Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 6

Curso 2023-2024

Marta N. Gómez

Análisis del Coste Temporal

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

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, por ejemplo, están en **orden ascendente**.

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.

Búsqueda Binaria de un elemento de un Vector Ordenado

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 $\Omega(1)$

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: $n/2^k$.
- La **última iteración** se produce cuando el **número de elementos es 1**, es decir: $1 = n/2^k$

Análisis en el peor caso

Tomamos <u>logaritmos</u> 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)) {
            return i;                               ← O(1)
        }
        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)

O(1)

O(log n)

O(log n)

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.

Ordenación por selección

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;
    }
}
```

Análisis en el mejor caso

```
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;
    }
}
```

← *Ω(1)*

← *Ω(1)*

Ω(n)

← *Ω(1)*

Ω(n²)

Luego: *T(n)* es $\Omega(n^2)$

Análisis en el peor caso

```
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;
    }
}
```

← *O(1)*

← *O(1)*

O(n)

← *O(1)*

O(n^2)

Luego: ***T(n)*** es ***O(n^2)***

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.

Ordenación por intercambio - Método de la burbuja

Se basa en el **intercambio sistemático de elementos no ordenados** hasta lograr que la lista esté ordenada.

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

62	31	25	87	48	52	77

Se **comparan los dos primeros** y se **intercambian porque no están ordenados**:

62	31	25	87	48	52	77

31	62	25	87	48	52	77

Ordenación por intercambio- Método de la burbuja

Procedemos igual con el 2do. y 3ro.. **Repetimos el proceso hasta completar el vector** (1era. vuelta):

31	62	25	87	48	52	77

31	25	62	87	48	52	77

31	25	62	87	48	52	77

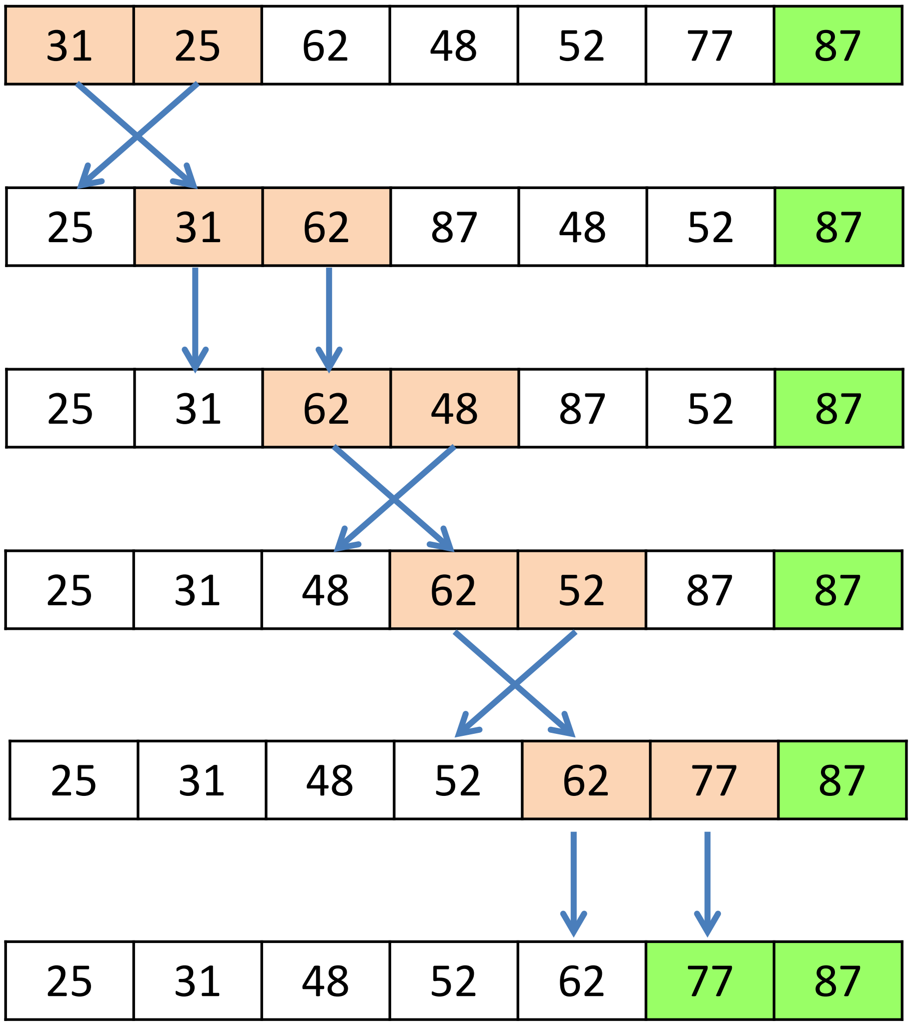
31	25	62	48	87	52	77

31	25	62	48	52	87	77

31	25	62	48	52	77	87

Ordenación por intercambio- Método de la burbuja

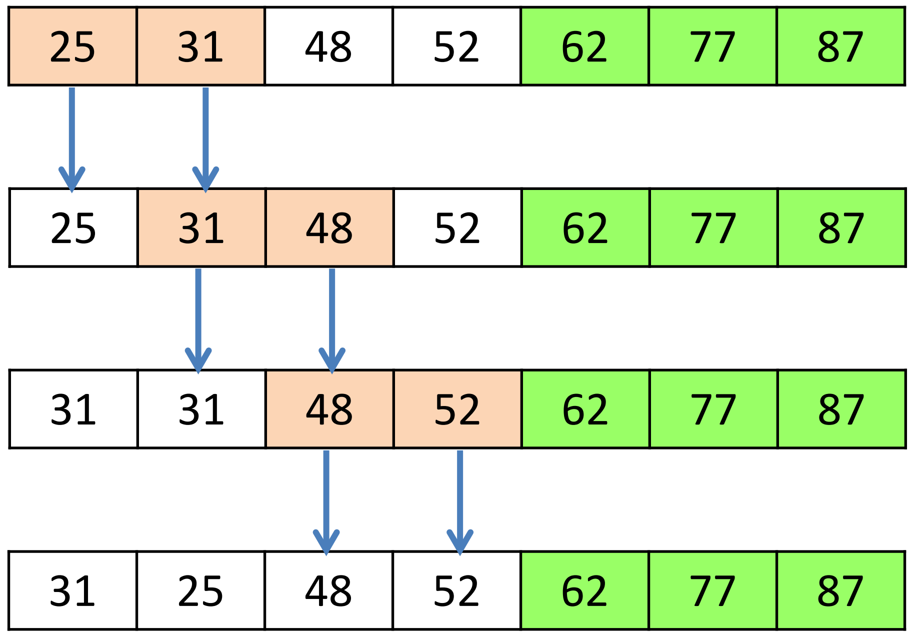
Empezamos la **2da. vuelta dejando fuera el último elemento** que está en su **posición correcta**:



UNIVERSIDAD NEBRIJA

Ordenación por intercambio- Método de la burbuja

Procedemos hasta el final:



Ordenación por intercambio- Método de la burbuja

