**Task 2**

**Array Sort Technique Runtimes**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **10 Items** | **100 Items** | **500 Items** | **5000 Items** | **25000 Items** | **50000 Items** |
| **Bubble Sort** | 0 sec | 0 sec | 0.0005 sec | 0.0696 sec | 2.01153 sec | 8.20133 sec |
| **Insertion Sort** | 0 sec | 0 sec | 0.000502 sec | 0.02053 sec | 0.523795 sec | 2.10258 sec |
| **Merge Sort** | 0 sec | 0 sec | 0 sec | 0.000501 sec | 0.003005 sec | 0.007009 sec |
| **Quick Sort** | 0 sec | 0 sec | 0 sec | 0.000501 sec | 0.002503 sec | 0.005175 sec |
| **Heap Sort** | 0 sec | 0 sec | 0 sec | 0.001001 sec | 0.004005 sec | 0.009002 sec |
| **Counting Sort** | 0 sec | 0 sec | 0 sec | 0 sec | 0.0005 sec | 0.000999 sec |
| **Radix Sort** | 0 sec | 0 sec | 0 sec | 0.0005 sec | 0.002003 sec | 0.003001 sec |

The table above shows the results of the required array sizes. It should be noted that any item with 0 seconds obviously did not actually take 0 seconds. These tests simply went faster than the C++ chrono timer’s capabilities. Specifically, any test faster than .0005 seconds could not be accurately recorded. Because of this limitation, an additional array size test of 50000 was added to the results to see the trends more clearly.

The test results are about what was expected. It’s common knowledge that bubble sort is a very inefficient method, and this is reflected in the data by its enormously higher runtimes. Although insertion sort is a slightly better method, it still has low performance capabilities and is significantly slower than the sorting algorithms below it.

Between merge sort, quick sort, and heap sort, it was difficult to predict which would go quicker since they are all of the efficiency O(n\*log(n)). In the end, it was decided that heap sort would be the slowest since it is more complex and requires the building of an entirely separate data structure. It was then decided that quick sort would be slower than merge sort since the worst case of quick sorts is still O(n2) compared to merge sort’s O(n). While the heap sort prediction was correct, quick sort actually proved to be slightly faster than merge sort in the 25000 and 50000 item tests. Since quick sort depends on the first pivot being properly chosen, it can be inferred from this that our first quick sort pivot is properly chosen.

Lastly, it was predicted that counting sort and radix sort would be the fastest. Counting sort has an efficiency of O(n+k) where k is the range of values, whereas radix sort has an efficiency of O(wn) where w is the length of the highest value. Since these are similar and could vary depending on the data set, it isn’t clear which one would be faster off-hand. However, since our radix sort calls upon a similar counting sort and by general inspection, it was predicted that radix would be slightly slower. Our general prediction was correct, but to our surprise, the counting sort was more than twice as fast!

**Task 3**

*For brevity, one type of each ascending/descending sort option is provided; two for each sorting algorithm.*

