EECS 268: Spring 2016

Laboratory 9

Due: This lab is due before your next lab begins

Introduction

In this lab you will investigate the performance of various sorting algorithms on several datasets, evaluate them using Big-O notation, and produce tables and graphs of actual measured execution times. You will be using the book's template-based implementation of the various sorting algorithms, however all arrays that you sort will contain double precision floating point numbers.

Code Starting Point

- Starting Code (gzipped tar file)

This tar file has implementations of the five sorting algorithms we have studied so far: selection sort, insertion sort, bubble sort, mergesort, and quicksort. (Note that the mergeSort implementation here generalizes the book's implementation in a small, but very important way by dynamically allocating the temporary array needed for the merge operation. Your code will therefore call: `mergeSort(anArray, n)` instead of `mergeSort(anArray, 0, n-1)` as is shown in the book.)

You will write a main program that does a `#include` for each of the five files. You will further elaborate this main program as described below.

Building Your Sort Algorithm Tester

Your main program will receive from the command line (via `argv[]`) the size of an array to be generated and sorted, the initial order of data, and the algorithm to be used to perform the sort:

`lab9` <*data_size*> <*random|ascending|descending*> <*selection|insertion|bubble|merge|quick*>

For example:

```
lab9 50000 random insertion
```

Generating an array to be sorted

Given the data size and the data requirement, you are to dynamically allocate an array of `double` of the specified size and populate it with the appropriate data. For "ascending", set `array[i] = 0.001*static_cast<double>(i)`; for "descending", set `array[i] = 0.001*static_cast<double>(SIZE - i - 1)`, where `SIZE` is the size of the array. Finally, for "random", set each element of the array to a unique random number, *x*, in the range $0.0 \le x \le 100000.0$.

When populating an array with random numbers, use a random number generator that returns uniformly distributed *floating point* (as opposed to integer) values. If you populate very large arrays with random integers, implicitly casting to `double` as you store them, you will oftentimes get a large number of identical values in the array. While the sorting algorithms will still work, having a large number of identical values tends to skew the statistics you will gather. On the linux machines, you ***must use*** the `drand48` function defined in `<stdlib.h>`. You must also call `srand48` with a constant at the start of your program to make sure you get the same sequence of random numbers each time you execute the program.

Performing and timing the sort

Call the specified algorithm, passing it the array you generated. Record the time taken to sort the data. Your lab GTA will discuss how to retrieve the actual cpu time used by your process. Be sure you do not include any I/O operations, time taken to generate the array to be sorted, or any other extraneous operation. Time only the sorting algorithm itself by inserting the appropriate call immediately before and immediately after the call to the sorting algorithm. Format and print a message to the console reporting the size of the array, the sorting algorithm used, the initial array condition (sorted, reverse sorted, or random), and the total time taken to perform the sort.

Evaluation

For each sorting algorithm and initial array condition, get times for at least 8 different array sizes, none of which should be less than 50000. For quicksort and mergesort (as well as bubble sort and insertion sort for the "ascending" case), record times for much larger sizes – at least 500000 and 1000000. Record these results in a spread sheet. Create five sets of three graphs each; each set is for one of the algorithms (bubble, insertion, etc.), and the three graphs within each set are for "random", "ascending" and "descending", respectively.

For each of the cases (random, ascending, descending) and each of the sorting algorithms:

1. State whether the times you observed demonstrate O(N), O(N*log(N)), or O(N^2) growth rates. Justify your statement. Be as quantitative as possible.
2. Predict the time required to sort an array of size 10,000,000. Justify your answer.

Submission

Create a tarball with your source code in the usual way. Include in the same directory as your source code a report that includes the tables, graphs, and statements mentioned above. Your report must be a PDF file, and it must be named: *YourLastName*`_Report.pdf`. We will not accept `.doc`, `.docx`, `.odf`, `.txt`, or any other format!

Email this tarball to your TA. The email subject line must look like "[EECS 268] SubmissionName" as follows:

```
[EECS 268] Lab 09
```

Note that the subject should be exactly like the line above. Do not leave out any of the spaces, or the bracket characters ("[" and "]"). In the body of your email, include your name and student ID.