```
void metodoBurbuja (string & f)
{
    char menor;

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

Ordenación por intercambio- Método de la burbuja

Análisis en el mejor caso

```
void metodoBurbuja (string & f)
{
    char menor;

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

$\leftarrow \Omega(1)$

$\Omega(n)$

$\Omega(n^2)$

Luego: $T(n)$ es $\Omega(n^2)$

Análisis en el peor caso

```
void metodoBurbuja (string & f)
{
    char menor;

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

← $O(1)$

$O(n)$

$O(n^2)$

Luego: $T(n)$ es $O(n^2)$

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.

Método preferido de los jugadores de cartas.

Se basa en insertar un nuevo elemento en una lista que ya está ordenada manteniendo el orden.

62	31	25	87	48	52	77

31	**62**	25	87	48	52	77

25	**31**	**62**	87	48	52	77

25	**31**	**62**	**87**	48	52	77

25	**31**	**48**	**62**	**87**	52	77

25	**31**	**48**	**52**	**62**	**87**	77

25	**31**	**48**	**52**	**62**	**77**	**87**

Ordenación por inserción

- Se va tomando un elemento de la **parte no ordenada** para **colocarlo en su lugar en la parte ordenada**.
- **El primer elemento se considera ordenado** (la lista inicial consta de un elemento).
- A continuación, se procede a **insertar el segundo elemento en la posición correcta** (delante o detrás de primero) dependiendo de que sea menor o mayor.
- Se **repite esta operación sucesivamente** de tal modo que se va colocando cada elemento en la posición correcta.
- El proceso se **repetirá n-1 veces**.

