Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 4

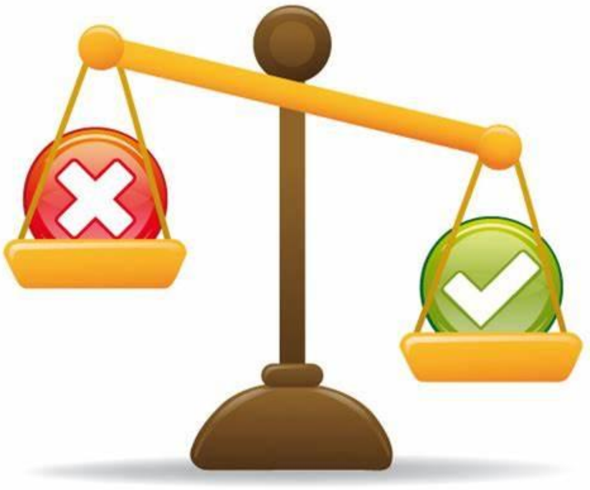
Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

Análisis del Coste Temporal

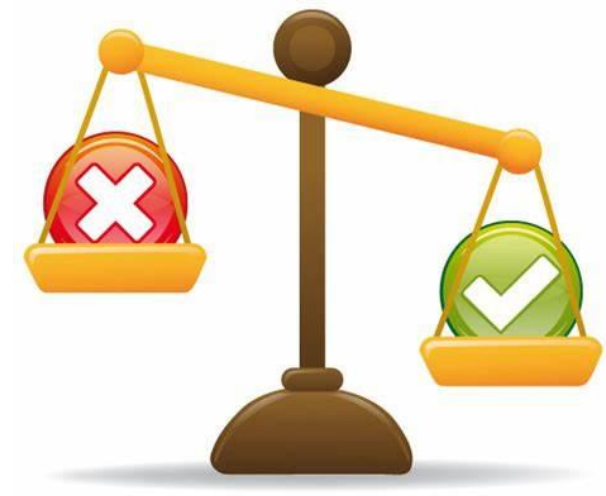
- Suma de una Serie Aritmética.
- Algoritmos de Búsqueda.
- Algoritmos de Ordenación.



UNIVERSIDAD NEBRIJA

ÍNDICE

Análisis del Coste Temporal



- **Suma de una Serie Aritmética.**
- Algoritmos de Búsqueda.
- Algoritmos de Ordenación.

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

Siendo $\boldsymbol{a_1}$ y $\boldsymbol{d}$ dos valores dados.

Suma de una Serie Aritmética

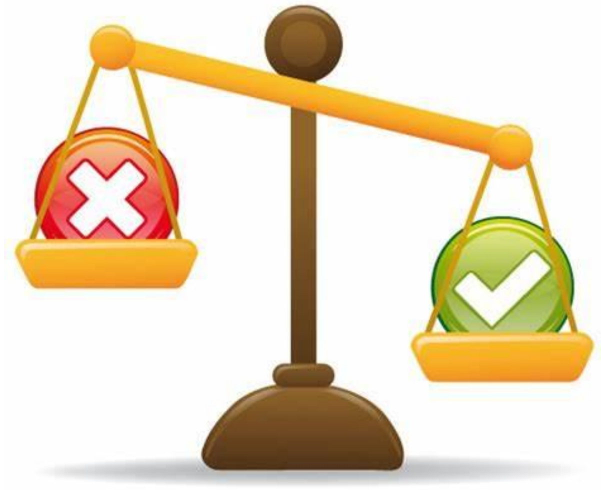
```
double sumaSerieAritmetica (double a1, double d, int n)
{
    double suma, an;

    an = a1;                                ← 1 paso
    suma = a1;                              ← 1 paso

    for (int i{2}; i <= n; i++)             ← 2(n-1) + 2 pasos
    {
        an += d;                            ← 1 paso (n-1) veces
        suma += an;                         ← 1 paso (n-1) veces
    }

    return suma;                            ← 1 paso
}
```

El coste (nº de pasos) es ***T(n)*** = 3 + 2n + 2(n-1) = **4n + 1**.

```
double sumaSerieAritmetica (double a1, double d, int n)
{
    double suma, an;

    an = a1;                                ← Θ(1)
    suma = a1;                              ← Θ(1)

    for (int i{2}; i <= n; i++)             ← Θ(1) (n-1) veces
    {                                                                  Θ(n)
        an += d;                            ← Θ(1)
        suma += an;                         ← Θ(1)    Θ(1)      Θ(n)
    }

    return suma;                            ← Θ(1)
}
```

Luego: *T(n)* es *Θ(n)* por ser ***O(n)*** *y* ***Ω(n)***, al mismo tiempo.

