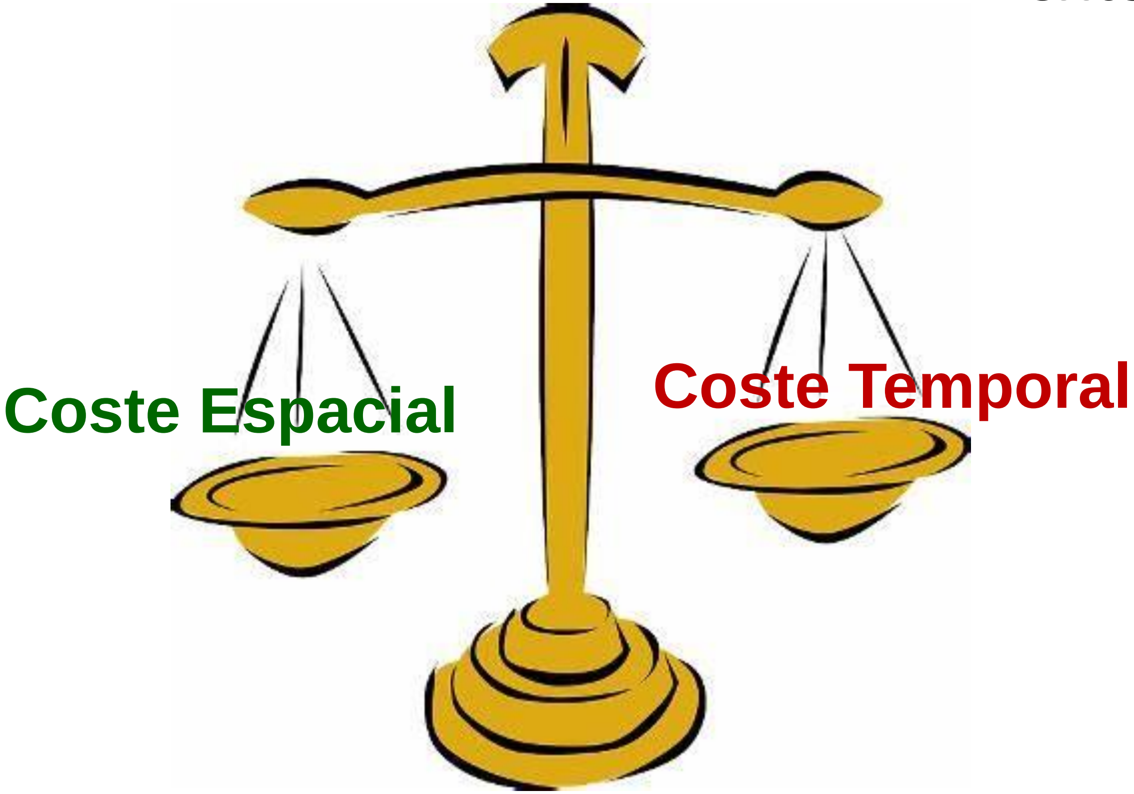
Criterios de eficiencia:

- **Espacio**
- **Tiempo**

Criterios de valoración



Complejidad Computacional



Principios básicos:

- Elegir las **estructuras de datos** adecuadas
- Utilizar **algoritmos** eficientes para manejar las estructuras de datos elegidas.

Principios del Análisis de Algoritmos

- Independiente del **ordenador.**
- Independiente del **lenguaje de programación.**
- Independiente de **detalles de la implementación** (tipos de datos, sentencias, etc.).

Hay que analizar **algoritmos** y **no programas**

- Criterios de valoración.
- **Coste Temporal.**
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

Calcular 10^2

Complejidad Temporal

Forma 1: Producto

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia = 10 * 10;

    cout << "\n\n\t10^2 = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

Forma 2: Suma

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia{0};

    for (int i{0}; i < 10; i++)
    {
        potencia += 10;
    }

    cout << "\n\n\t10^2 = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

Forma 3: Incrementos

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia{0};

    for (int i{0}; i < 10; i++)
    {
        for (int j{0}; j < 10; j++)
        {
            potencia ++;
        }
    }

    cout << "\n\n\t10^2 = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

El tiempo de cada programa **depende** de la duración de las **sentencias elementales utilizadas** (dependen del ordenador).

