Guion Grupo 11 AAD 6

# Presentación

Improvisación

# Selección

## Resumen

Comenzamos con el análisis del algoritmo de selección.  
Este algoritmo posiciona el elemento seleccionado en el lugar correcto dentro del vector. Para hacer esto, lo recorremos entero para encontrar el elemento más pequeño, y nos lo llevamos a la primera posición. Después volvemos a recorrer todo el vector desde la segunda posición hasta la última, volvemos a seleccionar el elemento menor y nos lo llevamos a la segunda posición. Este proceso se repite sucesivamente empezando a hacer el recorrido por el primer elemento no ordenado hasta el último, y llevándonos el elemento menor a la posición de comienzo del proceso, hasta que todos los elementos están ordenados. Esto lo veremos más claro con el ejemplo de ejecución.

## Código

Una vez explicado el procedimiento con palabras, pasemos a traducirlo a pseudocódigo, para después implementarlo en C++.  
Comenzamos el procedimiento pasando a la función por parámetro el array de tamaño n que queremos ordenar. Aquí establecemos un bucle “para” que recorre desde la primera posición (1) hasta la penúltima (n-1), en el que establecemos en un principio la primera posición no ordenada como aquella que contiene el elemento menor. Con otro bucle “para”, recorremos el vector desde la posición siguiente al mínimo que hemos predefinido hasta la última, y si encontrásemos un elemento menor que éste, lo asignamos a la variable del elemento mínimo. Una vez que hemos conseguido hallar el entero más pequeño, con el uso de una variable auxiliar, cambiamos de posición el valor mínimo y el primero no ordenado, quedando así en la posición correcta. La implementación en C++ es similar al pseudocódigo, pero hay una diferencia, en esta línea (señalar segunda) vemos que comenzamos en la posición 0 en vez de en la 1, ya que C++ comienza a numerar las posiciones del vector a partir de 0, por ello decrementamos todos los índices en 1. Se inicia en 0, se termina en size – 2 en vez de en size – 1. En cuanto a lo demás, vemos que se puede implementar directamente tal y como se describe en el pseudocódigo.

## Traza

Vamos a ver ahora la traza del algoritmo y un ejemplo de ejecución. Partiendo de un vector desordenado:

1º Seleccionamos el menor elemento (36) del vector y lo intercambiamos por su posición correcta (la primera).

2º Volvemos a recorrer el vector buscando el menor entre los desordenados (1060), en este caso ya se encuentra en su posición correcta.

3º Recorremos de nuevo el vector buscando el menor elemento (2505) y lo intercambiamos por la primera posición a partir del cual el vector esta desordenado.

4º Nuevamente recorremos la parte del vector desordenada y seleccionamos el elemento menor (4127), en este caso ya se encuentra en su posición correcta.

5º El último elemento queda siempre ordenado, como consecuencia de haber ordenado los anteriores.

Finalmente tendríamos nuestro vector ordenado

Tenemos una animación para mostrar la traza de una forma fácil y sencilla, que además ilustra muy bien el funcionamiento de este procedimiento. La flecha roja sobre el 4 indica que empezamos a recorrer el vector a partir de ahí, en busca de un elemento menor que este. Vemos que, tras terminar, encuentra el 1, que es el elemento más pequeño del array, lo intercambia con el 4, y así tenemos el 1 ya ordenado. Ahora comenzamos en el segundo elemento, volvemos a realizar el recorrido, pero no encontramos ninguno menor, por lo que este también está ordenado. Volvemos a realizar lo mismo empezando por el tercer elemento, encuentra el 3 que es el elemento menor, lo intercambia con el 4 y vemos que ya tenemos el vector ordenado.

## Análisis

Vamos a hacer a continuación un análisis de la eficiencia teórica. En este caso, el algoritmo no detecta si el vector esta total o parcialmente ordenado. Por tanto, siempre realiza el mismo número de comparaciones e intercambios. Las comparaciones tienen (leer fórmulas en la diapositiva)

## Mejoras

Para terminar el estudio de este algoritmo, vamos a proponer dos mejoras con el objetivo de mejorar su eficiencia. La primera sería añadir al algoritmo una comprobación, que impide que se realicen los intercambios si los valores almacenados en las dos posiciones son iguales con una sentencia if (señalar). El resto del algoritmo no se modifica con respecto al inicial.