Análisis del Coste Temporal



- Suma de una Serie Aritmética.
- **Algoritmos de Búsqueda:**
  - **Búsqueda de un elemento de un Vector sin Ordenar.**
  - Búsqueda de un elemento de un Vector Ordenado.
  - Búsqueda Binaria.
- Algoritmos de Ordenación.

Búsqueda de un elemento de un Vector sin Ordenar

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***.

Búsqueda de un elemento de un Vector sin Ordenar

```
int buscarCaracterSinOrdenar (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) return i;
    }

    return -1;
}
```

El **coste depende de la instancia del problema**, luego hay que **analizar el mejor** y **peor caso.**

Búsqueda de un elemento de un Vector sin Ordenar

Análisis en el mejor caso: está en la primera posición.

```
int buscarCaracterSinOrdenar (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)          ← 2 pasos
    {
        if (l == f.at(i)) return i;                     ← 2 pasos
    }

    return -1;
}
```

El coste (nº de pasos) es ***T(n)*** = **4**

Búsqueda de un elemento de un Vector sin Ordenar

Análisis en el mejor caso: está en la primera posición.

```
int buscarCaracterSinOrdenar (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) return i;
    }

    return -1;
}
```

Ω(1)

Luego: *T(n)* es *Ω(1)*

Búsqueda de un elemento de un Vector sin Ordenar

Análisis en el peor caso: no está en el ***string***.

```
int buscarCaracterSinOrdenar (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)      ← 2n + 2 pasos
    {
        if (l == f.at(i)) return i;                 ← 1 paso (n veces)
    }

    return -1;                                      ← 1 paso
}
```

El coste (nº de pasos) es ***T(n)*** = **3n + 3**

Búsqueda de un elemento de un Vector sin Ordenar

Análisis en el peor caso: no está en el ***string***.

```
int buscarCaracterSinOrdenar (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) return i;
    }

    return -1;
}
```

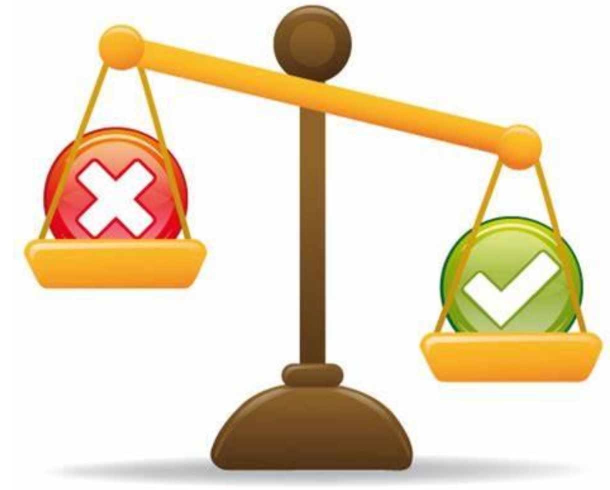
O(n)

O(n)

Luego: ***T(n)*** es ***O(n)***

UNIVERSIDAD NEBRIJA

Análisis del Coste Temporal



- Suma de una Serie Aritmética.
- **Algoritmos de Búsqueda:**
  - Búsqueda de un elemento de un Vector sin Ordenar.
  - **Búsqueda de un elemento de un Vector Ordenado.**
  - Búsqueda Binaria.
- Algoritmos de Ordenación.