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones
forma1.cpp	1			1	
forma2.cpp		10	10	12	11
forma3.cpp			210	12	121

Calcular n^2

Complejidad Temporal

Forma 1: Producto

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia, n;

    cout << "\n\n\tIndique un numero entero: ";
    cin >> n;
    potencia = n * n;
    cout << "\n\n\tPotencia = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

Forma 2: Suma

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia{0}, n;

    cout << "\n\n\tIndique un numero entero: ";
    cin >> n;
    for (int i{0}; i < n; i++) {
        potencia += n;
    }
    cout << "\n\n\tPotencia = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

Forma 3: Incrementos

```cpp
#include <iostream>

using namespace std;

int main()
{
    int potencia{0}, n;

    cout << "\n\n\tIndique un numero entero: ";
    cin >> n;
    for (int i{0}; i < n; i++)
    {
        for (int j{0}; j < n; j++) {
            potencia ++;
        }
    }
    cout << "\n\n\tPotencia = " << potencia;

    cout << "\n\n\t";
    return 0;
}
```

UNIVERSIDAD NEBRIJA

El **tiempo** de alguno de los programas **depende** del **valor de *n*:**

- Cuanto mayor es ***n***, más cuesta resolver el problema.
- El tamaño del problema es ***n***.
- Hay que determinar el coste del algoritmo en función de ***n***.
- Por tanto, se puede representar gráficamente el tamaño temporal de los algoritmos en función del tamaño de ***n***.

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparaciones
forma1.cpp	1			1	
forma2.cpp		n	n	n + 2	n + 1
forma3.cpp			$2n^2 + n$	n + 2	$n^2 + 2n + 1$

- Criterios de valoración.
- Coste Temporal.
- **Coste asintótico.**
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

El **coste asintótico** indica el coste del un algoritmo en **función del tamaño del problema** para valores grandes ($n \rightarrow \infty$).

Los estudios asintóticos son **independientes** del coste de cada operación que se realiza en el algoritmo.

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- **Cálculo del tamaño del problema.**
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

Se llama **paso (*step*)** al segmento de código cuyo **tiempo de proceso no depende del tamaño del problema** considerado y que **está acotado** por alguna **constante**.

Se consideran ***pasos***:

- Operaciones aritméticas y lógicas,
- Comparaciones entre escalares,
- Acceso a: variables escalares, elementos de vectores o arrays,
- Lectura/escritura de un valor escalar, etc.

Coste Computacional Temporal de un programa: número de pasos del programa expresado en función del tamaño del problema (*n*).

Calcular n^2

Forma 1: Producto

```
#include <iostream>

using namespace std;

int main()
{
    int potencia, n;

    cout << "\n\n\tIndique un numero entero: ";      <--  1 paso
    cin >> n;                                         <--  1 paso
    potencia = n * n;                                 <--  1 paso
    cout << "\n\n\tPotencia = " << potencia;          <--  1 paso

    cout << "\n\n\t";                                 <--  1 paso
    return 0;                                         <--  1 paso
}
```

El coste (nº de pasos) es 6.

Calcular n^2

Forma 2: Suma

```
#include <iostream>

using namespace std;

int main()
{
    int potencia{0}, num;                               <--  1 paso
    cout << "\n\n\tIndique un numero entero: ";         <--  1 paso
    cin >> num;                                         <--  1 paso
    for (int i{0}; i < num; i++) {                      <--  2n+2 pasos
        potencia += num;                                <--  2 pasos (n veces)
    }
    cout << "\n\n\tPotencia = " << potencia;            <--  1 paso

    cout << "\n\n\t";                                   <--  1 paso
    return 0;                                           <--  1 paso
}
```

El coste (nº de pasos) es 4n + 8.

