Métodos de ordenamiento

Método *Shell Sort*

Este método lleva el nombre de su inventor, D. L. Shell. Se suele denominar también ordenación por inserción con incrementos decrecientes. Se considera que el método Shell es una mejora del método de inserción directa.

El algoritmo Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño, y con ello se consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial n/2 (siendo n el número de elementos), luego se reduce el salto a la mitad en cada repetición hasta que el salto es de tamaño 1.

6 1 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2), (5,4), (6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	*ninguno*	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2), (4,3)	0 1 2 3 4 5 6
5	1	*ninguno*	0 1 2 3 4 5 6

Los pasos a seguir por el algoritmo, para una lista de *n* elementos, son:

1. Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.
2. Clarificar cada grupo por separado, comparando las parejas de elementos, y, si no están ordenados, se intercambian.
3. Se divide ahora la lista en la mitad de grupos (n/4), con un incremento o salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando se consigue que el tamaño del salto sea 1.

```
Algoritmo Shell
Entrada: Vector V, Tamaño entrada n
Salida: Vector ordenado V
  intervalo ← div(n, 2) //division entera
  para intervalo > 0 hasta 0 hacer
    para k ← intervalo hasta n hacer
      y ← V[k]
      i ← k
      mientras i >= 0 ∧ y < V[i − intervalo] hacer
        V[i] ← V[i − intervalo]
        i ← i − intervalo
      fin mientras
      V[i] ← y
    fin para
    intervalo ← div(intervalo, 2)
  fin para
```

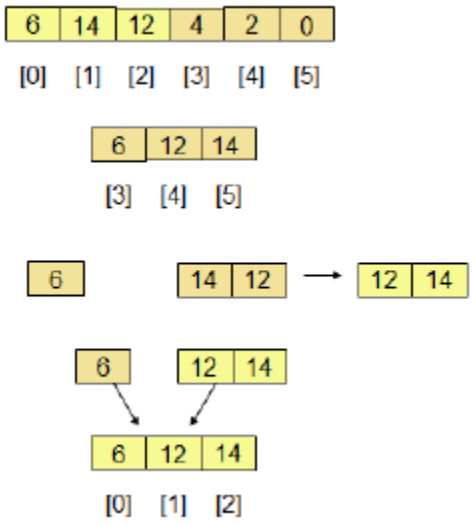
Complejidad del método: $O(n^2)$

Método *Merge Sort*

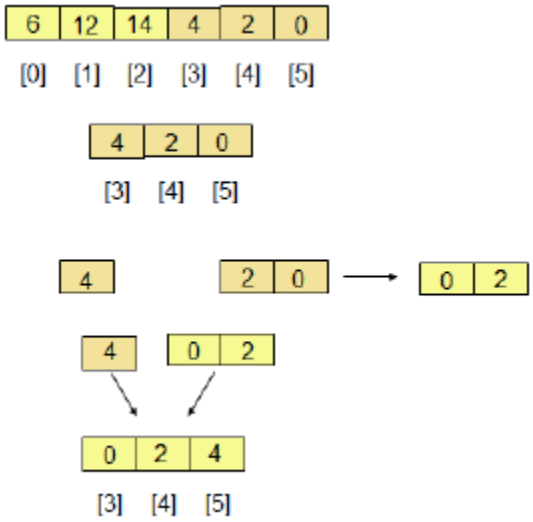
El algoritmo de ordenamiento por mezcla utiliza la técnica de "divide y vencerás" para realizar la ordenación de un arreglo. Su estrategia consiste en dividir un arreglo en dos subarreglos que sean de un tamaño tan similar como sea posible, dividir estas dos partes mediante llamadas recursivas, y finalmente, al llegar al caso base, mezclar los dos partes para obtener un arreglo ordenado.

