Dr. Hasmik Gharibyan
CSC/CPE349

Project 1: Empirical Analysis of Algorithms

Deadlines: submit your files **electronically** by **midnight** (end of the day) on **Monday, 4/12/21.**

Late submission: you can submit your work within 24 hours after deadline (penalty applies): by 2 a.m. on 4/13/21 for **10-point penalty**; from 2:01 a.m. till *noon* on 4/13/21 for **25-point penalty**, from 12:01 p.m. till *midnight* (end of the day) on 4/13/21 for **50-point penalty.**

Programming Language and Environment: All programs should be written in ***Java***. You can use any Java program development environment you are comfortable with (any text editor, any IDE you like). However, you must make sure your programs can be compiled and executed from the command line on our departmental servers as the grading will be done in this environment.

> **DON'T CHANGE** any class/method names, sequence of steps in the assignment. Failure to do so will hinder the grading process and will end in penalty.

Goals of the assignment:

1. Review algorithms learned in previous courses: selection sort, bubble sort, mergesort, quicksort.
2. Implement algorithms constructed by different strategies, using different techniques. Selection and bubble sorts are iterative algorithms while mergesort and quicksort both are recursive algorithms representing two great examples of divide-and-conquer strategy.
3. Analyze performance of implemented algorithms empirically, observing differences of their performances: selection and bubble sorts are $O(N^2)$ algorithms while mergesort and quicksort are O(NlogN) algorithms[1]. You will evaluate their performance based on their running time.

---

[1] **Reminder**: quicksort's worst case takes $O(N^2)$ time. However, with some effort the worst case is made exponentially unlikely which means that quicksort's expected performance is as in its average case and thus it is classified as O(NlogN) algorithm.

PART 1: I. Design and implement a class named *Sorts*

The *Sorts* class contains the implementation of **4 sorting algorithms** and **NOTHING else** (the implementation of these algorithms must be **consistent** with the posted handout). This class does NOT have any instance or class variables, any constants (public or private), or any *public* methods except the ones stated below. *Sorts* class contains only **four *public* methods** for the four sorting routines **plus** all necessary ***private*** methods that support them. The four public methods are:

public static void selectionSort (int[] arr, int N)
//Sorts the list of *N* elements contained in *arr[0..N-1]* using the ***selection sort*** algorithm.

public static void bubbleSort (int[] arr, int N)
//Sorts the list of *N* elements contained in *arr[0..N-1]* using the **improved *bubble sort*** algorithm
//(see the handout).

public static void mergeSort (int[] arr, int N)
//Sorts the list of *N* elements contained in *arr[0..N-1]* using the ***merge sort*** algorithm.

public static void quickSort (int[] arr, int N)
//Sorts the list of *N* elements contained in *arr[0..N-1]* using the ***quick sort*** algorithm **with**
//**median-of-three pivot and rearrangement of the three elements** (see the handout)**.**

Notes: 1. The list given as a parameter may not fill the *arr* array: it occupies *arr's* 0 to *N-1* cells.
2. No need for validity check – you can assume that *arr.length* >0 and *N* <= *arr.lenght*.

Attention: you may **NOT** change the names or signatures of specified four **public** methods; **BE CAREFUL** about the upper/lowercase letters in method names (misspelled name will make the grading program fail). You can define *private* methods as you see fit.

II. Design and implement an application class (or classes) to test your 4 sorting methods and make sure they work correctly. Run *a lot of tests* for each sort.

Attention: Do **NOT** submit this application class (or classes) – it only serves as a testing tool for *you*. **The grader will run his own driver to test your methods.**

PART 2: Design and implement an application class named *SortTimes* to generate the running times of *Sorts* class' 4 routines.

This class only needs a *main* method designed to run *Sorts* class' public methods multiple times for different size lists and **output the execution time** in **milliseconds** for each method in each run.

Attention:
1. To get the execution time of a method, you can capture the system time immediately before and immediately after the method call, and then take the difference.
2. To obtain the **system time**, for more accuracy, use *System* class' *static* method *nanoTime()* which returns the system time in nanoseconds as a ***long*** type value. To obtain the running time in **milliseconds**, you need to **divide the obtained running time by 1,000,000**. Note: time values should be represented as **integer numbers.**

Here is the description of steps that need to be done in *main* (variable N represents the number of elements in the list).

