1 1 120

Q1

$h(x) = x \bmod 13$

a)

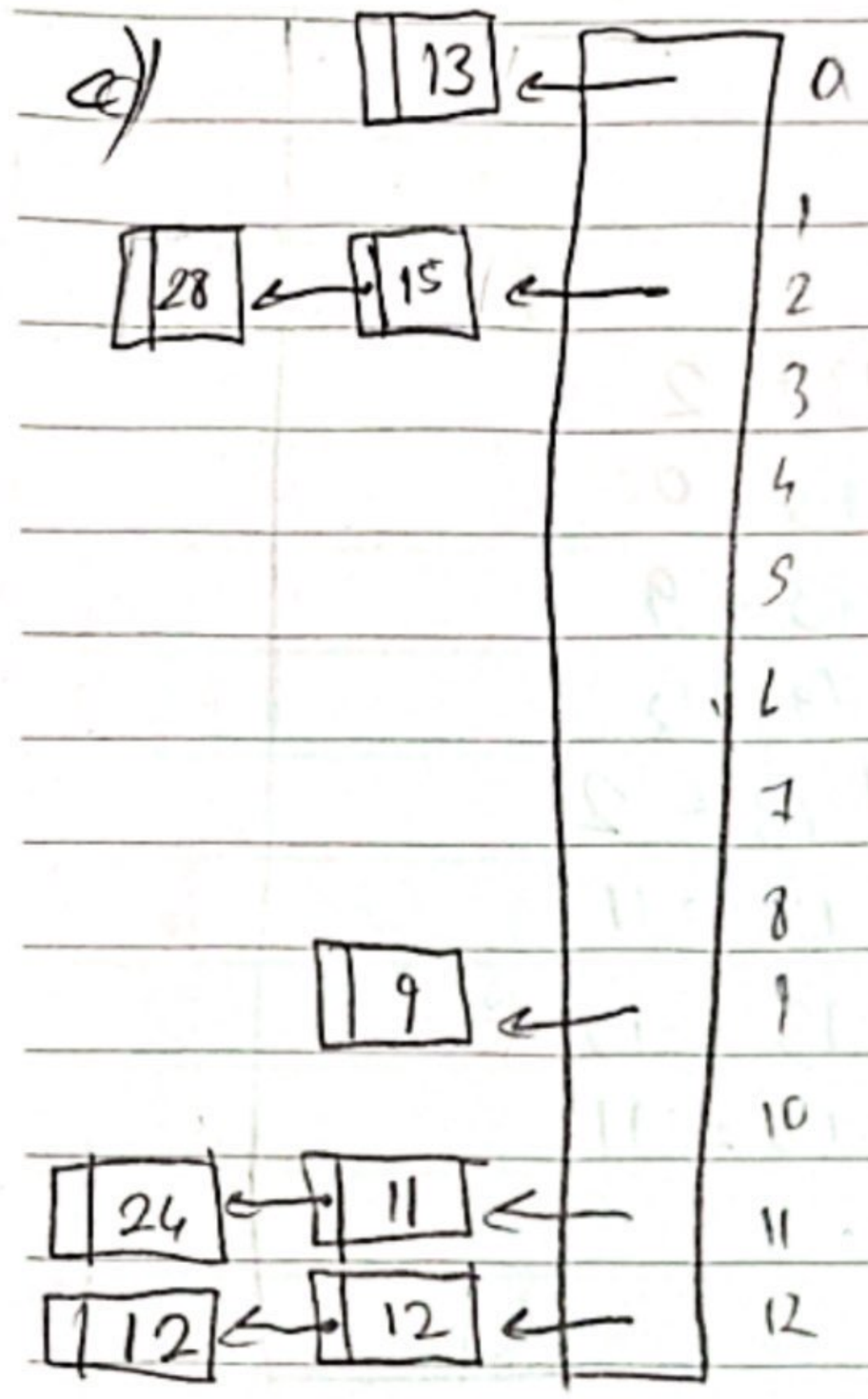
13	0
25	1
15	2
28	
24	
9	9
	10
11	11
12	12

15 mod 13 = 2
13 mod 13 = 0
9 mod 13 = 9
12 mod 13 = 12
28 mod 13 = 2
11 mod 13 = 11
25 mod 13 = 12
24 mod 13 = 11

b)

13	0
	1
15	2
28	3
	4
	5
	6
24	7
25 ~~5~~	8
9	9
	10
11	11
12	12

15 mod 13 = 2
13 mod 13 = 0
9 mod 13 = 9
12 mod 13 = 12
28 mod 13 = 2 $2+1^2=3$ ✓
11 mod 13 = 11
25 mod 13 = 12 $12+1=13 \to 0$
$12+4=16 \to 3$
$12+9=21 \to 8$
24 mod 13 = 11 $11+1=12$
$11+4=15 \to 2$
$11+9=20 \to 7$



