Comp 2140 - Data Structures
Lab 3 - Quick Sort.

‘Play is our brain’s favorite way of learning’ Diane Ackerman

Please print your name and your student id in this box. Return this paper (with your answers to the written questions) to the TA at the end of the lab. The TA will comment on your programming assignments, also on this paper.

your name: your id:

The objective of this lab is to explore the quick sort algorithm.
Please take the following steps in this lab. Download the file Lab3.java. This file contains partial implementation of quick-sort (including partition method) along with methods for initializing arrays randomly and printing them.

1. Take a look at the method `public static void testPartition()`. This method creates an array of size \( n = 10 \), places random integers in the range \([0..10]\) in \( a \), and prints it. Then, it calls `partition (a, 0, n-1, 0)` and prints the output. Note that the pivot's index is 0, i.e., the pivot is the first element in the input. Run the code 5 to 10 times. Explain whether the output is always as you expect or not. If not, I) in the box below write an example that you encountered for which the partition is not working properly (before and after partition). II) then go through the partition method and fix the error. Show the result of your correct test to the TA.

   Answer: The output is Not as expected. Here is a possible output:
   before partition:
   0 7 4 4 8 10 6 3 2 1
   after partition around 0:
   7 0 4 4 8 10 6 3 2 1

   Note that 0 (the pivot here) is not in its correct position and 7 has appeared before 0 while it had to be on the right of 0 after partitioning around 0. The problem is that `finalpivot` is blindly set to `right` while it should be `right-1` if \( A[right] > A[lo] \). In the above example, pivot \( (A[lo]) \) is smaller than \( A[right] \) and it should be swapped by the index before 7 (the 0 itself). To get the full mark, you had to indicate that the answer is not what you expect and fix the code.

2. Note that code for `quickSort (int [] A)` and partial code for `quickSortAux ( int [] A, int lo , int hi)` are provided. But in the code for `quickSortAux` the pivot is not set (its value is -1) and the code is missing a line (one of the recursive calls). Set the value of the pivot to be the first element in the subarray that is being sorted (what is that index?), and add the missing recursive call to `quickSortAux`. Then uncomment `test1()` in the main function. Note that `test1` creates an array of size \( n = 10 \), places random integers in the range \([0..100]\) in \( a \), and prints it. Then, it runs quick-sort on the array and prints the sorted array. Show the result of your working code to the TA. Does it matter which recursive calls are made first? (write your Yes/No answer in the box below):

   Answer: The missing lines are: `int pivot = lo;` and `quickSortAux (A, pivot + 1, hi);`
   It does not matter which recursive call is made first (but they obviously have to be made after partitioning the array).
3 Uncomment test2 in the main function. This method creates two arrays $A$ and $B$ of $n = 100,000$ random integers. $A$ is initiated with random numbers while $B$ is initially sorted. test2 runs the quick-sort to sort these two arrays. The time that it takes to run the quick-sort in the two cases is reported in milli-second. Note that your pivot is chosen to be the first element in the array. Do you think the first or the second run of the function quickSort take more time? Show your numbers to the TA and write them in the following box; justify your observation in 2-3 sentences.

**NOTE:** Depending on your system setting, it is likely that when you run `java Lab3`, the code crashes. This happens when the stack of your system becomes full (which is very likely for large values of $n$). If you encountered this (interesting/disappointing) phenomenon, you can run `java -Xss9m Lab3` to increase the size of your stack.

```
Answer: The numbers are much higher when the array is sorted (second run). This is because when the array is sorted, each recursive call decrease the size of sub-arrays by 1, and the running time becomes $O(n^2)$ while in the random case the running time is $O(n \log n)$. See the sample output below.
```

4 Before doing this part, make sure to show your work for previous parts to the TA. Change the quickSortAux so that the pivot is a random value in the range $[lo, hi]$. Also, increase $n$ from 100,000 to 10,000,000. As we saw in the class you can do it using `int pivot = lo + (int)(Math.random() * (hi - lo + 1))`. Repeat part [3] (running test2) with these updates and report the running times the random and sorted cases. Show your numbers to the TA and write them in the following box; justify your observation in 2-3 sentences.

```
Answer: The numbers are asymptotically similar because the algorithm runs in $O(n \log n)$ in both cases. Note that the algorithm that is called second might take a constant factor (e.g., three times) less time. That is because compiler optimizations. (as long as you observe similar numbers and mention that the two cases have similar time complexity (or number of recursive calls), you will get the full mark).
```