I222698

SE - F

Design and Analysis of Algorithm

Assignment - 2

Report

Problem 3: PakFlora Garden Selection

Part (a): Top k Highest-Scoring Gardens

Algorithm Explanation

The problem requires finding the k highest-scoring gardens from an unsorted list in O(|A|+klog|A|) time. I implemented a solution using a min-heap of size k.

The algorithm follows these key steps:

1. Create a min-heap to store (score, registration_number) pairs
2. Insert the first k gardens into the min-heap
3. For each remaining garden, if its score is higher than the minimum score in the heap:
    - Remove the garden with the minimum score
    - Insert the current garden
4. Extract all k gardens from the heap to get the result

This approach ensures we maintain the k highest-scoring gardens at all times, with the minimum of these k gardens at the top of the heap for easy comparison and replacement.

Time Complexity Analysis

- Building initial min-heap with k elements: O(k)
- Processing the remaining (|A|-k) gardens:
    - Each comparison and potential insertion/deletion: O(log k)
    - Total for this step: O((|A|-k) × log k)
- Extracting all k elements from the heap: O(k log k)

Total time complexity: O(k + (|A|-k) × log k + k log k) = O(|A| + k log k)

Since k ≤ |A|, this satisfies the required O(|A| + k log |A|) time complexity.

Part (b): Gardens with Scores Exceeding Threshold

Algorithm Explanation

For this part, we need to extract all gardens with scores greater than a threshold x from a max-heap in O(nx) time, where nx is the number of gardens that meet the criteria.

The optimized solution uses a recursive traversal of the max-heap:

1. Start at the root of the max-heap
2. If the current node's score exceeds threshold x:
    - Add its registration number to the result
    - Recursively check its left and right children
3. If the current node's score is not greater than x, none of its descendants will have scores > x (by max-heap property), so skip that subtree

This approach leverages the max-heap property to prune branches that cannot contain qualifying gardens.

Time Complexity Analysis

- We only visit nodes with scores > x and their immediate children
- In the worst case, we examine nx nodes that qualify (with scores > x) and at most nx additional nodes that don't qualify
- This gives a total of O(nx) nodes visited
- Each node operation is O(1)

Total time complexity: O(nx)

This meets the requirement of O(nx) time complexity.

Problem 4: Counting Ordering Discrepancies

Algorithm Explanation

The problem requires counting the number of times a higher quality score appears before a lower one in a sequence. This is equivalent to counting inversions in an array, which can be efficiently solved using a modified merge sort algorithm.

The key insight is that we can count inversions while merging two sorted halves of the array:

1. Split the array into two halves recursively
2. Count inversions in the left half
3. Count inversions in the right half
4. Count split inversions (across the two halves) during the merge step

During the merge step, when an element from the right half is placed before an element from the left half, it forms inversions with all remaining elements in the left half.

Pseudocode for the Modified Merge Sort

1. If array size is 1, return 0 (no inversions)
2. Recursively count inversions in the left half
3. Recursively count inversions in the right half
4. Merge the two halves and count split inversions
5. Return the sum of all inversions

Time Complexity Analysis

- The recurrence relation for merge sort is $T(n) = 2T(n/2) + O(n)$
- This resolves to $O(n \log n)$ using the master theorem
- Space complexity is $O(n)$ for the temporary array used during merging

This solution is optimal as it's been proven that counting inversions requires $\Omega(n \log n)$ time in the comparison model.

Problem 5: Repair Shop Optimization

Algorithm Explanation

The problem involves minimizing the maximum time it takes to repair all phones across K repair shops. Since phones must be repaired in the order they appear in the pile, this is a problem of partitioning an array into K subarrays to minimize the maximum subarray sum.

I used a binary search approach on the answer:

1. Define the search space:
    - Lower bound: Maximum repair time of any single phone
    - Upper bound: Sum of all repair times
2. For each mid-point in the binary search:
    - Check if it's possible to repair all phones with at most K shops, with no shop exceeding the mid-point time
    - Use a greedy approach: assign phones to the current shop until adding another phone would exceed the mid-point time, then use a new shop
3. If the mid-point works, try a smaller time limit; otherwise, try a larger time limit
4. The final answer is the smallest time limit that works

This approach efficiently narrows down the optimal solution through binary search.

Time Complexity Analysis

- Binary search range: From max_element to sum_of_all_elements
    - This range could be up to 10^12 in the worst case (10^6 phones with maximum time 10^6 each)
    - Number of iterations: O(log(sum of repair times))
- Each iteration requires O(n) time to check if a solution is possible
- Overall time complexity: O(n × log(sum of repair times))
- Space complexity: O(n) for storing the repair times

The solution efficiently handles the constraints specified in the problem (n up to 10^6, k up to 1001, and repair times up to 10^6).