Estructura de Datos y Algoritmos 1

Teórico #6:

Algoritmos de ordenación

Sumario

1. Definición del problema de ordenación
2. Métodos simples o de orden $O(n^2)$
   - Bubble-Sort
   - Insertion-Sort
   - Selection-Sort
3. Métodos complejos o de orden $O(n \log n)$
   - Merge-Sort
   - Quick-Sort
4. Otros métodos

Introducción

- La ordenación es una de las operaciones más comunes que se realizan sobre una colección de datos.
- Es el proceso de reorganizar un conjunto dado de objetos en una secuencia determinada.
- Uno de los propósitos principales del ordenamiento de datos es facilitar luego la búsqueda sobre ellos.
- En problemas prácticos es vital. Es necesario para esto, tener un conocimiento previo de los datos a ordenar, herramienta a usar, lenguaje de programación, cuántas veces hay que aplicar el algoritmo y en qué momento, entre otros aspectos.

Definición del problema de ordenación

Ordenar significa mover los datos para que se logre una secuencia tal, que queden ordenados de forma ascendente o descendente. Para poder realizar el ordenamiento es necesario que exista un criterio de ordenación de dos elementos cualesquiera.

EJEMPLO

Para ordenar un conjunto de personas según su nombre se definiría la relación $\leq$ de modo que dos personas cualesquiera P1 y P2 están relacionadas P1 $\leq$ P2 si el nombre de P1 es léxicográficamente menor o igual al nombre de P2.

Definición del problema de ordenación

Dada una secuencia de n elementos de un mismo tipo $S = <e_1, e_2, \ldots, e_n>$ y una relación de orden $\leq$ ($\geq$), el problema del ordenamiento consiste en encontrar una permutación $<e_1', e_2', \ldots, e_n'>$ de S tal que

- $e_1' \leq e_2' \leq, \ldots, \leq e_n'$ (orden ascendente)
- $e_1' \geq e_2' \geq, \ldots, \geq e_n'$ (orden descendente)

NOTA:

La relación de orden se establece a partir de los valores del atributo clave de los elementos.

No es necesario todos los elementos tengan valores distintos entre sí en su atributo clave

Estabilidad de un método de ordenación

Un método de ordenación es estable, si para objetos con atributos clave de igual valor, tras la ordenación, el orden inicial entre los mismos se mantiene.

No.	Original	Ordenación estable	Ordenación no estable
1	Juan Pérez	Jacinto Fernández	Jacinto Fernández
2	**Ana Rodríguez**	José García	José García
3	José García	Juan Pérez	Juan Pérez
4	**Andrés Rodríguez**	**Ana Rodríguez**	**Andrés Rodríguez**
5	Jacinto Fernández	**Andrés Rodríguez**	**Ana Rodríguez**

Método Bubble-Sort

Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor (o menor) que el que está en la siguiente posición se intercambian.

La idea básica es imaginar que los objetos a ordenar están en un arreglo "vertical" y que en tal sentido, los objetos con claves menores son "más ligeros" y por tanto "suben a la superficie" primeramente.

Método Bubble-Sort

```
void Intercambiar(int* L, int n, int i, int j) {
  int aux = L[i];
  L[i] = L[j];
  L[j] = aux;
}
```

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 0

9	12	18	2	5

j

12 > 9 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 0

9	12	18	2	5

j

12 > 18 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 0

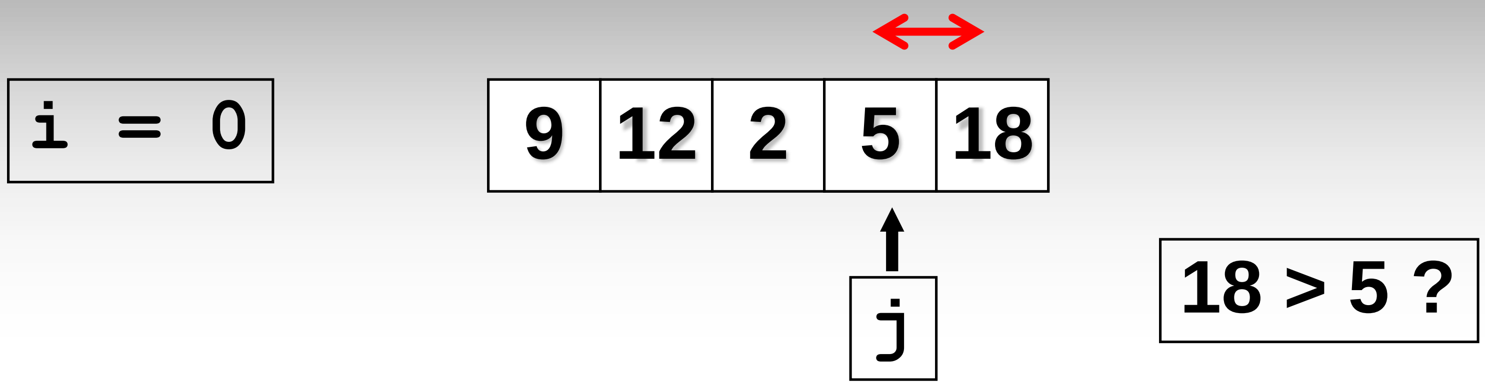
9	12	2	18	5

j

18 > 2 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución




```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 1

9	12	2	5	18

j

9 > 12 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 1

9	2	12	5	18

j

12 > 2 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 1

9	2	5	12	18

j

12 > 5 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 2

2	9	5	12	18

j

9 > 2 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 2

2	5	9	12	18

j

9 > 5 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 3

2	5	9	12	18

j

2 > 5 ?

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Ejemplo de ejecución

i = 4

2	5	9	12	18

j

Fin!

