Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 5

Curso 2023-2024

Marta N. Gómez

UNIVERSIDAD NEBRIJA

Análisis del Coste Temporal

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

Análisis del Coste Temporal

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

Suma de una Serie Aritmética

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 numéricos dados.

Suma de una Serie Aritmética

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

    an = a1;
    suma = a1;

    for (int i{2}; i <= n; i++)
    {
        an += d;
        suma += an;
    }

    return suma;
}
```

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 $T(n) = 3 + 2n + 2(n-1) = \mathbf{4n + 1}$.

Suma de una Serie Aritmética

```
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
    {
        an += d;                      ← Θ(1)
        suma += an;                   ← Θ(1)
    }

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

Θ(1) (bloque de las dos líneas dentro del for)

Θ(n) (bloque del for)

Θ(n) (todo el cuerpo)

Luego: $T(n)$ es $\Theta(n)$ por ser $O(n)$ y $\Omega(n)$, al mismo tiempo. No hay mejor caso ni peor caso.

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 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 (*string*), ***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;
}
```

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++)
    {
        if (l == f.at(i)) return i;
    }

    return -1;
}
```

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;
}
```

← 2 pasos

← 2 pasos

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;
}
```

$\Omega(1)$ $\Omega(1)$

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

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++)
    {
        if (l == f.at(i)) return i;
    }

    return -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++)
    {
        if (l == f.at(i)) return i;
    }

    return -1;
}
```

← **2n + 2 pasos**

← **1 paso (n veces)**

← **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(1) — `if (l == f.at(i)) return i;`

O(n) — `for (unsigned int i{0}; i < f.size(); i++) { ... }`

O(1) — `return -1;`

O(n) — todo el bloque

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 de un elemento de un Vector Ordenado

Determinar el **número de pasos** y el **coste temporal** de la **búsqueda** de un determinado carácter, ***c***, considerando 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;
}
```

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

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;
}
```

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;
}
```

← 2 pasos

← 2 pasos

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;
}
```

$\Omega(1)$

$\Omega(1)$

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

Búsqueda de un elemento de un Vector Ordenado

Análisis en el peor caso: no está en el ***string*** o 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)) {
            return i;
        }
        else if (l < f.at(i)) {
            return -1;
        }
    }

    return -1;
}
```

Búsqueda de un elemento de un Vector Ordenado

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

```
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;
}
```

← 2n + 2 pasos

← 1 paso (n veces)

← 1 paso (n veces)

← 1 paso

Búsqueda de un elemento de un Vector Ordenado

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

```
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;
}
```

← *O(1)*

← *O(1)*

O(n)

← *O(1)*

O(n)

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

Búsqueda de un elemento de un Vector Ordenado

Análisis en el peor caso: es mayor que el mayor de los elementos.

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

    return -1;
}
```

Búsqueda de un elemento de un Vector Ordenado

Análisis en el peor caso: 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(1)
            return i;
        }
        else if (l < f.at(i)) {                ← O(1)
            return -1;                         ← O(1)
        }
    }

    return -1;
}
```

O(n)

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