*Students are manually seeded in Lab12Task3.cpp*

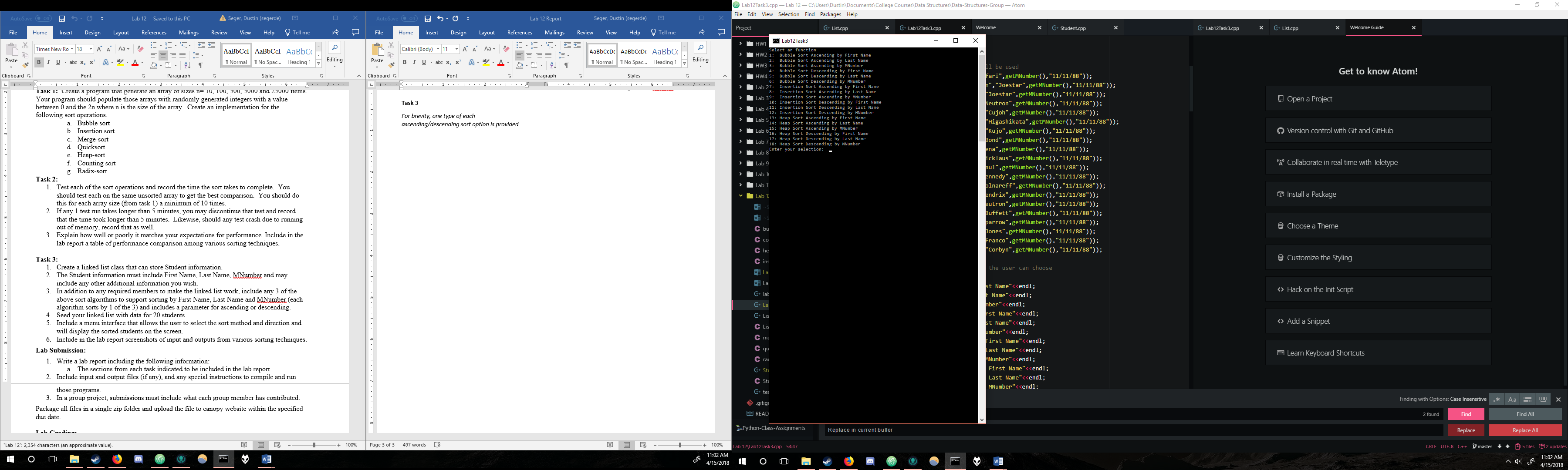


Figure 1- Selection menu (for reference)

