# ANÁLISIS DE CASOS

Comparativa de órdenes de complejidad
Comparativa de tiempos de ejecución

# ORDENAMIENTO POR BURBUJA

- El algoritmo de la “burbuja” consiste en recorrer el *arreglo* analizando la relación de precedencia que existe entre cada elemento y el elemento que le sigue para determinar si estos se encuentran ordenados entre sí y, en caso de ser necesario, permutarlos para que queden ordenados.

- Es necesario revisar varias veces todo el *arreglo* hasta que no se necesiten más intercambios, lo cual significa que el *arreglo* está ordenado.

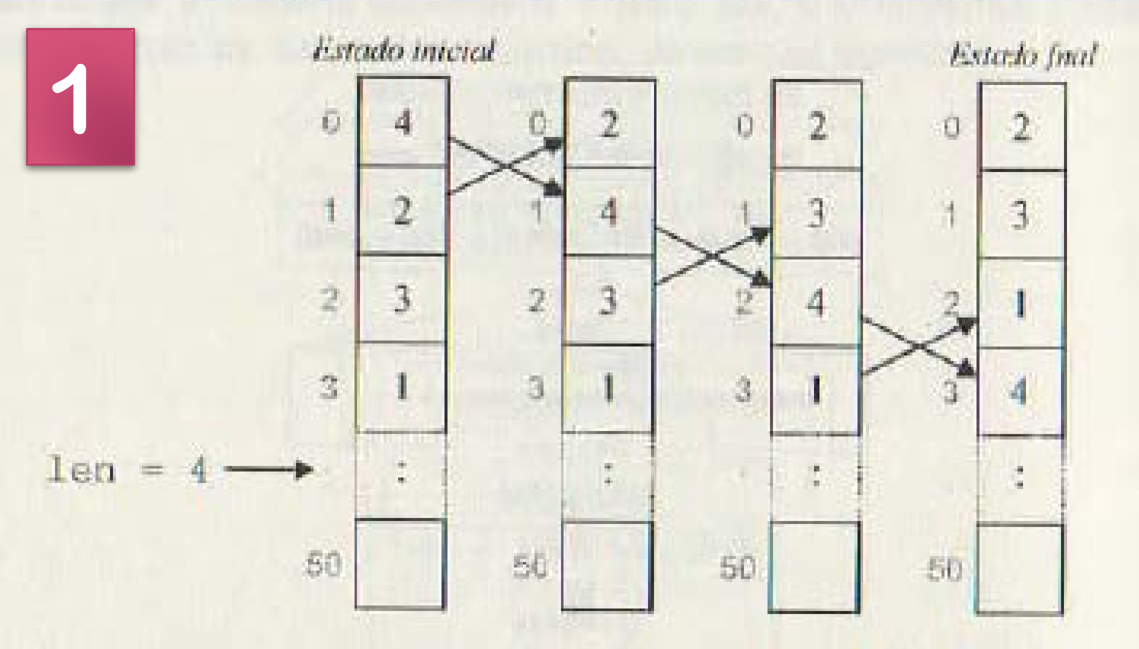
- Si ***a*** es el *arreglo* que vamos a ordenar e ***i*** es un valor comprendido entre 0 y len-1 entonces:

  - Si a[i]>a[i+1] significa que estos dos elementos se encuentran desordenados entre si y habrá que permutarlos.