Ordenación por inserción

```
void ordenarInsercion(vector<int> & v)  {
     int temp;
     unsigned int i;

     for (unsigned int e=1; e < v.size(); e++){
          temp=v.at(e);
          i=0;
          while (v.at(i)<temp) {
               i++;
          }
          if (i < e)
          {
             for (unsigned int k=e ; k>i; k--)  {
                  v.at(k) = v.at(k-1);
             }
             v.at(i) = temp;
          }
     }
}
```

UNIVERSIDAD NEBRIJA

Análisis en el mejor caso

```
void ordenarInsercion(vector<int> & v)  {
    int temp;
    unsigned int i;

    for (unsigned int e=1; e < v.size(); e++){
        temp=v.at(e);                            Ω(1)
        i=0;
        while (v.at(i)<temp) {
            i++;                                 Ω(1)
        }                                                        Ω(n)
        if (i < e)
        {
            for (unsigned int k=e ; k>i; k--)  {
                v.at(k) = v.at(k-1);                      Ω(1)
            }
            v.at(i) = temp;
        }
    }
}
```

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

Análisis en el peor caso


```
void ordenarInsercion(vector<int> & v)  {
    int temp;
    unsigned int i;

    for (unsigned int e=1; e < v.size(); e++){
        temp=v.at(e);
        i=0;
        while (v.at(i)<temp) {
            i++;
        }
        if (i < e)
        {
            for (unsigned int k=e ; k>i; k--)  {
                v.at(k) = v.at(k-1);
            }
            v.at(i) = temp;
        }
    }
}
```