```
void BubbleSort(int* L, int n){
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n-i-1; j++)
      if (L[j] > L[j+1])
        Intercambiar(L, n, j, j+1);
}
```

Bubble-Sort: Análisis de la complejidad

El método consiste en dos ciclos anidados cada uno de n iteraciones, donde n es la cantidad de elementos de la lista, y las operaciones que se realizan dentro del ciclo interno tienen complejidad $O(1)$. Por tanto, el algoritmo tiene complejidad $O(n^2)$.

Ventajas

- Fácil de implementar.
- Estable.

Desventajas

- Complejidad temporal $O(n^2)$ en el peor de los casos.

Método Selection-Sort

Su principio de funcionamiento es el intercambio de objetos, pero en este caso, no necesariamente adyacentes.

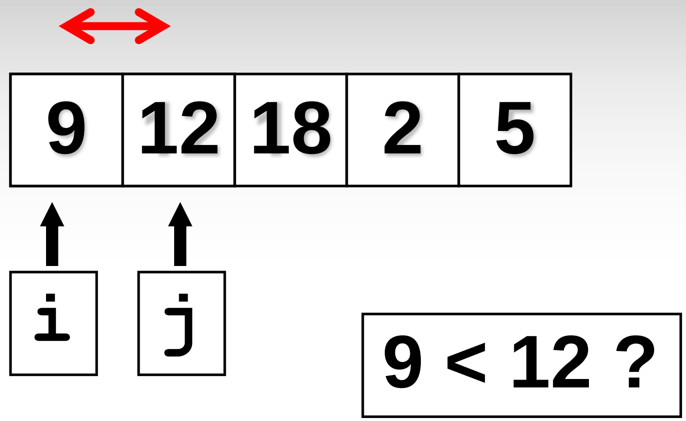
En la primera pasada se halla el mínimo de los *n* elementos y se coloca en la primera posición del arreglo.

En la segunda pasada se halla el mínimo de los *n*-1 elementos restantes y se coloca en la segunda posición, y así sucesivamente.

Método Selection-Sort

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución




```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución

9	12	18	2	5

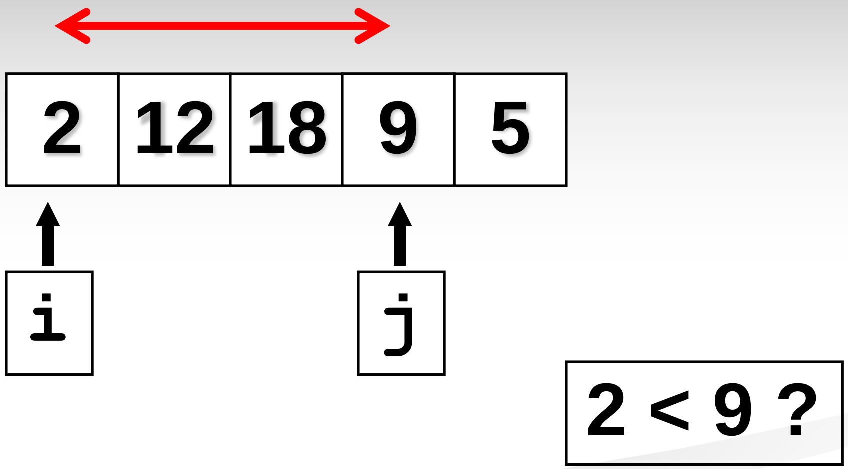
i

j

18 < 9 ?

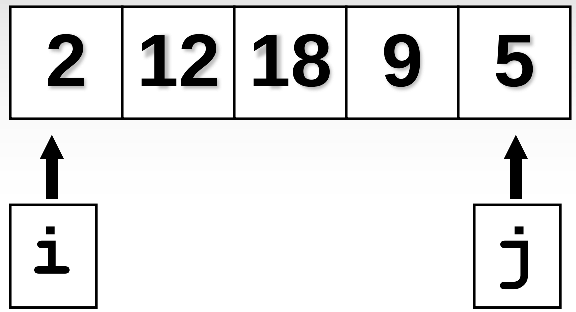
```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución




```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución



5 < 2 ?

