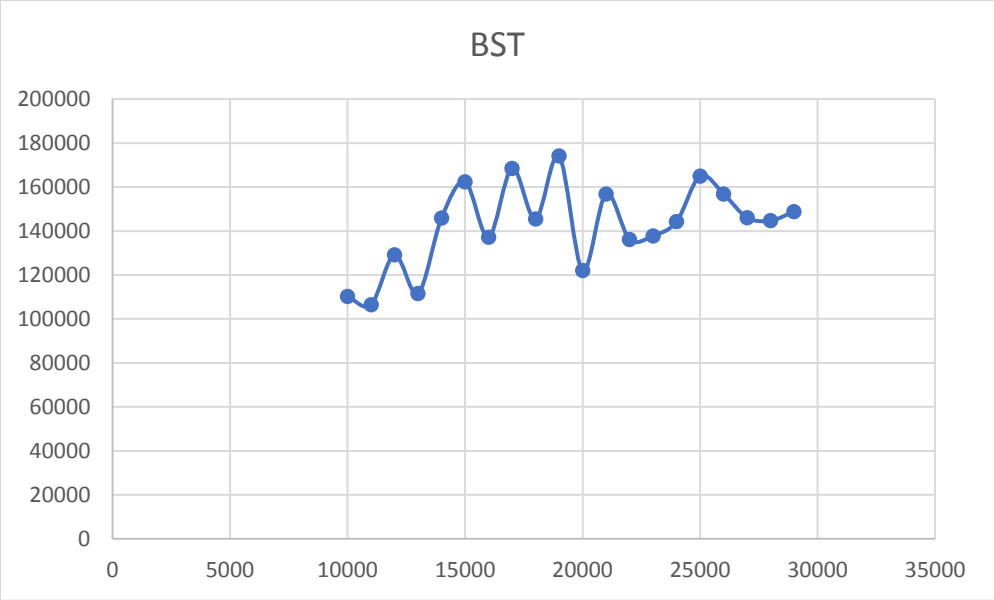
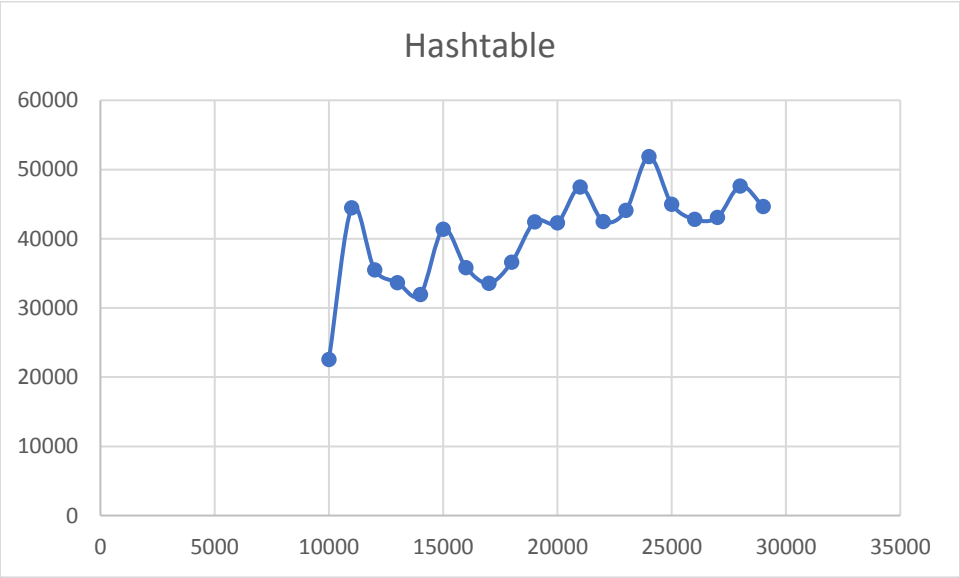
Part 3:



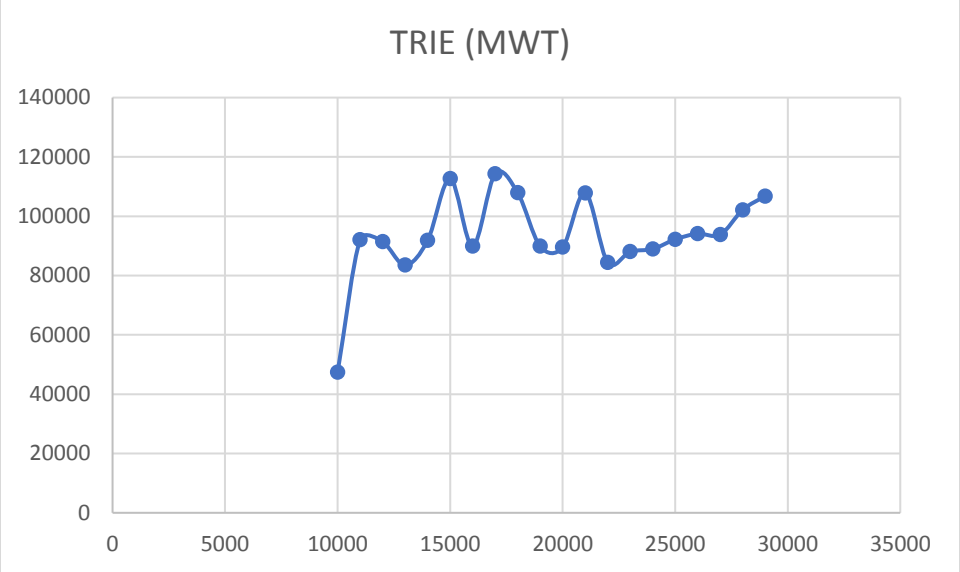
From the above graph, it shows that we have taken 10000 as the minimum size, 1000 as the step size and 20 as the number of iterations. The runtime ranges from 110000 to 180000 ns disregarding the uniquely large or small numbers. We can see a little bit of the trend that larger size will cost us more time somehow to finish the process. The average runtime of find in BST is O(log(n)). We can also see our graph grow faster in smaller values and slower in larger values. That also matches the expected output despite the effect by the background programs to make our results look kind of unstable.

Hashtable:



From the above graph, it shows that we have taken 10000 as the minimum size, 1000 as the step size and 20 as the number of iterations. The runtime ranges from 33000 to 50000 ns disregarding the uniquely large or small numbers. The average runtime of find in Hashtable is O(1). We can also see our graph is not even close to this average runtime. Maybe it's due to multiple collisions through separate chaining that makes our results closer to the worst-case scenario. Maybe it's also due to the unstable background programs in the computer. Nevertheless, that still matches the expected output despite the effect by the background programs to make our results look kind of unstable.

Trie(MWT):



From the above graph, it shows that we have taken 10000 as the minimum size, 1000 as the step size and 20 as the number of iterations. The runtime ranges from 83000 to 116000 ns disregarding the uniquely large or small numbers. We can see a little bit of the trend that larger size will cost us more time somehow to finish the process. The average runtime of find in MWT is O(k), where k is the longest word to be searched. Our result is not close to the expected result which should be way faster. The reasons behind might be due to our implementation of find method and also the effect of the background programs. We can also see our graph don't differentiate a lot regarding the change of the values of size. The runtime only relates to the longest length of the string in the dictionary, but not how large the dictionary is. This could be also shown in our graph.

Part 4:

(a)

Hash Function 1 (Quick Hashing): This hash function takes all the ASCII values of each characters of the input strings and sums them together. We can get the hash value by finding the modulus of the final sum divided by the size of the table.

Hash Function 2 (Unique Hashing): This hash function takes in a specific value of seed chosen by the user. And it sums up the old sum times the seed and the ASCII values of each characters. We take the final sum and get the modulus of it divided by the size of the table again.

Source for both functions:
https://stackoverflow.com/questions/8317508/hash-function-for-a-string

(b)

We calculate the expected values for all the three cases with the two hash functions in the scratch paper first, and then we can compare our outputted results to our calculations.

The first test case is using “hello”, which has a total ASCII value of 500. We also set the table size as 10. In this way, no matter how large is the seed we input for the second function, we will still get 0 as the modulus for both functions. Our output also matches the expected result.

The second test case is using “Whatever…” which consists of upper case letter and non-symbol value. It should not make a difference from using pure alphabets since dots and upper-case letters should also contain specific ASCII values. We set the table size as 20. Quick Hashing hashes the string to index 16 and Unique Hashing hashes it to index 12. Our output also matches the expected one.