← $O(1)$

$O(n)$

$O(n)$

$O(n)$

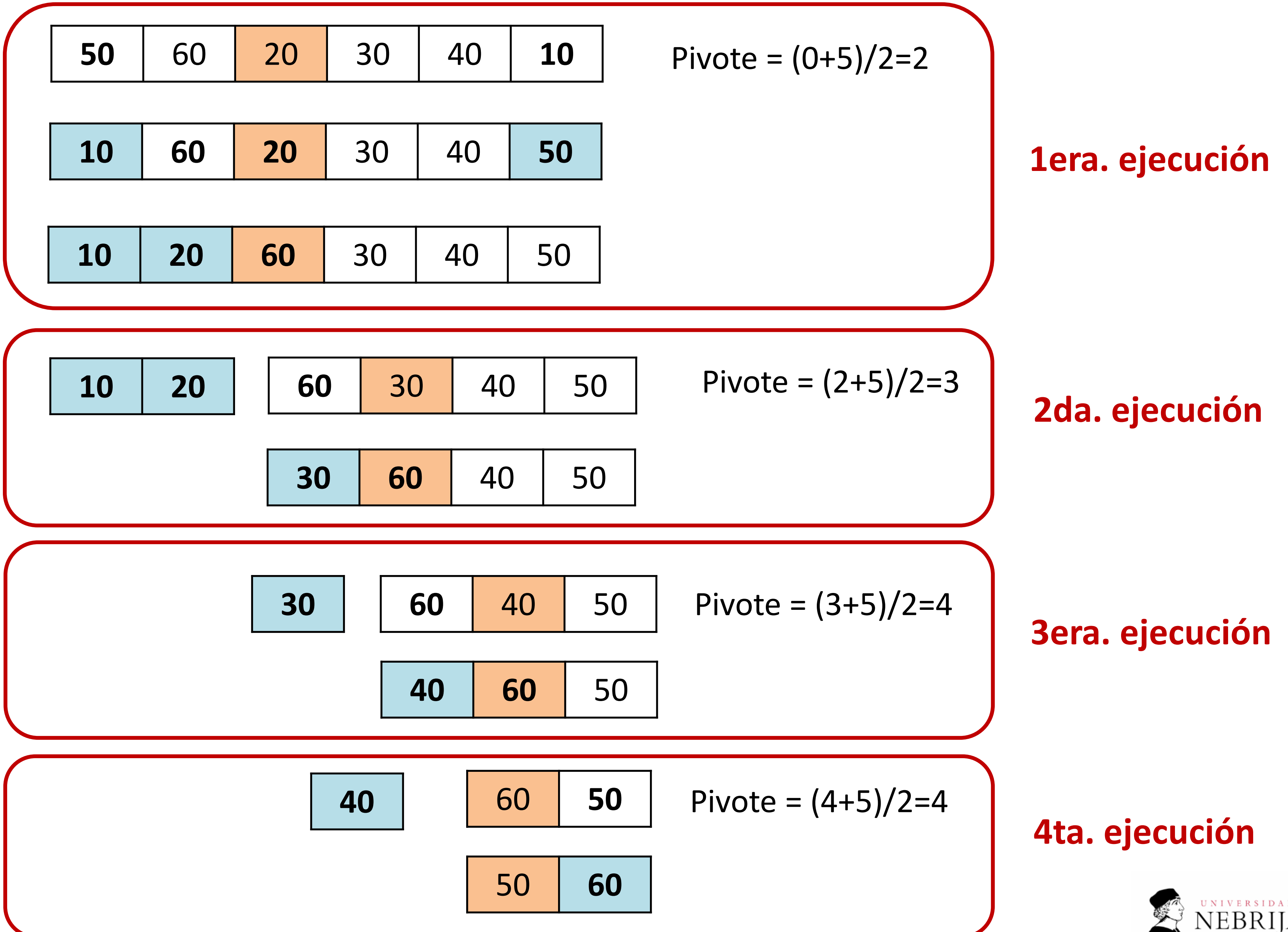
Luego: $T(n)$ es $O(n^2)$

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 (quicksort).**

- El algoritmo consiste en **dividir el vector** que se desea ordenar en **dos bloques**. En el primero se sitúan todos los elementos que son **menores** que un valor que se toma como referencia (**pivote**) y en segundo bloque el resto.
- Este procedimiento se repite dividiendo a su vez cada uno de estos bloques y repitiendo la operación anteriormente descrita.
- La condición de parada se da cuando el bloque que se desea ordenar está formado por un **único elemento** (bloque ordenado).
- El resultado se obtiene de la combinación de todos los resultados parciales.
- El esquema seguido por este algoritmo es el de "***divide y venceras***".

Ordenación rápida (Quicksort)



Ordenación rápida (Quicksort)

```
void ordenQuickSort(vector <int> & v, int izq, int der)
{
    int i{izq}, d{der}, pivote,aux;

    pivote = v.at((i+d)/2);

    while (i < d)
    {
       while (v.at(i) < pivote) { i++;  }

       while (pivote < v.at(d)) { d--;  }

       if (i <= d)  // intercambio de elementos
       {   aux = v.at(i);
           v.at(i) = v.at(d);
           v.at(d) = aux;
           i++;  d--; // se ajustan las posiciones
       }
    }

    if (izq < d) {  ordenQuickSort(v, izq, d);   }
    if (i < der) {  ordenQuickSort(v, i, der);   }
}
```

Análisis del mejor caso: Se produce cuando el pivote divide al vector en **dos partes iguales** y el orden de los elementos es aleatorio:

$$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n - 1}$$

T(n/2) es el coste de ordenar una de las mitades y ***n*** el número de comparaciones realizadas.

Si desarrollamos la recurrencia se obtiene:

$$\boldsymbol{T(n) = n\,logn + n - 1}$$

Que es de orden ***T(n) ∈ Ω(n logn)***

Análisis del mejor caso: Supongamos que el tamaño del vector es una potencia de 2: $n = 2^k \rightarrow \log n = k$

Recorrido	Comparaciones
1º	(n-1) comparaciones * 1 vector ≈ n 2 vectores de tamaño n/2
2º	(n/2) comparaciones * 2 vector = n 4 vectores de tamaño n/4
3º	(n/4) comparaciones * 4 vector = n 8 vectores de tamaño n/8
…	…
k-ésimo	($n/2^{k-1}$) comparaciones * 2^{k-1} vector = n 2^k vectores de tamaño $n/2^k$

$$n + n + \cdots + n = kn = n \log n$$

Luego es de orden $T(n) \in \Omega(n \log n)$

Análisis del peor caso: Si se elige como pivote el **primer elemento del vector** (elemento menor) y además se considera que el vector está **ordenado** entonces, el bucle ***while*** se ejecutará en total:

$$(n-1)+(n-2)+(n-3)+\cdots+1$$

Cada término de esa suma proviene de cada una de las sucesivas **ordenaciones recursivas**. Este sumatorio da lugar a la siguiente expresión:

$$(n-1)+(n-2)+(n-3)+\cdots+1=$$

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

Luego es de orden cuadrático $T(n) \in O(n^2)$