15 mod 13 = 2

13 mod 13 = 0

9 mod 13 = 9

12 mod 13 = 12

28 mod 13 = 2

11 mod 13 = 11

25 mod 13 = 12

24 mod 13 = 11

Q2)

```
bool isPrime(int num){
    if(n<=1)
        return false;
    for(int i=2; i<=n/2; i++)
        if(n%i==0)
            return false;
    return true;
}

int nextPrime(int num){
    if(num<=1)
        return 2;
    int prime=num;
    bool found=false;
    while(!found){
        prime++;
        if(isPrime(prime))
            found=true;
    }
    return prime;
}
```

```
int minTableSize(int[] array, int n) {
int current;
if (isPrime(n))
current = n;
else
current =     nextPrime(n);
bool found = false
while (!found){
 for(int i=0;i<n;i++){  bool a=false;
    for(int j=i+1;j<n;j++){
      if(array[i]%current == array[j]%current){
        current = nextPrime(current)}

        a= true;
        continue;
      }
      if (a){ continue;}
    }
    return current;
}
}
```

Alara Zeybek
22102544
cs202-hw3

Question 4 - 15 points In this question, you will analyze your solutions for Question 3 for each subtask. You must discuss the question, discuss about the worst case scenarios. You also need to discuss your solution. You need to talk about your hash table and features of hash table. You are expected to mention hash functions. Also, you need to talk about how you handle collisions. What have you done for the collision scenarios? You MUST NOT write this part in only 2-3 sentences. It should be detailed enough to get a full credit

Implementing a solution for Question 3 probably entails efficient data storage and retrieval, frequently with an emphasis on string manipulation and search functions. Large datasets or input strings are usually used in the worst-case scenario of this kind of problem, which can lead to higher memory utilization, slower runtime, and possible performance bottlenecks. However, I will talk about the worst cases of each subtask separately.

Hash tables are perfect for situations where having rapid access to data is crucial since they offer average-case lookup and insertion operations in a consistent amount of time. Therefore, in each subtask I worked with hash tables and hash functions.

Subtask 1
In this task I implemented a hash table using separate chaining. To handle collisions, implementing chaining is straightforward and effective for most scenarios. (I coded the linked list for the buckets in the hashtable.) In this question my main idea was first adding the input words into the hash table. Then getting the hash value of each prefix and suffix of each word and checking if there is a collision or not in the hash table. As a hash function I get the ASCII values of each char and take the module respectively to the table size. This process takes O(total_num_of_characters). Since I am doing the process for all the inputs, the overall complexity is O(input_num*total_num_of_characters) but here input_num is a respectively a small number therefore I consider it as O(1) and the overall runtime complexity becomes **O(total_num_of_characters).**

Subtask 2
In this task I am mainly using the hash function. Inorder to increase the time management, I am not storing each string in a hash table and instead of this , I am creating a struct object that contains {tuple_id, tuple_string, tuple_hash_val1, tuple_hash_val2}. I am storing these objects in a tuple_array (while putting them in array, I also calculate the k value). I went with this approach because all I needed was the hash_function results and implementing the whole separate-chaining hash table was unnecessary. In addition in this subtask I am also using an additional array called frequency_arr to keep the frequency values of each tuple.
After I initialized the tuple array, I sort them by first looking at their tuple_hash_val1 value and if they have the same, I look at the tuple_hash_val2 value. Here I use std::sort() and it takes O(nlogn) time. After sorting, I traverse the substrings of text having length of k from left to right and do it with k+1,k+2,k+3,k+4 and k+5. Each time I take the hash_val1 value and hash_val2 value of the substring and check if the same hash_values exist in the tuple_array. Here I am

Alara Zeybek
22102544
cs202-hw3
using binary search and it takes O(logn) time. If I found an object having the same hash values, I return its id and I increment the value of frequency_arr[tuple.id]. At the end, I am simply print the values in the frequency array.

Overall this approach takes n O((n + m) * log m + total_number_of_characters). O(total_number_of_characters) comes from hash functions and traversing the whole string. O(mlogm) comes from sorting (binary search is O(logm) since it is smaller than O(mlogm) we can ignore). In this calculation the relatively smaller values are ignored (i.e. k).

In this subtask, I handled the collisions by using two hash_functions. The main template of my hash functions is polynomial rolling hash formula.This hash function aims to produce unique hash values for different strings while minimizing collisions. However, it's essential to note that collisions are still possible and therefore I am using two hash functions with different modulo values. One is 1e9+7, and the other is 1e9+9.

Subtask3
In this task, first I collect all the input strings into a tuple array with their unique_hash_value1 and unique_hash_value2. Let's say the tuple array size is n. I also have a visited array with the same size. Then it iterates through each tuple, rotating and verifying that there are no clashes between the tuples to determine minimum subsets. By rotating and reversing tuples, it counts reverse operations and updates counters appropriately. The computed results are printed when it finishes. Since each for loop operation is less than O(total_num_char), we can ignore the complexity summation. Then, we have O(total_num_char) from the calculation of hash values. To prevent possible collision, I used two hash_functions. The main template of my hash functions is polynomial rolling hash formula.This hash function aims to produce unique hash values for different strings while minimizing collisions. However, it's essential to note that collisions are still possible and therefore I am using two hash functions with different modulo values. One is 1e9+7, and the other is 1e9+9.