UNIVERSIDAD NEBRIJA

ÍNDICE

Búsqueda de un elemento de un Vector Ordenado

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

Búsqueda de un elemento de un Vector Ordenado

```
int buscarCaracterOrdenado (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {
            return -1;
        }
    }

    return -1;
}
```

Búsqueda de un elemento de un Vector Ordenado

Análisis en el mejor caso: está en la primera posición.

```
int buscarCaracterOrdenado (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {
            return -1;
        }
    }

    return -1;
}
```

Ω(1)

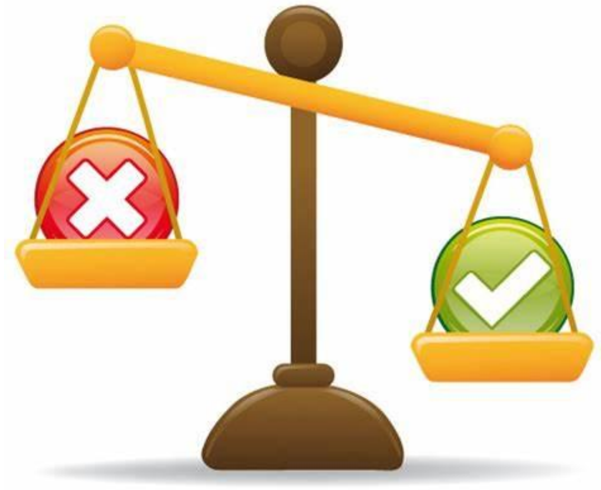
Luego: $T(n)$ es $\Omega(1)$

Búsqueda de un elemento de un Vector Ordenado

Análisis en el peor caso: no está en el ***string*** y es mayor que el mayor de los elementos.

```
int buscarCaracterOrdenado (const string & f, char l)
{
    for (unsigned int i{0}; i < f.size(); i++)
    {
        if (l == f.at(i)) {                          ← O(n)
            return i;
        }
        else if (l < f.at(i)) {                      ← O(n)      O(n)
            return -1;
        }
    }

    return -1;
}
```

Luego: $T(n)$ es $O(n)$



Análisis del Coste Temporal

- Suma de una Serie Aritmética.
- **Algoritmos de Búsqueda:**
  - Búsqueda de un elemento de un Vector sin Ordenar.
  - Búsqueda de un elemento de un Vector Ordenado.
  - **Búsqueda Binaria.**
- Algoritmos de Ordenación.

UNIVERSIDAD NEBRIJA

ÍNDICE

Búsqueda Binaria de un elemento de un Vector Ordenado

La búsqueda binaria consiste en buscar un elemento en un ***vector*** (o ***array***) donde los datos están en **orden ascendente** (por ejemplo).

La búsqueda se hace **dividiendo sucesivamente** el vector. Así, se **busca sobre la mitad** de los elementos del vector y eso **reduce a la mitad el número de comprobaciones** en cada iteración.

Algoritmo de búsqueda binaria:

Se compara el **dato a buscar** con el **elemento central del vector**:

- Si es el elemento buscado se **finaliza**.
- Si no, se sigue **buscando en la mitad del vector** que determine la relación entre el valor del elemento central y el buscado.

El algoritmo finaliza cuando se localiza el dato buscado en el vector o se termina el vector porque no existe.

Búsqueda Binaria de un elemento de un Vector Ordenado

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15>7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15>12

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15=15

Búsqueda Binaria de un elemento de un Vector Ordenado

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;

    final = f.size()-1;

    while (ppio <= final)
    {
        i = (ppio+final)/2;

        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;
        }
    }

    return -1;
}
```

Búsqueda Binaria de un elemento de un Vector Ordenado

Análisis en el mejor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;                       ← Ω(1)

    final = f.size()-1;                          ← Ω(1)

    while (ppio <= final)                        ← Ω(1)
    {
        i = (ppio+final)/2;                      ← Ω(1)

        if (l == f.at(i)) {
            return i;                            ← Ω(1)
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;
        }
    }

    return -1;
}
```

$\Omega(1)$

Luego: $T(n)$ es $\Omega(1)$

NEBRIJA UNIVERSIDAD

Búsqueda Binaria de un elemento de un Vector Ordenado

Análisis en el peor caso

Hay que determinar el número de veces que se hace el bucle:

- **Cada iteración** del bucle **reduce**, aproximadamente, a la **mitad el número de elementos donde se busca** (tamaño del vector).
- Después de **k iteraciones** el número de elementos sobre el que se busca será a lo sumo $\boldsymbol{n/2^k}$**.**
- La **última iteración** se produce cuando el **número de elementos es 1**, es decir: $1 = \boldsymbol{n/2^k}$