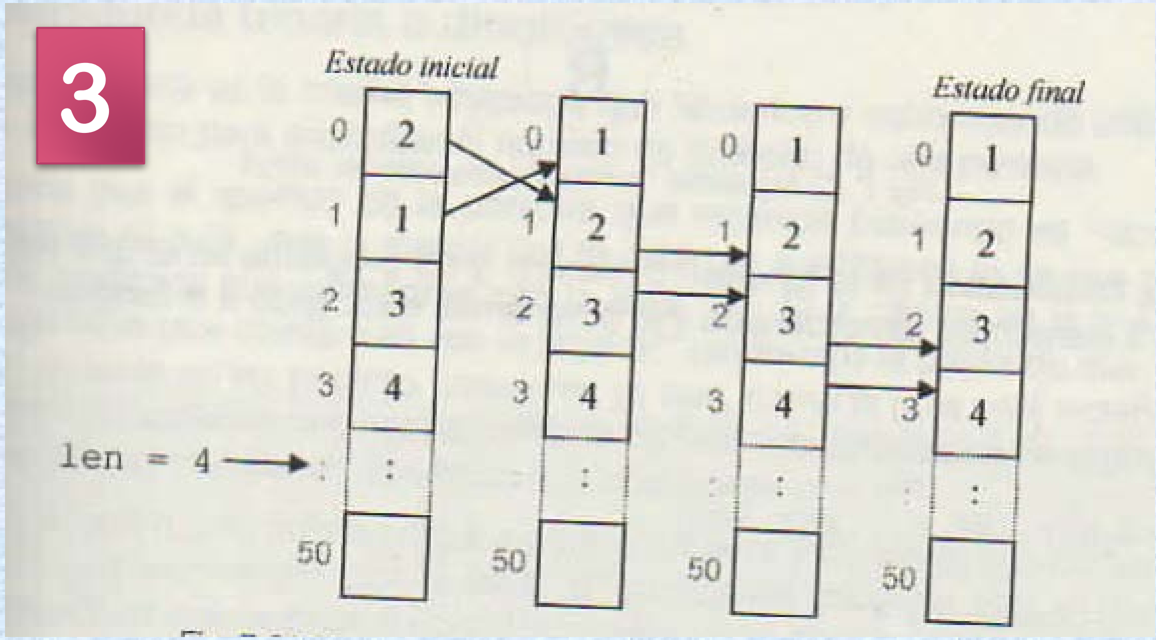
  - Entonces, se toman de a pares los elementos del *arreglo*, se comparan y si corresponde los permutamos para que cada par de elementos quedé ordenado entre sí.

- El algoritmo será recorrer el arreglo comparando **a[i]** con **a[i+1]** para permutarlos si no están en orden.

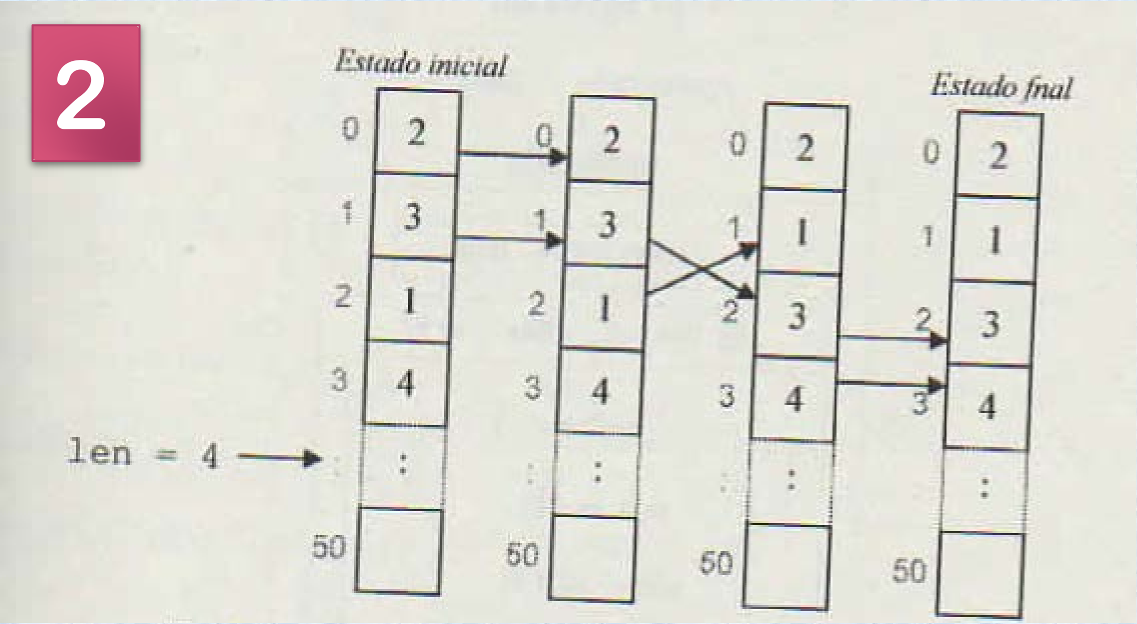
- El proceso finalizará cuando realicemos una iteración en la cual no haya sido necesario realizar ninguna permutación.




