Antecedentes II

En la primera parte ya viste dos algoritmos para ordenar los valores de un arreglo de enteros, pero ambos tienen limitaciones importantes. ***Selection Sort*** tiene una complejidad $\mathcal{O}(n^2)$ y, como pudiste experimentar en la parte pasada, el tiempo requerido para su ejecución crece bastante a medida que aumenta el tamaño de los arreglos. ***Counting Sort*** por otro lado está limitado porque necesita información extra del arreglo (el elemento mínimo y máximo) y ocupa memoria auxiliar en su ejecución.

En esta segunda parte implementarás el ordenamiento veloz (***Quick Sort***) 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.

Ordenamiento veloz

El ordenamiento veloz (*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 ordenan los subarreglos izquierdo y derecho de forma **recursiva**. En nuestra implementación, el pivote siempre va a ser el primer elemento del arreglo.

Ejemplo 7.3. *Sea el arreglo* A*:*

4	7	0	6	5	2	3	-4	1	8	2
0	1	2	3	4	5	6	7	8	9	10

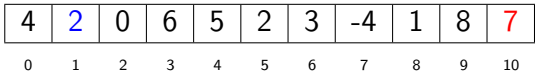
El pivote es `A[0] = 4`.

Para encontrar la posición del pivote 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`.

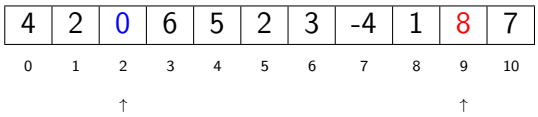
En el ejemplo, el índice `i` inicia en el valor `1`, el `j` en `10`.

4	7	0	6	5	2	3	-4	1	8	2
0	1	2	3	4	5	6	7	8	9	10
	↑									↑

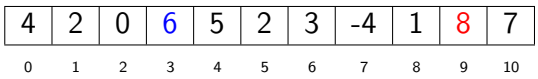
Para los valores del índice `i` (azul) y el índice `j` (rojo) vamos a considerar 3 casos. El primero de ellos, que es el que se presenta en este punto es: `A[i] > pivote` y `A[j]` $\leqslant$ `pivote`. En este caso, intercambiamos ambos valores:



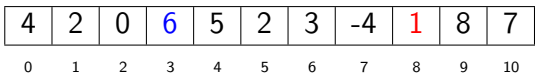
Una vez hecho esto, movemos i a la derecha y j a la izquierda.



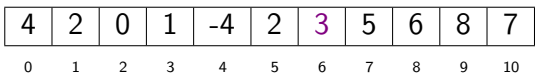
El segundo caso, que es el que se presenta ahora es: `A[i]` $\leqslant$ `pivote`. En este caso únicamente movemos i a la derecha:



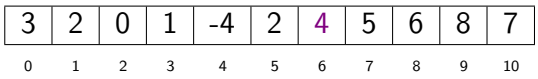
Finalmente, el tercer caso es el que se presenta ahora: `A[i] > pivote` y `A[j] > pivote`. En este caso, únicamente movemos j a la izquierda:



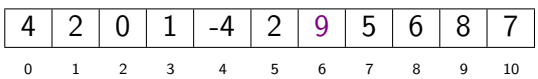
Vamos a repetir esto hasta que i $\geqslant$ j, es decir hasta que i (el azul) esté en la misma posición o a la derecha de j. En este caso, si repetimos el proceso ambos quedan sobre la misma posición:



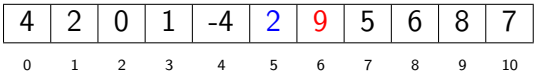
Al final de esto, vamos a intercambiar El pivote $A[0]$ con $A[i]$:



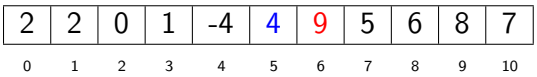
Existe **otro** caso para este intercambio. Si $A[i] > A[0]$. Por ejemplo, si el ejemplo anterior hubiera terminado en:



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



E intercambiamos, resultando en:



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 *Quick Sort* en el subarreglo izquierdo (que va de 0 hasta $i-1$) y el derecho (que va de $i+1$ hasta $A.length-1$).

Búsqueda binaria

Deberás implementar este algoritmo de forma **recursiva**. Búsqueda 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, para ser implementado recursivamente requiere ser programando con dos funciones. La primera, pública, con entradas A (un arreglo) y $elem$ el entero que se está buscando, esta función sólo manda llamar a la otra comenzando con una búsqueda en el arreglo completo. La segunda, privada, es la versión recursiva que resuelve el problema, esta necesita como entradas A (el arreglo), $elem$, $inicio$ el índice del extremo izquierdo del subarreglo donde se realiza la búsqueda y fin el índice del extremo derecho:

1. Si $inicio > fin$ regresa -1 pues no logró encontrar a $elem$.
2. Obtiene $m = (inicio + fin)/2$, el índice que corresponde a la mitad del subarreglo de A.
3. Si $elem$ es igual a $A[m]$, regresa m.
4. Si $elem < A[m]$ regresa el resultado de la búsqueda binaria de $elem$ en el subarreglo que va de $inicio$ a $m-1$.
5. Si $elem > A[m]$ regresa el resultado de la búsqueda binaria de $elem$ en el subarreglo que va de $m+1$ a fin.

Desarrollo II

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

La implementación de `búsquedaBinaria` debe hacerse forzosamente de manera recursiva. La recomendación que te damos es que crees una *función auxiliar* que sea recursiva y que pueda recibir los parámetros que necesites. No podrás usar `for` o `while` ni en la función `búsquedaBinaria` 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 la función.`
- Importar clases o funciones externas adicionales.
- Modificar las firmas de las funciones que se te proporcionan. Si quieres añadir más o menos parámetros ocupa 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 tus algoritmos estén ordenando un arreglo correctamente. Las puedes correr con el comando `ant test`. Es normal que las pruebas fallen si no has terminado tu código, pero una vez que hayas implementado un algoritmo, las pruebas correspondientes deben pasar.

Entregables II

1. Deberás entregar el código con las funciones que se piden implementadas.
2. La respuesta a la pregunta:
   - Observa los tiempos que toma ejecutar `quickSort` y compáralos con los tiempos de la ejecución de *Selection Sort* que obtuviste en la parte I.

     ¿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.