Calcular n^2

Forma 3: Incrementos

```
#include <iostream>

using namespace std;

int main()
{
    int potencia{0}, num;                           <--  1 paso
    cout << "\n\n\tIndique un numero entero: ";     <--  1 paso
    cin >> num;                                     <--  1 paso
    for (int i{0}; i < num; i++)                    <--  2n+2 pasos
    {
        for (int j{0}; j < num; j++) {              <--  2n+2 pasos (n veces del bucle anterior)
            potencia ++;                            <--  1 paso (n * n veces)
        }
    }
    cout << "\n\n\tPotencia = " << potencia;        <--  1 paso

    cout << "\n\n\t";                               <--  1 paso
    return 0;                                       <--  1 paso
}
```

El coste (nº de pasos) es $3n^2 + 4n + 8$.

- El **valor concreto de cada término** de las expresiones **no importa**.
- El concepto de paso hace que el **valor de las constantes no sea significativo**. Luego, es igual 4n + 8 que c_2n + c_1.
- Cualquier número de pasos que no dependa del tamaño del problema, ***n***, se considera una **cantidad constante** de pasos.

Programa	forma1.cpp	forma2.cpp	forma3.cpp
Coste en tiempo	6	4n + 8	$3n^2 + 4n + 8$

Programa	forma1.cpp	forma2.cpp	forma3.cpp
Coste en tiempo	c_0	$c_2n + c_1$	$c_5n^2 + c_4n + c_3$

- Cuando se define una entrada concreta de tamaño ***n***, hablamos de la **instancia de un problema**.
- Por ejemplo, buscar un determinado elemento en un vector. El tamaño del problema es la **longitud del vector**.
- Por tanto, el **coste temporal de un algoritmo depende**:
  - **Tamaño del problema.**
  - **Instancias del problema**.

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- **Análisis del Mejor y Peor caso.**
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

Se trata de estudiar dos situaciones extremas del algoritmo para un valor tamaño del problema (*n*) dado:

- El ***mejor caso***
- El ***peor caso***

Ejemplo: la búsqueda de un elemento en un vector:

- Mejor caso: si el elemento que se busca es el primero.
- Peor caso: si el elemento que se busca no existe.

El Mejor y el Peor de los casos

- **Mejor caso** para un tamaño ***n***: el elemento es el primero.

```
bool pertenece (vector<int> const &V, int n)  {
    for (int elem:V)  {                      <-- 1 paso
        if (elem == n) return true;          <-- 2 pasos
    }

    return false;
}
```

Su coste es constante: $\mathbf{c_0}$ **pasos**

- **Peor caso** para un tamaño ***n***: el elemento no existe.

```
bool pertenece (vector<int> const &V, int n)  {
    for (int elem:V)  {                      <-- n pasos
        if (elem == n) return true;          <-- n pasos
    }

    return false;                            <-- 1 paso
}
```

Su coste es lineal: $\mathbf{c_2 n + c_1}$ **pasos**

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- **Notaciones asintóticas.**
- Cálculo de la eficiencia.
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

Orden de *f(n)*

Un algoritmo tiene un tiempo de ejecución de orden $f(n)$, para una función dada f, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un **tiempo acotado superiormente por** $c \cdot f(n)$, donde n es el **tamaño del problema** considerado.

Notación *O (cota superior)*

Se dice que una función $T(n)$ es $O(f(n))$ si existen constantes n_0 y c tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$:

$$T(n) \text{ es } O(f(n)) \Leftrightarrow$$

$$\exists c \in R,\ \exists n_0 \in N, \text{ tal que } \forall n \geq n_0 \in N,\ T(n) \leq cf(n)$$

Ejemplos de Notación *O*

$T(n) = 3n + 1$

$T(n)$ es $O(n)$, ya que $T(n) <= 4n$ para $n>=1$.

$T(n) = (n+1)^2$

$T(n)$ es $O(n^2)$, ya que $T(n) <= 2n^2$ para $n>=3$

$T(n) = 12n^2 + 7n + 3$.

$T(n)$ es $O(n^2)$ pero no es $O(n)$.