Primera pasada del ordenamiento por burbuja




Tercera pasada del ordenamiento por burbuja




Segunda pasada del ordenamiento por burbuja

## Código del ordenamiento por burbuja

```
void ordenar(int a[], int len)
{
	int i, aux;
	int ordenado = 0;
	while ( !ordenado )
	{
		ordenado=1;
		for(i = 0; i<len-1; i++)
		{
			if(a[i]>a[i+1])
			{
				aux = a[i];
				a[i] = a[i+1];
				a[i+1] = aux;
				ordenado = 0;
			}
		}
	}
}
```

- El ciclo interno se ejecuta len-1 iteraciones por cada una de las pasadas que se realicen sobre el arreglo.

- Una “pasada” implica recorrer el arreglo comparando cada elemento respecto del elemento siguiente para determinar si ambos están en orden o no y entonces permutarlos.

- Si durante una pasada se realiza al menos una permutación se forzará la realización de una nueva pasada.

- Así hasta que no sea necesario permutar ningún elemento, situación que nos permitirá determinar que el arreglo quedo ordenado.

El peor de los casos se dará cuando los elementos del arreglo se encuentren justamente en el orden inverso al cual los queremos ordenar.

Por ejemplo, el siguiente arreglo ordenado descendentemente.

arr = {5, 4, 3,  2, 1}

Si este fuera el caso, luego de la primera pasada el arreglo quedará así:

arr = {4, 3, 2, 1, 5}

Luego de la segunda pasada quedará así:

arr = {3, 2, 1, 4, 5}

Luego de la tercera pasada será:

arr = {2, 1, 3, 4, 5}

Y una nueva pasada lo dejará así:

arr = {1, 2, 3, 4, 5}

- Aunque el arreglo quedo ordenado, la permutación del elemento 2 por el elemento 1 nos obligará a realizar una pasada adicional.

- De este análisis se desprende que para ordenar un arreglo de 5 elementos totalmente desordenados, debemos realizar 5 pasadas y por cada una de estas el ciclo interno iterará 4 veces.

- Para un arreglo de longitud n, en el peor de los casos debemos atenernos a:

- n(n – 1) iteraciones del for interno, lo que equivale a decir: $n^2 - n$ iteraciones.

- Dado que esta función esta acotada superiormente por $n^2$ decimos que el ordenamiento por burbujeo tiene una complejidad cuadrática.

# BURBUJA OPTIMIZADO

Si aprovechamos el hecho de que luego de cada iteración el elemento más "pesado" se ubica en la última posición del arreglo (es decir, en su posición definitiva), podemos mejorar el algoritmo evitando comparar innecesariamente aquellos elementos que ya quedaron ordenados.

1. La primera iteración comparamos todos los elementos del arreglo hasta llegar a comparar arr[len-2] con arr[len-1].
2. La segunda iteración comparamos todos los elementos del arreglo pero solo hasta comparar arr[len-3] con arr[len-2].
3. En otras palabras, podemos comparar **arr[i]** con **arr[i+1]** si hacemos variar a ***i*** entre **0** y ***len-j-1*** siendo j una variable cuyo valor se inicializa en 0 y se incrementa luego de cada iteración.

# ORDENAMIENTO POR SELECCIÓN

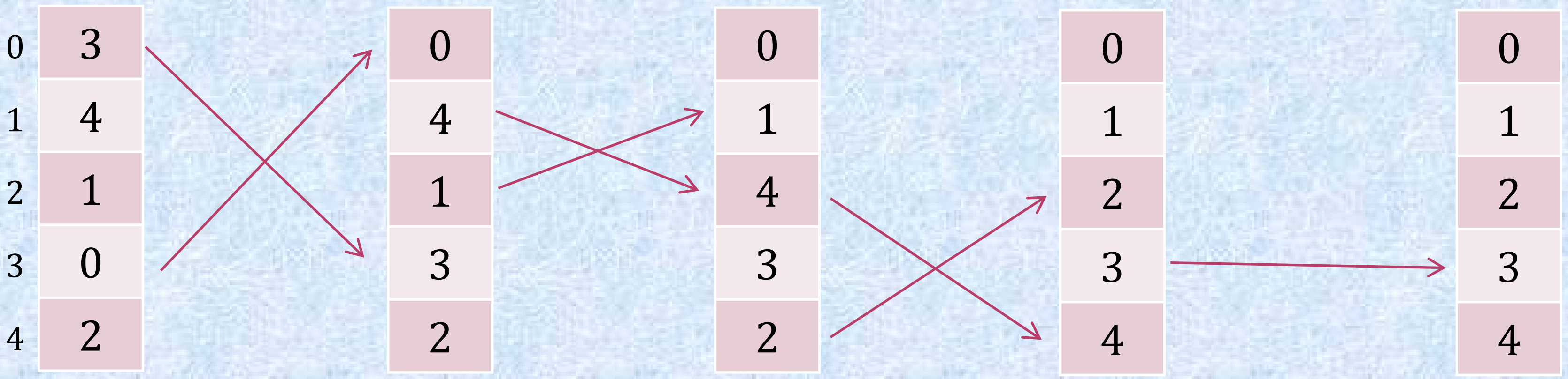
- Este algoritmo consiste en recorrer el arreglo buscando el menor elemento para intercambiarlo con el primero.

