CS1110 Lab 11. Timing execution Fall 2011

Name _____ NetId _____

Introduction

The goal of this lab is to show you how to time execution of a program and, with this new skill, to investigate the difference in execution time between linear search and binary search, between selection sort and insertion sort, and between insertion sort and quicksort. Have a blank sheet of paper ready to write on. Start a new folder for the .java files for this program and download these files from the course web page for labs: Sorting.java TestArrays.java

Step 1. Learn about class Date

Package `java.util` contains a class `Date`, which can be used to record the current time. An instance of this class records a date in milliseconds (there are 1,000 milliseconds in a second) since 1 January 1970 (Greenwich mean time). Since a day has 24 * 60 * 60 * 1,000 = 86,400,000 milliseconds, a lot of milliseconds have flowed through your clock since 1 January 1970! So, use type **long** to record such a number.

To see how this class is used, look at method `times` in class `TestArrays`. The body includes two statements:

```
// Store in timeStart a Date with the time at which the statement is executed.
Date timeStart= new Date();

// Store in timeEnd a Date with the time at which the statement is executed.
Date timeEnd= new Date();
```

The next set of statements prints the values of these times in two forms: first, using method `toString` of class `Date` (applied automatically); second, as an integer, which is obtained using function `getTime`. So now, in case you were curious, you know how many milliseconds have elapsed since 1 January 1970 (divide by 1000 to get the seconds).

To see the results of execution of these statements, execute a call on procedure `times`. You'll see the results in the Java console. Note that the two times are exactly the same (or differ by at most 1). It takes very little time to create a new `Date` object and store its name in a variable.

Now, this determination of execution time is not exact: the computer is handling many chores at the same time —various bookkeeping things, allocating memory, dealing with the hard disk, communicating with the internet, repainting components on the computer monitor, etc.— and all this processing is included in the execution time. Nevertheless, this determination of execution time is good enough for our purposes of showing the difference between algorithms.

Class `Date` is *deprecated* (obsolescent, lessened in value, replaced by something better), but it can still be used. The problem with `Date` is that it doesn't extend easily to international times. It is fine for our use here. If compiling the program results in warnings about deprecated classes and methods, you can turn the warning off using the `Edit -> Preferences` menu item.

Step 2. Experiment with searches

The first experiment to run compares linear search with binary search. Look at procedure `testSearches` and its specification. It executes linear search `m` times on the million-element array `b` and then executes binary search `m` times on `b`. Execute the call `TestArrays.testSearches(10);` and see what is printed in the Interactions Pane (and Java console).

The number 10 for `m` may be too small to see any results. It depends on how fast (or slow) your machine is. Increase it to 50, to 100, etc. until it takes between 5 and 10 seconds for linear search. When you get a reasonable number, write down `m` on your sheet of paper.

To get a non-zero reading for binary search, keep increasing the value of `m`. For this purpose, you may want to fix the linear search experiment so that it always does linear search 0 times, so you don't have to wait so long. How many times did binary search have to execute in order to get an elapsed time of around 5 or ten seconds? Write down these facts. Isn't it interesting how fast binary search really is compared to linear search?

Step 3. Experiment with insertion sort and selection sort

Study method `testSorts` and its spec. It creates an array of size 10,000, and then:

(0) runs selection sort `m` times on array `b`, each time initializing the array to random values.

(1) runs insertion sort `m` times on array `b`, each time initializing the array to random values.

Execute a call on `testSorts` with argument 5. Try higher values, like 10, 20, 30, etc., until it takes about 10-15 seconds to execute. Remember, we don't know how fast your computer is. (Computing this many random numbers takes a bit of time—try commenting out the calls to the sorting algorithms to find out how much time this takes.)

Write down on a piece of paper the value `m` and the times for each of the sorts.

Step 4. Experiment with sorting an already-ascending array

In method `testSorts`, method `fillRand` is used to fill array `b` with random values. There is also a method `fillPos`, which fills the array to {0, 1, 2, 3, ...}. Change the method call `fillRand();` (in two places) to `fillPos();` , so that both selection sort and insertion sort will work on arrays that are already sorted and run the experiment again.

You will see that insertion sort takes a lot less time! Keep increasing `m`, the number of times each sorting method is executed, until finally you have a nonzero number for the insertion-sort time. Write down the results of the experiment on your sheet of paper.

Figure out **why** insertion sort is so quick when the array is already sorted. This requires looking at the code of insertion sort and the method it calls and determining what happens if the array is already sorted. Write your explanation on your sheet of paper.

Step 5. Experiment with insertion sort and quick sort

Study method `testSorts2` and its spec. It creates an array of size 75,000, and then:

(0) runs selection sort `m` times on array `b`, each time initializing the array to random values.

(1) runs quicksort sort `m` times on array `b`, each time initializing the array to random values.

Execute a call on `testSorts2` with argument 1 —that should be high enough. It may take 30 seconds to a minute to time selection sort. Remember, we don't know how fast your computer is.

Write down the value `m` and the times for each of the sorts. Remember, for an array of size `n`, selection sort takes time proportional to `n*n` and quicksort, on average, takes time proportional `to n * log n`.

If you have some extra time, repeat the already-sorted test from Step 4 with insertion sort and quicksort. What happens? Why? (Look at the code to determine what happens when the input is sorted.)