Atakan Akar
22203140
HW 3

Question 6)
In this problem, the goal is to efficiently match multiple patterns within a large text. Initially, the problem simplifies preprocessing and ensures optimal time complexity by restricting the difference in pattern lengths to at most five characters. This limitation reduces the number of unique substring lengths for which rolling hashes need to be computed, making the approach scalable and efficient. However, when the restriction on pattern lengths is removed, the complexity of the problem increases significantly. Pattern lengths can now vary widely, potentially spanning the entire length of the text, resulting in up to O(n) unique pattern lengths, where n is the size of the text. For each unique pattern length L, the algorithm must compute rolling hashes for all substrings of length L in the text, which takes (O(n)) time. This leads to a total preprocessing time of (O(Kn)), where K is the number of unique pattern lengths. After computing the hashes, they must be sorted to allow efficient pattern matching using binary search. For each input pattern, the hash value of the pattern is searched in the sorted list, and all occurrences are counted by traversing left and right. When this process is repeated for M patterns in a text of length N, the overall time complexity becomes (O(MN)), considering the combined costs of hashing, sorting, and searching for each pattern. This analysis demonstrates that removing restrictions on pattern lengths introduces significant scalability challenges, as the algorithm must handle potentially (O(n) unique pattern lengths, each requiring separate hash computations, sorting operations, and searches. Consequently, the time complexity grows rapidly, making the approach less feasible for large inputs without further optimization.
Question 7)
Insertion order rises to maximum number which is n! when all input numbers are valid and their hash positions align perfectly with their modulo values this give rises to every permutation to form a valid order. The minimum insertion order count is 1 no valid numbers exist, there is only one valid number and it directly goes to its hash position, or multiple valid numbers exist but are all dependent on each other due to collisions are resolved by linear probing method.

Question 8)

Question 1)
For this problem, considering the constrained inputs, I calculate hash values for substrings with lengths varying from the length of the first input to five characters beyond it. Then, I determine how frequently each hash value occurs within the inputs. My code uses polynomial hashing to efficiently count the occurrences of patterns in a given text, thereby making it comparable to other strings much faster by transforming substrings into fixed-size numerical hash values. Hashing reduces the time complexity of string matching because it does not require direct character-by-character comparisons. The polynomial hash function with (P = 31) and (MOD = 10^9 + 7) ensures low chances of collisions; for example, by using modulo arithmetic, it prevents overflow. Rolling hash is a technique where hash values are precomputed for all substrings of certain lengths, so that it can be updated efficiently as the

substring slides over the text. Hash values are then sorted and a binary search is performed in the sorted array to find a match to further improve the running time. Now for each pattern, compare its hash value with pre-computed hashes; by extending it is able to find all matching substrings to get the total occurrences. This, therefore, allows the solution to run in O((N + M) * log N + total characters) time where N is the length of the text and M is the length of the pattern, thus it is highly scalable for large inputs while keeping computational overhead to a minimum.

Question 2)

My solution as like this. The program efficiently processes n strings using Booth's algorithm to find their canonical forms and by hashing for fast storage and comparison. First, the program reads all strings, calculating the sum of their total character count, T, in time O(T). For each string of length m, it applies Booth's algorithm to find lexicographically smallest rotation. It runs in O(m) for one string of length m and thus has a time complexity of O(T) for all strings. To handle reverse equivalence, each string is reversed in O(m) and its smallest rotation is computed again in O(m), giving a combined complexity of O(2T) for both forward and reversed canonical forms. Comparing these two forms to select the lexicographically smaller form is done using a custom string comparison function in O(m). The canonical form selected is stored in the hash table, which does support efficient insertion and lookup in average time O(1). In this case, hashing becomes important because, by avoiding the quadratic complexity due to direct pairwise comparisons, it allows for fast checks of duplicates and grouping of strings that are equivalent, regardless of their orientation. After processing, the hash table is traversed in O(k), where k is the number of unique canonical forms, to compute the minimal subset size and the number of reverse operations. Generally, the time complexity of the program is O(T + n + k) for input reading, string processing, and result computation. Using Booth's algorithm for fast rotation computation and hashing for fast handling of data, the program scales well for large datasets to ensure the best performance even for the upper bounds of input constraints. The code uses hashing to efficiently store and compare strings, particularly their canonical forms. Hashing comes into play here in the form of storing and retrieving data in constant average time, without the computational overhead of explicit pair comparisons between strings.

Question 3) Not implemented.

Question 4)
This is my solution. To find the hash table's lexicographically minimal insertion order, I used a matrix and a counter array. The initialization of the of the matrix takes O(n^2). The counter array stores the number of dependencies for each index, and it is updated dynamically as dependencies are resolved. To find the next smallest element with no dependencies, I iterated through the counter array, which takes O(N)for each iteration. When an element is selected, its dependencies are removed, requiring updates to the dependency matrix and counter array, which also take O(N) per iteration. This process is repeated N times until all elements are processed, leading to an overall complexity of O(N^2). Each element's starting position in the hash table is determined by the hash function h(x)=x mod N, which is used for mapping values. I used linear probing, which cyclically finds the next available slot to resolve conflicts, to deal

with collisions. The dependency matrix is used to track the dependencies that are introduced between slots. These dependencies are resolved by the counter array while preserving the insertion sequence's lexicographical order. I there is no element whose dependency value is 0 but there are some other elements in it which are not -1 than it becomes impossible. Since managing and resolving collisions is important for reconstructing the lexicographically minimum insertion order, hashing is important for this problem. By getting benefit hashing and combining it with the dependency tracking, the solution maintains correctness and ensures time complexity of O(N^2). And in the worst case scenario the solution has a time complexity of O(n^2). This arises because generating the dependency array involves O(n^2) operations. Additionally, if every element meets the required conditions, the loop that calculates the solution also executes with O(n^2) complexity. Consequently, the overall process is dominated by O(n^2) time.

Question 5)

This is my solution. To compute the number of possible insertion orders for the given hash table in problem 5, I used modular operations and collision handling strategies. The solution begins with creating a difference array that determines the positional offsets required to resolve collisions caused by linear probing. For each element in the hash table, the difference between its current index and its hash value (table[i]modN) is calculated and stored. By getting benefit from this difference array which was created before the relationships between the indices to identify patterns that indicate dependencies or constraints on the insertion order are analyzed. Based on these patterns, the total number of valid permutations is calculated by dividing the current possibility by modular inverses (which are calculated before)(mod_inv2…), which handle overlapping or dependent positions. The loop through the difference array also runs in O(N), which give rise to complexity of O(N). The hash function is used to determine the elements to their initial positions in the hash table if there was no collisions. Collisions are solved using linear probing, which ensures all elements are placed in the table. The restriction that is given in the question is also utilized which a element is at most iterated at most 2 from its original position. The algorithm determines the number of valid insertion orders while taking these collisions into consideration by utilizing the characteristics of modular arithmetic.