COMP161 - Project 2 - Exploring Theory and Practice

Spring 2014

> For your final project you will generate and present a data set that aims to shed light on the performance of C++ implementations of key algorithms in searching and sorting. To generate the data you'll write a series of programs that can be used to measure execution time and again to get instruction execution counts with Valgrind's callgrind. All this data will be presented in a series of well crafted tables.

The Data Set

The primary goal of this project is to generate a data set that allows one to think about the relationship between algorithm complexity and actual procedure performance. Towards that and you'll be writing a series of programs to gather data about the *execution time* and *number of instructions executed* for the following procedures[1]:

[1] procedures are written by you unless otherwise specified

- binary search
- *binary_search* from the C++ *algorithm* library
- linear search
- *find* from the C++ *algorithm* library
- bubbleSort
- selectionSort
- insertionSort
- mergesort
- quicksort
- *sort* the C++ *algorithm* library

All of the algorithms listed above are discussed at length in the textbook where the complexity of each procedure is clearly listed and complete array-based implementations are common. The algorithms for which implementations aren't given are at least discussed and presented using pseudo-code. These textbook implementations clearly highlight the core logic of the algorithm but are not optimized for maximum efficiency. To explore optimized implementations, you'll look turn to *find*, *binary_search*, and *sort* from the C++ STL *algorithm* library[2].

[2] `http://www.cplusplus.com/reference/algorithm/`

Execution Times

Your first data set is the *average* of five randomly selected procedure inputs for each of the 10 algorithms listed above and for each of the following input sizes: 10, 50, 100, 500, 1000, 5000, 10000, 25000, 50000, 75000, 100000, and 250000. To be clear thats 12 sizes, 10 algorithms, and 5 executions per size-algorithm combination for a total of *120 average execution time data points* computed from 600 timed executions[3].

[3] 60 per algorithm

As programmers, we learn that execution time is a function of many variables, is best viewed as a rough measure of overall performance, and is not a precise measure of the efficient use of computational resources. However, time data has the added benefit of being easily digestible for experts and non-experts alike; everyone has a good sense of how to interpret time data and has a feel for the difference between 30 seconds and 5 minutes. This is why time data is still useful and still widely cited; it provides easy to understand context for more technical, nuanced data.

We want to measure average performance and do do so we'll run our procedures on randomly generated inputs, or VECTOR<INT>s in this case, of various sizes. To get a better feel for overall performance for each problem size, we'll collect the execution time for several different executions of the procedure and then compute and report the average. Averaging only five instances is really too few, but we're being hyper conservative about scaling programs up to larger numbers of executions when the input vectors get bigger. Several of the procedures that you'll be dealing with can and will take on the order of minutes to execute. So, a procedure that takes 5 minutes repeated 20 times will take over an hour to run. Given that we're all working off the same server, we'd like to avoid overloading the server with 14 compute jobs each needing hours of compute time.

Instructions Executed

Your second data set is the number of instructions executed by a single representative of your averaged groups for each of the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, and 25000. This amounts to 8 sizes for 10 procedures for a total of *80 instructions executed data points*. You'll need to use callgrind to collect this data and then use callgrind_annotate to get the inclusive instruction execution counts for each procedure

Programs executed by Valgrind run considerably slower than they would otherwise. This means that you don't want to profile a program that takes a long time to execute unless you really want or need the profiler data for that program because the program in question will take a very, very long time to execute with Valgrind.

You're gathering instruction counts in order to give the execution time data some context and not to dig into the details underlying the counts themselves. This means you're going to be a little less rigorous in gathering data with Valgrind than you were gathering execution time data.

You'll continue to run the procedures on random inputs, but this time you'll only do a single execution per size and restrict yourself to smaller sizes. It's important that we reuse data used in gathering timing data so that it is correlated with the execution time data we're gathering. In short, you'll pick one of the 5 executions you timed, and get counts on the number of instructions needed for that execution.

The Program

The code you're writing for this project has four components:

1. A library of the 7 algorithm implementations discussed in the text, i.e. the 7 non-C++ standard library procedures listed in the beginning.
2. A library for generating random *vector<int>* data and reading and writing that data from and to files.
3. A program for generating the random test data used to gather your performance data
4. A series of 10 programs, one per procedure, used to gather performance data.

Algorithm Library

The first library should contain seven procedures. One for each of following algorithms on a *vector<int>* variable.

