Introduction:

The objective of this assignment is to implement the map ADT using both a binary tree and a hash map. The hash map is implemented using a bucket array and separate chaining. The Binary Tree is implemented using a linked structure. These implementations are used to store character distributions designed to "learn" the probability of characters coming after certain sequences of characters. These various character distributions are used to create an output that should try and simulate the English in the input file.

Theoretical Analysis:

Both the hash table and binary search tree implementations of implement the size(),empty(),find(k),put(k,v),and erase(k) functions. The hash table implementation also includes an additional hash function to select which index of the array it should go to. The BST implementation includes a couple of helper functions for both find and put.

I will start with the HashMap's functions. Size() just returns the number of values currently stored in the map. This function takes O(1) time since a counter is stored separately and incremented or decremented when items are added or removed. The empty() function returns true if and only if the number of items in the hashmap is 0. This takes O(1) time for the same reason as size(). The find(k) function takes O(n) time in the worst case (when all of the values hash to the same bucket), but takes about O(1) time when the hash function is good and the bucket array is large enough. This function performed is to go to the index of the bucket associated with key k and loop through the values stored in the index. It will return the pair associated with k if it is found and will return a dummy pair otherwise. The put(k,v) function works in a similar way. It will first use find to see if there is already an item with key k and will update the value associated if the pair exists. Otherwise it will add a new pair into the index associated with key k. This will also take O(n) time in the worst case due to using find, but will also take O(1) time if the same good conditions for find are applied. The erase function will also take O(1) time because it will need to use find to see if an item with key k exists and if it does will delete it. This will also take O(n) time in the worst case, and O(1) time in the best case.

The BSTMap's functions have similar worst case run times. The size() and empty() functions perform the same function and run in the same time as the HashMap implementation. The find(k) function will have an O(n) worst case when all of the items form a long linear chain, but will have an O(log n) runtime if the tree is relatively balanced. The find(k) function in this case will start at the root and will perform a binary search for the key, going to the left child if the key being searched for is less than the current nodes key and will go to the right child if the key is greater. If the key is equal the node holding it will be returned and if it is not found, the dummy node where it would fit is returned. The put(k,v) function takes the same amount of time to run as the find function does. This is due to put using find in order to either update an existing node holding key k or to just insert a new node to the dummy node location found by find if no such key-value pair exists. The erase(k) function takes the same amount of time as find due to using find to locate the node that needs to be removed. It will take very slightly longer than put due to having to find the successor node if the node being removed is an internal node, but it doesn't take more time asymptotically.

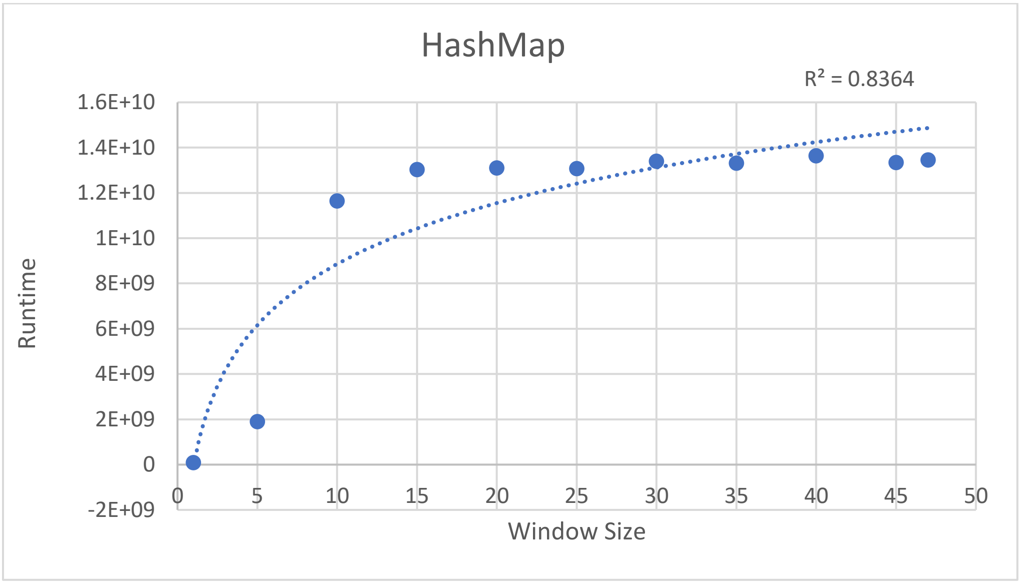
Experimental Setup:

