Report on Minimum Deletions and Time Complexity Analysis

Zhiwei Jiang

October 22, 2024

1 Problem Analysis

In this assignment, I tackled two distinct problems. The first problem involved determining the minimum number of deletions required from a given list such that the remaining elements matched a certain pattern. The second problem was focused on computing the time complexity for various nested loops in a custom programming language. My approach for each problem involved careful analysis of possible solutions, choosing the most efficient method, and implementing algorithms that can handle the given constraints.

This report will walk through my thinking process, detailing possible approaches, the final solutions, and why these solutions were effective.

2 Problem 1: Minimum Deletions in a List

2.1 Problem Overview

The first problem involved a list of n elements, where each element in the list belongs to the range $[1, n]$. We were required to ensure that all elements from 1 to n appeared at least once in the modified list by removing the minimum number of elements. This required calculating how to best reduce the list so that it contains all necessary numbers, while minimizing the deletions.

2.2 Possible Solutions

1. **Naive Approach**: Iterate over each number from 1 to n and check if it exists in the list. If it does not, then continue iterating through the list until that number is found or until all elements are considered. This approach requires multiple scans through the list, resulting in a complexity of $O(n^2)$ in the worst case.

2. **Optimal Approach (Queue and Hashing)**: The more optimal approach would use auxiliary data structures such as a frequency count list and a queue. The frequency count tracks occurrences of each element in

the list, and a queue is used to maintain the numbers that are missing from the list. This allows us to manage deletions in a systematic way, ensuring minimal deletions. This solution runs in $O(n)$ time complexity, making it suitable for larger input sizes.

2.3 Final Solution

To solve this problem efficiently, I used an approach involving both frequency counting and a queue:

- **Step 1: Initialize Count and Index Map**: I initialized a count array to track the frequency of each element in the list. I also used an index map to record the indices of occurrences of each number.
- **Step 2: Initialize Delete Queue**: I created a delete queue to track which numbers were missing from the list. The queue contained numbers from 1 to n that had zero occurrences in the list.
- **Step 3: Process Deletions**: Using the delete queue, I iterated through the elements that needed to be deleted, updating the count and index map accordingly.

This approach ensures that each number is visited only once, and all operations related to counting, mapping, and queuing are performed in constant time. The overall complexity is $O(n)$.

2.4 Why This Solution is Better

The solution based on a queue and hashing is more efficient in both time and space. By keeping track of missing elements through a queue and efficiently updating their occurrences, we avoid the repetitive checks inherent in the naive approach. This allows us to solve the problem in linear time, which is crucial given the constraint $n \leq 2 \times 10^5$.

3 Problem 2: Calculating Time Complexity of Programs

3.1 Problem Overview

The second problem required determining the time complexity of several programs written in a simplified programming language. Each program consisted of nested loops, and we were tasked with deriving their complexity in the form $O(n^w)$.

3.2 Possible Solutions

1. **Manual Parsing and Calculation**: Manually parse the loop structure, track nesting levels, and calculate the contribution of each loop to the overall complexity. This approach would be tedious and error-prone, particularly for deeply nested loops.

2. **Stack-Based Automated Calculation**: Use a stack to simulate the behavior of loop nesting and keep track of the complexity contributed by each loop. This approach ensures that we can dynamically manage nesting, particularly when loops are exited or entered.

3.3 Final Solution

The stack-based automated approach was chosen:

- **Step 1: Parse Loop Lines**: Each line was parsed to identify the type of loop (F i x y) or the end of a loop (E). For each F line, the loop bounds (x, y) were also extracted.

- **Step 2: Manage Stack for Complexity**: A stack was used to keep track of the complexity contribution of each nested loop. If the upper bound (y) was n, the complexity was incremented by 1 for each nested loop level. The stack was updated dynamically to reflect entering and exiting loops.

- **Step 3: Calculate Maximum Complexity**: The maximum complexity encountered during the execution was stored to determine the overall complexity of the program.

3.4 Why This Solution is Better

The stack-based approach is systematic and well-suited to handling nested structures, particularly when the nesting depth is unknown beforehand. By using a stack, we can easily track the contribution of each nested level and ensure that complexities are properly accounted for. The complexity of parsing and managing the stack is $O(L)$, where L is the number of lines in the program, making it efficient for the input constraints ($L \leq 20000$).

4 Conclusion

The two problems required thoughtful consideration of efficiency and scalability. For the first problem, the queue and frequency-based solution provided a robust and optimal approach for minimizing deletions, while for the second problem, the stack-based complexity calculation allowed us to systematically determine the time complexity of nested loops.

These solutions were chosen based on their efficiency and ability to handle the worst-case constraints effectively. By reducing repetitive computations and ensuring systematic tracking of necessary operations, both problems were solved within the optimal time bounds.