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución

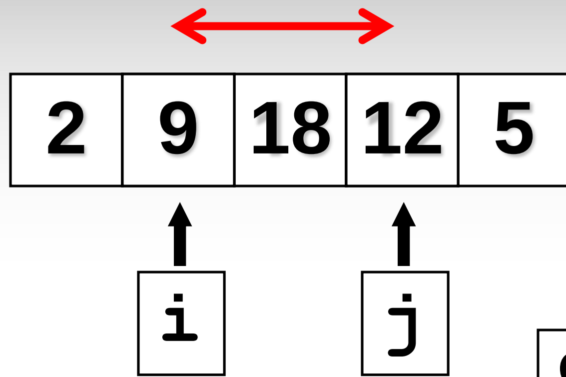
2	12	18	9	5

i j

18 < 12 ?

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

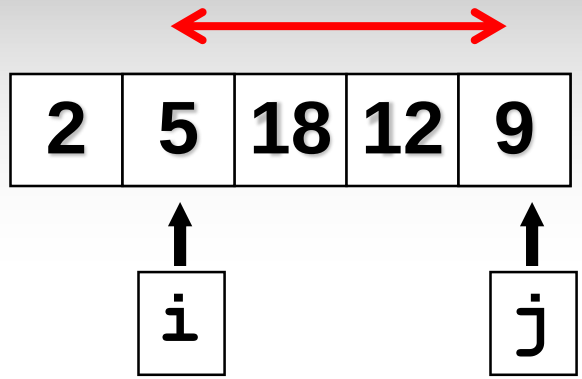
Selection-Sort: Ejemplo de ejecución



9 < 12 ?

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

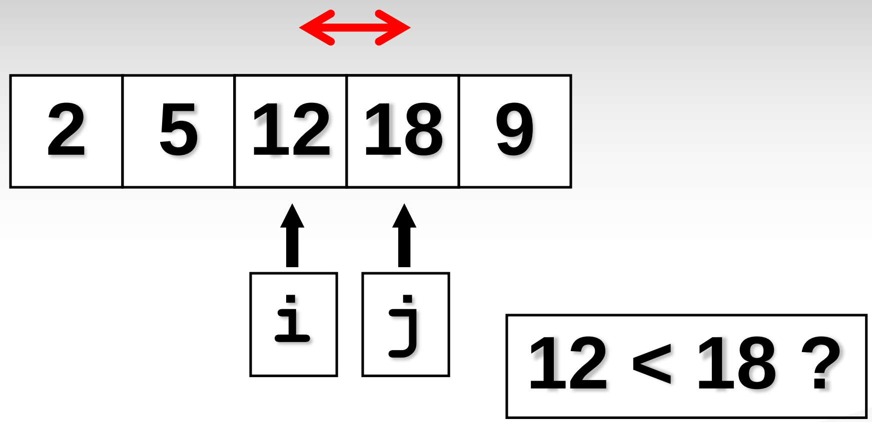
Selection-Sort: Ejemplo de ejecución



5 < 9 ?

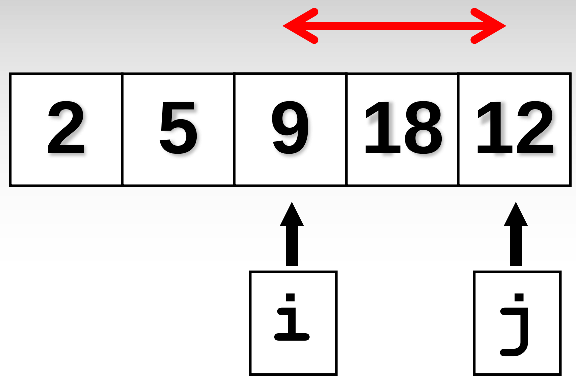
```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución




```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución



9 < 12 ?

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución

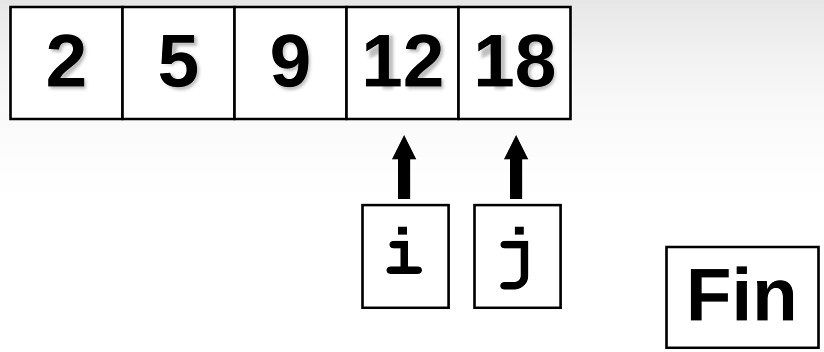
2	5	9	12	18

i j

12 < 18 ?

```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Ejemplo de ejecución




