EECS 268: Fall 2011

Project 6: Time Complexity of Sorting Algorithms

Lecture Topics: Algorithmic Efficiency and Sorting
Lab Topics: Formatted output, Timing Measurement of Program Behavior, Random Number Generation
Outside Topics: Data Presentation

Part 1: Introduction

Through this lab, you will examine a number of sorting algorithms, specifically: (1) selection sort, (2) insertion sort, (3) bubble sort, (4) mergesort, and (5) quicksort. Your goal in this assignment is to create a testing framework within which you can explore the best, worst and average case behavior of several sorting algorithms, and in which you can also compare the performance of the algorithms to each other by evaluating their performance on the same data sets.

The source code for these algorithms is provided as part of the starter code TAR file. Note that the C++ source for the algorithms is provided in separate source files. The header_file "data_type.h" provides a place to specify a typedef for the data type being sorted. The utility.cpp and utility.h files provide some useful enumeration definitions and routines for translating input arguments to equivalent enumerations and routines for printing enumeration values to printable strings for structured output.

An implementation for the **swap( )** operation is provided, which is used by some of the sort algorithms to operate on data elements in an encapsulated way that leaves the sorting code unaware of what DataType actually is. We have set the code up so that integers are being used, see data_type.h, but you might find it educational to try other data types, including your own classes, as long as you overload the necessary comparison operators. You will have to read the existing Make file and create or complete steps as necessary to make the **one_test** and **do_experiment** executables. The "test1", "test2", and "test3" make file targets give examples of their intended use. The bulk of your programming task is to fill in missing portions of code in the quicksort pivot code, several of the routines in utility.cpp, and most of the code in do_experiment.cpp.

This assignment concentrates *less* on programming and *more* on evaluating how programs behave. However, to test the performance of code you have written, or that of code other people have written, one often has to write code that "glues together" the code you want to test. Sometimes this additional code provides a framework for controlling when the code being tested is executed, often called a testing framework,skeleton, or harness. Other times the code you write provides any missing capabilities or operations that are required by the new environment, but which were not available in the environment from which the code has come.

Note also that to use the routines defined in the separate source files, the utility.h file provides function prototypes for routines in utility.cpp and for the sorting algorithms defined in the separate ".cpp" file. Note that, in theory, each sorting algorithm should be described with a separate header file, but we have put all the sort function prototypes in utility.h to save time and trouble for the assignment.

Choose quicksort and three of the four other algorithms to evaluate. Note, however, that your "Sort Algorithm Tester" program should be able to run all five algorithms. You will have to modify the do_experiment code accordingly as it currently runs tests for all algorithms.

For each algorithm, you will empirically evaluate the time complexity of these algorithms in terms of best (already ascending), worst (starts in descending order), and average case (random order) data sets. This will be accomplished by measuring and comparing their actual performance on a variety of specifically constructed data sets. You will also look specifically at the quicksort algorithm and evaluate how the choice of pivot affects performance. The code you write to accomplish this will be filling in the missing parts of the starter code provided which are marked with "IMPLEMENT ME" tags.

Part 2: Building the "Sort Algorithm Tester"

For this project you will complete the utility programs that generate an array of values of a particular size and then sorts that array with a specified sorting algorithm. In the starter code given you, the program **one_test** illustrates how one test is run and the program **do_experiment** executes a set of tests. The array values will be initially sorted in one of 3 ways:

- **Acending** – The first element in the array contains the smallest value and the values incrementally increase, with the largest value being in the final position of the array.
- **Decending** – The largest value is the first element of the array and the values decrease until you reach the last element.
- **Random** – Each element in the array contains a randomly generated value as described below.

Look at **one_test** and see how it receives command line arguments, presented in **argv[ ]**, specifying the following input parameters:

1. The size of the data set to be tested,
2. The initial order of data to be included within the data set,
3. The algorithm you wish use to sort the data set.

