Question 7 Whitepaper

Umar Kagzi

3/25/21

Question 7: C++ has several containers (https://www.cplusplus.com/reference/stl/). Pick three of them and compare similar operations by timing them on a large scale. **Explain a hypothesis, your results, and your explanation. –** I did this format for the whitepaper.

In this question, I decided to use the array, vector, and list containers in the C++ STL Library to test the time it took for each container to fill itself with random numbers, sort itself using std::sort, and finally printing themselves out with a print function.

My hypothesis was that a vector container with the aforementioned program would take the shortest time to complete since they are quite fast when inserting data into itself. After the vector, the fastest container would obviously be an array and then the final container, the list.

My method to test each container was simple. First, I timed and instantiated an array container with size of 10,000 and filled it with random numbers from 0-100. I then sorted the array using std::sort (uses quicksort, complexity N log(N)) and finally printed the array using a print function and calling the timer. I believed this process would take the containers a long time(seconds) and would give me adequate data, so I continued the same process with the other containers (vector, list).

The timings for the function are listed below, and I was surprised to see that the list had the fastest average time. Throughout the program, it had the most consistent timings, but I still feel that the vector should be faster overall since I used the push_back() function which allocates memory beforehand so the program doesn't need to earlier. I believe if I did a few more test

runs, I would see more accurate numbers. It is possible that the first test run might have some issues (which could be possible since the vector is much faster in the following tests). All in all, this was a great problem to examine the timings of different containers in C++.

In order to have average results**, I tested the function 3x to make sure they were accurate. The average results in the first three test runs were:**

Container Name	**Average Time (in Microseconds)**
Array	47,279
Vector	36,611
List	16,764

First Run (Microseconds)

List: 19331

vector: 23364

array: 39257

Second Run (Microseconds)

List: 14274

vector: 5758

array: 35853

Third Run (Microseconds)

List: 16687

Vector: 7489

Array: 66727

Question 4 Whitepaper

Umar Kagzi

3/25/21

Q4. Compare the times it takes to sort an array filled with random numbers vs a linked list via bubble sort and insertion sort.

In this question, I simply wanted to make an array and a linked list go through the two sorts and time them to find out which is faster in each case. Each array and LL was filled with random numbers and timed when each of them were sorted.

I first worked on sorting the array using bubble sort and insertion sort by filling them up with random numbers and timing the process. The given timing function by Professor Fried kept giving me the result of 0 whenever I timed it, so instead I used testing code I had made myself. After I had tested the array and logged the timings, I went on to the linked list and timed it with the same sorts.

- BubbleSortArray was a function I made to sort the array using bubbleSort. I had adapted the code from Prof. Fried's templates and GeeksforGeeks. It has the workings of a regular Bubblesort, and is simple to understand.
- insertionSortArray was a function I made to sort the array using insertionSort. I had adapted the code from Prof. Fried's templates and GeeksforGeeks.
- bubbleSortList was a function I made to sort the list using bubblesort. I had adapted the code from Prof. Fried's templates and GeeksforGeeks. This was implemented through much of Geeks4Geeks, since my own function had a number of errors.
- For the Insertion for LL, I entered in a sorted list which would perform insertions on each element. I had difficulty in implementing the actual algorithm for insertion sort, so I went with this method and timed it.

Timings:

```
Bubble Sort Array timing: 1717 milliseconds
Insertion Sort Array timing: 0 milliseconds
```

```
979 9980 9981 9982 9983 9984 9985 9986 9987 9988
Insertion Sort List timing: 78 milliseconds


...Program finished with exit code 0
Press ENTER to exit console.
```

```
[Running] cd "c:\Users\umark\Documents\VSCode\CS
Bubble Sort Array timing: 402 milliseconds
Insertion Sort Array timing: 0 milliseconds
Bubble Sort LL timing: 0 milliseconds
Insertion Sort List timing: 0 milliseconds
```

As you can see, the timings I got when running the program were incorrect and did not seem to be working besides the first sort. I had tested all the sorts individually and they all had worked fine, but when I compiled the code together, only the first timing was shown. I am not exactly sure of the issue here.

Question 5 Whitepaper

Umar Kagzi

3/25/21

Question 5. Create a multi-level sort. For instance, for all selections of n > 10 you do sort X and within sort X, when you have a situation with n < 10 you do sort Y. Be creative. Time your sort against two "reasonably comparable" sorts (you may use libraries for the "reasonably comparable sorts").

In this question, I made a simple sorting algorithm that would sort an array from lowest to highest. My purpose of the program was to make a price checking algorithm that would sort the lower and higher prices of items in a store, and at the end both sides would be sorted together.

The sorting for my algorithm was structured around a sort of linear sort, which is fast only when smaller quantities are there. In my timings, it took an average of 4110 milliseconds to sort. However, testing the function with built in libraries did not go exactly to procedure – I was having difficulty in getting the proper syntax for those separate built-in sorts. Therefore, I created an exact copy of the array I had made for testing my own sort and used it to test the libraries, which now tested with ease due to similar syntax being used in them.

The itemPrice function is the function I used to test out my function. It created the array needed to test and then sorted my array by swapping larger values. If the next value was larger, it would simply swap places and continue on with the program until there are no other values left in the array.

The other two sorts I used were from the STD library, the std::sort and std::stable_sort. Both sorts are comparable because of how quickly they sort unsorted arrays, and that is why I chose them.

For my timings, the std::sort ended up being the fastest sort, followed by stable_sort and finally my own. Because my sort had the condition (if n > 50, do a side sort), this made my sort slower than the others. In addition, I had created the array inside of the same function and timed it, which also might have led to increased timings.