BILKENT UNIVERSITY CS 202 HOMEWORK 3 REPORT

NAME: Simay Uygur **ID:** 22203328
SECTION: 1 **DATE:** 13.12.2024

Question 6

The restriction in the patterns' lengths enabled the implementation to have fixed size precomputed power and substring arrays according to the first minimum length pattern. However, if there was no restriction on patterns, using a precomputation array wouldn't work as the minimum and maximum pattern length is unknown and the space complexity would be enormous. The substrings' hash values of the text for each pattern is repeatedly calculated for all patterns and that also applies to the powers of the base. Even if modular arithmetic reduces the multiplying overhead with bigger and bigger powers, taking powers and calculating substrings' hashes every time increases the complexity. To decrease the capacity a bit, I could store 10 least used unique lengths that are called at the early stages of the program and while processing others, there is a possibility that same length patterns can be used recently and will be used again. That can be a small optimization of the solution. However, first finding the hash value of the first substring of the text and with rolling hash finding the others eases the burden of calculating from the start of the substring and adding each char's value to it according to the hash formula. Calculating hashing with the sliding window kind of calculating would make the computation almost O(1). Also calculating for all substrings of the text becomes O(N), which N is the text's total number of characters for one pattern. For all patterns, it becomes O(N*M). The first substring can be at most N or N-1 long, therefore first computation of rolling hash by looking every char of the substring needs that time. Also sorting and searching needs O(MNlogN + MlogN +N*M) time because sorting one substring hashes array takes NlogN time and doing this for every pattern would be MNlogN, searching will take logN time for one pattern, also grouping for binary search is needed and that takes NM time.The complexity overall will be O(N + 2*N*M + (N+1)*MlogN) $\approx$ O(N*M).

Question 7

The maximum number of insertion orders should be (validIntegerNumbers)! because there can be -1 in the hash table which means that place is empty. At most, there will be a perfect combination of numbers in the hash table such that there is no dependency between any numbers so the factorial wouldn't be divided by the dependency's permutation therefore it reaches the peak value permutations is minimum, which is 1 and the insertion order will become validIntegerNumbers!/permutations = (validIntegerNumbers)! (permutations =1). In the minimum case, every element is dependent and the dependency should be 0 1 1 or 0 1 2 (the numbers represent the difference between the indexes that elements are currently at and the indexes they should be obtained by taking the modulo of the element's value.). Selecting those combinations for the elements' dependencies would enable to divide the factorial by a power of

6 because every time these combinations are encountered the factorial should be divided by the 6 because those 3 elements that are dependent cannot change places and be inserted in another way around each other. So that will give insertion orders = (validIntegerNumbers)!/(6^(validIntegerNumbers/3)). As for each three elements are grouped for one dependency and one dependency means dividing by 6, the minimum number should be like this in the formula. If all of the integers are valid in the size N, the formula should have N instead of validIntegerNumbers.