The last test case is using “sleep late” which consists of space between two words. It should not make a difference from using pure alphabets or even other symbols since space also contains its specific ASCII value. We set the table size as 30. Quick Hashing hashes the string to index to 1 and Unique Hashing hashes it to index 27. Our output matches the expected one.

(c)

Printing the statistics for QuickHashing with hash table size 1000

#hits	#slots receiving the #hits
0	898
1	82
2	18
3	2

The average number of steps for a successful search for QuickHashing would be 1.75

The worst case steps that would be needed to find a word is 3

Printing the statistics for UniqueHashing with hash table size 1000

#hits #slots receiving the #hits
0 911
1 76
2 11
3 2
The average number of steps for a successful search for UniqueHashing would be 1.75

The worst case steps that would be needed to find a word is 3

Printing the statistics for QuickHashing with hash table size 2000

#hits #slots receiving the #hits
0 1799
1 158
2 38
3 3
4 1
5 1
The average number of steps for a successful search for QuickHashing would be 2.66667

The worst case steps that would be needed to find a word is 5

Printing the statistics for UniqueHashing with hash table size 2000

#hits #slots receiving the #hits
0 1806
1 170
2 24
The average number of steps for a successful search for UniqueHashing would be 1.33333

The worst case steps that would be needed to find a word is 2

Printing the statistics for QuickHashing with hash table size 20000

#hits #slots receiving the #hits
0 18267
1 316
2 251
3 211
4 173
5 189
6 182
7 131
8 92
9 76
10 46
11 36
12 11
13 9
14 5
15 4
16 1
The average number of steps for a successful search for QuickHashing would be 8.05882

The worst case steps that would be needed to find a word is 16

Printing the statistics for UniqueHashing with hash table size 20000

#hits #slots receiving the #hits
0 18174
1 1557
2 240
3 26
4 3
The average number of steps for a successful search for UniqueHashing would be 2.2

The worst case steps that would be needed to find a word is 4

Printing the statistics for QuickHashing with hash table size 200000

#hits #slots receiving the #hits
0 197157
1 209
2 138
3 109

4	84
5	77
6	78
7	72
8	70
9	50
10	32
11	54
12	43
13	38
14	39
15	21
16	26
17	27
18	27
19	31
20	30
21	27
22	28
23	33
24	19
25	28
26	24
27	22
28	16
29	33
30	18
31	24
32	22
33	26
34	18
35	26
36	23
37	23
38	23
39	24
40	18
41	13
42	19
43	19
44	7
45	14
46	21
47	14

48	20
49	25
50	17
51	22
52	24
53	23
54	12
55	36
56	26
57	23
58	29
59	27
60	25
61	15
62	27
63	21
64	38
65	27
66	29
67	32
68	23
69	28
70	26
71	38
72	29
73	24
74	20
75	28
76	23
77	28
78	17
79	20
80	18
81	27
82	22
83	12
84	20
85	9
86	11
87	11
88	7
89	16
90	4
91	6

92 9
93 6
94 5
95 4
96 1
97 3
98 2
99 1
100 1
101 3
102 3
106 2
110 1
The average number of steps for a successful search for QuickHashing would be 52.0952

The worst case steps that would be needed to find a word is 110

Printing the statistics for UniqueHashing with hash table size 200000

#hits #slots receiving the #hits
0 182005
1 15216
2 2434
3 311
4 31
5 2
6 1
The average number of steps for a successful search for UniqueHashing would be 3.14286

The worst case steps that would be needed to find a word is 6

(d)

We have tested four cases with four significantly different input sizes. 1000,2000,20000 and 200000. For the case of 1000, the number of steps and the worst case of steps taken between two functions have not shown a difference yet. But from 2000 till 200000, we can notice a significant difference. Unique Hashing is way more efficient than Quick Hashing with just an extra seed to find the modulus. The difference is growing like a graph of $y=x^n$, whereas n is an integer and $n > 0$. When we use larger size of table eventually, it shows a larger difference. This depicts that a too simple hash function might not be as efficient as you think, it's important to pick the right one. The output also matches our expectation since we intentionally find two hash functions that have huge different outputs, and the second one should be faster.