CSCI-1200 Data Structures — Fall 2019
Lab 13 — Big 'O' Notation & Performance

In this lab we will carry out a series of tests on the five fundamental data structures in the Standard Template Library we have studied this semester: `vector`, `list`, binary search tree (`set` / `map`), `priority_queue`, and hash table (`unordered_set` / `unordered_map`). We will hypothesize and evaluate the relative performance of these data structures and solidify our understanding of data structure and algorithm complexity analysis.

For each data structure you will predict and measure the runtime for three different moderately compute-intensive operations: *sorting*, *removing duplicates* (without changing the overall order), and *finding the mode* (most frequently occurring element). You will perform these tests on a collection of STL `string` objects read from an input file.

Checkpoint 1

estimate: 15-30 minutes

Before doing any implementation or testing, fill in the table below with the big 'O' notation for completing the operation using that datatype. If it is not feasible/sensible to use a particular data structure to complete the specified operation put an X in the box. *Hint: There should only be two X's!* If two or more data structures have the same big 'O' notation for one operation, predict and rank the structures by faster running time for large data. We combine set & map (and unordered_set & unordered_map) in this analysis, but be sure to specify which datatype of the two makes the most sense for each operation.

"Rules" for comparison: For each operation, analyze the cost of a function that reads the input from an STL input stream object (e.g., `std::cin` or `std::ifstream`) and writes the answer to an STL output stream (e.g., `std::cout` or `std::ofstream`). The function should read through the input only once and construct and use a single instance of the specified STL data structure to compute the output. The function may *not* use any other data structure to help with the computation (e.g., storing data in a C-style array).

	sort	remove duplicates	mode
vector			
list			
BST (`set`/`map`)			
priority queue / binary heap			
hash table (`unordered_set`/`unordered_map`)			

To complete this checkpoint: Present your analysis to a TA or mentor and be prepared to explain your answers and an outline/pseudocode of the implementation as necessary.

Checkpoint 2

estimate: 15-30 minutes

Next, let's dive into the provided code. We have implemented the 3 operations for the `vector` datatype.
http://www.cs.rpi.edu/academics/courses/fall19/csci1200/labs/13_big_o_notation/provided_files.zip

We provide a small standalone program to generate input data files with random strings. Here's how you compile and use this program to generate a file named `medium_input.txt` with 10,000 `string`s, each with 5 random `char`s:

```
 g++ -g -Wall generate_input.cpp -o generate_input.out
./generate_input.out  1000 5  > medium_input.txt
```

For the actual performance testing program, we specify the data structure and operation on the command line. The input strings will come from a file, redirected to `std::cin` on the command line. Similarly we can redirect the output to `std::cout`. Some basic statistics will be printed to `std::cerr` for analysis. Here's how to compile and run & test the program:

```
 g++ -g -Wall performance*.cpp -o performance.out

./performance.out  vector  mode  < test_input.txt

./performance.out  vector  remove_dups  < test_input.txt  > my_remove_dups_output.txt
diff my_remove_dups_output.txt test_remove_dups.txt

./performance.out  vector  sort  < medium_input.txt  > my_medium_vector_sort_output.txt
./performance.out  list    sort  < medium_input.txt  > my_medium_list_sort_output.txt
diff my_medium_vector_sort_output.txt my_medium_list_sort_output.txt
```

The first example reads input from `test_input.txt`, uses an STL vector to find the most frequently occurring value (implemented by first sorting the data), and then outputs that string (the mode) to `std::cout`. The second example uses an STL vector to remove the duplicate values (without otherwise changing the order) from `test_input.txt` storing the answer in `my_remove_dups_output.txt` and compares that file to the provided answer. And the third example sorts the random strings generated above first using an STL vector, then using an STL list and confirms that the answers match.

Compile and test the provided code on a variety of tests of different sizes for the `vector` datatype for each of the 3 operations. The provided code uses the `clock()` function to measure the processing time of the computation. The resolution accuracy of the timing mechanism is system and hardware dependent and may be in seconds, milliseconds, or something else. Make sure you use large enough inputs so that your running time for the largest test is about a second or more (to ensure the measurement isn't just noise). Record the results in a table like this:

Sorting random 5 letter `strings` using STL `vector`

# of strings	operation time (sec)
10000	0.031
20000	0.067
50000	0.180
100000	0.402

As the dataset grows, does your predicted big 'O' notation match the raw performance numbers? We know that the running time for sorting with the STL `vector` sorting algorithm is O($n \log_2 n$) and we can estimate the coefficient from the collected numbers.

$$operation\ time(n) \quad = \quad k_{operation} \ * \ n \log_2 n$$

Thus, coefficient $k_{operation} \approx$ 2.3 x 10^{-7} sec. Of course these constants will be different on different operating systems, different compilers, and different hardware! These constants will allow us to compare data structures / algorithms with the same big 'O' notation.

Also, be sure to try different random string lengths because this number will impact the number of repeated/ duplicate values in the input. The ratio of the number of input strings to number of output strings is reported to `std::cerr` with the operation running time. Which operations are impacted by the number of repeated/duplicate values? What is the impact?

To complete this checkpoint: Show the results of your testing and analysis to one of the TAs. The data should be neat & well organized in order to receive this checkpoint.

Checkpoint 3

Checkpoint 3 will be available at the start of Wednesday's lab.