$T(n) = 4n^3 + 3n^2$

$T(n)$ es $O(n^3)$ pero no es $O(n^2)$.

$T(n) = 5^n$

$T(n)$ es $O(5^n)$ pero no es $O(4^n)$.

Omega de $f(n)$

Un algoritmo tiene un tiempo de ejecución de Omega de $f(n)$, para una función dada f, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un **tiempo acotado inferiormente por $c \cdot f\ (n)$**, donde n es el **tamaño del problema** considerado.

Notación Ω *(cota inferior)*

Se dice que una función $T(n)$ es $\Omega(f(n))$ si existen constantes n_0 y c tales que $T(n) \geq cf(n)$ para todo $n \geq n_0$:

$$T(n) \text{ es } \Omega(f(n)) \Leftrightarrow$$

$$\exists c \in R,\ \exists n_0 \in N, \text{ tal que } \forall n \geq n_0 \in N,\ T(n) \geq cf(n)$$

Ejemplos de Notación Ω

$T(n) = 12n^2 + 7n + 3$

$T(n)$ es $\Omega(n^2)$, ya que $T(n) \geq 12n^2$ para $n>=0$.

$T(n) = 5n^2+3^n$

$T(n)$ es $\Omega(3^n)$

Zeta de *f(n)*

Un algoritmo tiene un tiempo de ejecución de Zeta de $f(n)$, para una función dada f, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un **tiempo acotado superior e inferiormente por** $c \cdot f\ (n)$, donde n es el **tamaño del problema** considerado.

Notación Θ *(orden exacto)*

Se dice que una función $T(n)$ es $\Theta(f(n))$ si es $O(f(n))$ *y* es $\Omega(f(n))$, al mismo tiempo.

Ejemplos de Notación Θ

$T(n) = 4n + 1$ es $\Theta(n)$

$T(n) \leq 5n$ es $O(n)$ para $n \geq 1$

$T(n) \geq 3n$ es $\Omega(n)$ para $n \geq 0$.

$T(n) = n^2$ si n es par y n si n es impar

$T(n)$ no es Θ ya que es $O(n^2)$ y $\Omega(n)$.

Jerarquía de Cotas en las notaciones O y Ω

$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$

$\Omega(n^n) \subset \Omega(n!) \subset \Omega(2^n) \subset \Omega(n^3) \subset \Omega(n^2) \subset \Omega(n \log n) \subset \Omega(n) \subset \Omega(\sqrt{n}) \subset \Omega(\log n) \subset \Omega(1)$

Por ejemplo, $3n+2$ es de $O(n)$ y también es $O(n^2)$ y $O(n^n)$, pero se indica siempre la **cota que más se ajusta a la función**.