1) Define 4 arrays for **160,000** ***int*** type elements (the **max** size list you need to sort is 160,000).
Notes:
1. You **need 4 copies** of the original list: a sorting method alters the original list (sorts it), therefore you cannot use it as an input list for another method, hence the need for 4 copies of the original list. To hold 4 copies of the list, you need 4 arrays so you can pass different copies/arrays to different sorting methods.
2. To test an algorithm on different size lists, it's more efficient to have **one** array of **max** size (and fill it partially) rather than using a different N-size array for each N.

2) For different values of N (starting with 5,000 and doubling N's value until 160,000 **including**), do the work defined below:

Repeat the following steps 5 times (i.e. 5 times for each N-value):

a) Create 4 copies of a list of N random integer values in the range [0, N-1], saving them in the first N cells of each of the four arrays (i.e. repeatedly select/generate a random number and save it under the **same** index **in each** of the four arrays).
Note: to obtain random integers, you can define a *Random* class object and call its *nextInt* method. You can also use *Math* class's *random()* method; in this case make sure to multiply the result by N and then cast it to an integer.

b) Run each of the 4 methods of *Sorts* class (*selectionSort, bubbleSort, mergeSort,* and *quickSort*); make sure to **give different arrays** as a parameter **for different sorts.** Make arrangements to compute the execution time of each method (as described above). Output **ONE line** containing the value of N, followed by the execution times of all 4 sorts, separated by commas or spaces, in a descriptive sentence. For example, your output line may look like this:

N=*value***: T_ss=***time1***, T_bs=***time2***, T_ms=** *time3***, T_qs=** *time4*

where *value* is the current value of N, and *time1*, *time2*, *time3*, *time4* are the execution time values for *selectionSort*, *bubbleSort*, *mergeSort,* and *quickSort* correspondingly.

OUTPUT: the output produced by your *SortTimes* program should look something like this:

Running Times of four sorting algorithms:

N=*5000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*5000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*5000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*5000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*5000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

N=*10000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*10000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*10000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*10000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*10000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

N=*20000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*20000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*20000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*20000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*20000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

N=*40000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*40000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*40000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*40000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*40000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

N=*80000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*80000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*80000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*80000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*80000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

N=*160000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*160000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*160000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*160000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*
N=*160000*: **T_ss**=*time1*, **T_bs**=*time2*, **T_ms**=*time3*, **T_qs**= *time4*

End of output

Note: of course instead of all *time1, time2, time3, time4* terms you'll have actual **integer** numbers:

Attention: Your output should be **EASY TO READ**; e.g. print a blank line at the end of each iteration of the N-loop (as shown in the above example).

IMPORTANT!

1. **ANALYZE YOUR OUTPUT THOROUGHLY.** Make sure that your output reflects the expected behavior for each algorithm. Remember that selection and bubble sorts are expected to execute in $O(N^2)$ time while mergesort and quicksort are expected to do the job in O(NlogN) time. You should notice that (i) the bubble sort is noticeably **slower** than the selection sort due to the large number of swaps it has to do at each iteration, and (ii) the mergesort is noticeably **slower** than the quicksort due to the fact that at each step of recursion it uses a temporary array for merging the two sorted halves of the list-segment.

2. Prepare a text file called ***times.txt*** containing the output produced by *SortTimes* program. **You will need to submit this file together with your program.** You can get your output in a file if you run the program on the command line and direct your output to the mentioned file:
   javac SortTimes.java
   java SortTimes > times.txt

Submitting your work:

Turn in the following 3 files electronically via "*handin*" procedure by the deadline (see the top of the first page of this document for the due date and time):

1. **Two** source files called ***Sorts.java*** and ***SortTimes.java*** containing *Sorts* and *SortTimes* classes.
2. **One** text file called ***times.txt*** containing the outputs produced by the *SortTimes* program.

The account *id* is: ***hg-cpe103*** (please do not be concerned with the "weird" account name)

The *name* of the assignment is: ***Project1***

The *name* of the assignment for late submissions is: ***Project1_late.***

Important:

1. Your programs must **compile and run from command line** on our departmental servers. So, before submitting, make sure your programs compile and run on *unix* servers.
2. Submit **individual files** – do **NOT** create packages for your source code, do **NOT** zip, do **NOT** use folders for your submission (all these complicate the automated grading process).
3. Each file must have a **comment-header** which should include both authors' **names** and **ids** (as in ***id***@*calpoly.edu*), as well as the **date** and the **assignment name** (e.g. *Project 1*).

   Reminder: only one submission needs to be made for a pair/team.
4. The header of each file must **start at the first line** (do **NOT** put any empty lines or *import* statements before the header).
5. Leave **at least one empty line** after the header, before the first line of the code.