CS202 Homework 3

Hilal Coşgun

22201571

Section 1

Q6.

In the first question, the difference in length between the patterns was at most 5, and each pattern consisted of the letters a, b, c, and d.

If the difference in lengths were not been restricted to 5, I would need to detect all possible pattern lengths and apply them to sliding window approach. In other words, I would use every possible window sizes to scan the text. Therefore, if the number of possible lengths were not a fixed constant like 5 but instead a variable k, the system's complexity would be multiplied by k. I would not achieve the required efficiency. Probably, efficiency would be $O((N + M) * klog\ N + total\ number\ of\ characters)$.

If the alphabet was not limited with 4 letters, hash values of patterns would grow higher. They would need to applied mod much more times, therefore the collision probability would also grow. In other words, bigger numbers would be squeeze into the mod values(10e9+9, 10e7+7) and would collide more frequently. Because of this, even double hashing would not be enough and I would probably check equality char by char and if max pattern length is k, complexity would be $O((N + M) * klog\ N + total\ number\ of\ characters)$.

Q7.

Maximum number of possible insertion orders of a given hash table is N!. This scenario may occur if every value in the hash table is in the right place, no one shifting each other. In this sceneario, there would be no dependencies and every value would be free to be inserted any time. Therefore the number of possible insertions would be N!.

If a valid hash table with size N is given, the scenario for minimum number of insertions is like that: Each three value would be dependent each other, but not all N values. Because, if all values were dependent then the one at the end of depedency chain could only be inserted at Nth order, which is against the condition of question. Therefore, I believe the most dependent hash table with this condition would have A->B->C dependency for each block of size 3. Therefore each block of size 3, would have only 1 insertion order. Dividing the N! to 3! for each block we acquire:

$N!/ 6^{N/3}$

If an invalid hash table is given, minimum number of insertions becomes zero.

Q8.

q1- In the first question, I began by reading the input file and computing the hash values for all patterns. To calculate these hash values, I used the Rabin-Karp hashing algorithm combined with double hashing. This algorithm is well-suited for minimizing hash collisions,

and the addition of double hashing made collisions nearly impossible. By resolving hash collisions in this way, I avoided to cost of directly comparing strings char by char. Finding hash of one individual pattern is O(n) with Robin-Karp algorithm due to handling of each position and adding them up. I stored this hash values in a 2D array, the outer array is consisted of arrays of size 3, holding hash value 1, hash value 2 and index of pattern. Then I sorted this array with respect to first hash value to make it easier to find any hash value if necessary. I used c++ implemented sorting algortihm which Works in O(NlogN). While I process the patterns I also detected the smallest length of patterns. Then I started to scan the text with a sliding window, for window sizes starting from smallest index to smallest index+5. Sliding window approach can scan a text in O(N) times. For each window size I calculated the first string's hash value with Rabin-Karp with O(length), after finding the first one, I used the rolling-hash algorithm which takes only O(1) for each string. After finding hash value of a string I tried to find a match in the sorted array using binary search. Binary search costs O(logN), when I found a match of first hash value, I also checked the equality of second hash value. If true, I incremented the occurence count of corresponding pattern. As a result, total complexity of this program becomes $O((N + M) * log\ N + total\ number\ of\ characters)$. Using hash is advantageous because without hash, I would need to compare each string to each pattern, hash makes it possible to finding a match just by looking at the hash value.

q2- In the second question, I again used Rabin-Karp's method of hashing and an rolling hash algorithm. Hash was preferable because common way of shifting a string requires shifting all of the chars by one position and it is costly. With rolling hash, shifting can be done in O(1) time and by prioritizing shifting over reversing we may gain much efficiency compared to string implementation. Also, since the possibility of collision with double hashing is very low, only finding same hash values is enough for a match. For string, char by char comparison is required.

My implementation logic resembles to first question. This time I have a 2D array consisting of arrays size of 5. I kept the size, isVisited Boolean, hash value 1, hash value 2 and index of pattern. I also kept the patterns itself in an array to use in reverse operations. First, I calculated hash of each pattern with Rabin-Karp in O(length) time and sort the array according to first hash values. Then for each not visited hash, I incremented subset size and I shifted the hash with O(1) time until it circles back. For shifting, I used the same logic with Rolling hash, I substracted the place value of first char, multiply rest with base and then add the first char to the end. Then I checked if there are any matches with binary search in O(logN). I marked the matched hashes as visited, then continue with reversing the pattern. I calculated the hash of reversed pattern with O(length) and shift it with O(1) and find matches using same steps. In each iteration I added minimum number of matches whichever method gives (shifting the original or shifting the reverse) to reverse count because reverse is costly and we try to minimize it. Total complexity of solution is O($total\ number\ of\ characters * log\ N$).

q4- In this question, by considering the logic of the linear probing algorithm, I tried to determine how a given hashtable may be inserted at the first place. To do this, I checked where each element should ideally be positioned and where they actually are. The values between the ideal position and the actual position of an element must have come earlier then

the element. Based on this, I recorded the dependencies of values in a 2D isBefore array as booleans. This step takes $O(N^2)$ complexity. I also have a int array keeping the number of dependencies of each element. Then for the elements having zero elements before them, I search for the lexicographically smallest one and insert it. Update its dependencies and iterate other elements until no value left uninserted. If at some point, there are uninserted valid values but neither of them has 0 dependency, program outputs “Impossible”, since no insertion order is possible for such of hash table. If all values are valid(not “-1”) this step also takes $O(N^2)$ complexity. Total complexity of program becomes $O(N^2)$.

q5- In this question I used a similar logic to question 4, but since this time an element may be inserted at at most 3rd element, it may shift 2 positions at most. Therefore, I did only check the previous 2 positions of each element to detect dependencies. This part takes O(N) time. If each value would be in the right place, every insertion would be possible and insertion number would be N!. However, dependencies between groups decreases this number. I determined 4 different dependency types as following:

A->B->C, A->C and B->C, A->B, A(no dependency)

I choose the dependency type of each element from one of these and reduce the complexity accordingly. For example, for A->B->C there is only one possibility, so I multiplied result with 1/3! Since each can be dependent to at most 2, this part again takes O(N). Since factorials grows so fast, it easily overflows, therefore I took mod in each step of factorial calculation. For dividing N! to dependency, I used inverse mod, so that the actual result is preserved. Total complexity of program is O(N).