Lab 3

Due 9 February

Handout 2
CSC 062: Spring, 2007
2 February

Complexity Analysis

1 Short Answers

Bring a written answer to the following question to lab on Tuesday. Pseudo-code (english written in an algorithmic style) is fine. Don't bother to write and debug Java.

- R-4.17 through R-4.19 on page 182 of Goodrich and Tomassia.
- R-4.21 on page 182 of Goodrich and Tomassia.

2 Lab

In this assignment I want you to measure the actual complexity of some basic sorting algorithms. Two of these algorithms are $O(n^2)$ while the last is $O(nlogn)$.

- Selection Sort
- Insertion Sort
- Merge Sort

I will provide you with a class `SortingDemo` that includes a method isort(int[] array) that can be used to call different sorting routines that are supplied as methods of the class. The idea is to run each sort on random arrays of different sizes. Because the length of time it takes to sort a routine will sometimes depend on the initial ordering of the elements, you should run each size trial many times (e.g., 5 or more) in order to get a reasonable estimate for the average time for the sort to work on an array of that size.

The idea is that if you get this data for enough different sizes of arrays, you can determine whether the algorithm truly is $O(n^2)$ or satisfies some other upper bound.

You can set the following constants at the top of the program in order to control your experiments:

- `START_SIZE` - the size of the smallest array to be sorted
- `END_SIZE` - the size of the smallest array to be sorted
- `DELTA` - the difference in size between successive arrays to be sorted
- `NUM_TRIALS` - the number of times arrays of a fixed size should be created and sorted.

The constructor for the program calls a routine `isort`, which itself calls the actual sort to be measured. For example, to test the recursive version of insertion sort, you should have:

```
public void isort(int[] array) {
   insertionSort(array, array.length);
}
```

The code in `SortingDemo` includes code to perform iterative and recursive versions of insertionSort, and recursive versions of selection sort and merge sort. You are to add a methods that performs an iterative version of selection sort.

You can either run the tests for each of these separately or modify the program so that you test them all with a single run.

The main point of this problem is clearly not just the coding. I want you to compare the running times of each of these algorithms and try to explain why they are different. In your README you should describe a series of experiments you ran, what data you collected, and what your conclusions about running times are. Please plot the data that you collect. You can do this by hand or use a computer application to do the graphing. Things to address: Does the actual running time correspond to the asymptotic complexity as you would expect? What do you think explains the practical differences between these algorithms? (I.e., explain why you think each is better or worse than the others.) Does recursion make programs run significantly slower?

Accurate Timing: When you run your programs, use `java -Xint`. The -Xint flag will turn off dynamic optimizations. This will give you more accurate results. (I found a difference in results of a factor of 6 to 8 when the dynamic optimizations were turned off!)

3 Testing tips

Computers tend to be busy doing lots of different things all the time. This makes it a challenge to get good timing data. Here are some things to think about.

- The Java command

  ```
  System.currentTimeMillis();
  ```

  returns a `long` (an integer with twice as many bits as an `int`) that represents the number of milliseconds (thousandths of a second) since sometime in 1970. Calling this before running an experiment, and again afterward, and then subtracting the two gives a good estimate for the number of milliseconds that the intervening code took to execute. Our computers can run 1000s of instructions in a millisecond so you may need to run tests many times before you get a non-zero value. See the code below for an example.

- Different runs of the experiment can generate different times due to the different things going on in the background processes of the computer. To get a sense of how much is happening go to Applications/Utilities and double click on the Activity monitor. Try to make sure as few things are happening on the computer as possible when doing your resting. For example, you might want to reboot the computer before running your tests to make sure there are no stray processes grabbing processor time. Be sure not to do anything else on the computer (including running the activity monitor) when doing your timing.