- Luego recorrer el arreglo pero comenzado desde la segunda posición para buscar el menor elemento e intercambiarlo por el segundo y así sucesivamente.

- Es decir que si consideramos un arreglo ***arr*** y un índice **i = 0**, debemos buscar el menor elemento de **arr** entre las posiciones **i** y **len-1** e intercambiarlo **con arr[i]**.

- Luego incrementamos **i** para descartar el primer elemento porque ya contiene su valor definitivo.

En el siguiente gráfico vemos como se ordena un arreglo arr = {3, 4, 1, 0, 2} luego de len-1 iteraciones.



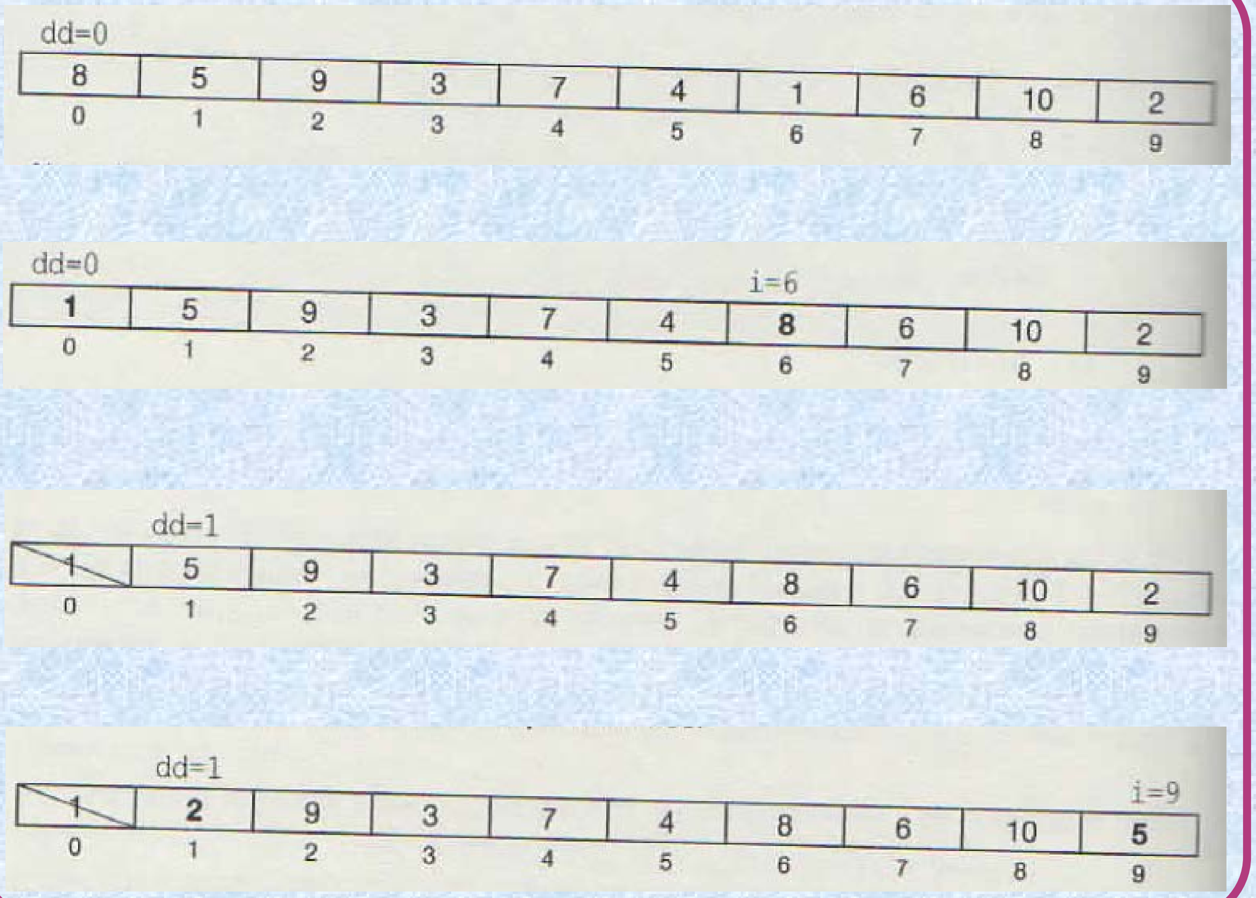

El algoritmo de ordenamiento por selección es de orden cuadrático **O(n$^2$)** y tiene un rendimiento similar al de Burbuja Optimizado.

# Función ordenar por selección

```
void ordenar_seleccion(int arr[], int len, int dd)
{
	int posMin;
	if(dd<len)
	{
		//buscamos el menor valor entre dd y len
		posMin = buscarPosMinimo(arr,len,dd);

		//permutamos arr[dd] por arr[posMin]
		int aux = arr[dd];
		arr[dd] = arr[posMin];
		arr[posMin] = aux;

		//invocación recursiva
		//ordenamos el array pero descartando la posición dd
		ordenar_seleccion(arr, len, dd+1);
	}
}
```

