Hashing Report

Ho Suk Lee (Max), 719577

1.
Assuming we are using separate chaining using a MTF linked list, insertion takes O(1) because you only have to make the new element as a head of the list by making it point to the head of the current, then make the bucket point to the new head (the new element). These operations take constant time to do.

The upper bound of the search is O(n) since the element finding could be at the end of the list and to reach the end in linked list, we are required to traverse through all the linked elements.

2.
Insertion still takes O(1) for the same reason as number 1.
Upper bound of search is O(n/size) because each bucket has now n/size elements as the input was perfectly spread over all buckets and the element to find is potentially is the last element in the linked list.

3.
Insertion takes O(1) for the same reason as number 1, only difference is that the new element has the same value as the existing ones.

Search takes O(1). Two different inputs never hash to same bucket, meaning each input has its own unique bucket. Therefore, the element we search is guaranteed to be either in the first hash if it exists in the table. If it is not, the index must be empty since no other value can take place in it.

4.
Any string that starts with the same character hashes into the same bucket, making the search potentially extremely inefficient since search in linked list takes O(n), n = elements in list, as explained previously.

Example of such string input is:

Algorithms are fun
Alistair Moffat
Alice in the Wonderland
Adafruit
Ani

5.
During initial insert, O(1 * MAXSTRLEN) = O(1) time is taken to generate the random numbers, assuming mod operation takes O(1). Each input has maximum length, MAXSTRLEN. Then, each input takes upper bound O(MAXSTRLEN) to calculate the sum then mod, assuming all integer operations take O(1) time. Therefore, first insertion takes O(MAXSTRLEN + MAXSTRLEN * n) = O(n). After that, the

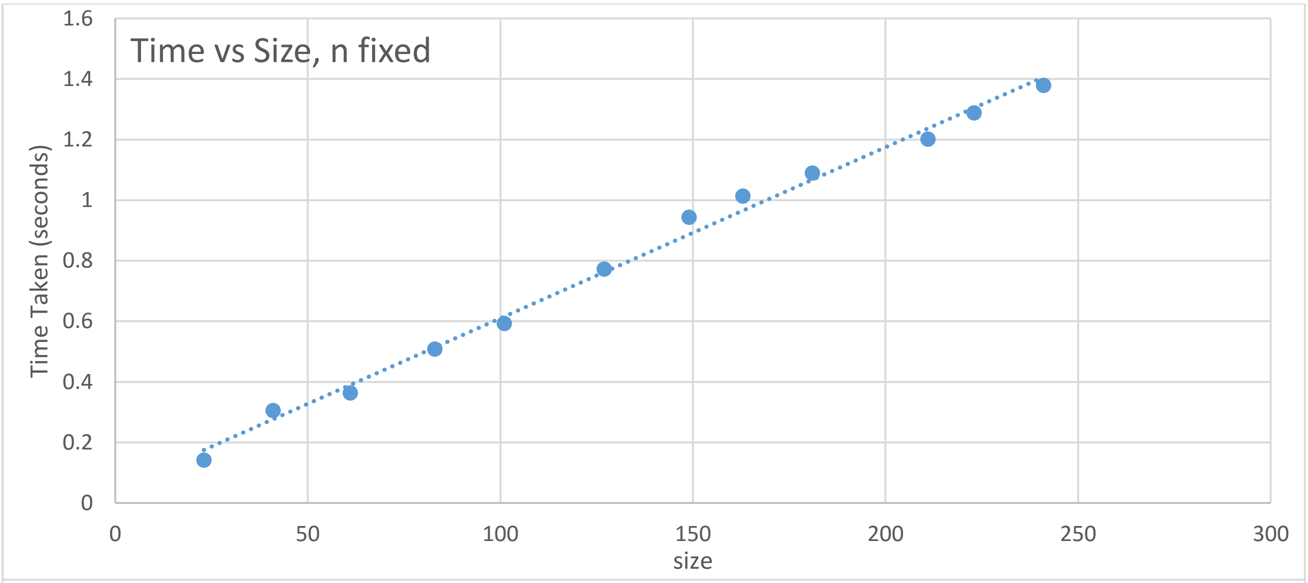
insertion takes O(MAXSTRLEN * n) = O(n).