Question 8

For the first question, I have calculated with base 31 and used double hashing to prevent collisions. Selecting base a prime was crucial because of hashing's calculation process because while hashing even numbers can lead to collusions and also selecting a great and prime modulo is important. In my implementation, I precomputed the powers of the base until maximum length -1. That is because the difference between the minimum and the maximum length was 5. That enabled me to construct an array containing precomputed base powers with modulo 10^9+7 and 10^9+9. Creating a HashTuple class, I stored two hash values and calculated each string and text's pattern with rolling hash formula and using that precomputed array for the base powers for easy computation instead of multiplying with the calculated power every time at that instant. Only multiplying the precomputed powers with the current character's value and taking the modulo of this operation takes less time than the instant computation of base powers and multiplying. So with restricted lengths, I opened a precomputed array and took the powers from that and for each different length, as each test case can have elements with differing lengths at most 6, and placed the substring's hash values according to those lengths into 6 arrays corresponding to each unique length. That gave me the chance to only search those arrays for each pattern and count the matching hashes with the pattern's hash. Computing base powers in precomputed array takes O(4*(minLengthOfOnePattern + 5)) ≈O(minLengthOfOnePattern), there are 4 different characters that need base power, powers should be computed from base^0 until base^(minLengthOfOnePattern + 4). Also initializing 6 arrays for 6 lengths starting from minLengthOfOnePattern until minLengthOfOnePattern+5 and for computing one array, first substring of the text with the given length is found. Then Rolling Hash is used and the rest of the array's values are computed in a short time. That computation took O(6*N) ≈ O(N) time. Also sorting hash values of belonging to each pattern length (substrings) takes O((minLengthOfOnePattern + 5)*log(N - minLengthOfOnePattern - 4)) ≈ O(N*logN). I grouped the same hash values in the substring hash arrays and that took O(6*(N - minLengthOfOnePattern - 4)) ≈ O(N). Also using binary search for each pattern's hash value in the substring hash arrays take O(M*logN) time because for one pattern, searching takes O(log(N - minLengthOfOnePattern - 4)) ≈ O(logN) time at most (if no substring is grouped). My implementation's overall complexity is O((N+M)*logN + N + minLengthOfOnePattern) ≈ O((N+M)*logN + total Characters).

For the second question, I used the same rolling hash logic while calculating cyclic shifts. For each unvisited pattern, I calculated at the start, the hash of all patterns, then for each unvisited pattern, the hashes of all cyclic shifts of that specific pattern, and its reverse and related cyclic shifts were calculated. First calculating hashes for all patterns and their reverses took O(2*totalPatternLength) time. But with rolling hash computations become O(1) for the cyclic shifts and for one pattern O(2*patternLength) is the complexity for one O(unvisited*2*patternLength) is the overall cyclic shift complexity. Sorting the all pattern's hash values take MlogM time to apply binary search on them. Searching all hash permutations of the specific pattern in sorted all pattern's hashes takes O(logM) time for each pattern and O(M*logM) time if no element can be turned to each other. While searching if a hash is found I made a flag in a HashTuple and make them visited and not to look at that element and its permutations anymore. Also by comparing the left and right side (reversed and non reversed and related cyclic shifts, I took the minimum of those to decide the reverse count and add that to reverse count. In the worst case, we write all permutation hashes and search for them O(totalCharacters*logN).

For the fourth question, I utilized a 2D matrix binding the elements if they need to come before or after one another. The dependency matrix was populated based on the given relative dependencies. A loop was needed to traverse the dependency matrix and set the counts of all dependencies for each element. There is another loop that traverses the count array and identifies the smallest element (according to their integer values) with zero dependencies and adds it to the result array. If no such element exists, the algorithm concludes that the solution doesn't exist. Hashing was applied to determine the position of elements in the hash table, which is they should be placed at, where the hash function calculates the modulus of the number with the size of the hash table. To handle collisions effectively, linear probing was used. The worst-case time complexity for this approach is O(N^2) because linear probing needs searching for the fitting or near that position and placing it there. Populating the dependency matrix takes also O(N^2) because for each element we check the dependency of the neighbors. Additionally, in cases where all elements are valid, the loop to generate the solution also has a complexity of O(N^2). Overall complexity becomes O(N^2) for this.

For the fifth question, the solution begins by calculating the factorial of the number of valid elements, a step that has a worst-case complexity of O(N) without being affected by the dependency number of elements. Next, an array is created to record how far each element's index deviates from its modular position. If the deviation is negative, the hash table size is added to it to make it positive and making count cases common with 0, 1, or 2. The hash function calculates the modular position of a number based on the hash table size, and linear probing is applied to resolve any potential collisions. Since direct division is not feasible during modular arithmetic, precomputed modular inverses of 2, 3, and 6 are used with mod 10^9+9 . The relevant values are multiplied by these modular inverses, instead of dividing directly 2, 3 and 6. So calculating the factorial dominates the time complexity, but dividing has also at most O(N).