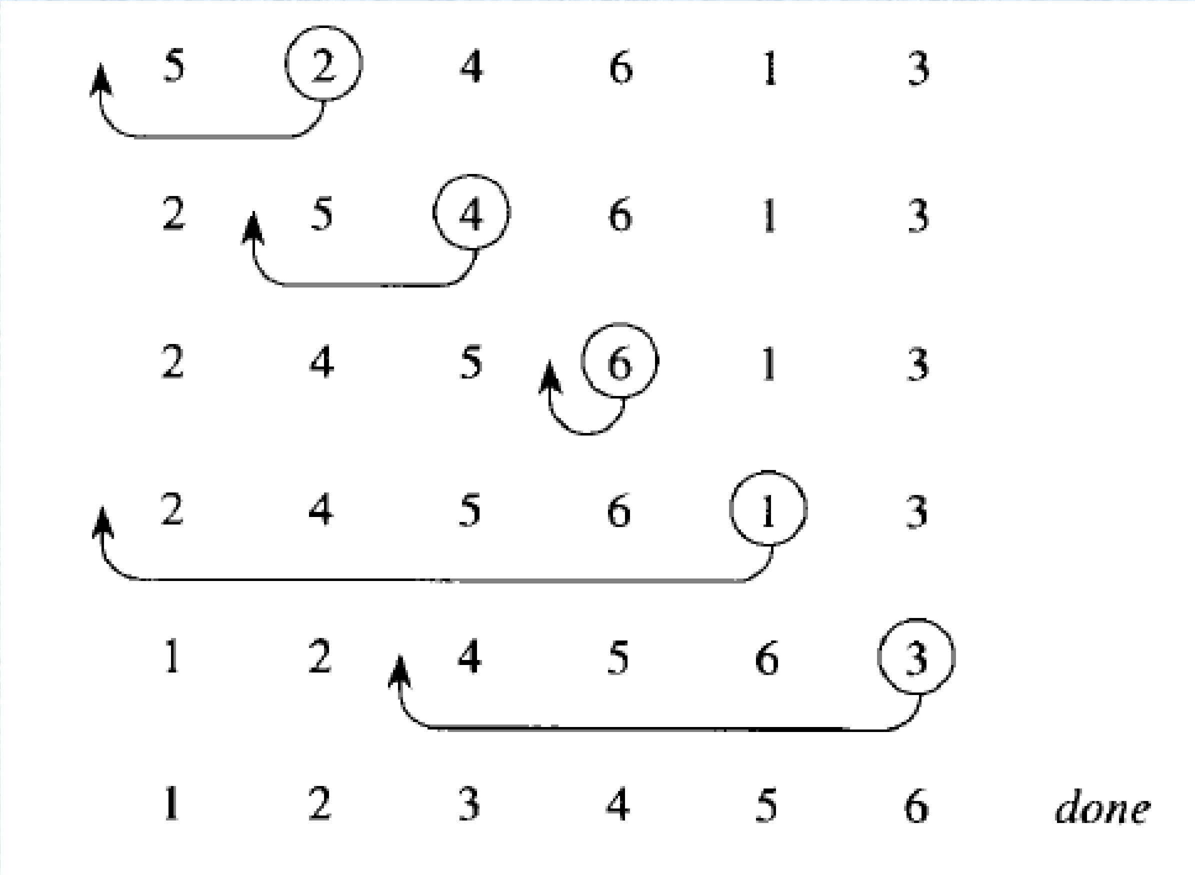


# Función buscar la posición del mínimo

```
//buscamos el menor elemento entre dd+1 y len y retornamos su posición
int buscarPosMinimo(int arr[], int len, int dd)
{
	//por ahora el menor es primero
	int posMin = dd;
	int min = arr[dd];
	int i;
	for(i=dd+1; i<len; i++)
	{
		if(arr[i]<min)
		{
			min = arr[i];
			posMin = i;
		}
	}
	return posMin;
}
```

# ORDENAMIENTO POR INSERCIÓN

- Algoritmo eficiente para ordenar una pequeña cantidad de elementos.



# PSEUDOCÓDIGO INSERTION SORT

```
INSERTION-SORT(A)
for j ← 2 to length[A]
     do key ← A[j]
          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                    i ← i − 1
          A[i + 1] ← key
```

| INSERTION-SORT($A$) | *cost* | *times* |
| --- | --- | --- |
| **for** $j \leftarrow 2$ **to** *length*$[A]$ | $c_1$ | $n$ |
|     **do** *key* $\leftarrow A[j]$ | $c_2$ | $n-1$ |
|         ▷ Insert $A[j]$ into the sorted | | |
|         ▷ sequence $A[1 \,..\, j-1]$. | $0$ | $n-1$ |
|         $i \leftarrow j-1$ | $c_4$ | $n-1$ |
|         **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
|             **do** $A[i+1] \leftarrow A[i]$ | $c_6$ | $\sum_{j=2}^{n}(t_j - 1)$ |
|                 $i \leftarrow i-1$ | $c_7$ | $\sum_{j=2}^{n}(t_j - 1)$ |
|         $A[i+1] \leftarrow key$ | $c_8$ | $n-1$ |

Para calcular *T(n)*, el tiempo de ejecución del ordenamiento por inserción, sumamos los productos de los costos y los tiempos obteniendo:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,.
\end{aligned}
$$

# Mejor de los casos

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \,.
\end{aligned}
$$

- Este tiempo de ejecución se puede expresar como :

***an + b*** para las *constantes a* y *b* que dependen de los costos ***Ci***, por lo tanto, es una **función lineal** de *n*.

Si el arreglo está en orden inverso, es decir, en orden decreciente, el resultados es el **peor de los casos**.

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}$$

Este tiempo de ejecución en el peor de los casos se puede expresar como:

$$an^2 + bn + c$$

para constantes a, b y c que dependen de los costos Ci.

Por lo tanto es una **función cuadrática de n**

# ORDENAMIENTO POR MEZCLA

- El algoritmo de ordenación por mezcla sigue de cerca el paradigma de divide y vencerás.