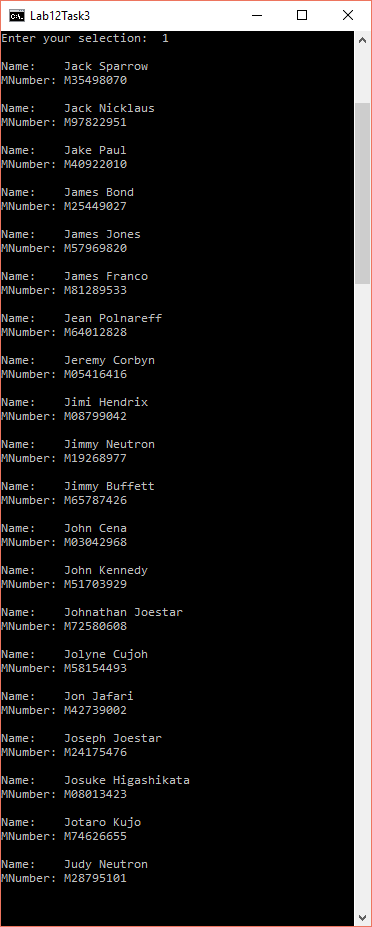


Figure 2 - Bubble Sort Ascending by First Name

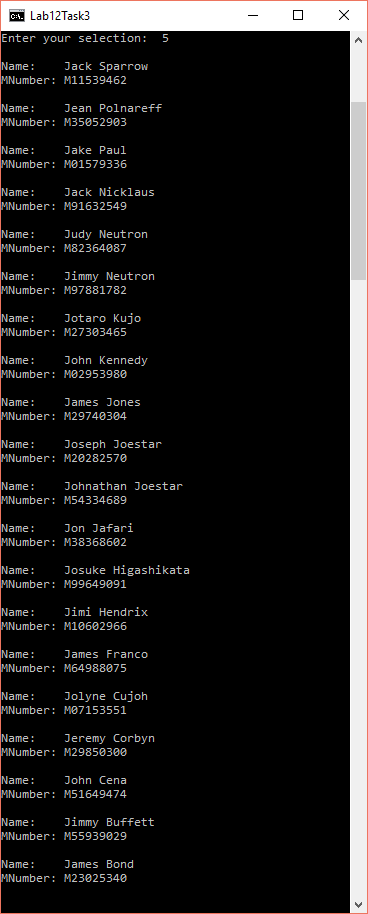


Figure 3 - Bubble Sort Descending by Last Name

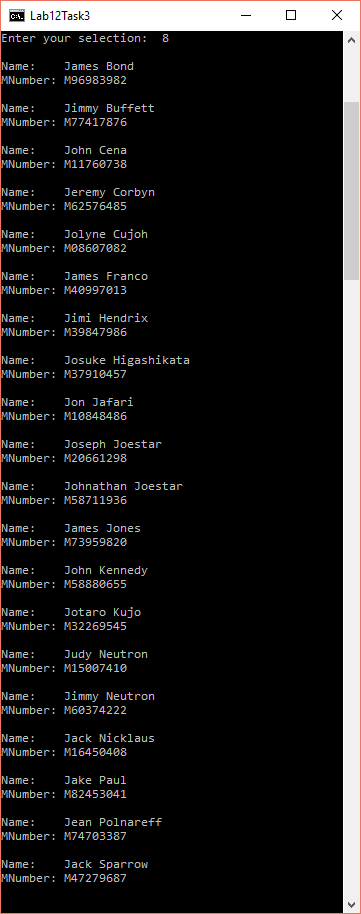


Figure 4 - Insertion Sort Ascending by Last Name

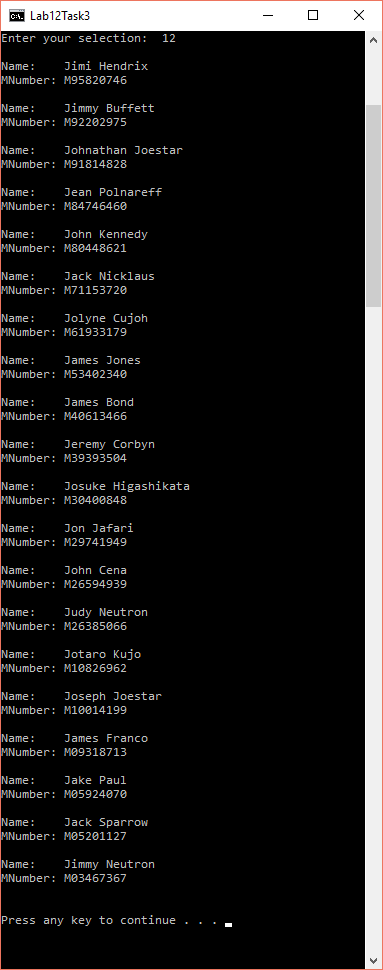


Figure 5 - Insertion Sort Descending by M-Number

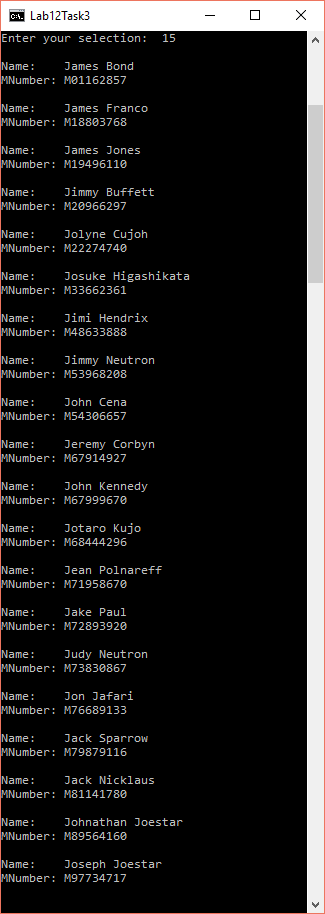


Figure 6 - Heap Sort Ascending by M-Number

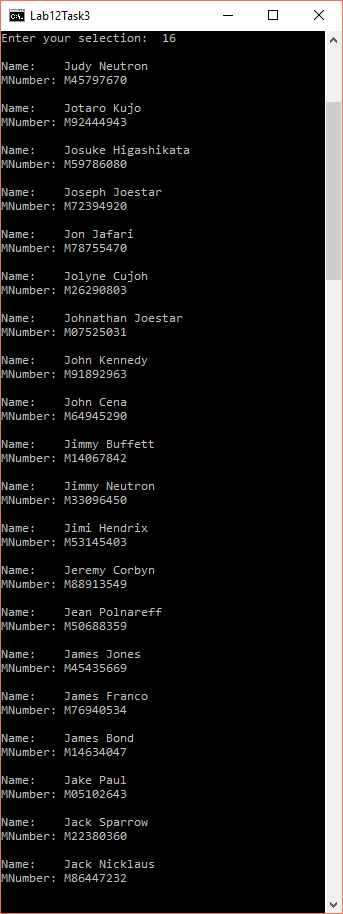


Figure 7 - Heap Sort Descending by First Name

**Compilation**

Lab12Task2.cpp contains the main for Task 2

Lab12Task3.cpp contains the main for Task 3

It should be compiled with standard gpp settings for windows.

**Contributions**

Zach wrote the heap, merge, and quick sorts, implemented the groundwork of the Task 2 and 3 mains, and adapted the sorting for our list class.

Dustin wrote the code for the bubble, insertion, counting, and radix sorts, fixed sorting bugs, and did the remainder of the Task 2 and 3 mains.

The work is equitable.