Dr. Hasmik Gharibyan
CSC/CPE349

Project 1: Empirical Analysis of Algorithms

Deadlines: submit your files **electronically** by **midnight** (end of the day) on **Friday, 10/5/18**

Late submission: you can submit your work within 24 hours after deadline (penalty applies):
- by 2 a.m. on Saturday, 10/6/18, for **25 point penalty**
- from 2:01 a.m. to midnight (end of the day) on Saturday, 10/6/18, 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 result in penalty.

Goals of the assignment:

1. Review some sorting algorithms learned in previous courses (CPE102-103 or CPE202-203): selection sort, mergesort, and quicksort.
2. Implement algorithms constructed by different strategies, using different techniques. Selection sort is an iterative algorithm while mergesort and quicksort both are recursive algorithms representing two great examples of divide-and-conquer strategy.
3. Analyze the performance of implemented algorithms empirically, observing differences of their behavior; note that selection sort is an $O(N^2)$ algorithm while mergesort and quicksort both are O(NlogN) algorithms[1]. You will evaluate performance of algorithms in two different ways: 1) via their actual running time and 2) via number of *element*-comparisons during the execution.

---

[1] Reminder: quicksort's worst case takes $O(N^2)$ time. However, with some effort the worst case is made exponentially unlikely so quicksort's typical performance is expected to be no worse than its average case. Thus it is classified as an O(NlogN) algorithm.

PART 1

I. Design and implement a class named *Sorts.*

The *Sorts* class contains the implementation of **3 sorting algorithms** and **NOTHING else.**
ATTENTION: You are **required** to use these algorithms **as described** in the provided **handout.**

The *Sorts* class does NOT have any instance or class variables, constants, or any *public* methods other than the ones stated below. This class contains only **three *public* methods** for three sorting routines, **plus** all necessary ***private*** methods that support them. The three 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 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 three **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 3 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 yourself. **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' 3 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, you need to 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 **integers**.

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 3 arrays for 160,000 ***int*** type elements (the **max** size list you need to sort is 160,000).
Notes: 1. You **need 3 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 3 copies of the original list. To hold 3 copies of the list, you need 3 arrays so you can pass different copies/arrays to different sorting methods.
2. To test an algorithm on different size lists, it is more efficient to use **one** array of **max** size rather than using different N-size arrays for each value of 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) Fill in the first N cells of your 3 arrays with 3 copies of a list of N random integer numbers in the range [0, N-1] (i.e. repeatedly select/generate a random number and save it under the **same** index **in each** of the three 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) Call each of the 3 methods of *Sorts* class (s*electionSort, 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 3 sorts, separated by commas and spaces. Your output line should look like this:

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

where *value* is the current value of N, and *time1*, *time2*, *time3* are the execution time values for sorting methods *selectionSort*, *mergeSort*, and *quickSort* correspondingly (the order of terms and the formatting must be as shown; for spacing use 2 to 4 spaces)

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

Running Times of three sorting algorithms:

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

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

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

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

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

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

End of output

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

Attention: Your output should be **EASY TO READ** so 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 sort is expected to execute in $O(N^2)$ time while mergesort and quicksort both are expected to do the job in O(NlogN) time. You will notice that the quicksort will actually be **faster** than the mergesort due to the fact that at each step of recursion mergesort deals with a temporary array (when merging two sorted halves of a list segment, elements are saved in a temp array and later copied back into the main array).

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

PART 3

I. Design and implement a class named *Sorts1*.

This class is very similar to the *Sorts* class: it contains the routines implementing algorithms for *selectionSort*, *mergeSort*, and *quickSort* (3 public methods plus their supporting private methods). **The difference is** that for each algorithm you must keep track of and **return** the number of **element-comparisons** performed in it (i.e. for each algorithm you need to count every step that compares **two elements of the list** and return that count). To implement this change:

1. *Sorts1* class's 3 *public* methods (see page 1) will **return** a ***long*** value instead of being *void*:
2. *Sorts1* class may have a ***private*** **class**-variable(s) to carry out the accounting.

II. Design and implement an application class named *SortCounts* to generate the average number of element-comparisons of *Sorts1* class' 3 algorithms.

This class only needs a *main* method designed to run *Sorts1* class' three routines multiple times for different size lists and **output the average number of element-comparisons performed by** each algorithm for each N-value (as before, N is the number of elements in the list). Thus you are going to evaluate the average running times of these algorithms expressed in terms of the number of element-comparisons.

This program is very similar to *SortTimes* program above, except for the **following differences**:

- In step 1) define 3 arrays of size 12,800 (instead of 160,000).
- In step 2) take N's values to start with 100 and double N's value until 12,800 **including**.
- In step 2) for each value of N run the three algorithms 100 times (instead of 5).
- For each value of N output only **one** line including the **average** number of **element-comparisons** performed by each algorithm over 100 test runs (for each N-value compute and output one number per algorithm to represent the average of 100 test runs).

OUTPUT: the output of your *SortCounts* program should look something like this:

Average number of element-comparisons in three sorting algorithms:

N=*100***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*200***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*400***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*800***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*1600***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*3200***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*6400***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*
N=*12800***: C_ss=***number1***, C_ms=***number2***, C_qs=** *number3*

End of output

Note: of course instead of all *number1, number2, number3* terms you will have actual numbers:

IMPORTANT!

1. **ANALYZE YOUR OUTPUT THOROUGHLY.** Make sure your output reflects the expected behavior for each algorithm. Note that quicksort does more comparisons than mergesort.

2. Prepare a text file ***counts.txt*** containing the output produced by your *SortCounts* 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 SortCounts.java
   java SortCounts > counts.txt

Submitting your work:

Turn in the following 6 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. **Four** source files containing classes**:** *Sorts, Sorts1, SortTimes, SortCounts.*
2. **Two** text files called ***times.txt*** and ***counts.txt*** containing the outputs produced by the *SortTimes* and *SortCounts* programs respectively.

The account *id* is: ***hg-cpe103*** *//I know, it is weird, but I have to use this grader account for now.*

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 lab machines.
2. Do not create packages for your source code and do not use folders for your submission – it complicates 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** (no empty lines or *import* statements before it). Leave at least one empty line after the header, before the first line of the code.