La segunda mejora consiste en hacer que el algoritmo sea estable, es decir, en hacer que conserve la posición relativa de los elementos con la misma clave.

Si comparamos la eficiencia del algoritmo inicial con los que acabamos de proponer, vemos que estabilizar el algoritmo hace que el tiempo de ejecución se dispare, mientras que realizar la comprobación antes de intercambiar apenas lo modifica.

# Quicksort

## Resumen

Ahora, tras haber estudiado detenidamente el algoritmo de Selección, haremos lo mismo con Quicksort. (>>)

Quicksort es un método de ordenación recursivo, es decir, que se llama a sí mismo, basado en la técnica “divide y vencerás”. Este algoritmo sigue los siguientes pasos:

1º Elegimos un elemento pivote dentro del vector.

2º Se agrupan todos los elementos menores o iguales que el pivote a un lado del vector, y los mayores al otro lado. Tras hacer estos agrupamientos, tenemos el elemento pivote en la posición correcta dentro del array. Así, el vector queda separado en dos subrangos, uno formado por los elementos a un lado del pivote, y otro por los elementos del otro lado.

3º. Repetimos este proceso de forma recursiva para cada subrango mientras éstos contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados. Cabe destacar que la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

## Código

A continuación, el pseudocódigo de Quicksort y su implementación en lenguaje C++.

Tenemos una interfaz, que recibe el vector a ordenar y el número de elementos que contiene. Llama a la función principal; que, además del vector, requiere saber dónde empieza y termina. Se le pasa todo el vector. Desde el primer elemento, que en C es el 0; hasta el último, que en C es el tamaño menos uno.

Si el fin es mayor que el principio, es decir, hay más de un elemento. Busca un punto por el cual dividir el vector y manda a ordenar los subconjuntos por separado.  
En el caso base, un solo elemento a ordenar, no realiza ninguna operación.  
Esta función se traduce directamente a C.

A diferencia de la especificación original; en este caso no se asume un pivote previamente, sino que se toma el primer valor que cumple la condición para ser pivote.  
Guardamos el valor del primer elemento, lo llamaremos X. Tenemos dos contadores, uno desde la primera posición, que avanzará mientras el elemento sea menor que X; el otro desde el final que avanza mientras sea mayor. Al final, selecciona el primer mayor o igual y el último menor o igual. Si no se cruzan, los intercambia; si se cruzan, ha encontrado un pivote.

//Posible aclaración.  
Un punto, desde el cual todos los elementos a la izquierda son menores a él y a la derecha son mayores.

## Traza

Vamos a ver una instancia.  
Esta es la salida del algoritmo mostrando sus acciones, pero la explicaré con una animación.  
Encuentra el 3 y el 2, los intercambia. Encuentra el 4 y el 1, los intercambia. Las selecciones se cruzan, se descubre que el 1 es un pivote.  
Ordenamos los dos primeros. Encuentra el 2 y el 1, los intercambia. Las selecciones se cruzan, descubre que el 2 es un pivote.  
La división genera dos subconjuntos de un elemento, por separado están ordenados.

//Si has gastado más de 1:15 minutos, esto sáltatelo.  
Ordenamos los 3 últimos. Encuentra el 5 y el 3, los intercambia. Las selecciones se cruzan, descubre que el 4 es un pivote.  
Ordena el 3 y el 4. Las selecciones se cruzan, descubre que el 4 es un pivote. Las divisiones son el caso base, quedan ordenadas.  
Ordenar un elemento es trivial.  
Ya esta todo el array ordenado.

## Análisis

La eficiencia depende de lo centrado que quede el pivote. Si fuera justo el centro, el orden del algoritmo sería NlogN, como puede verse en el desarrollo por expansión de recurrencia. Esto es lo que llamaremos caso mejor. Que el array este ya ordenado, no significa que se dé el caso mejor; de hecho, dependiendo del método de elección del pivote, puede ser el peor.