- binary search
- linear search
- bubbleSort
- selectionSort
- insertionSort
- mergesort
- quicksort

Each of the algorithms listed above is well documented in our textbook[4]. Your goal is to modify the textbook presentations, which are written for arrays, so that they work on *vector<int>*s. You'll then need to write a set of gTest unit tests to demonstrate the correctness of your implementations. They are a few typos in the textbook, so be sure to check the author's errata page here: `http://homepage.cs.uri.edu/~carrano/WMcpp6e/`

[4] See chapter 11 for sorts and chapter 2 for binary search. We've already seen *linear search* under the name *find*

Random Vectors and Files Library

The second library is used too generate our random inputs, write them to files for easy reuse, and then read them in from files for actual testing. It should be comprised of the following procedures:

1. *fillWithRand*
   A vector mutator that fills a vector<int> with randomly generated integer values
2. *writeToFile*
   An output procedure which takes a vector<int> and an output file stream and then writes the contents of the vector to the file. Each integer should be separated by whitespace in the file.
3. *readFromFile*
   A vector mutator and input procedure which takes a vector<int> and an input file stream and fills the vector<int> with the integers found in the file stream. Integers in the file are assumed to be separated by whitespace.

These three procedures can be used to generate all the test data we need. Testing *fillWithRand* is tricky as it makes heavy use of random integers. You should simply test that the mutator changes the contents of a vector and leaves the size unchanged. The file based procedures do not rely on random ints, but they are I/O procedures and I/O procedures require special testing. Check the lecture notes on I/O procedures for a recap on issues related to testing I/O based procedures[5]

[5] It is possible to automate the testing of file based I/O procedures by cycling between by cycling between reading and writing.

Test Data Generation Program

The program that generates our test data for us is a fairly straightforward use of the second library. Let's just be clear about exactly how many random inputs we need and the parameters of those inputs.

- 5 random and unsorted sets of integers for each of these sizes (60 data sets)
  10, 50, 100, 500, 1000, 5000, 10000, 25000, 50000, 75000, 100000, and 250000

This amounts to 60 data sets. Each data set should be stored in its own file so that we do not accidentally mix data.

One of your goals in writing this program should be to minimize the number or recompilations and re-executions of the program needed to get your data. In an ideal world you would run this program once and it would produce 60 files. A step down from this would be to run it 12 times, once for each size. You should not have to run it 60 times, once per file. If you do need to run the program multiple times, then you should not have to recompile it. Instead you should pass in parameters from the CLI. For example, you could pass in the size on the CLI such that for a program called *genDataSet*, calling *genDataSet 10000* would generate your 5 files worth of randomly generated sets of 10000 integers.

The main impediment to this kind of automation is probably *programmatically generating file names*. You'd like to be able to write a loop such that each loop produces a different file and therefore generates a new filename. To do this you'll need to construct *strings* on the fly. This is easy enough to do with the C++ *string* library and even easier to do with some new C++11 features that allow the conversion of integers to strings. We'll discuss this further in class, but the main idea is to append or insert the string form of a numbers to a common prefix. For example, you could generate the names *set-1000-1.txt* and *set-1000-2.txt* with inside a loop that counts from 1 to 2 by converting the integers 1 and 2 to strings and then combining them with the strings "set-1000-" and ".txt".

Remember, your goal is to easily generate data for testing. The more time you spend modifying things like sizes and file names in the code, recompiling the code, and then rerunning the newly changed code, the less easy data generation will be and the longer it takes you to achieve your goal.

Final Data Gathering Programs

By pre-generating random inputs and writing them to files, we've really added some flexibility to the design space for our main data gathering program. At one end of the spectrum you could write one uber-program that could be run once to get time data and again with valgrind to get instruction counts. The problem with this approach is that these programs can and will take a very long time to execute and get pretty complex for the amount of data we'd like to collect. Imagine your program crashing after one hour and you losing all the data you gathered in that time. You wouldn't be happy. It's probably better that we break up our data gathering tasks in to more manageable chunks. This allows us to break up the computing time into manage-

able chunks, which reduces the load on the server and hopefully the risk of accidental data loss.

Your goal will be to write 10 programs, one per procedure, instead of one. Once again, the goal is to avoid recompiling code to generate new data. This program will once again require us to programmatically generate file names. To ensure that Valgrind doesn't combine instruction counts from different input sizes, we'll have to run it multiple times on different sizes to get the instruction counts. This means passing size as a CLI parameter. Furthermore, when we run it under Valgrind, we only want to execute the program once. This implies another CLI parameter. It could be as simple as a 0 for getting time and repeating the execution five time, and 1 for profiling, executing just once. You could also get more creative, it's up to you. Either way, the goal is to make it easy to get your data. Rewriting code and recompiling is a recipe for spending extra time and running into potential problems like forgetting to recompile with *-g* for profiling.

