CS3334 Project Report

1. Introduction

The compiling command for compiling programs is std=c++11.

This report 3 different approaches are implemented to solve the problem and time complexity is analyzed

2. Approaches

2.1. Basic input and output

For simplicity, I use the same main and parseLine functions to process the input data and output the result for all methods, and the function() is the core that applies different methods.

The main function reads input lines until an empty line is encountered, parsing each line and calling the function to process the numbers and threshold. It then prints the results according to the specified format. The time complexity of the main function depends on the total number of lines and the length of each line, but it is dominated by the complexity of the function since this is called for each line.

The parseLine function is responsible for parsing the input string and converting it into a vector of integers. It handles negative numbers by checking for a '-' sign and appropriately adjusting the sign of the resulting integer. The time complexity of this function is O(n), where n is the length of the input string, since each character in the string is processed once.

2.2. Brute Force Approach

The function implements a brute-force method. A result vector is initialized to store the numbers that exceed the threshold T, and a boolean vector a of the same size as nums is initialized to false. This vector is used to mark elements that have already been considered to avoid counting them multiple times. Then, it iterates over each element to check if it is marked, and for each unmarked element, the function counts how many times it appears. After counting the occurrences of the current number, the function checks if the count exceeds the threshold T and returns the result vector.

This function has one outer loop and inner loop for each of the n elements; considering together with the time complexity of main and parseLine function in the worst case, the time complexity of the function is O(n^2), where n is the number of elements in the nums vector.

2.3. Linked list

This method employs a linked list data structure to track the frequency of each number. The linked list is composed of nodes, where each node contains an integer value (data), the frequency of that integer (frequency), and a pointer to the next node (next). The function

begins by initializing a head pointer to nullptr, indicating an empty list. It then iterates through each integer in the input vector nums. For each integer, it searches for an existing node with the same value using the findNode function. If the node is found, its frequency is incremented; otherwise, a new node is created and appended to the list using the append function. After constructing the linked list, the function iterates through the list again to collect numbers whose frequency exceeds the threshold T. These numbers are added to the result vector, which is then returned.

The time complexity of this method is O(n^2) in the worst case, where n is the number of elements in the input vector nums. This is because the findNode function, which is called for each element in the input vector, has a time complexity of O(m), where m is the number of elements already in the linked list at the time of the search. Since the findNode function is called for each of the n elements, and m can grow up to n in the worst case (when all elements are unique), the overall time complexity becomes O(n^2). The second pass through the linked list to collect results has a time complexity of O(n), but it does not dominate the overall complexity.

2.4. Binary Search Tree

The function uses a binary search tree (BST) to track the frequency of each number in the input vector nums. The BST is implemented using a TreeNode structure, where each node contains an integer value (data), the frequency of that integer (frequency), and pointers to the left and right child nodes (left and right). The insert function is a recursive helper function that inserts a new number into the BST. If the tree is empty, it creates a new node; otherwise, it finds the correct position for the new node by comparing the number with the current node's value and recursively inserting it into the left or right subtree. If the number already exists in the tree, it increments the frequency of that node.

After building the BST, the function iterates through the original input vector nums to collect numbers that exceed the given threshold T. For each number, it performs a search in the BST to find the node corresponding to that number and retrieve its frequency. If the frequency is greater than T and the number has not been added to the result vector yet (tracked by the added vector), it adds the number to the result. To maintain the order of first appearance, it also marks all occurrences of the number in the added vector.

The time complexity of this method is O(n^2) in the worst case, where n is the number of elements in the input vector nums. This is because for each of the n elements, the function performs a search in the BST, which can take O(h) time in the worst case, where h is the height of the tree. In a skewed tree, the height can be O(n), leading to a total time complexity of O(n * n) = O(n^2). However, if the tree remains balanced and the height is kept to O(log n), reducing the search and insertion operations to O(log n) time. This would bring the overall

time complexity down to O(n log n), making it more efficient for larger datasets.

3. Test

From the time complexity explanation in the previous paragraph, we can summarize that the brute-force method, which involves comparing each element with every other element, exhibits a worst-case time complexity of $O(n^2)$ when all elements are unique, necessitating a full traversal for each element. The linked-list approach also reaches a worst-case time complexity of $O(n^2)$ under the same condition, as each insertion could potentially require traversing the entire list. Although the average time complexity for linked lists is $O(n^2)$, it is important to note that this does not reflect the best-case scenario. The tree-based method, utilizing a Binary Search Tree (BST), has an average time complexity of O(n log n), which is efficient for most cases. However, in the worst-case scenario of $O(n^2)$, this occurs when the input is already sorted, causing the BST to degenerate into a linked list, thereby eliminating the benefits of binary search and leading to inefficient traversals.

Based on this, we use several test cases that include the worst situation and other situations to further evaluate the performance of these methods.(detailed number is in test file)

Test Case 1: 1 -1 1 -1 8

Test Case 2: 2 13 4567 2 2 2 3 10 13 13 13 -4 4 4 5

Test Case 3: 1000 distinct random numbers

Test Case 4: 1000 same integers

Test Case 5: 1000 random integers with both negative and positive

Test Case 6: List of sequential integers from 1 to 1000

The three-method execution time(microseconds) has been summarized in a table

Test Case	Brute Force Approach	Linked List Approach	Tree Approach
1	3	7	9
2	4	17	10
3	3327	827	4568
4	17	419	391
5	2294	252	3502
6	4187	9083	9083

For Test Case 1, both the Brute Force Approach and the Linked List Approach perform well,

with relatively shorter execution times. The Tree Approach is slightly slower due to the need to construct the tree structure.

In Test Case 2, the Brute Force Approach and the Tree Approach perform relatively better, with the Linked List Approach being slightly slower, potentially because linked list operations are more frequent.

In Test Case 3, the Linked List Approach performs best, with the Brute Force Approach coming in second, while the Tree Approach takes longer due to the need for frequent insertions and lookups.

For Test Case 4, the Brute Force Approach performs the best due to the high number of repeated numbers, which allows for quick counting. Both the Linked List Approach and the Tree Approach require more operations to handle repeated insertions.

Test Case 5 sees the Tree Approach as the slowest due to the need to construct and traverse the tree. The Linked List Approach performs relatively better, with the Brute Force Approach being slightly slower.

In Test Case 6, all methods have longer execution times because the input data is fully ordered, causing the Tree Approach to degrade into a linked list, which leads to a decrease in performance.

4. Conclusion

In summary, the Brute Force Approach performs well when working with small or repetitive data-heavy lists but performs poorly on large datasets. The linked list method performs well when working with random data but may degrade in performance on ordered data. The tree method performs stably in most cases but may degrade to a chained list on fully ordered data, resulting in performance degradation. Choosing the right algorithm based on the characteristics of the input data can significantly improve performance. For large datasets, the tree method is usually the better choice, but it needs to be used with caution on fully ordered data.