CSCI-1200 Data Structures — Spring 2022
Homework 10 — Performance and Big 'O' Notation

In this final assignment for Data Structures, you will carry out a series of tests on the fundamental data structures in the Standard Template Library to first hypothesize and then measure the relative performance (running time & memory usage) of these data structures and solidify your understanding of algorithm complexity analysis using Big 'O' Notation. The five fundamental data structures we will study are: `vector`, `list`, binary search tree (`set` / `map`), `priority_queue`, and hash table (`unordered_set` / `unordered_map`).

Be sure to read the entire handout before beginning your implementation.

Overview of Operations

We will consider the following six simple, but moderately compute-intensive, operations that are common subtasks of many interesting real-world algorithms and applications. We will test these operations using integers and/or STL `string`s as noted below.

- **Sort** - we'll use the default `operator<` for the specific data type.
- **Remove duplicates** from a sequence - without otherwise changing the overall order (keeping the first occurrence of the element).
- Determine the **mode** – most frequently occurring element. If there is a tie, you may return any of the most frequently occurring elements.
- Identify the **closest pair** of items within the dataset - *integer data only.* We'll use `operator-` to measure distance. If there is a tie in distance, you may return any of the pairs of closest distance.
- Output the **first/smallest f items** - a portion of the complete sorted output.
- Determine the **longest matching substring** between any two elements - *STL string data only.* For example, if the input contains the words '`antelope`', '`buffalo`' and '`elephant`', the longest substring match is '`ant`' (found within both '`antelope`' and '`elephant`'). If there is a tie, you may return any of the longest matching substrings.

See also the provided sample output for each of these operations.

"Rules" for comparison: For each operation, we will analyze the cost of a program/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).

Your Initial Predictions of Complexity Analysis

Before doing any implementation or testing, think about which data structures are better, similarly good, or useless for tackling each operation.

Fill in the table on the next page with the big 'O' notation for both the runtime and memory usage to complete each operation using that data structure. If it is not feasible/sensible to use a particular data structure to complete the specified operation put an X in the box. *Hint: In the first 3 columns there should only be 2 X's!* If two or more data structures have the same big 'O' notation for one operation, predict and rank the data 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.

For your answers, n is the number of elements in the input, f is the requested number of values in the output (only relevant for the '`first sorted`' operation), and l is the maximum length of each string (only use this variable for the '`longest substring match`' operation). Type your answers into your README.txt file. You'll also paste these answers into Submitty for autograding.

	sort	remove duplicates	mode	closest pair	first f sorted	longest substring match
vector						
list						
BST (`set`/`map`)						
priority queue/ binary heap						
hash table (`unordered_set`/ `unordered_map`)						

Provided Framework

We provide a framework to implement and test these operations with each data structure and measure the runtime and overall memory usage. The input will come from a file, redirected to `std::cin` on the command line. Similarly, the program will write to `std::cout` and we can redirect that to write to a file. Some basic statistics will be printed to `std::cerr` to help with complexity analysis. Here's examples of how to compile and run the provided code:

```
clang++ -g -Wall -Wextra performance*.cpp -o perf.out

./perf.out vector mode string < small_string_input.txt

./perf.out vector remove_duplicates string < small_string_input.txt > my_out.txt
diff my_out.txt small_string_output_remove_duplicates.txt

./perf.out vector closest_pair integer < small_integer_output_remove_duplicates.txt
./perf.out vector first_sorted string 3 < small_string_input.txt
./perf.out vector longest_substring string < small_string_output_remove_duplicates.txt

./perf.out vector sort string < medium_string_input.txt > vec_out.txt 2> vec_stats.txt
./perf.out list sort string < medium_string_input.txt > list_out.txt 2> list_stats.txt
diff vec_out.txt list_out.txt
```

The first example reads string input from `small_string_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 `small_string_input.txt` storing the answer in `my_out.txt`, and then uses `diff` to compare that file to the provided answer.

The next 3 command lines show examples of how to run the `closest_pair`, `first_sorted` and `longest_substring` operations. Note that the `first_sorted` operation takes an additional argument, the number of elements to output from the sorted order. Also note that the `closest_pair` and `longest_substring` operations are more interesting when the input does not contain duplicate values.

The final example sorts a larger input of random strings first using an STL vector, and then using an STL list and confirms that the answers match.

Generating Random Input

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_string_input.txt` with 10,000 `string`s, each with 5 random letters ('`a`'-'`z`'). And also a file named `medium_integer_input.txt` with 10,000 integers, each with 3-5 digits (ranging in value from `100-99999`).

```
clang++ -g -Wall -Wextra generate_input.cpp -o generate_input.out

./generate.out string  10000 5 5 > medium_string_input.txt
./generate.out integer 10000 3 5 > medium_integer_input.txt
```

Measuring Performance

First, create and save several large randomly generated input files with different numbers of elements. Test the vector code for each operation with each of your input files. 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	vector sort 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 k in front of the dominant term from the collected numbers.

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

Thus, on the machine which produced these numbers, coefficient $k_{\textit{vector sort}} \approx 2.3 \text{ x } 10^{-7}$ 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. The STL `list` sorting algorithm is also O($n \log_2 n$), but what is the estimate for $k_{list\ sort}$?

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 relative impact?

Operation Implementation using Different Data Structures

The provided code includes the implementation of each operation (except `longest_substring`) for the `vector` datatype. Your implementation task for this assignment is to extend the program to the other data structures in the table. You should carefully consider the most efficient way (minimize the running time) to use each data structure to complete the operation.

Focus on the first three operations from the table first (`sort`, `remove_duplicates`, and `mode`). Once those are debugged and tested, and you've analyzed the relative performance, you can proceed to implement the other operations.

Estimate of Total Memory Usage

When you upload your code to Submitty, the autograder will measure not only the running time, but also the total memory usage. Compare the memory used by the different data structures to perform the same operation on the same input dataset. Does the total memory usage match your understanding of the relative memory requirements for the internal representation of each data structure?

You can also run this tool on your local GNU/Linux machine (it may not work on other systems):

```
clang runstats.c -o runstats.out

./runstats.out ./perf.out vector sort string < medium_string_input.txt > my_out.txt
```

Results and Discussion

For each data type and each operation, run several sufficiently large tests and collect the operation time output by the program. Organize these timing measurements in your README.txt file and estimate the coefficients for the dominant term of your Big 'O' Notation. Do these measurements and the overall performance match your predicted Big 'O' Notation for the data type and operation? Did you update your initial answers for the Big 'O' Notation of any cell in the table?

Compare the relative coefficients for different data types that have the same Big 'O' Notation for a specific operation. Do these match your intuition? Are you surprised by any of the results? Will these results impact your data structure choices for future programming projects?

Submission

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity". If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file. Important Note: Do not include any large test datasets with your submission, because this may easily exceed the submission size. Instead describe any datasets you created, citing the original source of the data as appropriate.