CS 1110, LAB 12: TIMING EXECUTION

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

There is an online version of these instructions at

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

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.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab12/sorting.py)
- `lab12.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab12/lab12.py)

Despite the importance of these files, there is no code to turn in for this lab. In fact, this **lab is entirely optional**. With that said, you will find the exercises useful for understanding the various sorting algorithms, which will appear on the final.

The Module `time`

The module `time` has several functions to record the current time. For example, the function `time()` records the current time in seconds since since 1 January 1970 (Greenwich mean time). The value returned is a float, as sometimes you want determine the time in intervals of less than a second. Indeed, a common interval for timing programs is milliseconds (there are 1,000 milliseconds in a second)

To see how this module is used, look at function `times()` in module `lab12`, which has two statements:

```
# Store in timestart the time at which the statement is executed.
timestart = time.time()

# Store in timeend the time at which the statement is executed.
timeend = time.time()
```

The next set of statements prints the values of these times in two forms: first, using seconds (as a float), and second as a milliseconds (rounded to the nearest int). So now, in case you were curious, you know how many seconds and or milliseconds have elapsed since 1 January 1970.

To see the results of execution of these statements, execute a call on procedure `times()`. You will see the results in the interactive shell. Note that the two times are exactly the same (or differ by at most 1). It takes very little time to call the ¡code¿time()¡/code¿ function 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.

Exercise 1: Experiment with Searches

The first experiment to run compares linear search with binary search. Look at procedure `testsearches(m)` 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 `lab12.testsearches(10)`; and see what is printed in the Python interactive shell.

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(m)` and its specification. It creates an array of size `m`, and then:

(1) runs selection sort on list `b`, after initializing the array to random values.
(2) runs insertion sort on list `b`, after initializing the array to random values.

Execute the call `testsorts(1000)`. Try higher values for the argument – like 5000, 10000, 15000, 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.

Exercise 3: Experiment with Sorting an Already-Ascending Array

In method `testsorts(m)`, method `fill_rand(b)` is used to fill list `b` with random values. There is also a method `fill_pos(b)`, which fills the list to [0, 1, 2, 3, ...]. Change the function call `fill_rand(b)` (in two places) to `fill_pos(b)`, 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 list 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(m)` and its specification. It creates an array of size `m`, and then:

(1) runs selection sort on list `b`, after initializing the array to random values.
(2) runs quicksort on list `b`, after initializing the array to random values.

Execute a call on `testsorts2(m)` with argument 20000 – 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 a list of size `n`, selection sort takes time proportional to $\texttt{n}^2$ and quicksort, on average, takes time proportional to $\texttt{n} \log \texttt{n}$.

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.