Design and Analysis of Algorithms
Assignment 1 (Due Date: 15/02/2016)

The objective this assignment is to study how the theoretical analysis of a variety of sorting algorithms compares with their actual performance. The major emphasis of this assignment is on analyzing the performance of the algorithms, **NOT** on coding the algorithms. Therefore, you will be given a template program which implements the following algorithms:

- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Radix Sort

You will need to make only a few modifications to test out different strategies. Most of your time should be spent on designing careful test cases and analyzing your results in order to draw useful conclusions regarding the performance of the various algorithms.

Template Program

The code is documented well enough for you to take it and modify it. The template program can read the input from a file, or to create three types of input lists: **(i)** random generated elements**, (ii)** elements sorted in increasing order, **(iii)** elements sorted in decreasing order. If needed, you can add additional functionality to it. The format of the input files is as follows:

[n] /*number of positive integers to
sort*/ [element 1]
[element 2]
:
[element n]

The time value returned is in terms of number of cpu clock ticks and seconds *on my machine*. However, the value returned is system dependent. If you are unsure of the units you will have to check out the man page for your machine. However, primarily what it would be of interest to you is the *relative* time – i.e., the units should not matter except for determining the constants. To understand the mechanism of timing the function refer to the accompanying help document *timing*.

The random number generator function used in this template returns a number in a specified range, say (0, *rand_max*) and uses a seed number for initialization. You will need to do a man on "rand," "drand", and "random" and its counterparts to see how they work. To get a number in the range you want, say [F, L], you would need to find a way to translate the random number r in (0, *rand_max*) to something in [F,L].

There are a number of ways to do this (like using mod function). Here is just one:

r1 = rand() / random number between 0 and 1 */*
t1 = (r1/ rand_max)(L-F) /* random number between 0 and L-F*
**/ r2 = F + t1 /* random number between F and L */*

To compile the program, copy **sort.c** into your directory and use **gcc** compiler (it is written in C).

Coding

The version of *QuickSort* in the template program uses the first element in the array as the pivot element. You will study different strategies for selecting the pivot:

- **Pivot Choice 1:** The first element in the list (used in the template program)
- **Pivot Choice 2:** A random element in the array.
- **Pivot Choice 3:** The median of the first, middle, and last elements in the array.

You should implement the Pivot Choices 2 and 3 listed above. You will then have three different versions of QuickSort (you would need to add them to the template program as additional menu options).

Also Implement and compare two more sorting algorithms: **Selection Sort** and **Bucket Sort.**

Analysis

1. Theoretical question:

   Assume the *n* input elements are integers in the range [0, n-1]. For each algorithm, determine what are best, average, and worst-case inputs. Your write-up should list these for each algorithm. Include a sentence or two of justification for each one. You should answer what you expect to be true based on a theoretical analysis (and you should not refer to experimental results). In the subsequent questions we will compare the experimental results to these theoretical predictions

.

2. Data generation and experimental setup.

   The choice of test data is up to you (i.e., for each sorting algorithm, which input sizes should be tested, how many different inputs of the same size, which particular inputs of a given size.) Be smart about which experiments to run, i.e., don't run larger or more tests than you need to answer the above questions

reasonably well. Also, note that you will need to run your experiments several times in order to get stable measurements (i.e., times will vary depending upon system load, input, etc.). Your experimental setup must be described in terms of the following:

- What kind of machine did you use?
- What timing mechanism?
- How many times did you repeat each experiment?
- What times are reported?
- How did you select the inputs?
- Did you use the same inputs for all sorting algorithms?

3. Which of the three versions of Quick sort seems to perform the best?

   - Graph the best case running time as a function of input size **n** for all three versions (use the best case input you determined in each case in part 1).
   - Graph the worst case running time as a function of input size **n** for all three versions (use the worst case input you determined in each case in part 1).
   - Graph the average case running time as a function of input size **n** for all three versions.

(Note: refer to the attached help document "plotting" to know how to generate professional quality graphs)

4. Which of the five (+ two) sorts seems to perform the best (consider the best version of Quicksort)?

   - Graph the best case running time as a function of input size **n** for the five sorts (use the best case input you determined in each case in part 1).
   - Graph the worst case running time as a function of input size **n** for the five sorts (use the best case input you determined in each case in part 1).
   - Graph the average case running time as a function of input size **n** for the five sorts.

5. To what extent does the best, average and worst case analyses (from class/textbook) of each sort agree with the experimental results?

   To answer this question you would need to find a way to compare the experimental results for a sort with its predicted theoretical times. One way to compare a time obtained experimentally to a predicted time of **O(f(n))** (e.g., $f(n)= n^2$) would be to divide the time for a number of runs with different input sizes by **f(n)** and see if you get a horizontal line (after some input size n_0). That n_0 would

represent the n_0 value for the asymptotic analysis. The value on the y-axis (assuming you put input size on the x-axis) will give you the constant value of the big-O.

For each sort, and for each case (best, average, and worst), determine whether the observed experimental running time is of the same order as predicted by the asymptotic analysis. Your determination should be backed up by your experiments and analysis and you must explain your reasoning. If you found the sort didn't conform to the asymptotic analysis, you should try to understand why and provide an explanation.

6. For the comparison sorts, is the number of comparisons really a good predictor of the execution time? In other words, is a comparison a good choice of basic operation for analyzing these algorithms?

   To answer this question you would need to analyze your data to see if the number of comparisons is correlated with execution time. Plot **(time / #comp)** vs. **n** and refer to these plots in your answer.

Deliverables

- **A detailed REPORT** (which have to be submitted online only) addressing the points mentioned above. Your write-up must include a coherent discussion of which experiments you ran, how many times you ran them, etc. Grading on this assignment will put the greatest weight on the choice of test data and the quality and insightfulness of your discussion of your results. Don't be put off too much if there are some discrepancies between the theoretical results and the experiments. If that happens, try to explain why it occurred.

  Answer the questions in the order presented. Use meaningful titles for each subsection and figure captions to explain the graphs. Also, graphs should be numbered and must be in the same section where they are discussed.

- **AN ELECTRONIC COPY OF YOUR CODE.** Provide instructions of how to run your code. Please note that *I do **not** want a hardcopy of your report, code, your raw output, or a log of your program's execution.*