Si los subconjuntos se dividen de la forma menos equilibrada, se necesitan tantas divisiones como elementos, luego el orden es N^2.

F(n) es el tiempo de división, que es lineal. G(n) es la solución directa, es cuadrática; pero Quicksort recurre a ella cuando la talla es, g(1) = 1.  
//Si quedan menos de 30 segundos, saltar esta parte.

El resultado es que se ordenan 2 elevado a i subconjuntos de talla 2 elevado a k-i, siendo k la talla original. Es muy sencillo. Ordenar un vector de 8 elementos es ordenar dos vectores de 4, que es ordenar 2 de 2; pero son 2, luego 4 de 2. Más, al final, 8 veces el caso base.

## Mejoras

Mejora pivote aleatorio:  
Esta mejora se centra en el uso de un pivote aleatorio, esto favorece a que el caso peor sea un caso medio.

Mejora Tail call:   
Reduce el tamaño de stack de N a logN, reduce la recursividad del algoritmo, haciendo que ocupe menos memoria.

Comparación de las mejoras:  
Podemos observar que la mejora del pivote aleatorio ralentiza al código original, pero en cambio la mejora de Tail call consigue mejorar la rapidez del código original.

# Comparación

Respecto a la implementación el método de selección es mucho más sencillo de implementar que el Quicksort.

Ahora usaremos la gráfica para observar la eficiencia de cada algoritmo, en tallas pequeñas, inferiores a 100, Selección es incluso más rápido que Quicksort, pero no es muy significativa. La verdadera diferencia la encontramos cuando las tallas son mayores, donde podemos apreciar que el método Quicksort es muy superior.

# Problema 6

Como hemos podido ver en la comparación de los dos algoritmos, Selección resulta más rápido que Quicksort ene tallas pequeñas. Lo que hace esta modificación, es llamar a Selección cuando el tamaño es menor que una cota dada, la llamaremos M.  
En esta gráfica se observa que el valor óptimo para M está entre 50 y 100.

# Problema 7

Esta modificación detiene la ejecución del algoritmo si detecta que todos los elementos son iguales. La comprobación es rápida y, en caso de detectarlo en una talla alta, curta numerosas ramas; lo que reduciría considerablemente el tiempo.  
En el caso medio habitual, habiendo 10.000 elementos diferentes; este cambio únicamente añade el tiempo de la comprobación. Conforme disminuye el numero de elementos distintos, es más posible que sean todos iguales y la mejora se hace notoria por debajo de los 100 elementos diferentes.

# Conclusiones

Y para terminar, vamos a sacar algunas conclusiones de todo lo que hemos visto.

En primer lugar, se observa que los algoritmos lentos, de orden n2, no pueden competir contra los rápidos, de orden n·log n. Una comparación más justa sería QuickSort vs Mergesort o Heapsort.

En segundo lugar, podemos afirmar que ambos algoritmos son útiles, pero el rendimiento de cada uno varía dependiendo de la situación, ya que la búsqueda por selección es más eficiente en vectores de poco tamaño, debido a que cuanto mayor sea el vector el tiempo de ejecución es mayor. Quicksort, sin embargo, es rápido independientemente del tamaño de este, pero es dependiente del pivote, y un fallo implementando el código es bastante difícil de detectar. Además, el orden de elementos idénticos variaría, cosa que intentamos optimizar en el problema 7, pero vemos que para tallas grandes la diferencia es irrisoria.

Otra conclusión que sacamos es que el método de búsqueda por selección es bastante difícil de mejorar, ya que cualquier optimización empeoraría el tiempo de ejecución. En todo caso, lo único que podríamos hacer es que este algoritmo sea estable. Sin embargo, al método Quicksort si se le puede añadir una mejora: hacer una llamada a cola, consiguiendo disminuir el tiempo de ejecución. La otra optimización planteada, establecer un pivote aleatorio, empeora la eficiencia del algoritmo.

# Despedida (introducción a preguntas)

Improvisación