Ejemplo: se quiere ordenar el vector 6 14 12 4 2

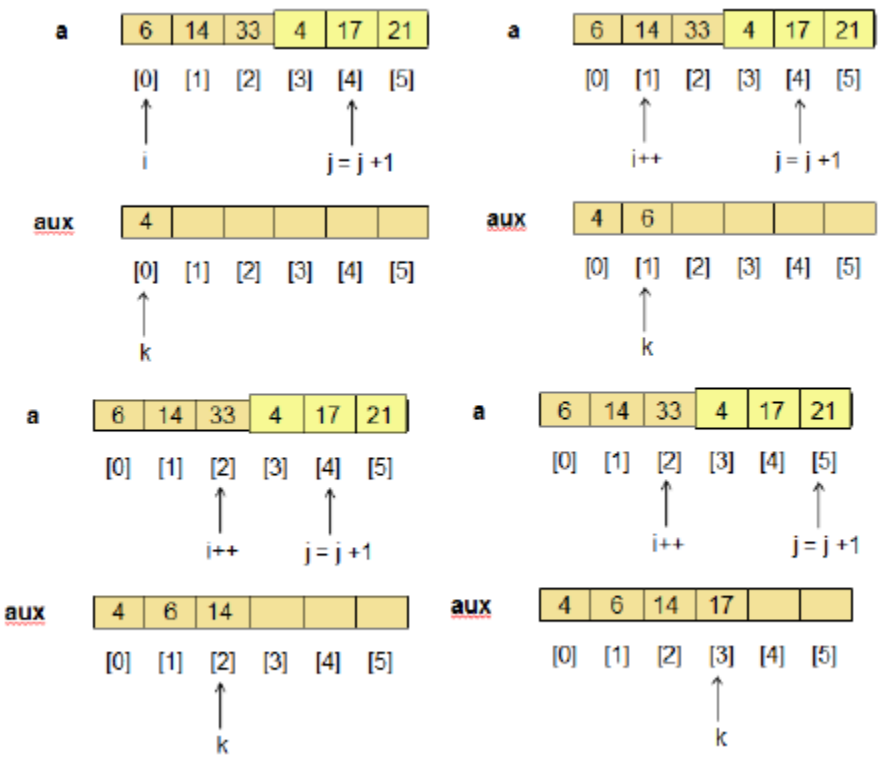
- Se hacen llamados recursivos a *Merge Sort* partiendo los sub-arreglos por la mitad con cada llamada recursiva, hasta que se obtiene un sub-vector de tamaño 1; entonces se llega al caso base:



- Subdivisión y mezcla del sub vector:



- Ya están los dos sub-vectores principales ordenados, por lo que sólo resta mezclarlos:



Tiempo de ejecución de Merge Sort

Los tres pasos a ejecutar son:

1. Dividir recursivamente la primera mitad del arreglo.
2. Dividir recursivamente la segunda mitad del arreglo.
3. Mezclar los dos subarreglos para ordenarlos.

El tiempo de ejecución es: $T(n) = 2T(n/2) + 4n + 2$ es $O(n\log_2(n))$.

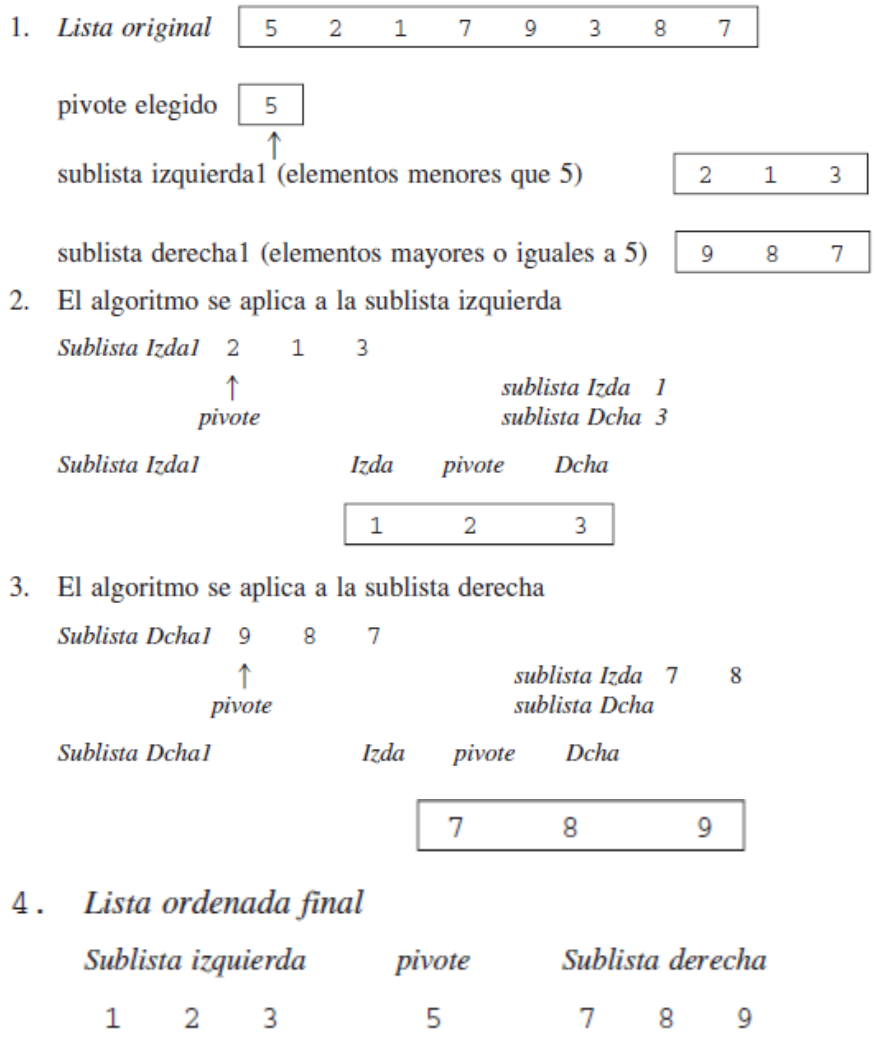
Método *Quick Sort*

El algoritmo conocido como *Quick Sort* (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple: se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar que aplica la técnica "divide y vencerás". El método es, posiblemente, el más pequeño de código, más rápido, más elegante, más interesante y eficiente de los algoritmos de ordenación conocidos.