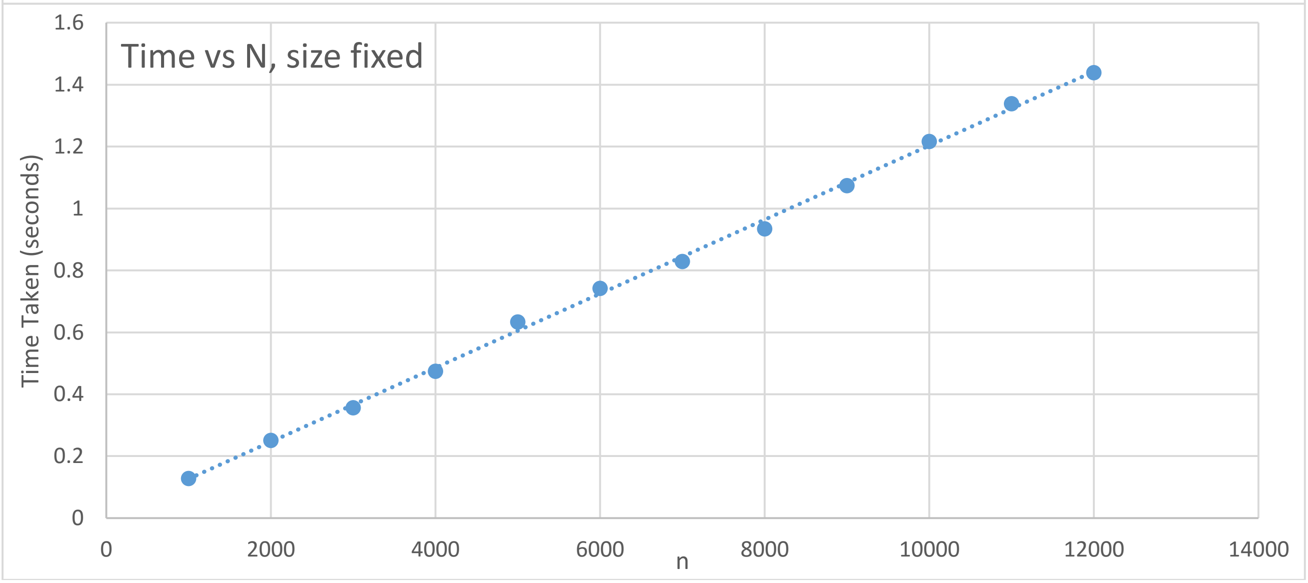
Search takes O(1) because calculating the sum takes O(1) time as explained previously, assuming no collision has taken place during hashing.

6.
The function first initialises the universal hash function with the provided seed, making sure it has random constants we expect it to. This is important since collide_dumb uses rand().

Then, it randomly assign random size within range, then assign random values within range to each characters, then check if it hashes to 0. If it does, print the value and start again with new size. If it doesn't, for the same size, we randomly re-assign values and try again. If we tried until certain number of times and still could not find a string that hashes to 0, we change the size and try again.

Theoretically, the function can run for indefinite amount of time due to its randomness. However, in real life, it shows O(n + size) behaviour according to the data collected.





7.
The algorithm works with the following steps:

1. Set random size

2. Fill in the characters between index 0 to \n with random characters

3. Calculate constant[i] * char[i] from index 1 to \n and add them up.

4. Calculate the value (let as A) required to make (sum + A) % size = 0 by A = size – (sum % size).

5. Now we know that we should satisfy constant[0] * char[0] = A mod size.
   constant[0] * char[0] = A + size * d for any integer d.
   constant[0] * char[0] + size * d = A

   Then, solve Extended Euclid for constant[0] and size.
   This gives X and Y such that constant[0] * X + size * Y = 1 because size is prime and gcd(size, any integer) gives 1.