A typical execution of the program from the command line would thus follow this specification:

bash> ./one_test *<data_size> <initial_arrangement> <sorting-algorithm>*

Where the <initial_arrangement> and <sorting-algorithm> choices are:

<initial_arrangement> = random | ascending | descending
<sorting-algorithm> = selection | insertion | bubble | merge | quick

Thus, a reasonable invocation of your program creating a 1000 item data set randomly arranged initially and then sorted using the insertion sort algorithm, would be:

```
bash> ./one_test 1000 random insertion
```

Or, another invocation might specify 10,000 data items initially in ascending order that would be sorted into ascending order using merge sort:

```
bash> ./one_test 10000 ascending merge
```

Study the one_test.cpp code to see how several of the utility routines used with the enumerations work, and study the Make file to see several calls to **one_test**. Modify the Make file to try your own calls when you have the implementation done enough for **one_test** to do something interesting.

One important architectural feature to note is that most of the work of **one_test** is to figure out the arguments to give to the utility routine **do_test( )**, defined in utility.cpp. You will be implementing much of the **do_test( )** routine.

Data Generation

Given the data set size and the data requirement, you are to dynamically allocate an array of the appropriate size and populate it with the appropriate data. For the "random" arrangement of the test data set, you must generate numbers in the range $0 \leq x \leq$ sample_size. We might use floating point numbers, but for purposes of this assignment and discussion we will assume the DataType is “int”.

Using the information covered in lab and in the sample code concerning random numbers, finish the implementation of the **create_data_set( )** routine. You will first have to create an array of the correct size and type, and then fill it in a “case” statement for each of the initial conditions. Note that you have to distinguish the different data arrangements using the **starting_conditions** parameter which is of type **start_t** which is one of the enumerations.

Algorithm Execution

Once your program has generated the space for the data, it must be arranged in the proper initial conditions. The random generation of integer values can be used to fill the array for the “random” arrangement. The ascending initial order can be accomplished by looping though the array elements and filling them with 1 through N where N is the data set size. Descending initial conditions can be accomplished by looping through the array elements in

reverse order and filling them with 1 through N. A little arithmetic is sufficient and how to do it should be obvious after you get ascending working.

When the data is properly arranged, you should write the code in **do_test** that will then execute the specified sort algorithm on the test data by calling the proper sorting algorithm. Most of the implementation effort in **do_test** is to implement the cases in the switch statement for each of the possible values for the **sort_selected** parameter which is of the **sort_t** enumeration type.

Your implementation of **do_test( )** must be able to execute all of the five sorting algorithms listed above. For execution of the quicksort algorithm, you can use the default pivoting scheme for this section of the project, which is used if NO_PIVOT is specified. In Part 3, the second component of this project describes how you will consider the implementation and comparison of two other pivoting methods.

Timing

The main purpose of this lab is to experimentally measure the time complexity of the sort operation as performed by each algorithm. Obviously, the measured behavior is expected to match the theoretical complexities for each algorithm. However, you may find that at first this is not perfectly true. Learning to measure program performance and correctly interpret the data is not too hard, but it is often a bit harder than it first appears. To that end, you use the information provided in the lab slides and the sample code concerning time interval measurement to record a time stamp value before and after the switch statement that calls the selected sort algorithm. After the second time stamp is taken, note the call to the **microsecond_difference( )** routine. You will have to complete its code before execution intervals will be correctly calculated in microseconds.

Be certain to *exclude* from your measurement of sort timing the amount of time taken to generate the array of test data to be sorted.

Evaluation

You will collect timing information on each of the sorting methods using various sizes and different initial array orderings. This is what the **do_experiment** program does. Note that the **main( )** routine of **do_experiment** is nothing but a set of calls to **do_table( )**. You will have to fill in the internals of **do_table( )**.

For each data set ordering: ascending, descending, and random, you will record the execution time for your four algorithms, using each of the following data set sizes: 100, 500, 1,000, 10,000, and 100,000. Note that the **elapsed_times** 2D array in **do_table( )** in do_experiment.cpp is set up to hold this table's worth of data. You will then use formatted C++ output as discussed in the lab slides to print a table of values in an orderly way.