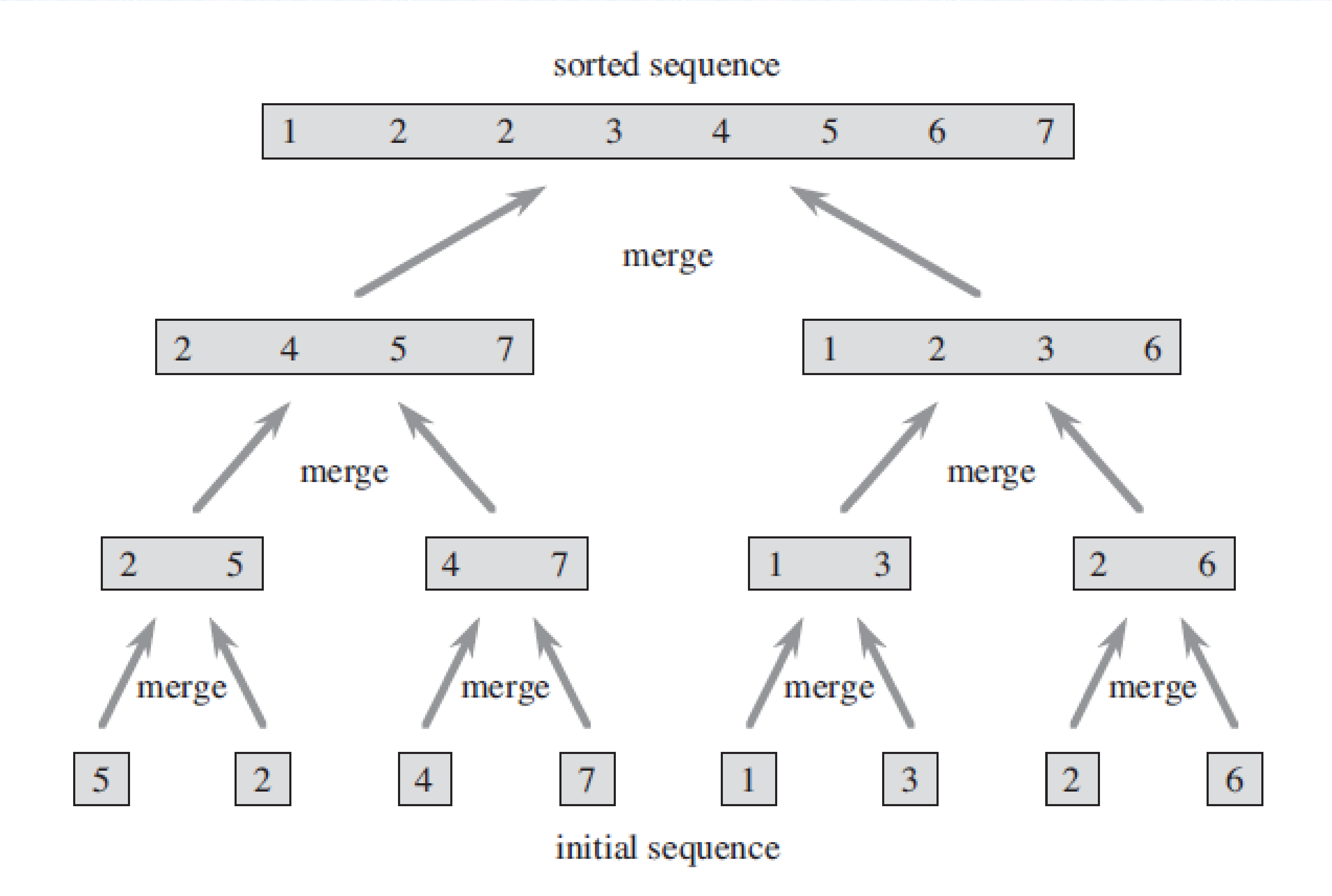
  Intuitivamente, funciona de la siguiente manera:

- **Dividir**: Divide la secuencia de n elementos para clasificar en dos subsecuencias de n/2 elementos cada una.

- **Conquistar**: Ordena las dos subsecuencias de forma recursiva utilizando merge sort.

- **Combinar**: Combina las dos subsecuencias ordenadas para producir la respuesta ordenada.