Análisis en el peor caso

Tomamos logaritmos para resolver: 1 = $\boldsymbol{n/2^k}$

$$\frac{n}{2^k} = 1 \Leftrightarrow log_2 \frac{n}{2^k} = log_2 1 = 0$$

$$0 = log_2 \frac{n}{2^k} = log_2 n - log_2 2^k = log_2 n - k$$

$$k = log_2 n$$

Luego, el **número de iteraciones (*k*) está acotado superiormente por $\boldsymbol{log_2 n}$.**

Búsqueda Binaria de un elemento de un Vector Ordenado

Análisis en el peor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;                                  ← O(1)

    final = f.size()-1;                                     ← O(1)

    while (ppio <= final)                                   ← O(log n)
    {
        i = (ppio+final)/2;                                 ← O(1)

        if (l == f.at(i)) {                                 ← O(1)
            return i;
        }
        else if (l < f.at(i)) {   // se busca en la mitad izquierda
            final = i-1;                                    ← O(1)
        }
        else {                    // se busca en la mitad derecha
            ppio = i+1;                                     ← O(1)
        }
    }

    return -1;                                              ← O(1)
}
```

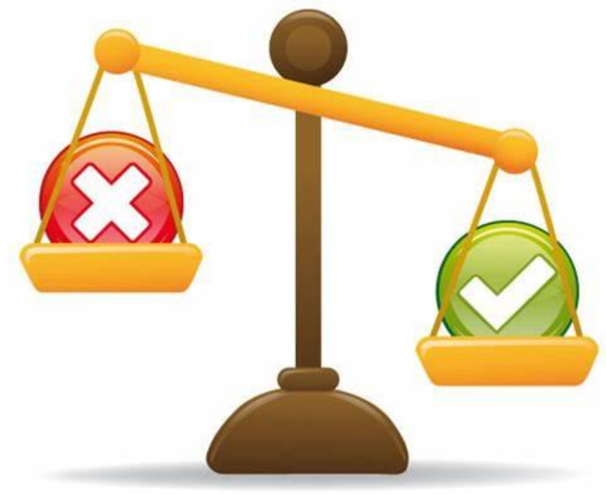
O(1) (if / else if / else)

O(log n) (while)

O(log n) (función completa)

Luego: ***T(n)*** es ***O(log n)***

Análisis del Coste Temporal



- Suma de una Serie Aritmética.
- Algoritmos de Búsqueda.
- **Algoritmos de Ordenación:**
  - **Ordenación por selección.**
  - Ordenación por intercambio. Método de la burbuja.
  - Ordenación por inserción.
  - Ordenación rápida.

UNIVERSIDAD NEBRIJA

ÍNDICE

Se basa en **realizar varias pasadas** e ir **localizando el elemento** que **hay que reubicar** en su **posición correcta**.

Ejemplo, se quiere ordenar de menor a mayor y se **localiza el menor** de los valores: **25**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
62	31	**25**	87	48	52	77

Se **intercambia** con valor de la **primera posición**:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	62	87	48	52	77

Se recorre la sublista de elementos desde la siguiente posición al elemento ubicado correctamente, buscando el **valor menor: 31**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	**31**	62	87	48	52	77

Se **intercambia** con valor de la **segunda posición**:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	**31**	62	87	48	52	77

Ordenación por selección

Se **procede de igual manera** hasta llegar al final de la lista.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	62	87	**48**	52	77

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	48	87	62	**52**	77

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	48	52	**62**	87	77

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	48	52	62	87	**77**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
25	31	48	52	62	77	87

Ordenación por selección

```
void ordenacionSeleccion (string & f)
{
    int  posmenor;
    char menor;

    for (unsigned int i{0}; i < f.size(); i++)
    {
        posmenor = i;
        menor = f.at(posmenor);
        for (unsigned int j{i+1}; j < f.size(); j++)
        {
            if (f.at(j) < menor)
            {
                posmenor = j;
                menor = f.at(posmenor);
            }
        }
        f.at(posmenor) = f.at(i);
        f.at(i) = menor;
    }
}
```