```
void SelectionSort(int* L, int n){
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
      if (L[j] < L[i])
        Intercambiar(L, n, i, j);
}
```

Selection-Sort: Análisis de la complejidad

Independientemente de la cantidad de intercambios que haga, siempre se ejecutarán íntegramente los dos ciclos que son los que aportan $n(n - 1)/2$ operaciones determinando su orden cuadrático, llegando a que la complejidad en el peor caso seria $O(n^2)$

Ventajas

- Fácil de implementar.

Desventajas

- Complejidad temporal $O(n^2)$ en el peor de los casos.
- No es estable

Método Insertion-Sort

Sea $L = (x_0, x_1, ..., x_{n-1})$ la lista a ordenar.

Se parte de la base de que para $i = 0$ la sublista (x_0) está ordenada y desde $i = 1$ hasta n se inserta x_i en el lugar que le corresponde en la sublista $(x_0, x_1, ..., x_{i-1})$ de forma que se obtiene la lista desde 0 hasta n-1 ordenada

$$\overbrace{x_0 \quad x_1 \quad x_2 \quad . . . \quad x_{i-1}}^{\text{Sublista ordenada}} \quad \boxed{x_i} \quad \ldots \quad x_{n-1}$$

Método Insertion-Sort

```
void InsertionSort(int* L, int n){
  for (int i = 1; i < n; i++) {
    int j = i;
    while ((L[j-1] > L[j]) && (j > 0)) {
      Intercambiar(L, n, j, j-1);
      j--;
    }
  }
}
```

Insertion-Sort: Ejemplo de ejecución

i=1 x←55

Insertar *x* en el lugar que le corresponde

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
i=1	**44**	**55**	**12**	**42**	**94**	**18**	**06**	**67**

Como 55 > 44 se mantiene el 55 en el lugar 1

Insertion-Sort: Ejemplo de ejecución

i=2 x←12

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=2	**44**	**55**	**12**	**42**	**94**	**18**	**06**	**67**

Como 12 < 55 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=2 x←12

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=2	**44**	**12**	**55**	**42**	**94**	**18**	**06**	**67**

Como 12 < 44 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=2 x←12

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
i=2	**12**	**44**	**55**	**42**	**94**	**18**	**06**	**67**

Insertion-Sort: Ejemplo de ejecución

i=3 $x \leftarrow 42$

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=3	**12**	**44**	**55**	**42**	**94**	**18**	**06**	**67**

Como 42 < 55 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=3 x←42

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=3	**12**	**44**	**42**	**55**	**94**	**18**	**06**	**67**

Como 42 < 44 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=3 x←42

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=3	**12**	**42**	**44**	**55**	**94**	**18**	**06**	**67**

Como 42 > 12 NO se intercambian, y se pasa a la siguiente iteración

Insertion-Sort: Ejemplo de ejecución

i=4 x←94

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=4	**12**	**42**	**44**	**55**	**94**	**18**	**06**	**67**

Como 94 > 55 NO se intercambian, y se pasa a la siguiente iteración

Insertion-Sort: Ejemplo de ejecución

i=5 x←18

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=5	**12**	**42**	**44**	**55**	**94**	**18**	**06**	**67**

Como 18 < 94 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=5 x←18

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=5	**12**	**42**	**44**	**55**	**18**	**94**	**06**	**67**

Como 18 < 55 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=5 x←18

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=5	**12**	**42**	**44**	**18**	**55**	**94**	**06**	**67**

Como 18 < 44 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=5 x←18

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=5	**12**	**42**	**18**	**44**	**55**	**94**	**06**	**67**

Como 18 < 42 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=5 x←18

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=5	**12**	**18**	**42**	**44**	**55**	**94**	**06**	**67**

Como 18 > 12 NO se intercambian, y se pasa a la siguiente iteración

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=6	**12**	**18**	**42**	**44**	**55**	**94**	**06**	**67**

Como 6 < 94 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=6	**12**	**18**	**42**	**44**	**55**	**06**	**94**	**67**

Como 6 < 55 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=6	**12**	**18**	**42**	**44**	**06**	**55**	**94**	**67**

Como 6 < 44 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
i=6	**12**	**18**	**42**	**06**	**44**	**55**	**94**	**67**

Como 6 < 42 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=6	**12**	**18**	**06**	**42**	**44**	**55**	**94**	**67**

Como 6 < 18 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
i=6	**12**	**06**	**18**	**42**	**44**	**55**	**94**	**67**

Como 6 < 12 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=6 x←6

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
i=6	**06**	**12**	**18**	**42**	**44**	**55**	**94**	**67**

Insertion-Sort: Ejemplo de ejecución

i=7 x←67

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=7	**06**	**12**	**18**	**42**	**44**	**55**	**94**	**67**

Como 67 < 94 se intercambian

Insertion-Sort: Ejemplo de ejecución

i=7 x←67

Insertar *x* en el lugar que le corresponde desde j=i hasta 0

	0	1	2	3	4	5	6	7
i=7	**06**	**12**	**18**	**42**	**44**	**55**	**67**	**94**

Como 67 > 55 NO se intercambian.
Se debería pasar a la siguiente iteración
, pero es la última. FIN

Insertion-Sort: Análisis de la complejidad

En el peor caso (<u>caso en que la lista esté ordenada descendentemente</u>) cada elemento debe ser comparado con los previos n-1 elementos en el arreglo, por lo que en total se harían:

$$1+2+3+ \ldots\ldots. + (n-2)+(n-1) = 1/2*(n-1)*n$$

comparaciones. Llegando a que la complejidad en el peor caso seria $O(n^2)$

Ventajas

- Fácil de implementar.
- Estable

Desventajas

- Complejidad temporal $O(n^2)$ en el peor de los casos.

Sumario

1. Definición del problema de ordenación
2. Métodos simples o de orden $O(n^2)$
   - Bubble-Sort
   - Insertion-Sort
   - Selection-Sort
3. Métodos complejos o de orden $O(n \log n)$
   - Merge-Sort
   - Quick-Sort
4. Otros métodos

Método Merge-Sort

Este algoritmo está basado en la técnica de diseño de algoritmos *Divide y Vencerás*.

Consiste en dividir el problema a resolver en subproblemas del mismo tipo que a su vez se dividirán, mientras no sean suficientemente pequeños o triviales.

Método Merge-Sort

Ordenar una secuencia S de elementos

- Si S tiene uno o ningún elemento, está ordenada
- Si S tiene al menos dos elementos se divide en dos secuencias S1 y S2, S1 conteniendo los primeros n/2, y S2 los restantes.
- Ordenar S1 y S2, aplicando recursivamente este procedimiento.
- ***Mezclar*** S1 y S2 ordenadamente en S

Método Merge-Sort

Mezcla de dos secuencias ordenadas S1 y S2 en S

- Se tienen referencias al principio de cada una de las secuencias a mezclar (S1 y S2).
- Mientras en alguna secuencia queden elementos, se inserta en la secuencia resultante (S) el menor de los elementos referenciados y se avanza esa referencia una posición.

Método Merge-Sort

26	5	77	1	61	11	59	15	48	19

S1

26	5	77	1	61

S2

11	59	15	48	19

Secuencia ordenada

Método Merge-Sort