Tiempos calculados suponiendo 1μs por operación elemental

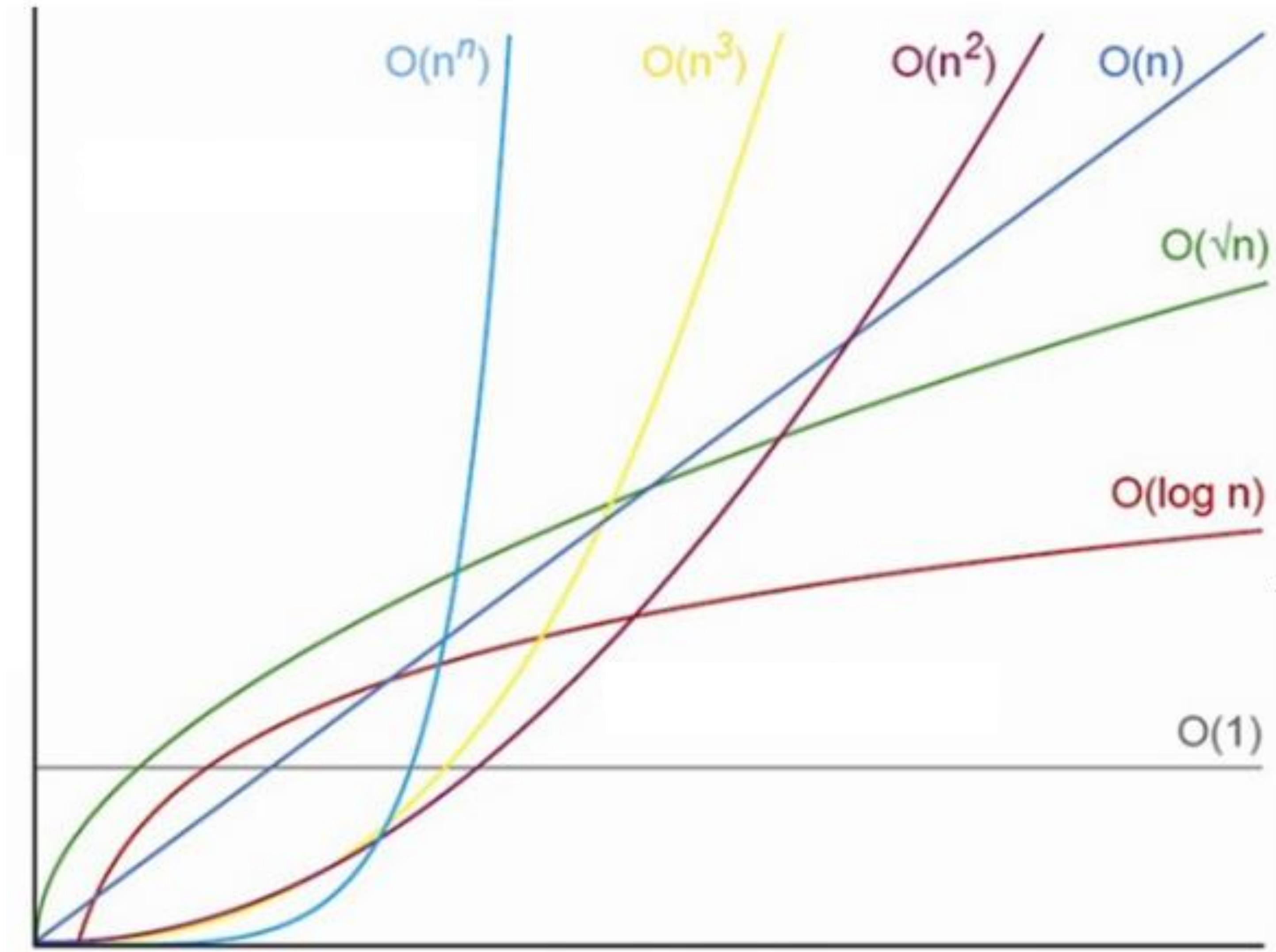
N	$O(\log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
10	3 μs	10 μs	30 μs	0.1 ms	1 ms	4 s
25	5 μs	25 μs	0.1 ms	0.6 ms	33 s	10^{11} años
50	6 μs	50 μs	0.3 ms	2.5 ms	36 años	...
100	7 μs	100 μs	0.7 ms	10 ms	10^{17} años	...
1000	10 μs	1 ms	10 ms	1 s	...	...
10000	13 μs	10 ms	0.1 s	100 s	...	...
100000	17 μs	100 ms	1.7 s	3 horas	...	...
1000000	20 μs	1 s	20 s	12 días	...	...

UNIVERSIDAD NEBRIJA

Clasificación de las funciones

Sublineales	Constantes	O(1)
	Logarítmicas	$O(\log n)$
	Raíces	$O(\sqrt{n})$
Lineales	Lineales	$O(n)$
Superlineales	Lineal-logarítmicas	$O(n \log n)$
	Polinómicas	$O(n^2)$ $O(n^3)$
	Exponenciales	$O(2^n)$ $O(n!)$ $O(n^n)$

Representación de las funciones O



Orden lineal – O(n): Tiempo de ejecución proporcional al tamaño del problema (n). Ejemplo, calcular el máximo de n números.

Orden cuadrático – O(n^2): Ejemplo, enumerar todas las parejas posibles de elementos de un conjunto.