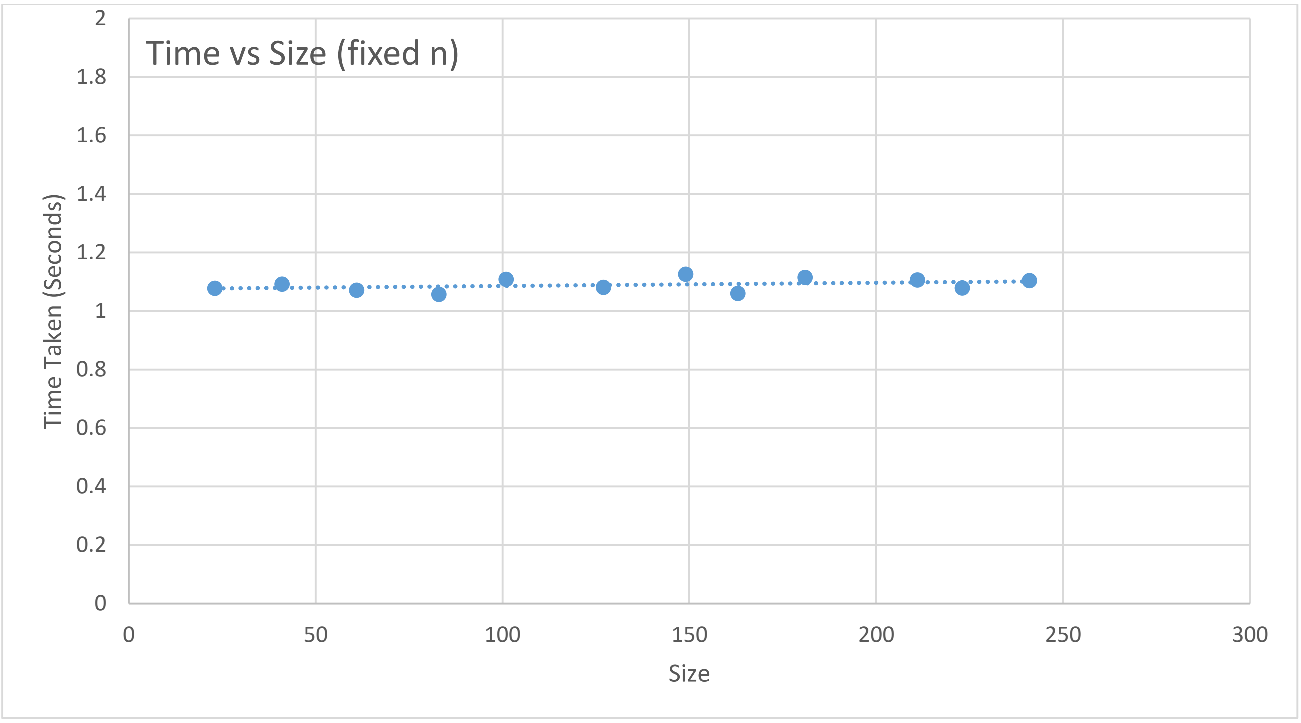
6. Multiply the above equation by A, constant[0] * X * A + size * Y * A = A.
   Now we know that A * X is the character to fill up for index 0.