The last point about this program concerns the averages. You program should compute your averages for you. It's a computer, it's really good at doing calculations. Furthermore, you might as well write the averages to a file in place of, or in addition to, printing them to the CLI. By recording your data to a file you can more easily manage it when you move on to making tables. With a little cleverness, you could get all 10 programs to write their data to a single file[6] and you can then move that file off the server to do you table production.

[6] don't forget the default writing mode is to overwrite, not append!

Putting all of this together, you should be able to run something like:

```
myBSearch 100 0
```

to get the computer to run your binary search on 5 different vectors of 100 integers and report the average either to the command line, a file, or both. Alternatively,

```
valgrind --tool=callgrind myBSearch 100 1
```

Would run your binary search on a single vector of size 100 and profile that execution using callgrind.

From random sequences to random inputs

For sorts, a randomly generated vector<int> is a pretty good representative of the average case performance scenario. The best and worst case typically occur when the data is already sorted one way or another and it's extremely unlikely that you'll randomly generate a sorted sequence of numbers.

For searching, we'll actually want to start with a sorted sequence. Binary search requires that the data is sorted, so in the interest of

fairness, we should do our linear searches on sorted data as well. To get random sorted vector<int>s you can simply take your unsorted data and sort it using the*sort* procedure from the *algorithm* library. There are better, more efficient ways of generating random sorted sequences, but this will serve our purpose for now.

The other problem with random inputs to search is that search requires two inputs: the vector<int> and the key value. If the key value is not in the vector, then we're actually seeing the worst case in terms of computational work. So, we need to pick a value that's in the vector and that is the same for all the searches. To do this, *you should select the first value in the pre-sorted vector<int> as the random key*. This way we avoid any kind of bias in terms of selecting the min or max and we choose a value that can be repeatedly chosen by each of our four searches.

Miscellaneous Notes

Remember to incrementally and iteratively develop your programs. Write it to time one execution. Then profile that execution. Now add a loop to run multiple executions. Somewhere in there consider adding CLI input handling and test that you can control the execution via command line programs. Be certain that you run the programs on the small cases to be sure they're working properly before you venture off to the tens or hundreds of thousands. These should be pretty simple little programs, but that doesn't mean you should though good programming habits out the window.

Another way of planning your programming time is to think about the end goal: 120 average times and 80 instruction counts. You can work on your program so that you can fill in your table one column at a time, one size, all algorithms, or one row at a time, one algorithm, all sizes. If you must turn in a partially complete project, it's better to have a partial data set and partial tables then all code and no tables. So, don't be afraid to write some code, use it to get data, use the data to fill in some of the tables, then repeat for more data.

The uber-tester program is a nice goal and if you want to go for it, but you should setup the program so that you can select the algorithm with by CLI input. Do not attempt to gather all your data with a single execution of the program. This project is a good chance to dogfood[7] you program. You're writing a program and also using it. If you find it difficult or annoying to gather the data, then perhaps you can make a change to allow for easier data collection.

[7] http://en.wikipedia.org/wiki/Eating_your_own_dog_food

The Data Presentation

You'll be presenting a series of nine well developed tables for your 200 numbers:

- A single table of average execution times by algorithm and input size
  In this table you'll list each procedure in a row and in that row you report the procedure name, algorithm complexity, and execution times for each of the 12 sizes. Including a header row for column labels, this table is 11 rows by 14 columns.
- A set of eight tables showing instruction counts. One size per table.
  Each table shows the data for a single vector size. In each table, a row once again presents data for a single procedure. In a row we'll show complexity, average execution time, and instruction count. Including a header row, each table is 11 rows by 4 columns.

Properly presented tables include some other, key information, typically presented in good old fashioned written words. So, also you'll need to write a few paragraphs of text to support the tables[8]. The exact content of these tables is discussed below, but it should be noted that this text plays a supporting role to the tables and the data in the tables.

[8] not a few paragraphs per table, just a few paragraphs

Making Good Tables

Tables allow you to present a large set of numbers in such a way that the reader can easily browse through that data and easily think about what they're seeing. Notice, the point is helping the reading to think, not telling them what to think. You don't design a table to present an interpretation of data, you design it to present data. Period. The interpretation should be self-evident from the data.

