IE 170 Laboratory 2: Selection Algorithms

Dr. T.K. Ralphs

Due February 6, 2006

1 Laboratory Description and Procedures

1.1 Learning Objectives

1. Understand each of the key terms listed below.
2. Understand the use of recursion in algorithm design.
3. Understand the use of the pair data structure.
4. Understand why randomization can be an effective tool in algorithm design.
5. Understand the difficulties that can occur in analyzing an algorithm empirically.

1.2 Key Words

1. Randomized
2. Recursive
3. Selection
4. Partial sorting

1.3 Scenario

Because of the large number of complaints that are received by the on-line retailer `ripoff.com` where you are employed, the management decides to establish a list of the complaints that have been made. The list contains the name of each customer who has registered a complaint, along with a numerical priority (known to insiders as the *squeaky wheel index*) that is constantly updated. The priority is based on a combination of the amount of merchandise the customer has purchased in the past six months, the number of times they've called, how recently they've called, and a number of other factors. In order to make the list easy to update, it is kept in order by the customers' last names. However, the customer service supervisor would like to be able to determine at the touch of a button who the top 10 complainers in the system are at any one time. Because the number of complaints is so large, it is not practical (or needed) to completely re-sort the list in order to determine this, so we would like to have an algorithm that could efficiently determine this information without resorting the entire list. Finding the i^{th} item in a list of n items is called the *selection problem.* Generating a sorted list of the best i items in a list of n items is called the *partial sorting problem.* These problem are essentially equivalent since either one of them can be solved easily using an algorithm for the other one (see the exercises).

1.4 Designing and Analyzing the Algorithm

In this lab, we will be comparing in more detail the results of the theoretical and empirical analysis of two algorithms. This time, however, we'll measure the actual running time of the algorithms in question. Because the subroutines we have available for measurement of running times are not that precise and because it's more convenient to work with small data sets, we will need to run each of the algorithms repeatedly in a loop. The running time of the algorithms we will study in this lab depend on both the number of items in the list and the number of items to be selected. In the analysis, you will examine the effect of both these factors.

One of the important concepts for this lab will be the usefulness of randomization in algorithm design. A *randomized algorithm* is one in which some step in the algorithm is taken at random. Although this may seem like a bad idea in general and also may seem like it violates our concept of a well-defined algorithm, we will see that it can work to our advantage at times.

Another important concept will be that of a *recursive algorithm.* We saw our first recursive algorithm in the last lab and we continue our study of such algorithms here. Note that although both of these algorithms are intended for the partial sorting problem, we can use them to perform a full sort by specifying that the number of items to be selected should be equal to the number of items in the list. However, this is unlikely to be a very good method of sorting.

The final important point to be made is that small details can make a big difference in algorithm performance. We will see that by making some small changes to the algorithms below and testing their effect. Changes in the type of input can also have a dramatic effect. We will see that as well.

1.5 Program Specifications

1.5.1 Program Function

Your program should perform as follows:

1. Read from the command line a file name and the number of items to be selected from the list in the file, as well as an optional random seed.
2. Read in the list of customer names and associated priorities from from the file.
3. Use the insertion algorithm to select the specified number of names and report them in *in priority order.*
4. Report the time required (looping as needed to increase accuracy).
5. Use the partition algorithm to select the specified number of names and report them in *in priority order* (the names don't come out in sorted order on the sublist, so you can call the insertion algorithm to sort them).
6. Report the time required (looping as needed to increase accuracy).

1.5.2 Algorithms

For this lab, you will implement and test two different partial sort algorithms. Let n be the total number of items in the list and let i be the number of items to be output. For the description, let's assume we are looking for the i largest items.

1. *Insertion*: In the insertion algorithm, we will loop through the input array, keeping an ordered list of the largest i elements seen so far, inserting a new element in the proper slot when one is found.
2. *Partition*: In the partition algorithm, we "guess" which element is the i^{th} one in the list—call this the partition element. We then rearrange the array so that

   - the partition element is in the position it would be in if the array were sorted (from largest to smallest),
   - the elements that are greater than the partition element come before it in the array, and
   - the elements that are less than the partition element come after it in the array.

   If the partition element happens to end up in position $i - 1$, then the partition element, along with the elements that come before it in the array are the i largest elements. These can then be (optionally) sorted using the insertion algorithm. If the partition elements is in position $j > i - 1$, then we can simply recursively apply the algorithm to the part of the array containing the j largest elements (the beginning of the array). If the partition element is in position $j < i - 1$, then we need to add the best $i - j - 1$ elements from the end of the array to the $j + 1$ elements in the beginning of the array. This can be done by applying the algorithm recursively to the top part of the array.

   There are a number of ways to "guess" the partition element. We can take it to be the last element in the array, the first element in the array, the i^{th} element in i^{th} position of the original array or select it randomly. Your program should be capable of any of these methods. The subroutines needed to perform the random algorithm are in the book on pages 146, 154, and 186.

1.5.3 Data Structures

The basic data structures required for this laboratory are pairs and arrays. In addition, you will be implementing a list data structure capable of performing a partial sort operation.

2 Laboratory Test Files

The test files for this laboratory are in the zip archive `Lab2.zip` available on the course Web site. The archive will unpack into a directory called `Lab2` with subdirectories `data`, `generator`, and `shell`. In the `data` subdirectory, the files named `input*.txt` contain lists of randomly generated "names" and priorities that can be used to test your code. The name of the file indicates the number of names in the list. The number of names in the list is also the entry on the first line of the file. The files `output*.txt` is the expected output from each file when asking for the top 10 items. You can test your code by comparing your output to these files. In the subdirectory `shell`, you will find files `main.cpp`, `select.hpp`, and `select.cpp`. The file `select.hpp` contains the definition of the class you are responsible for implementing in this lab. The file `select.cpp` contains part of the implementation. You will fill out the rest. The file `main.cpp` is a driver that tests the correctness and efficiency of the class.

3 Laboratory Assignments

3.1 Programming (50 points)

1. Implement a class for performing a partial sort operation using both the insertion and partition algorithms described above. This class should be written with adherence to the basic principles discussed in lecture regarding the separation of interface from implementation. A class for performing a partial sort operation has already been defined for you in the file `select.hpp`. Your assignment is to implement the class using the file `select.cpp` as a starting point.

2. Verify that your program compiles and runs correctly using the test files provided for the lab.

3.2 Analysis (30 points)

1. (10 points) Create graphs showing the running time needed to execute each of the algorithms as a function of the number of words in the list for the following scenarios using the files `input*.txt` (do not use the file `input1K.sorted.txt`. You will need to run your algorithm at least a few hundred times to get a good comparison.

   - Selecting the top 10%.
   - Selecting the top 50%
   - Sorting the list completely.

   For the partition algorithm, use a random partition element. To create the graphs, use Excel and then paste the graph into your laboratory write-up. Comment on what you find.

2. (10 points) Suppose the items are not needed in priority order. Comment out the calls to selectInsertion in the selectPartition function and do your experiments again. What happened. Comment on your observations.

3. (10 points) Now compare the two algorithms for the file `input1K.sorted.txt`, which is already sorted by priority. For the partition algorithm, try each different partitioning rule to determine the best one. Comment on your findings.

3.3 Follow-up Questions (20 points)

1. (10 points) CLRS 3.1-1

2. (10 points) CLRS 3-2