```
MergeSort (int* L, int n)
{
  if (n > 1) {
    L1 = subLista(L, n, 0, n/2 - 1); // Primera
                                     // mitad
    L2 = subLista(L, n, n/2, n - 1); // Segunda
                                     // mitad
    MergeSort(L1, n/2);
    MergeSort(L2, n – n/2);
    Merge(L1, n/2, L2, n – n/2, L);
  }
}
```

Método Merge-Sort

```
Merge(int* L1, int n1, int* L2, int n2, int* L)
{
  int i = 0, j = 0, k = 0;
  while (i < n1) && (j < n2) {
    if (L1[i] <= L2[j]) {
      L[k] = L1[i];
      i++; k++;
    }
    else {
      L[k] = L2[j];
      j++; k++;
    }
  }
  //…
```

Método Merge-Sort

```
//Continuación

  while (i < n1) {
    L[k] = L1[i];
      i++; k++;
  }
  while (j < n2) {
    L[k] = L1[j];
    j++; k++;
  }
} //Fin del Merge
```

Merge-Sort: Análisis de la complejidad

$$T(n) = \begin{cases} d & \text{Si } n \le 1 \\ 2T(n/2) + O(n) & \text{Si } n > 1 \end{cases}$$

Para ordenar una lista de tamaño *n* se ordenan 2 listas de tamaño *n*/2, de aquí el 2*T*(*n*/2), y luego se consume O(*n*) en realizar la mezcla.

Resolviendo la ecuación recurrente tenemos que

T(*n*) = *O*(*n* log *n*)

Ventajas

- Es estable.
- Complejidad *O*(*n* log *n*)

Desventajas

- Usa memoria auxiliar *O*(*n*).

Merge-Sort: Ventajas y Desventajas

Ventajas

- Es estable.
- Complejidad $O(n \log n)$

Desventajas

- Usa memoria auxiliar $O(n)$.

Método Quick-Sort

