

COMP161

Project 2

Basic Profilers

Spring 2017

Your final project requires that you write four programs to carry out basic profiling of average cases of some classic algorithms and run those programs for a variety of vector sizes. When you're done you'll submit your code and the results of your profiling experiments.

A Basic Average-Case Profiler

Each of your four programs runs in exactly the same way. The user passes two positive valued integers at the command-line. The first is the number of times the procedure being profiled should be run and the second is the size of the data set. For example, if you're going to use the program *profFind* to profile the C++ *std::find* procedure by running it 10 times on a vector of size 10000, then you'd run *profFind 10 10000* at the command-line.

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 average time taken should be given on a line of its own after all the individual times have been reported.

Let's say you had a vector of size 1000, ran seven runs of *std::find*, and they took 3, 5, 4, 3, 4, 7, and 9 milliseconds respectively¹ for an average execution time of 5 milliseconds. Then your program would print:

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

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

The Four Procedures

From the C++ standard library², you'll be profiling: *std::find*, *std::sort*, and *std::binary_search*. From the instructor's *searchsort* library you'll be profiling *searchsort::insertsort*.

The true average case for these procedures is determined from some 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

¹ these times are made up

² specifically the *algorithm* library

best case. For the two sorts this occurs when they are given random, unsorted data. For the two searches this occurs when the item you're looking for is in the middle of the vector somewhere.

The *labp2* library given to you by the instructor provides the procedure *labp2::rand_ints* which produces a vector containing the numbers 1 to n in random order as well as a procedure *labp2::sorted_ints* which gives the same range in sorted order. These procedures can be used as the basis for all your profiling runs. Sorting will require several instances of random vectors where searching can work with sorted vectors³

³ binary search requires sorted vectors

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. 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 is the index of the number for which we'll search. If you pick 17 and random, then you should search for the number 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.

Gather Data

Once your profiling programs work, you should use them to gather some basic data about these procedures and their associated algorithms. That data should be written to a file using command-line redirects such that each procedure has its own file. 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 procedure⁴.

⁴ binary search, find, sort, insertion sort

Logistics and TLDR

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* — Open work time. Submit what you end up with as *labp2*.
- *Lab 5/3* — Open lab time to work. No submission.
- *Saturday 5/6* Program code *and data files* submitted via handin as *proj2* by **noon**.