El método se basa en dividir los *n* elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición izquierda, un elemento central denominado

pivote o elemento de partición; y una partición derecha. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y, la otra, todos los elementos (claves) mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *Quick Sort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista:



Tiempo de ejecución del Quick Sort

El peor caso, en el *Quick Sort*, se presenta cuando el pivote se elige de tal forma que los sub-arreglos de la derecha y de la izquierda están extremadamente desequilibrados. En el peor caso, el tiempo de ejecución del *Quick Sort* es $\mathbf{O}(n^2)$. En el mejor de los casos, es $\mathbf{O}(n\log n)$.

Algoritmos de búsqueda

Son algoritmos que se utilizan para encontrar un dato dentro de una estructura o arreglo:

- Se ha desarrollado un conjunto de algoritmos de búsqueda que varían en complejidad, eficiencia y tamaño del dominio de búsqueda.
- Si se conoce por anticipado en qué tipo de "orden" inicial se encuentran los datos, es posible elegir un algoritmo que sea más adecuado.

Tipos de búsqueda:

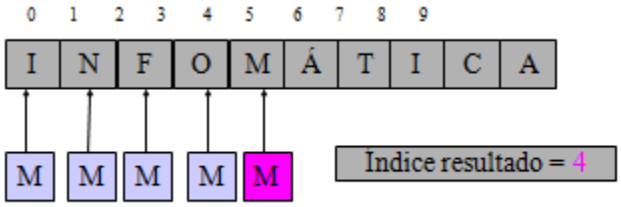
- Búsqueda secuencial.
- Búsqueda binaria.

Búsqueda secuencial

Consiste en ir comparando el elemento que se busca con cada elemento del arreglo hasta que se encuentra.

Ejemplo:

Se busca la M:



Complejidad de la búsqueda secuencial

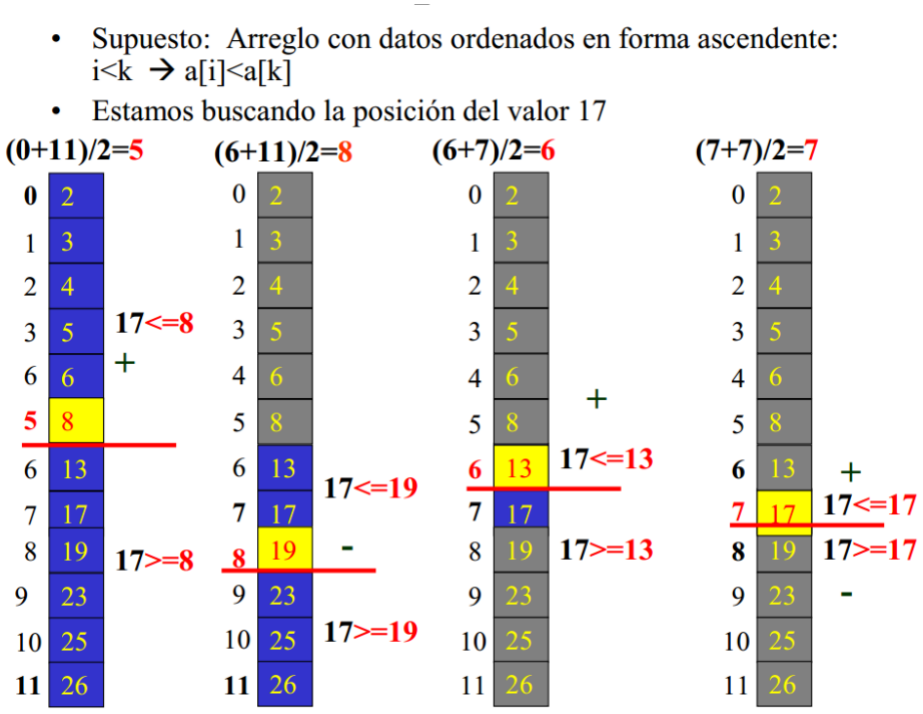
En el peor caso, el elemento buscado está en la última posición.

Complejidad: O(n).

Búsqueda binaria

La búsqueda binaria compara si el valor buscado está en la mitad superior o inferior del vector. En la que esté, subdivido nuevamente, y así sucesivamente hasta encontrar el valor. El vector debe estar ordenado.

Ejemplo:



Complejidad de la búsqueda binaria: O(log n)