- Si la lista a ordenar tiene más de un elemento, dividir la lista en dos partes (no necesariamente de igual tamaño) donde cualquier elemento de la primera es menor que cualquier otro de la segunda. Esta división se realiza usando un elemento *pivote*.
- Ordenar cada una de las partes aplicando este procedimiento recursivamente.

Método Quick-Sort

Particionar

1. Selección del elemento pivote
2. Buscar desde la primera posición hacia el final un elemento mayor o igual que el pivote (posición I)
3. Buscar desde última posición hacia el inicio un elemento mayor que el pivote (posición J)
4. Si I < J intercambiar los elementos en estas posiciones y volver al paso 2, sino particionar en la posición J.

Método Quick-Sort

Particionar

pivote = 26

26	5	77	1	61	11	59	15	48	19

I

J

A[I] >= pivote ?

SI NO

A[J] < pivote ?

SI NO

Método Quick-Sort

Ordenar las dos partes aplicando el mismo procedimiento

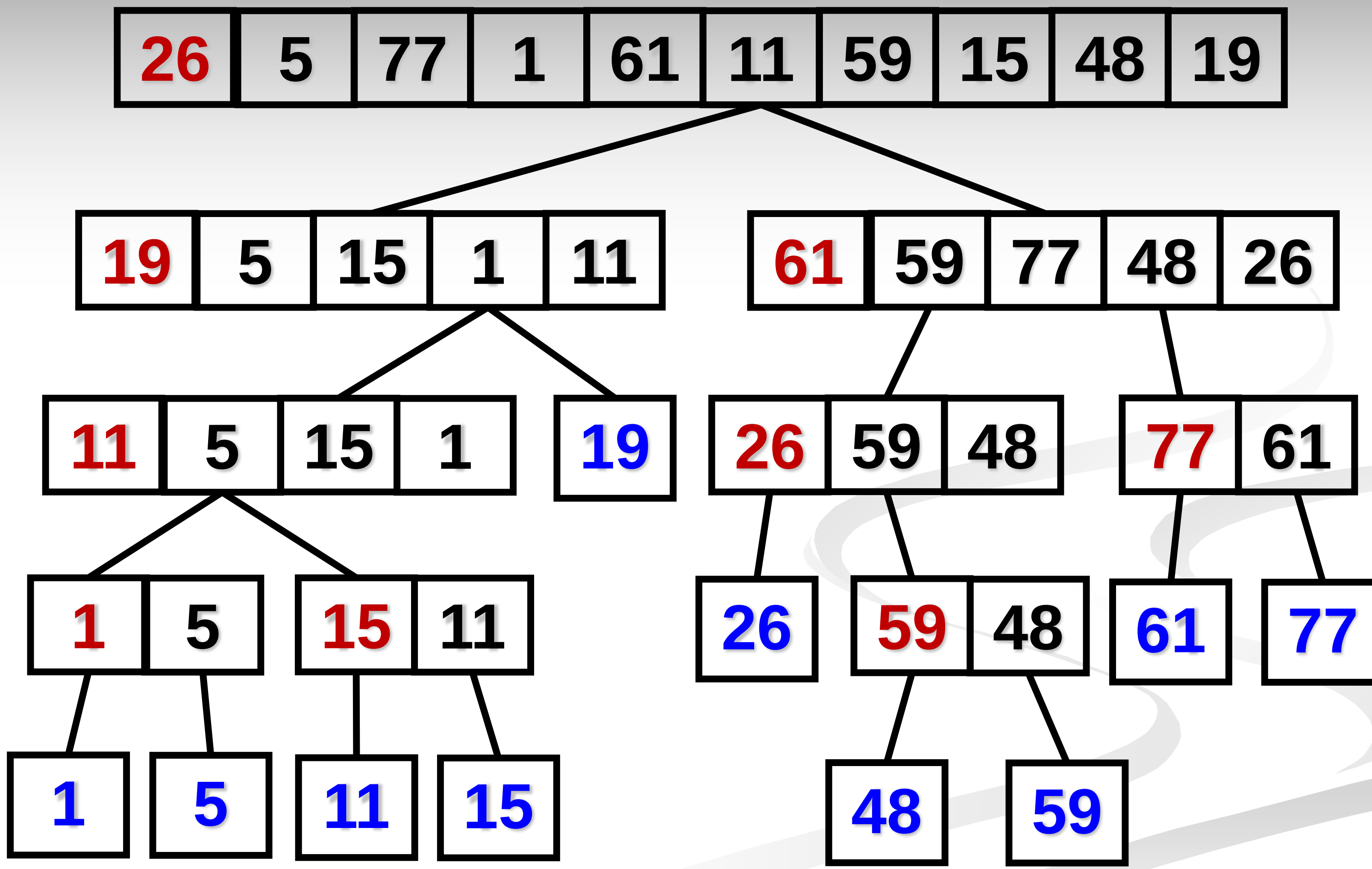
19	5	15	1	11	61	59	77	48	26

QuickSort **Ordenado**

QuickSort **Ordenado**

Ordenado

Método Quick-Sort



Método Quick-Sort

```
void QuickSort (int* L, int n){
  QuickSortAux(L, 0, n-1);
}
```

```
void QuickSortAux (int* L, int inicio, int fin){
  if (inicio < fin) {
    int k = Partition(L, n, inicio, fin);
    QuickSortAux(L, inicio, k); //Ordenar la
                                //primera mitad
    QuickSortAux(L, k+1, fin); //Ordenar la
                               //segunda mitad
  }
}
```

Método Quick-Sort