Ordenes O(log n) y O(n log n): Muchos algoritmos recursivos. Ejemplo, búsqueda binaria (O(log n)) y mergesort, heapsort, etc. (O(n log n).

Orden exponencial – O(2^n): Problemas de tipo combinatorio, con factoriales. Ejemplo, enumerar todo los subconjuntos de un conjunto.

Simplificación de Cotas y sus propiedades

- **Producto de una función por una constante:**
  - *Si* $\boldsymbol{T(n) \in O(f(n))}$ *entonces* $\boldsymbol{cT(n) \in O(f(n))}$
  - *Si* $\boldsymbol{T(n) \in \Omega(f(n))}$ *entonces* $\boldsymbol{cT(n) \in \Omega(f(n))}$
- **Suma de funciones:**
  - *Si* $\boldsymbol{T_1(n) \in O(f_1(n)), T_2(n) \in O(f_2(n))}$ *entonces*

    $$\boldsymbol{T_1(n) + T_2(n) \in O(max(f1(n), f2(n)))}$$
  - *Si* $\boldsymbol{T_1(n) \in \Omega(f_1(n)), T_2(n) \in \Omega(f_2(n))}$ *entonces*

    $$\boldsymbol{T_1(n) + T_2(n) \in \Omega(max(f_1(n), f_2(n)))}$$
- **Producto de funciones:**
  - *Si* $\boldsymbol{T_1(n) \in O(f_1(n)), T_2(n) \in O(f_2(n))}$ *entonces* $\boldsymbol{T_1(n)\ T_2(n) \in O(f1(n)f2(n))}$
  - *Si* $\boldsymbol{T_1(n) \in \Omega(f_1(n)), T_2(n) \in \Omega(f_2(n))}$ *entonces* $\boldsymbol{T_1(n)\ T_2(n) \in \Omega(f_1(n)f_2(n))}$
- Una consecuencia de estas propiedades es que cualquier polinomio de grado ***k*** es $\boldsymbol{O(n^k)}$ y $\boldsymbol{\Omega(n^k)}$, es decir, es $\boldsymbol{\Theta(n^k)}$.

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- **Cálculo de la eficiencia.**
- Coste Espacial.

UNIVERSIDAD NEBRIJA

ÍNDICE

Reglas de cálculo de la eficiencia

1. Sentencias simples.
2. Bloques de sentencias.
3. Sentencias condicionales.
4. Bucles.
5. Llamadas a funciones.
6. Funciones recursivas.

Sentencias simples:

Se consideran las asignaciones, operaciones aritméticas y lógicas, acceso a los elementos de arrays, vectores o estructuras, lectura, escritura, break, continue, etc., excepto las llamadas a funciones o métodos, tienen un **coste temporal de ejecución constante**, es decir: **O(1)**.

$$O(\text{sentencia simple}) = O(1)$$

Bloques de sentencias:

El **coste temporal de ejecución es la suma de los costes de cada sentencia**, es decir:

$$O(bloque\ sentencias) = \sum_i O(sentencia\ i)$$

Orden de eficiencia = Máximo de los órdenes de eficiencia de cada una de las sentencias del bloque.

Sentencias condicionales:

El **coste temporal de ejecución es el coste de la rama de mayor coste**, es decir, si la sentencia es un ***if*** con dos ramas (***if-else***), si el bloque ***if*** es ***O(f(n))*** y el bloque ***else*** es ***O(g(n))***, la sentencia condicional será ***O(max{f(n), g(n)})*****:**

$$O(condicional) = max(O(f(n)), O(g(n)))$$

$$\Omega(condicional) = min(\Omega(f(n)), \Omega(g(n)))$$

Si hay más ramas (switch) y cada rama $\boldsymbol{i}$ tiene un coste $\boldsymbol{f_i(n)}$ se puede generalizar:

$$O(condicional) = max(O(f_i(n)))$$

Bucles:

Los componentes generales de un bucle son:

