COMP161
Project 2
Basic Profilers
Spring 2018

For your second project you will write programs to profile the runtime of of some classic algorithms given random inputs for a variety of vector sizes.

A Basic Average-Case Profiler

You are to setup your profilers as simple command-line scripts that work from two command-line arguments. The first is the number of times the procedure being profiled should be run, i.e. the number of experiments, and the second is the size of the vector. For example, if the program executable for profiling std::find is named *profFind*, then you'd run *profFind 10 10000* at the command-line to profile std::find on 10 randomly generated input vectors of size 10000.

The programs should report the size of the vector and time taken in milliseconds for each execution writing not more than five execution times per line and placing a space between each time. Additionally, the minimum, average, and maximum time taken should be given on a line of its own after all the individual times have been reported. For example, say you had a vector of size 1000, ran std::find seven times, and they took 3, 5, 4, 3, 4, 7, and 9 milliseconds respectively[1] for a minimum of 3, an average of 5, and a maximum of 9 milliseconds. Then your program would print:

[1] these times are made up

```
1000
3 5 9
3 5 4 3 4
7 9
```

The basic output template is size on one line followed by the minimum, average, and maximum time on one line, and finally the individual times written five per line.

The Procedures

From the C++ standard library[2], you'll be profiling: *std::find*, *std::sort*, and *std::binary_search*. These represent well optimized versions of classic algorithms and give you a real feel for the linear, linearithmic, and logarithmic classes, respectively. From the instructor's *searchsort*

[2] specifically the *algorithm* library

library you'll be profiling *searchsort::insertsort*, *searchsort::selectsort*, *searchsort::mergesort*, and *searchsort::quicksort*. These represent textbook presentations of classic algorithms and give you some points of comparison within the quadratic and linearithmic complexity classes.

The true average case for these procedures is determined from some statistical knowledge of what kind of data they can expect to work with. We'll simply look at cases that are not guaranteed to be the worst or best case. For the sorts, this occurs when the input is random, unsorted data. For the two searches this occurs when the item you're looking for is in the middle of the vector somewhere[3].

[3] This can actually trigger Binary Search's best case, but we'll run with it for this project.

Given that the sorts are mutators, it is important that every individual sort start with a freshly generated vector. Resorting an already sorted vector can induce best or worst case behavior depending on the sort. For the searches you'll need to use the C++ *random* library to generate a random integer in the middle range of indexes for the vector. For example, if our vector has a size of n, then we want a random number drawn from the uniform distribution of $[\frac{n}{4}, \frac{3n}{4}]$. That number will be the index of the number for which we'll search. For example, if you randomly generate the number 17, then you should search for the number currently at index 17 in your vector. Each search should pick a new number to search for. The vector itself can be reused but the index containing the search key must be regenerated.

Pre-Project Lab: Generating Test Data

In */home/com161/sp18* you'll find the file *labp2-stater.zip*. It contains all the design work for the following procedures:

- *fill_rand*
  A procedure for filling a vector with a random permutation of sequential integers.
- *fill_sorted*
  A procedure for filling vector with integers in greatest to least order.

In your pre-project lab assignment you'll implement these two procedures.

These mutator procedures form the backbone of your project as they enable you to quickly generate new inputs for the procedures you'll be profiling. The procedure fill_sorted can be implemented with basic iteration. The procedure fill_rand requires randomness and therefore follows the design of randomized procedures as seen in lecture notes 18.

To implement fill_rand you should use fill_sorted to get a vector full of the right sequence of numbers and then use a Fisher-Yates

shuffle to randomize the order. This linear time algorithm works by traversing the vector from the greatest index to the least index while randomly selection a position prior to the current traversal index and swapping the value at that position with the current position. Fisher-Yates is already implemented as std::shuffle, but it's a simple enough randomized, iterative process that it makes for a good example to learn from so you'll write your own implementation.

Once the procedures are implemented and the tests provided in the starter pass, use the program written in *labp2_main* to write a unit test for a fill_rand with a vector size greater than 5. The program takes two positive integers at the command line. It will give you the exact sequence of numbers your shuffle will get from a given random number generator seed value. The second input is the seed value and the first is, in terms of shuffling, the vector size. So passing the program 15 and 2, gives you the random values your shuffle will be given for a seed value of 2 and a vector of size 15. From here you can predict exactly what your shuffle will do as you know exactly which elements will be swapped on each iteration of the shuffle.

Submit your labp2 code as assignment *labp2*. In the event you do not complete this lab, a working copy of the library will be provided to you in order for you to complete the project.

Gather Data

Once your profiling programs work, you should use them to gather some basic data about the seven procedures in question and their associated algorithms. That data should be written to a file using command-line redirects such that each procedure has it's own file for a grand total of seven files worth of data.[4]

[4] This means you should append each different run of your program to the procedure's data file.

Minimally, you must gather data on 10 executions for each procedure and for each of the following sizes: 100,500,1000,5000,10000,50000, and 100000. You're welcome to try more or less executions than 10 as well as different sizes. Be certain that you understand the performance of the procedure before you ramp things up. It is recommended that you work your way from least to most complex procedure.

Logistics

You are expected to use helper procedures and good program design practice where prudent. Code must be well documented and tested. You should not be cramming all the code into main. On the other hand, you do not need to do extreme decomposition into procedures. Find a happy medium that works for you. In the end, there should

be a clear sense of design and style. It should be easily read and followed by a human reader in addition to correctly carrying out the task at hand.

- *Lab 4/26* — Pre-Project lab (*labp2*) and open work time.
- ***Wednesday 5/2*** Program code *and data files* submitted via handin as *proj2*.