CS202
HW 3 Report
Halil Arda Özongun
22202709
Section 1
13 December 2024

Question 6

The question says that patterns only use a,b,c,d. The alphabet used only changes the hashing function. In case only a,b,c,d is used, the hashing methods can be optimized by using different hashing methods. But in my case, this does not change anything.

The second constraint, between minimum length and maximum length, is limited to 5. Knowing this constraint changes only the efficiency of the problem. If it were not so, the efficiency of the problem would include another multiplier directly proportional to the input size. That is, the given complexity is not done with O (($N + M$) * $log\ N$ + $total\ number\ of\ characters$), but it will be done with O(N*($N + M$) * $log\ N$ + $total\ number\ of\ characters$).

Question 7

For this question, the reason for doing a different calculation is that some numbers are dependent on other numbers, depending on the inputs coming in different places. If none of the inputs depend on each other, there is no problem and we don't need to deal with additional cases. We can find the max state by multiplying all the numbers directly: O(N!)

For minimum value, all numbers expect a group of numbers might be invalid (-1). So, you have only this group to have different combinations.That group might be so small, like 1 to 3, and this 1 or 2 or 3 sized group might be dependen to each other, so they may inserted only in 1 way.

Question 8

In this question, you will analyze your solutions for each question. 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 strategies. You are expected to mention hash functions. Also, you need to talk about how you handle collisions. What is the main advantage of hashing in these questions ? 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.

Question 1:

A brute force approach to this question would be to try the string for each pattern. In this question, by taking a hash-oriented approach, we can calculate the strings only once instead of comparing them continuously. Therefore, using hashing can help us in this question. Also, when we look for a certain size while iterating the string, we can get the hash values of the new window in O(1) by simply subtracting the subtracted element from the hash value and adding the added value, instead of recalculating the new string we get when we move it left and right.

Using all this information, let's solve the problem as follows. First we need to hash all the patterns and save them somewhere, then we need to examine this for each bit

window size. For this question this is given as a special case, it is said that the difference will be max 5, otherwise this would affect the efficiency by a factor of N. Here, we need to calculate the hash for a length m. Then we need to binary search all patterns to see if the hash we calculated exists (logM). Since we are going to look with binary search, we need to sort them first (MlogM). After looking, if we find it, we should increase the number, if not, we should move to the next window with Rolling hash. We should do this until the end of the string. O(N). O(NlogM) for length N, with O(1) calculation in each, and O(logM) search in total O(NlogM).

When implementing the hashing logic I kept two hash tables in case of collisions, and this number can be used for multiple hash functions depending on the amount of input.

Question 2:

Some of the pattlerns given here are shifted or reversed and some are independent of each other. We try to calculate the number of independent groups. Since we want the shifts to be done from the beginning, we first need to find a unique hash value for a string and all its shifts. We can do this by computing the hash of the string for all its shifts and then taking the minimum of these numbers. It is better to use Rolling hash here, so we need only length of string time operations. After hashing all the strings, we have more groups than we want because if we reversed some strings, we would get the same string as the other strings and this would reduce the number of groups. So we can first sort these hash values to reduce our search speed for each of them to logarithmic time, sorting o(nlogn). After sorting them, we can start the grouping process. With a linear traverse, we can do the following: We can move forward as long as the hash values are identical (because they are all in a group) and then we can save the count of the set, to an hash of their reverse. This way, as we go through all sorted hashes, we can look in the table of reverses to see if there is a reverse group that matches them, and when we find an intersection with reverses, we can take them as a group.

Question 4:

The logic here is that if a number is in a different place than it was, it is either impossible or the number is in a different place because it comes after other numbers. In order to insert a number, all values between that number and its original location must be inserted. So there is a prerequisite logic here. Here we can keep a graph, and with this graph we can do a topological sort. When doing topological sort, if we do it according to lexiographic order, we get the answer we are looking for.

Question 5:

We should take advantage of a very important detail in this question, the question says that each number must be insertable at the 3rd time at the latest. So if we think of the inserted numbers as a graph, the path between the numbers is max 3 long. What we can conclude from this is that the numbers are highly independent, and if we pay

special attention to the connected ones, we can calculate the total state. If there were no dependencies in the calculation, we could just calculate N! for N objects, but there are some cases that limit us, we can divide them and do the calculation correctly.

Special cases:

1- Number in the right place -> number one more than the right place

2- Number in the right place -> number in the right place -> number two more than the right place

3- Number in the right place -> number in one more than the right place -> number in two more than the right place

4- Number in the right place -> number in one more than the right place -> number in one more than the right place

For the first case we have to divide the total by two, for the second case by three, and for the third and fourth cases by six. We can count these cases with a for loop. O(N). After counting them, we can get the result directly by calculating both N! and the divisors. Here, since we may encounter interger overflow, we can get the correct result by multiplaying until end, and not multiplying these numbers as many times as necessary when we encounter two or three.