- Inicialización del bucle: $O(1)$
- Comprobación de la condición del bucle (**k veces**): $O(k)$
- Incremento de la variable del bucle (**k veces**): $O(k)$
- Cuerpo del bucle (**k veces**) y suponiendo que las sentencias que lo forman tiene un coste $O(f(n))$: $O(kf(n))$
- Finalización del bucle: $O(1)$

Bucles:

Su coste será:

$$O(bucle) = O(1) + O(n_{max}) \, [+O(n_{max})] + O(n_{max} f(n)) \, [+O(1)] = O(n_{max} f(n))$$

$$\Omega(bucle) = \Omega(1) + \Omega(n_{min}) \, [+\Omega(n_{min})] + \Omega(n_{min} f(n)) \, [+\Omega(1)] = \Omega(n_{min} f(n))$$

Siendo n_{max} y n_{min} el número de iteraciones del bucle en el peor y mejor caso, respectivamente. Los *[]* significa que es opcional, no siempre aparecerá ese elemento en el bucle.

El problema está en determinar el número de veces que se ejecuta el cuerpo del bucle.

Llamadas a funciones:

El **coste temporal de ejecución de una función o métodos viene determinado por el bloque de sentencias que tiene**. Si el coste de sentencias de la función es $f(n)$:

$$O(función/método) = O(f(n))$$

Funciones recursivas:

El **coste temporal de ejecución es una función recursiva también es recursiva:**

$$T(n) = T(n-1) + f(n)$$

Funciones recursivas:

Ejemplo:

```
long factorial (int numero)
{
    if (numero > 1) return (numero * factorial (numero -1));
    else return (1);
}
```

O(1)

$$T(n) = 2+T(n-1) = 2+(2+T(n-2)) = 2+2+T(n-2) =$$
$$2+2+(2+T(n-3)) = (2*3) + T(n-3) = \ldots =$$
$$(2*i)+T(n-i) = \ldots = 2*(n-1)+T(n-(n-1)) =$$
$$2*(n-1)+T(1) = 2*(n-1)+2 = 2*n$$

Por tanto, ***T(n)* es *O(n)*, de orden lineal**.

Funciones recursivas:

Ejemplo:

```
long Fibonacci (int num)
{   if ((num == 1)||(num == 2)) return (1);
    else return (Fibonacci(num-1) + Fibonacci(num-2));
}
```

$$\boldsymbol{T(n)} = 2+T(n-1)+T(n-2) \leq 2+2T(n-1) =$$

$$2+2(2+T(n-2)+T(n-3)) \leq 2+2(2+2T(n-2)) = 2+4+4T(n-2) =$$

$$2+4+4(2+T(n-3)+T(n-4)) \leq 2+4+4(2+2T(n-3)) =$$

$$2+4+8+8T(n-3) \leq \ldots \leq 2^1+2^2+2^3+\ldots+2^{n-1}+2^{n-1}T(n-(n-1)) =$$

$$\ldots = \left(\sum_{i=1}^{n-1} 2^i\right) + 2^{n-1}\ T(1) = = \left(\sum_{i=1}^{n-1} 2^i\right) + 2^{n-1}\ 2 =$$

$$\left(\frac{2^{(n-1)+1}-2}{2-1}\right) + 2^n = 2^n - 2 + 2^n = \mathbf{2^{n+1} - 2}$$

Por tanto, $\boldsymbol{T(n)}$ **es** $\boldsymbol{O(2^n)}$**, de orden exponencial**.

Funciones recursivas:

Ejemplo:

```
long FibonacciI (int num)
{   long f1{1}, f2{1}, fn{f1};
    for (int i{3}; i<=num; i++){
        fn = f1 + f2;
        f1 = f2;
        f2 = fn;
    }
    return fn;
}
```

$$T(n) = 3+2+2(n\text{-}3)+4(n\text{-}3)+1=6+6(n+3)=6n+24$$

Por tanto, ***T(n)*** **es** ***O(n)*****, de orden lineal**.

