# CS 1110, LAB 12: TIMING EXECUTION

**Name**: _____ **Net-ID**: _____

There is an online version of these instructions at

http://www.cs.cornell.edu/courses/cs1110/2012sp/labs/lab12.php

You may wish to use that version of the instructions.

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.

**Requirements For This Lab.** There are two files necesssary for this lab, and they are all available from the online version of these instructions at the course web page. You should create a new directory on your hard drive and download the following five files into this directory:

- `Sorting` (http://www.cs.cornell.edu/courses/cs1110/2012sp/labs/lab12/Sorting.java)
- `TestArrays` (http://www.cs.cornell.edu/courses/cs1110/2012sp/labs/lab12/TestArrays.java)

Despite the importance of these files, there is no code to turn in for this lab. The purpose of this lab is to experiment with the algorithms in these Java files; you are to write down your findings on this worksheet. When you are done, you should show your instructor the contents of the sheet. If you do not finish the lab, then you need to finish it **by the beginning of lab next week** and show it to your instructor at that time.

---

## The 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! Because there are so many milliseconds, you need type **long** to record such a number.

To see how this class is used, look at method `times()` in class `TestArrays`, which has 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 will 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.

For really, really fast operations, we can count the number of *nanoseconds* that have passed. However, in general we do not both with this because none of these ways to determine execution time is really 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, and so on. All of this processing is included in the execution time. This is why we content ourselves with looking at milliseconds, not nanoseconds, as a "rough guess" of how fast a program executes. In particular, it will be good enough to show the difference between the various algorithms in this lab.

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 does not 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.

---

## Exercise 1: Experiment with Searches

The first experiment to run compares linear search with binary search. Look at procedure `testSearches(int)` and its specification. It executes a 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` and the result that you get.

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 part so that it executes 0 times; that way you do not 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?

We talked in class about how much faster binary search is than linear search. This makes that clear.

---

## Exercise 2: Experiment with Insertion Sort and Selection Sort

Study method `testSorts(int)` and its specification. It creates an array of size 10,000, and then:

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

Execute the call `testSorts(5)`. Try higher values for the argument – like 10, 20, 30, and so on – until it takes about 10-15 seconds to execute. Remember, we do not know how fast your computer is. Write down the final value for `m` and the times for each of the sorts.

Some of the time involved is just in computing the random numbers. Try commenting out the calls to the sorting algorithms to find out how much time this takes for you chosen value of `m`.

---

## Exercise 3: Experiment with Sorting an Already-Ascending Array

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

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 below.

---

## Exercise 4: Experiment with Insertion Sort and Quick Sort

Study method `testSorts2(int)` and its specification. It creates an array of size 75,000, and then:

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

Execute a call on `testSorts2(int)` with argument 1 – that should be high enough. It may take 30 seconds to a minute to time selection sort. Remember, we do not 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 $n^2$ 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 quick-sort. What happens and why? Look at the code to determine what happens when the input is sorted.