```
int Partition (int* L, int n, int inicio, int fin){
  int pivote = L[inicio];
  int i = inicio, j = fin;
  while (i < j) {
    while (L[i] < pivote)
      i++;
    while (pivote <= L[j])
      j--;
    if (i < j)
      Intercambiar(L, n, i, j);
  }
  return j;
}
```

Quick-Sort: Análisis de la complejidad

El tiempo de ejecución del algoritmo QUICKSORT está determinado por la forma en que se fueron seleccionando los pivotes:

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1} & \textbf{Si } \boldsymbol{n = 1} \\ \boldsymbol{T(k) + T(n-k) + O(n)} & \textbf{Si } \boldsymbol{n > 1} \end{cases}$$

Para ordenar *n* elementos hay que primero particionar la lista consumiendo *O*(*n*) (orden de PARTITION) y luego ordenar recursivamente dos sublistas de longitudes *k* y *n* – *k*.

Quick-Sort: Análisis de la complejidad

El caso mejor

En cada momento se selecciona como pivote la ***mediana*** *de los elementos de la lista,* *k* siempre es aproximadamente igual a *n/2* obteniéndose la siguiente expresión:

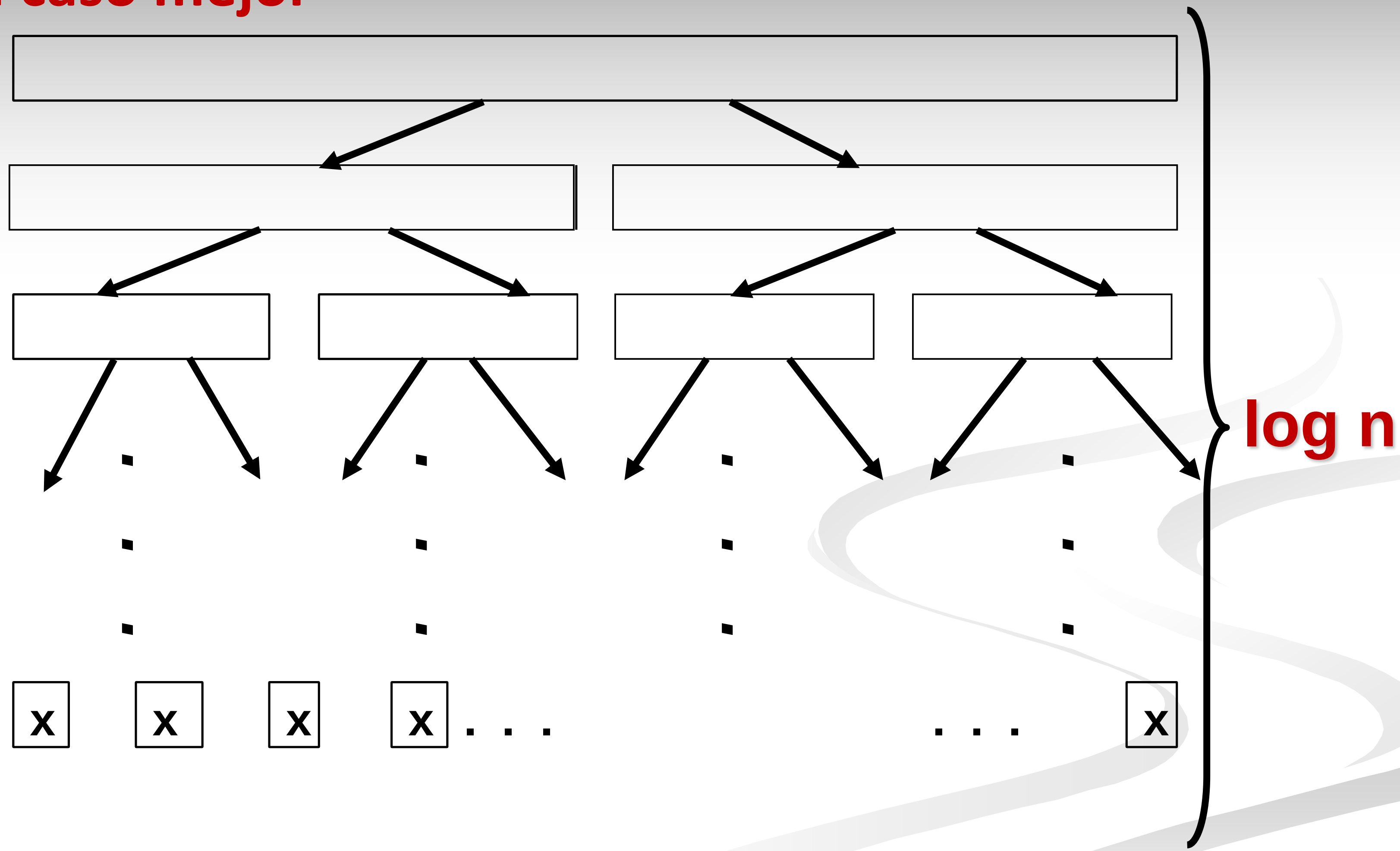