- Criterios de valoración.
- Coste Temporal.
- Coste asintótico.
- Cálculo del tamaño del problema.
- Análisis del Mejor y Peor caso.
- Notaciones asintóticas.
- Cálculo de la eficiencia.
- **Coste Espacial.**

UNIVERSIDAD NEBRIJA

ÍNDICE

- La Complejidad Espacial es estudiar la eficiencia de los algoritmos respecto a su consumo de memoria.
- El estudio asintótico debe mostrar la cantidad de memoria utilizada por un programa en relación al tamaño del problema, ***n***.
- El tamaño del problema es ***n***.
- Hay que determinar el coste del algoritmo en función de ***n***.
- Por tanto, se puede representar gráficamente el tamaño espacial de los algoritmos en función del tamaño de ***n***.

El análisis del coste espacial pone el foco en el concepto de celda de memoria que será lo que se considere "paso".

- Lo importante es saber el espacio que ocupan las variables en función al tamaño del problema, ***n***.
- Tampoco importa el número de variables utilizadas en el algoritmo porque su coste es constante.
- La complejidad espacial también maneja cotas superiores e inferiores sobre el consumo de memoria.

El análisis del coste espacial para algoritmos no recursivos:

- Si sólo se utilizan variables escalares, el coste es constante.
- Si se manejan estructuras (array, vectores, etc.) cuyo tamaño es proporcional al tamaño del problema, n, el coste espacial es $\boldsymbol{\Theta(n)}$.

El análisis del coste espacial para algoritmos recursivos deben considerar el espacio necesario para la pila de ejecución durante el proceso, así:

- Si sólo se utilizan variables escalares, el coste espacial puede ser $\Theta(n)$ si el proceso efectúa del orden de n llamadas recursivas para resolver un problema de tamaño del problema, n.

Progresiones aritméticas

$$a_{i+1} = a_i + d$$

$$\sum_{i=1}^{n} a_i = \frac{1}{2} n (a_1 + a_n)$$

$$\sum_{i=1}^{n} i = \frac{1}{2} n (n+1)$$

Progresiones geométricas

$$a_{i+1} = r\, a_i$$

$$\sum_{i=1}^{n} a_i = \frac{a_1 (r^{n+1} - 1)}{r - 1}$$

$$\sum_{i=1}^{n} b^i = \frac{b^{n+1} - b}{b - 1}$$

Fórmulas matemáticas útiles

Sumatorios

$$\sum_{i=1}^{n} a = na$$

$$\sum_{i=1}^{n} a f(i) = a \sum_{i=1}^{n} f(i)$$

$$\sum (a+b) = \sum a + \sum b$$

$$\sum_{i} \sum_{j} a_i b_j = \sum_{i} a_i \sum_{j} b_j$$

$$\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$$

$$\sum_{i=1}^{n} i^2 = \frac{1}{6} n(n+1)(2n+1)$$

$$\sum_{i=1}^{n} i^3 = \left[\frac{1}{2} n(n+1)\right]^2$$

$$\sum_{i=1}^{n} i(i+1) = \frac{1}{3} n(n+1)(n+2)$$

Fórmulas matemáticas útiles

Potencias

$$x^{y+z} = x^y \cdot x^z$$

$$x^{y-z} = x^y / x^z$$

$$x^{y \cdot z} = (x^y)^z = (x^z)^y$$

Logaritmos

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

$$\log_a(n\,m) = \log_a(n) + \log_a(m)$$

$$\log_a(n/m) = \log_a(n) - \log_a(m)$$

$$\log_a(n^p) = p \log_a(n)$$

$$n^{\log_a(m)} = m^{\log_a(n)}$$

1. Determinar el número de pasos y el coste temporal de la **suma** de n términos de la serie aritmética de razón d definida por: $a_{i+1} = a_i + d$

   Siendo a_1 y d dos valores dados.

2. Determinar el número de pasos y el coste temporal de la **búsqueda** de un determinado carácter, ***c***, en un array de caracteres, ***cadena***, de tamaño ***n***.
3. Repetir el cálculo si el array de caracteres está ordenado (orden alfabético).