  Given this background activity, think about whether it makes more sense to take the average timing over multiple runs, the minimum timing, or some other measure. *One thing to worry about is that different runs of insertion sort may take vastly different amounts of time if some of the data is already partially in order.*

- Never perform input or output while you are timing a computation. These operations are very slow and will greatly distort your time measurements. Thus all prints should take place before you start timing or (better yet) after you are done.

  Also if you keep your total experiment times relatively short (less than a few seconds, for example), you are less likely to get extra background activity distorting your results.

- Make sure that you are computing only the times for the operations you care about. In my program I have to refill the array between sorts. As a result I need to subtract off the time it takes to fill the arrays with random numbers. I use:

  ```
  long startTime = System.currentTimeMillis();
  for (int count = 0; count < NUM_TRIALS; count++) {
     generateRandomArray(size, array);
  }
  ```

```
long endTime = System.currentTimeMillis();
long overheadForCreating = endTime - startTime;
```

to calculate the amount of time it takes to fill the array `NUM_TRIALS` times so that I can substract it off from my timing of the sorts so that I only count the sorting time.

4 Your report

You will be turning in code, but the main item of importance will be your README file. You can turn in your graphs on paper or in pdf, word, or postscript format at your option. I am especially interested in having you address whether or not your data fits the asymptotic complexity (e.g. $O(n^2)$) that we derived in class for each sort. If you have software available, you can attempt to fit your derived data to a function (e.g. $an^2 + b^n + c$) or you can take a few pieces of data yourself and see if you can derive coefficients (i.e., plug in 3 values of n and solve for a, b, and c). It will be much harder to fit the $n * (log(n))$ sorts by hand, but see what you can do. Note that for those you should try to fit your data to $an(log(n)) + bn + c$. Either way, graph the actual data as well as the curve that you fit to the data to check the correspondence.

Here are some detailed tips:

- Your writeup should include a description of the machine you are using. Use the same computer for all of your experiments and try to keep the background activities the same for all runs (and as minimal as possible). Information about a Macintosh's hardware configuration can be found as the first item under the "apple" menu in the upper left hand corner of the screen.
- Be sure to discuss the amount of overhead time your program takes, and to subtract that off from your measurements.
- Provide a table of values for all the data generated by your program. Make sure it is well-labeled so that it is easy to see what the data means.
- Discuss the graphs of your data along with the curves that you fit with the data to illustrate whether you got a good fit.

5 Tools

Gnuplot is a tool to make graphs (see http://www.gnuplot.org). If interested, try out some of the examples from the gnuplot tutorial at `http://www.duke.edu/~hpgavin/gnuplot.html`.

Gnuplot is installed on the unix servers in the department, but not on the Macs. To execute it, you need to open a terminal window and type:

```
ssh linus.cs.pomona.edu
```

It will respond by asking you for your password. If all goes well, you will end up in your home directory. Navigate (using cd – dragging and dropping will not work) to the directory you wish to work in (the directories on linus are exactly the same as those on your Mac account) and run gnuplot from there.

The file `start.gp` from the handouts page contains a script for gnuplot to get you started. Once you generate a postscript file, you should be able to just double click on it to open it in Preview. You can print it to the printer using the lpr file.cps command. To practice, plot functions that are $O(1)$, $O(n)$, $O(n^2)$. In addition, copy the file `unknown.dat` from the web page. Plot the data in that file using the script `unknown.gp` to see how to plot data using gnuplot. In the tutorial you will also find information on how to fit curves to data.

6 What to hand in

While there is a bit of code to be written here, most of your work will involve an analysis of the data you collect. The analysis should be in a file README, which can be a plain text file, pdf, or (shudder) a Word file. You may also have other supporting files, such as graphs of your data. Of course you should also turn in the code that you are using.

Your program and report will be due on Friday, February 9 at 5 p.m. As usual place all of your code in a folder and drop it off in the CS 062 dropbox folder.

7 Acknowledgements

This assignment is based on similar assignments by Peter Frohlich of Johns Hopkins University and Duane Bailey of Williams College.