Problem C-4.16

To determine if there are duplicates in the sequence (S), we can use a dictionary (hash map) to keep track of the elements we encounter.

1. Initialize an empty dictionary called `seen`.

2. For each element `x` in (S):

  - If `x` is in `seen`, return `True` (indicating there's a duplicate).

  - Otherwise, add `x` to `seen`.

3. If the loop completes without finding any duplicates, return `False`.

Time Complexity:

- Each lookup and insertion operation in a hash map takes (O(1) \) on average.

- Since we're iterating over (n) elements, the algorithm has an overall time complexity of (O(n)).

Problem C-4.19

We can use a modified merge sort algorithm to count the number of inversions.

Algorithm:

1. Use the merge sort algorithm to sort \(S \).

2. During each merge step:

  - Count how many elements from the right half are less than an element in the left half.

  - For each such pair, an inversion is found.

  - Add this count to a running total of inversions.

Time Complexity:

- The modified merge sort runs in O(n log n)) time since it is essentially merge sort with an additional counting step during merging.

Problem A

1. Perform a depth-first search (DFS) traversal of the tree (T).

2. Keep track of the maximum depth encountered and maintain a list of nodes at this depth.

3. For each internal node (v) at the maximum depth, add ((v, d)) to the result list.

Time Complexity:

- Since DFS visits each node once, the time complexity is (O(n)), where (n) is the number of nodes in (T).

Problem B (a) - `isExclusiveOr(A, B, C)`

1. Initialize two dictionaries (or sets) `exclusiveA` and `exclusiveB` to store elements that are unique to (A) and (B), respectively.

2. Recursively populate these dictionaries.

3. Check if (C) contains exactly the elements in `exclusiveA` and `exclusiveB` (with duplicates allowed).

Time Complexity:

- Since each element in \(A \), \(B \), and \(C \) is examined only once, the time complexity is \(O(n) \), where \(n \) is the maximum length among \(A \), \(B \), and \(C \).

Problem C

1. Distribute the documents alphabetically across the tables (e.g., assign letters A-B to the first table, C-D to the second, etc.).

2. Within each table, sort the documents by name.

3. Once each table is sorted, combine the documents from each table in order.

Time Complexity:

- The sorting within each table is \(O((n/12) \log (n/12)) \), and combining the tables is \(O(n) \), giving an overall complexity of \(O(n \log n) \).

Problem D - `createBST(S)`

1. Find the middle element of \(S \) and make it the root.

2. Recursively:

  - Assign the middle of the left half as the left child.

  - Assign the middle of the right half as the right child.

3. Continue until all elements are inserted.

Time Complexity:

- Since each insertion involves dividing \(S \) in half, the time complexity is \(O(n) \).

Problem C-4.25

1. Choose a pivot nut and partition the bolts into those smaller, larger, and matching the pivot.

2. Recursively partition the nuts based on the matched bolt.

3. Repeat the process on smaller subarrays until all matches are found.

Time Complexity:

- This is similar to quicksort, so the running time is \(O(n \log n) \) in terms of comparisons.