To perform the experiments if this assignment I used a Dell Inspiron 5577 with an Intel Core i7-7700HQ CPU @2.80 GHz. This laptop has 8 GB of RAM and a 64-bit operating system. For the compiler I used Cygwin v3.0.7 and CMake v3.14.5 in combination with the CLion 2019.2.1 IDE. For this program I used C++14.

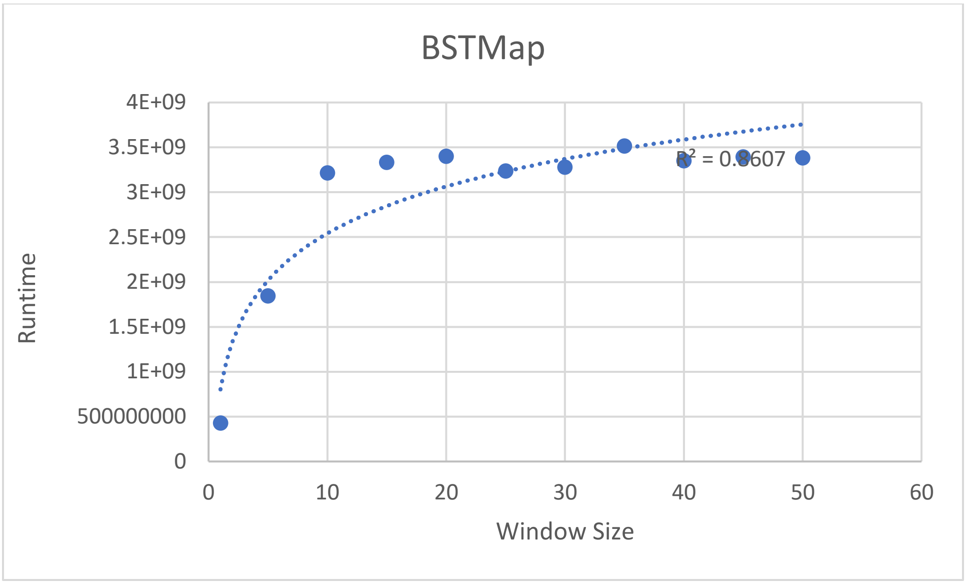
To measure the amount of time to sort n elements I used the high_resolution_clock from the C++ chrono library. I set it to nanoseconds in order to get accurate measurements on the lower test cases. I chose nanoseconds because milliseconds and coarser measurements did not give accurate results below 1000 elements.

I ran the program once for each window going from 1 to 50 in increments of 5. The length of output I selected was the length of the input "merchant.txt" file.

Experimental Results:



This graph shows the runtime of the charDistribution output using the hash table implementation of an unordered map. The function took about O(log(w)) time. As the window size increase, the amount of time actually taken did not increase much, if at all. This is most likely due to the fact that as the window size increased, more and more unique keys were created. This implementation of the hash map took longer than the BST implementation. This is almost certainly due to the size of the hash map chosen as well as the hash function chosen. With more experimentation it would likely be possible to bring down the runtime significantly. One issue with the implementation used here was that for some reason It would not work for windows above 47. I am not sure what is causing this issue, but I was unable to get it fixed due to discovering it late into the testing.



This graph shows the runtime of the charDistribution output using the binary search tree implementation of an unordered map. The function took about O(log(w)) time. As the window size increase, the amount of time actually taken did not increase much, if at all. This is most likely due to the fact that as the window size increased, more and more unique keys were created. This implementation of the algorithm was faster than the hash map implementation. This is likely due to not having to deal with a large number of collisions like the hash map implementation does.

For both implementations of the algorithm, the output text was much closer to actual English when the window size was larger. It took about an input size of 10-15 to get very close with errors happening consistently, but not making the output completely unreadable. At a window of 20-25, the output was very close to the input with scarce errors mostly concentrated on the beginning of words. This is due to the number of unique keys increasing dramatically as the window grows larger.