Your reader needs to know at least three things *before* digging in to the table data:

1. What's in the table
2. Where did it come from and how was it obtained
3. How should one read the table

These items will be addressed in the paragraphs that accompany the tables. First, provide a brief summary description of what's in the tables. Next, make it clear to the reader the exact tools and techniques you used to get the data. The goal here is to give them everything they need to know to evaluate the credibility of your methodology

and to recreate the experiment if need be. You do not need to give them the exact inputs and code. Focus instead on the source of the algorithms, the libraries used, the computer used, the tools used, and any other information that describes the provenance[9] of the data.

[9] look it up

Having a reader take your data seriously is a privilege that you must earn[10]. Think of this issue in another context, college. You take college classes from people with advanced degrees because their degrees are meant to demonstrate their credibility and the validity of the information they're giving you. You don't take college classes from random people off the street that you don't know because you can't tell if what they're saying is credible or crazy[11]. The same logic applies to reading data tables. Engaged readers are not going to sort through the data unless they have some assurances that it's worth their time.

[10] this is also why you cite credible sources in non-scientific papers and use good grammar and style in your writing

[11] those of us with advanced degrees are also prone to craziness, we have to re-establish credibility regularly

Once you've convinced the reader that you have interesting data gathered in a methodologically correct fashion, then you need to tell them how, exactly you've laid it out for them to read. Tell them how to read the tables, but do not tell them how to interpret what they're reading. If the conclusion you came to from reading the table is valid, then a smart reader will come to the same conclusion. *Always assume your reader is smart and let them come to their own conclusions based on the data you've given them*. It should also go without saying that you should *never*[12] withhold data for the purposes of steering the reader towards a questionable conclusion. Trickery and deception have no place in science.

[12] never ever ever

Finally, it's time to turn our eye to the table itself. What's makes a table a good table, a well designed table?[13] The following points are a good starting place:

[13] If you're serious about this topic seek out the works of Edward Tufte

- Always provide a *title and caption* that meaningfully describes the data. This can be the complete accompanying text described above or a summary of that text. Either way, the reader should be able to get a high level sense of what's in the table without having to search through paragraphs of text.
- Always make units and measures clear. This can be done in column/row headers, in the caption text, or by labeling each number. The reader should not have to guess or search for units and measures associated with the numbers in the table.
- Don't use horizontal and vertical lines to separate every row and column. All the user sees is the grid; tables are not grids their structured presentations of data. Do use lines, but use them purposefully. Use a horizontal line to separate the header row from the data rows. Use different styles of lines to break rows in to logical groups[14]. The point here is to make lines meaningful, not a

[14] by search and sort, complexity, etc.

distraction. If you must use a grid, then make the lines very light and grey so that they fade in to the background.

- Don't use shading and different colors unless they carry meaningful information. Proper text alignment within columns and rows is enough guidance for the human eye, don't "help" the reader out by shading every other row or column. Doing this just goes back to grid-land.

No matter what, always remember that *the purpose of the table is to clearly present data in order to assist the reader in the critical analysis of data*. Every thing you do should be done with that in mind. Underlying this purpose is the assumption that the reader is capable and desirous of analytical thought. Have faith in them and assume they are at least as capable as you. This also means that we, the data presenter, must earn the right to their critical thought, the right to use up some of their brain time, by establishing the credibility of the data and designing a table that attempts to free their brain power for analysis. Don't distract them with silly fonts and overuse of grid lines. Don't tell them how to interpret the data, just tell them how to read the table. If your table design assists thought, doesn't distract from data analysis, and more importantly of all contains good data, then you're well on your way to good tables.

Logistics and TLDR

For this project you'll write two libraries and ten small programs, use them to generate data about the performance of important algorithms in C++, and then present that data in table format. The important stuff is the data, so work in such a way that you're incrementally getting more and more data. We'll spend much of the remaining class time discussing issues related to this project. Two of the three remaining labs are dedicated to this project. The successful completion of the libraries will be counted as a quiz grade.

- *Lab 4/23* - Get general questions answered. Write toy programs to get comfortable with newer functionality (file I/O and CLI parameters). Begin work on libraries.
- *Monday 4/28* - Complete libraries with gUnit tests due. Submit as *proj2quiz* via *handin*.
- *Lab 4/30* - Free time to work on programs and tables.
- ***Monday 5/5*** Program code and tables due. Code submitted as *proj2* via handin. Tables due in hand at the start of class (8am)[15].

[15] I'm looking at you late comers