# EJEMPLO


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

# PSEUDOCÓDIGO FUNCIÓN MERGE

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

# MERGE SORT

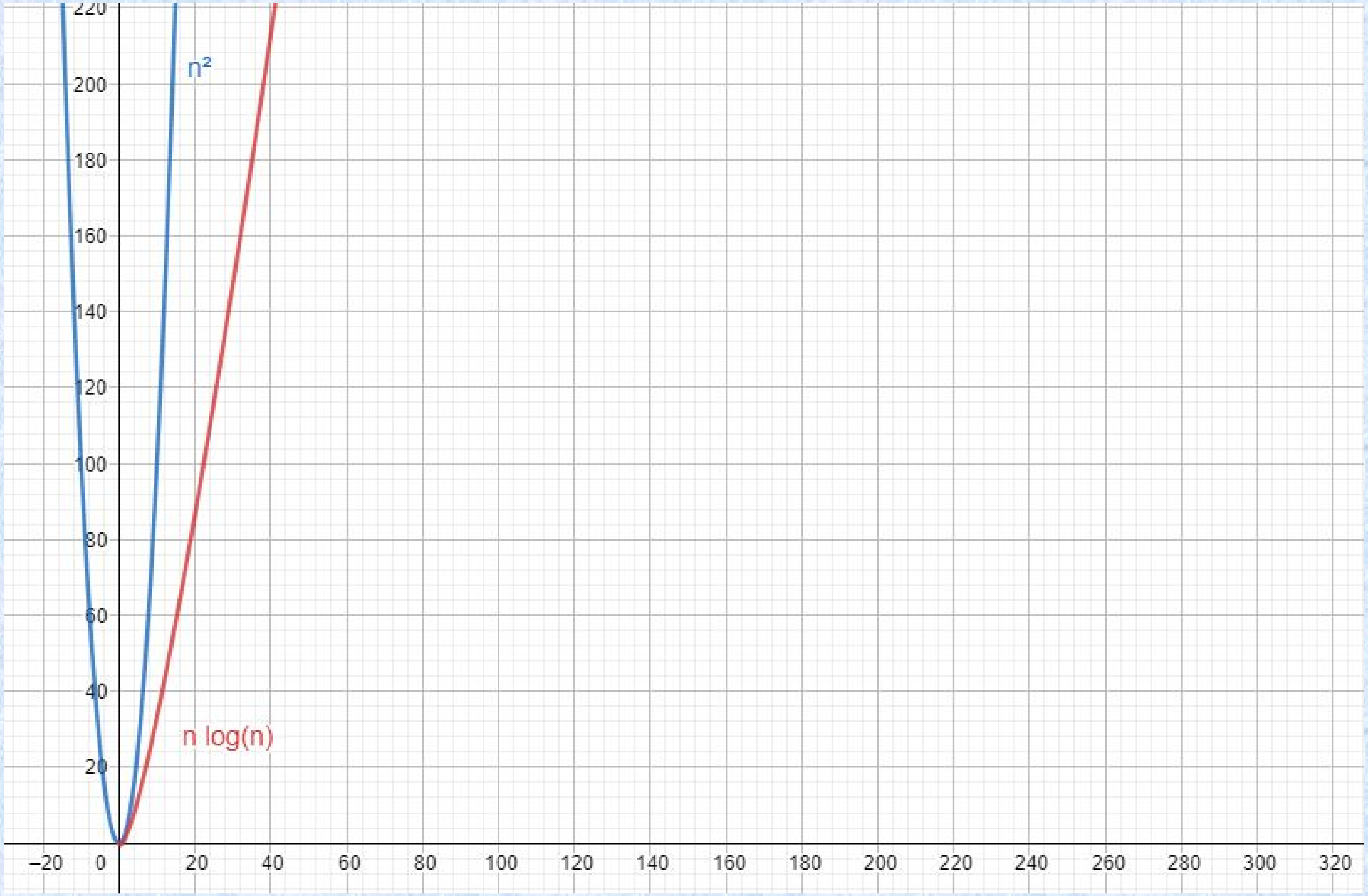
La complejidad del algoritmo esta definida por:

**T(n) = Θ(n lg n)**

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

Donde la constante *c* representa el tiempo requerido para resolver problemas de tamaño 1 como así como el tiempo por elemento del arreglo para dividir y combinar los pasos.

# COMPARACIÓN ENTRE LAS FUNCIONES DE LAS COMPLEJIDADES DE LOS ALGORITMOS DE ORDENAMIENTO



# TIEMPOS DE EJECUCIÓN

| Número de datos en la entrada | Burbuja | Selección | Inserción | Mezcla |
|---|---|---|---|---|
| *10 | 0.059 | 0.109 | 0.041 | 0.045 |
| 100 | 0.213 | 0.199 | 0.306 | 0.312 |
| 1,000 | 0.434 | 0.256 | 0.408 | 0.239 |
| 10,000 | 1.228 | 0.767 | 0.371 | 0.227 |
| 40,000 | 15.174 | 4.133 | 2.881 | 0.320 |
| 100,000 | 94.488 | 23.789 | 16.310 | 0.403 |
| 200,000 | 257.906 | 95.358 | 63.389 | 0.389 |
| 400,000 | | | | 0.468 |