Práctica 11: Ordenamientos: Parte II

1 Meta

Que el alumno conozca algunos algoritmos de ordenamiento y los implemente para ordenar arreglos de enteros.

2 Objetivos

Al finalizar la práctica el alumno:

- Será capaz de implementar en Java los algoritmos descritos en alto nivel.
- Tendrá las bases para implementar otros algoritmos de ordenamiento para arreglos de enteros.
- Comprenderá la importancia de la complejidad en tiempo de los algoritmos.

3 Antecedentes

Un algoritmo de ordenamiento tiene como objetivo ordenar las entradas de una estructura de datos. En este caso vamos a ordenar los valores de un arreglo de enteros. En la primer parte ya viste dos algoritmos que realizaban esto, pero ambos tienen limitaciones importantes. **Selection Sort** tiene una complejidad $O(n^2)$, y como pudiste experimentar la práctica pasada el tiempo requerido para su ejecución crece bastante a medida que aumenta el tamaño de los arreglos. **Bucket Sort** por otro lado está limitado porque necesita información extra del arreglo (el elemento mínimo y máximo) y que ocupa memoria auxiliar en su ejecución.
En esta segunda parte implementarás quickSort que como su nombre indica, es en promedio uno de los algoritmos de ordenamiento más rápidos.
También implementarás búsqueda binaria, un algoritmo cuya *correctez* depende de que su entrada esté ordenada.

3.1 QuickSort

QuickSort se basa en usar un elemento **pivote** del cual se encontrará la posición en el arreglo ordenado. Se pone el **pivote** en su posición adecuada; a todos los menores o iguales los colocamos a la izquierda; a todos los mayores a la derecha. Una vez hecho esto, se ordena los subarreglos izquierdo y derecho de forma **recursiva**.

En nuestra implementación, el pivote siempre va a ser el primer elemento del arreglo. En el caso del arreglo A ejemplo:

4	7	0	1	5	2	3	-4	1	8	2

El pivote es $A[0] = 4$. Para encontrar la posición de 4 en el arreglo ordenado y mover a los menores a la izquierda y los mayores a la derecha, vamos a recorrer el arreglo en dos direcciones, desde la posición 1 a la **derecha** con un índice i y desde la posición $A.length - 1$ a la **izquierda** con un índice j.

4	7	0	6	5	2	3	-4	1	8	2

Para los valores del índice i (azul) y el índice j (rojo) vamos a considerar 3 casos. El primero de ellos, que es el que se presenta en este punto es: $A[i] > pivote$ y $A[j] \leq pivote$. En este caso, intercambiamos ambos valores:

4	2	0	6	5	2	3	-4	1	8	7

Una vez hecho esto, movemos i a la derecha y j a la izquierda.

4	2	0	6	5	2	3	-4	1	8	7

El segundo caso, que es el que se presenta ahora es: $A[i] \leq pivote$. En este caso únicamente movemos i a la derecha:

4	2	0	6	5	2	3	-4	1	8	7

Finalmente, el tercer caso es el que se presenta ahora: $A[i] > pivote$ y $A[j] > pivote$. En este caso, únicamente movemos j a la derecha:

4	2	0	6	5	2	3	-4	1	8	7

Vamos a repetir esto hasta que $i \geq j$, es decir hasta que i (el azul) esté en la misma posición o a la derecha de j. En este caso, si repetimos el proceso ambos quedan sobre la misma posición:

4	2	0	1	-4	2	**3**	5	6	8	7

Al final de esto, vamos a intercambiar El pivote $A[0]$ con $A[i]$:

3	2	0	1	-4	2	**4**	5	6	8	7

Existe **otro** caso para este intercambio. Si $A[i] > A[0]$. Por ejemplo, si el ejemplo anterior hubiera terminado en:

4	2	0	1	-4	2	**9**	5	6	8	7

Entonces tenemos que mover i a la **izquierda** antes de hacer el intercambio:

4	2	0	1	-4	2	**9**	5	6	8	7

E intercambiamos, resultando en:

2	2	0	1	-4	4	**9**	5	6	8	7

En cualquiera de los dos casos, el pivote queda en su posición del arreglo ordenado. Y a partir de este punto, todos los elementos a la izquierda del pivote son menores o iguales. Aquellos a la derecha son mayores o iguales. De manera recursiva, ordenamos usando QuickSort el subarreglo izquierdo (que va de 0 hasta $i-1$) y el derecho (que va de $i+1$ hasta $A.length-1$).´

3.2 Búsqueda Binaria

Deberás implementar este algoritmo de forma **recursiva**.
Este algoritmo ya lo viste en el contenido teórico, por lo que la explicación se hará de manera muy breve:
Busqueda Binaria recibe un arreglo y un entero, y buscará dicho entero en el arreglo. Si el entero es un elemento del arreglo, deberá regresar el índice de ese elemento en el arreglo. Si no es un elemento del arreglo, deberá regresar -1.
Busqueda Binaria, con entradas A (un arreglo) y *elem* un entero:

1. Obtiene m, el índice que corresponde a la mitad del arreglo A.

2. Si *elem* es igual a $A[m]$, regresa m.

3. Si $elem < A[m]$ regresa el resultado de la búsqueda binaria de *elem* en la primer mitad del arreglo A.

4. Si $elem > A[m]$ regresa el resultado de la búsqueda binaria de *elem* en la segunda mitad del arreglo A.

4 Desarrollo

Deberás implementar en las funciones `busquedaBinaria` y `quickSort` los algoritmos correspondientes. Si el algoritmo implementado no corresponde al nombre de la función no será tomada en cuenta.

La implementación de `busquedaBinaria` debe hacerse **forzosamente** de manera recursiva. La recomendación que les damos es que creen una *función auxiliar* que sea recursiva y que pueda recibir los parámetros que necesiten. No podrás usar `for` o `while` ni en la función `busquedaBinaria` ni en la auxiliar; si lo haces tus funciones no serán tomadas en cuenta.

Además debes implementar la función auxiliar `intercambia` que intercambia los valores de dos índices de un arreglo.

ESTÁ PROHIBIDO:

- Modificar las funciones que no tienen el comentario *"//Implemente esta función".*
- Importar clases o funciones externas adicionales.
- Modificar las firmas de las funciones que se les proporcionan. Si quieren añadir más o menos parámetros ocupen funciones auxiliares.

Consejo para la implementación de `quickSort`: un arreglo de longitud 0 o menor (un arreglo vacío) **siempre** está ordenado.

Se te proporcionan **pruebas unitarias** para verificar que sus algoritmos estén funcionando correctamente. Las pueden correr con el comando `ant test`. Es normal que los test fallen si no han terminado su código, pero una vez que hayan implementado un algoritmo, las pruebas correspondientes deben pasar.

5 Entregables

- Deberán entregar el código con las funciones que se les piden implementadas.
- Las respuesta a las pregunta:
  - Observa los tiempos que toma ejecutar quickSort y compáralos con los tiempos de la ejecución de Selection Sort que obtuviste en la práctica pasada.
    ¿Esta información es suficiente para concluir algo respecto a la complejidad en tiempo de quickSort? Si tu respuesta es afirmativa, explica qué puedes concluir. Si tu respuesta es negativa, justifica.