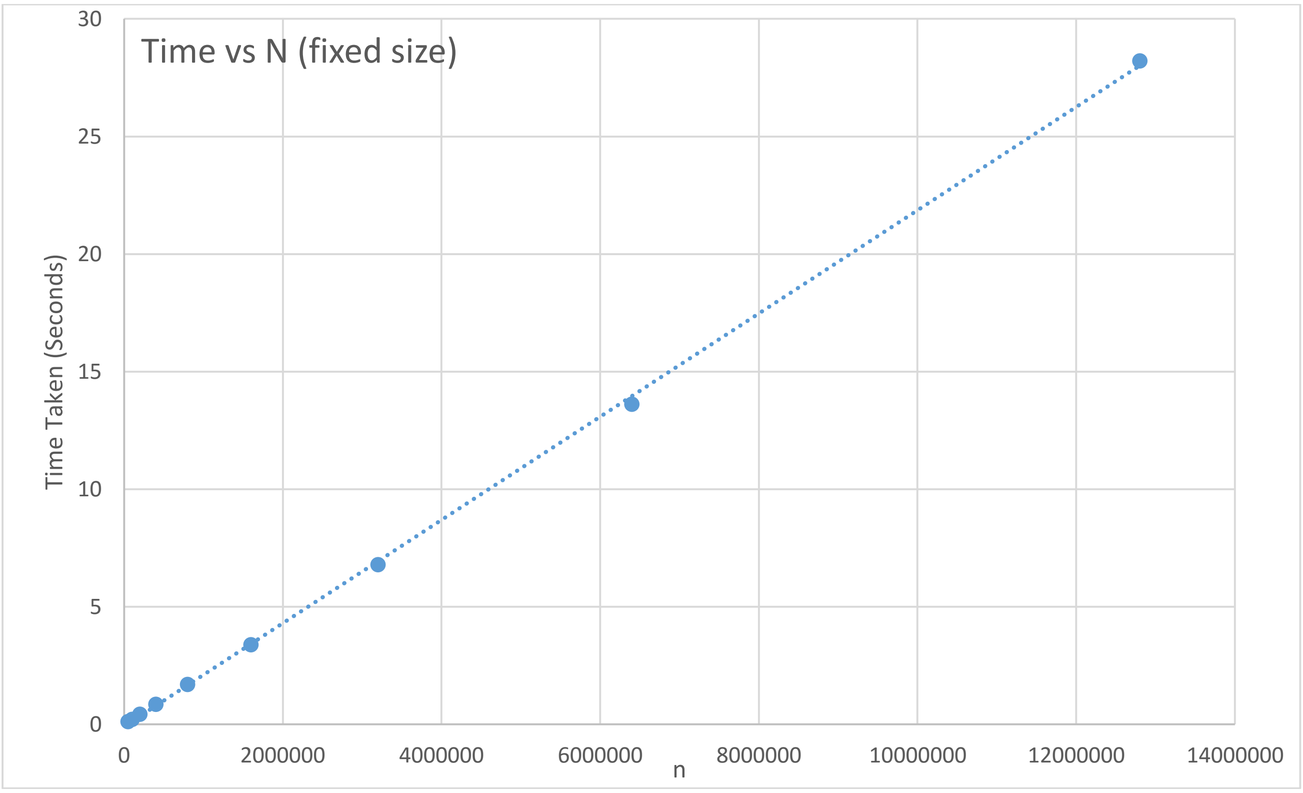
7. We try to make A * X fall into given range of ASCII code for output *prettiness*, by adding appropriate multiple of size. char[0] = A * X + d * size for some integer d.

   If char[0] = \n or DEL character, we attempt to add size to get another solution. If it fails, we ditch the solution and try again because those two characters break the string output therefore loads differently when reading. This has almost no impact on the over runtime of the algorithm because this has only 0.3% chance of happening.

As shown in above steps, the algorithm only uses size for the mod, + and – calculations. Therefore, complexity with size is O(1), assuming mod operation takes O(1) time. This is reflected in the data collected, as the time taken remains constant over fixed values of n with varying size.

True upper bound of the complexity is O(n) a constant time operation is carried out n times. This is also reflected in the data shown in the next page.


Time vs Size (fixed n)
Time Taken (Seconds)
Size
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
50
100
150
200
250
300


Time vs N (fixed size)
Time Taken (Seconds)
n
0
5
10
15
20
25
30
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000