$$T(n) = \begin{cases} 1 & \text{Si } n = 1 \\ T(n/2) + T(n - n/2) + O(n) & \text{Si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} 1 & \text{Si } n = 1 \\ 2T(n/2) + O(n) & \text{Si } n > 1 \end{cases}$$

La misma expresión del MergeSort

Quick-Sort: Análisis de la complejidad

El caso mejor



Quick-Sort: Análisis de la complejidad

El caso peor

En cada momento se selecciona como pivote el menor o el mayor elemento del arreglo, k siempre es 1, obteniendo la expresión:

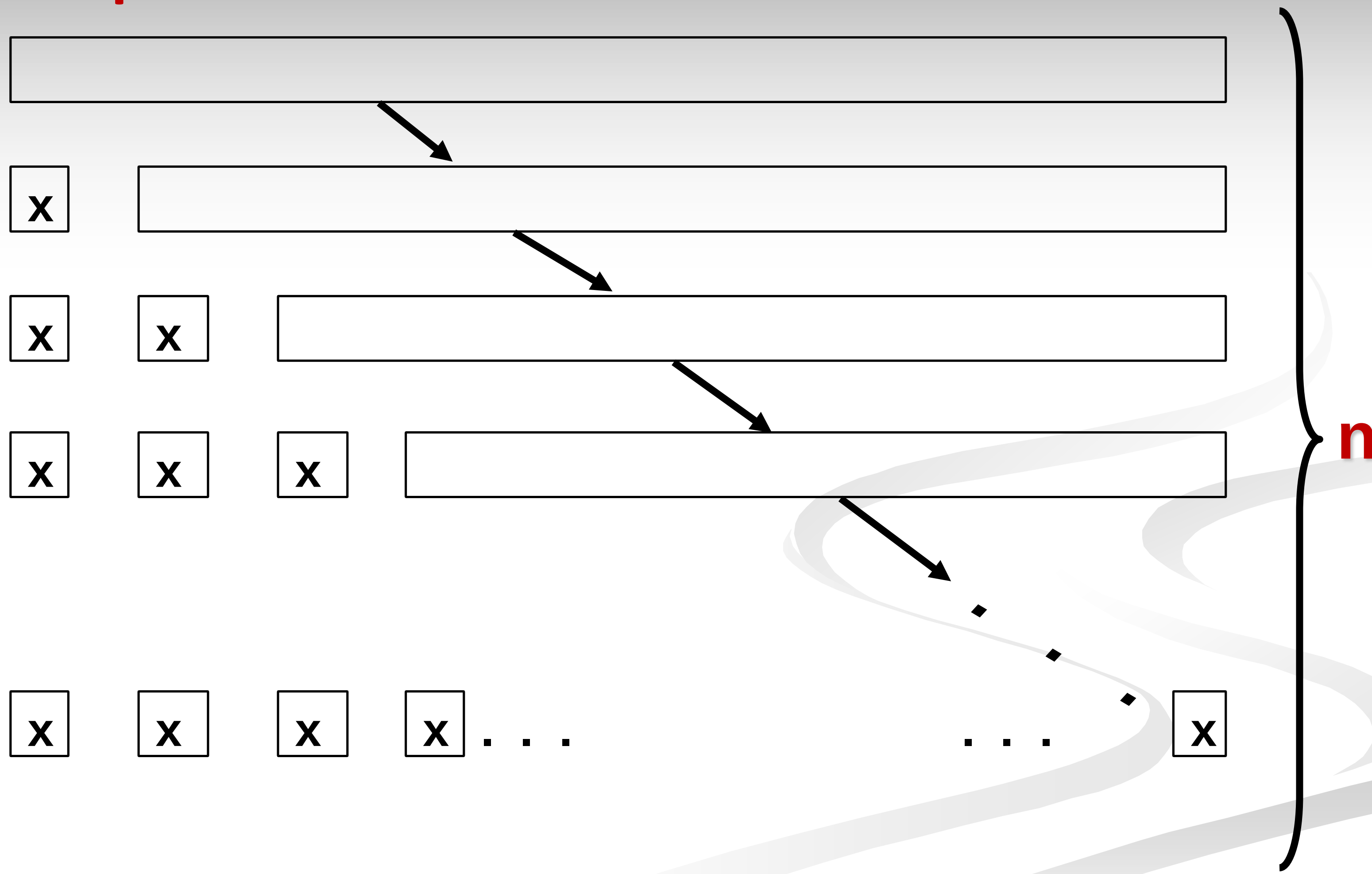
$$T(n) = \begin{cases} 1 & \text{Si } n = 1 \\ T(n-1) + O(n) & \text{Si } n > 1 \end{cases}$$

$n+(n-1)+(n-2)\ldots+1 = n(n+1)/2 = O(n^2)$

Este caso se puede dar cuando el arreglo está ordenado y tomamos el primer elemento como pivote

Quick-Sort: Análisis de la complejidad

El caso peor



Quick-Sort: Propiedades

- Es el algoritmo de ordenación más rápido que existe.
- Aunque su caso peor es $O(n^2)$, la probabilidad de su ocurrencia es muy baja si tomamos algunas de sus variantes aleatorias.
- En el caso promedio es un $O(n \log n)$ y es lo que determina fuertemente su eficiencia
- No es Estable
- Ordena en el lugar. No utiliza memoria auxiliar