So for example, for insertion sort, you will test the execution time of sorting an ascending array of size 100, then of size 500, 1000, 10000, and finally 100,000. You will then test the same sizes for an array sorted in descending order, and finally do the same for a randomly ordered data set.

In total, each of the four algorithms will have 15 time values associated with them, 5 data set sizes for each of 3 data set types. You will use the default pivot for quicksort, data_array[0], chosen when NO_PIVOT is specified. The do_experiment.cpp and **do_table( )** code is designed to make this conveniently automated.

When you have accumulated the raw data, you will analyze and describe your results by writing a report, as discussed in Part 4.

Part 3: Comparison of Pivot Point Selection in Quicksort

The second evaluation part of this project will be to implement two additional pivoting methods to be used in the quicksort algorithm. These two schemes are as follows:

- **Median of three** – discussed in class.
- **Random** – a random number generator will be used to specify a value to be used as an index into the sub-array you are examining. Ensuring that the value is within the range of the current sub-array.

To indicate which pivot selection to use, we allow a fourth input parameter to be passed into the application via command line. This parameter is only utilized when quicksort is the algorithm used for sorting.

The valid options for this new input parameter will be:

- default – indicates to use the default pivot method.
- median – indicates to use the median of three pivot method.
- random – indicates to use the random pivot method.

So one can call the **one_test** program to run quicksort on a randomly arranged data set of 1000 items using a random pivot:

bash> ./one_test `1000 random quick random`

Evaluation

For each of the three pivot methods you will gather timing information for each of the three types of arrays used (random, ascending, descending) and for each size of data set used (100, 500, 1000, 10000, 100000). This information will be used in the report as described below.

Part 4: Report

The timing information collected in this project will be combined together in the form of a report. This report will detail your findings and include explanations for why the algorithms performed as they did. The report will be broken down into two sections, corresponding to the two experiments of this project.

Section 1: A Comparison of Sorting Algorithms.

In this section you will write a brief analysis of what you found during the timing experiment. Discuss which algorithms do better with smaller data sets versus larger data sets. Also, write a summary about how each of the algorithms performed given the initial order of the data set. In your discussion, consider specific properties of each algorithm and why you feel that they may have contributed to the observed results.

Also, for this section include graphs of the performance of the various algorithms. You should create relevant graphs that illustrate the trends you discuss and which thus allow you to communicate effectively about the performance of your algorithms.

For example, you might want a graph for each sorting algorithm that compares the time required for each of the three different types of data arrays for all sizes of arrays. In this graph you would have a line for each of the three types of data sets, with the x-axis indicating the size of the data set and the y-axis relating the time it took for each test to run.

You could also generate graphs that allow for a more direct comparison between the various algorithms. For example, you could create graphs that displayed the timing differences between the four different algorithms for a particular data set type. In this case you would have four lines with the x-axis again being size and the y-axis being time. You would need 3 separate graphs corresponding to the three data set types examined in this lab.

Include these graphs within the report as you discuss the sorting algorithm behavior illustrated by the graphs. Also provide the spreadsheet or other source you used to create the graphs in your submission for this project.

Section 2: A Comparison of Pivot Selection in QuickSort

Here you will provide an analysis similar to that of Section 1 considering effects of the quicksort pivot selection. Discuss the performance of each pivot selection with respect to the initial order of the data set and the data set size. Is there one pivot selection method that you would choose? If so, which? If not, why not?

You will also include graphs to display the results of these tests, as you did in Section 1. Include in your spreadsheet for these graphs as well.

Part 5: What to turn in

You will upload a PDF file containing your report and the TAR file of your completed code as usual according to the instructions on the lab WWW page.

Part 7: Grading Criteria

60 Class points

20% 12 pts Implementation of the Sort Algorithm Testing Framework
- 10% 6 pts do_experiment
- 10% 6 pts utility routines

15% 9 pts Implementation of the pivot selection algorithms
- 5 pts mo3
- 4 pts random

30% 18 pts Report – Section 1 – Graphs and discussion

20% 12 pts Report – Section 2 – Graphs and discussion

10% 6 pts Coding Style and Standards

05% 3 pts Comments

---

100 60 pts Total