CS-700 Algorithms and Complexity

Assignment-2

ANAND M K
Roll Number: 242CS008
Department of Computer Science and Engg.- NITK Surathkal

Problem 1: Longest Common Prefix

Problem Statement: Given a set of strings, find the longest common prefix among all the strings.

Example:

- **Input:** {"apple", "ape", "april"}
- **Output:** "ap"

Pseudocode:

```
Function findLongestCommonPrefix(strings[], start, end):
    If more than one string is present:
        Calculate the midpoint: mid = (start + end) / 2

// Recursively find the longest common prefix in the left half

        prefixLeft = findLongestCommonPrefix(strings, start, mid)

// Recursively find the longest common prefix in the right half

        prefixRight = findLongestCommonPrefix(strings, mid + 1, end)

// Compare prefixLeft and prefixRight and find their common part

        i = 0
        limit = minimum(length of prefixLeft, length of prefixRight)
        While i < limit and prefixLeft[i] == prefixRight[i]:
            i++

// Return the common prefix found between the two halves

        result = prefixLeft[0...i-1]
        Return result
    Else:

// Base case:  If only one string is left, return it

        Return strings[start]
```

Time Complexity Analysis:

The recursive function divides the problem into two halves and compares the prefixes. Let n be the number of strings and m be the length of the common prefix between the two halves.

The recurrence relation can be written as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(m)$$

This is because the problem is divided into two subproblems, and merging the results (comparing the prefixes) takes $O(m)$.

Recursion Tree Analysis:

1. **Level 1 (Root):**
   - Work done: $O(m)$ for the root node.
2. **Level 2:**
   - 2 nodes each doing $O\left(\frac{m}{2}\right)$.
   - Total work: $2 \times O\left(\frac{m}{2}\right) = O(m)$.
3. **Level 3:**
   - 4 nodes each doing $O\left(\frac{m}{4}\right)$.
   - Total work: $4 \times O\left(\frac{m}{4}\right) = O(m)$.
4. **Level k:**
   - 2^{k-1} nodes each doing $O\left(\frac{m}{2^{k-1}}\right)$.
   - Total work: $2^{k-1} \times O\left(\frac{m}{2^{k-1}}\right) = O(m)$.

At each level, the total work remains $O(m)$. The depth of the recursion tree is $\log n$ since the problem size halves at each level. Therefore, the total time complexity can be calculated as:

$$T(n) = O(m) + O(m) + O(m) + \ldots + O(m) \quad (\log n \text{ levels})$$

$$\boxed{\mathbf{T(n) = O(m\ log\ \ n)}}$$

Thus, the overall time complexity of the algorithm is $O(m \log n)$, where m is the length of the common prefix between the two halves and n is the number of strings.

Problem 2: Longest Monotonically Increasing Subsequence

Problem Statement: Given a sequence of n numbers, find an $O(n^2)$ algorithm to determine the length of the longest monotonically increasing subsequence (LIS).

Pseudocode:

```
Function longestIncreasingSubsequence(arr[], size, resultSize):
    // Initialize dp array to store lengths of LIS ending at each index
    Create array dp of size ‘size‘
    Create array predecessor of size ‘size‘

    For i from 0 to size - 1 do:
        dp[i] = 1           // Initialize LIS length for each element as 1
        predecessor[i] = -1 // Initialize no predecessor for each element

    // Compute LIS for each element by comparing it with previous elements
    For i from 1 to size - 1 do:
        For j from 0 to i - 1 do:
            If arr[i] > arr[j] AND dp[i] < dp[j] + 1 then:
                dp[i] = dp[j] + 1          // Update LIS length
                predecessor[i] = j         // Update predecessor of element at i

    // Find the maximum value in dp[] which represents the length of the LIS
    maxLength = -1
    lastPos = -1
    For i from 0 to size - 1 do:
        If maxLength < dp[i] then:
            maxLength = dp[i]
            lastPos = i // Track the index of the last element in the LIS

    Print "Length of LIS: " + maxLength

    // Reconstruct the LIS by following the predecessor array
    Create array subsequence of size maxLength
    index = maxLength - 1
    While lastPos != -1 do:
        subsequence[index] = arr[lastPos]  // Store the current element
        lastPos = predecessor[lastPos]       // Move to the predecessor
        index = index - 1                     // Decrement index

    resultSize = maxLength // Update the size of the resulting LIS

    Return subsequence
```

Time Complexity Analysis:

The algorithm for finding the Longest Increasing Subsequence (LIS) has a time complexity analysis that can be broken down as follows:

1. Initialization: - The initialization of the dp and predecessor arrays takes $O(n)$ time.

2. Nested Loops: - The outer loop runs from 1 to $n-1$. - The inner loop runs from 0 to $i-1$, which leads to the total iterations:

$$1+2+3+\ldots+(n-1)=\frac{n\cdot(n-1)}{2}=O(n^2).$$

Combining these two components.

$$\boxed{\mathbf{T(n)=O(n^2)}}$$

Thus, the overall time complexity of the algorithm for finding the Longest Increasing Subsequence is $O(n^2)$, where n is the number of elements in the input array.

Problem 3: Counting Inversions in an Array

Problem Statement: Given an array of integers, count the number of inversions. An inversion is defined as a pair of indices (i, j) such that $i < j$ and $\text{arr}[i] > \text{arr}[j]$.

Pseudocode:

```
Function mergeAndCountInversions(array[], left, mid, right):
    size = right - left + 1          // Size of the merged array
    Create temp[size]                 // Temporary array for merging
    leftPtr, rightPtr, tempIndex, inversions = left, mid + 1, 0, 0

    // Merge the two subarrays while counting inversions
    While leftPtr <= mid AND rightPtr <= right do:
        If array[leftPtr] <= array[rightPtr]:
            temp[tempIndex++] = array[leftPtr++]  // No inversion
        Else:
            temp[tempIndex++] = array[rightPtr++]  // Found an inversion
            inversions += (mid - leftPtr + 1)       // Count inversions

    // Copy remaining elements
    While leftPtr <= mid: temp[tempIndex++] = array[leftPtr++]
    While rightPtr <= right: temp[tempIndex++] = array[rightPtr++]

    // Copy sorted elements back to the original array
    For i from 0 to size - 1: array[left + i] = temp[i]

    Return inversions                    // Return the count of inversions

Function countInversions(array[], left, right):
    if left >= right: return 0          // Base case: No elements to compare
    mid = (left + right) / 2            // Find the midpoint
    // Recursively count inversions in left and right halves, and merge
    return countInversions(array, left, mid) +
           countInversions(array, mid + 1, right) +
           mergeAndCountInversions(array, left, mid, right)
```

Time Complexity Analysis:

The algorithm utilizes a divide-and-conquer approach to count inversions in an array. The recurrence relation for the time complexity can be expressed as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here: - $T(n)$ represents the total time taken for n elements. - The term $2T\left(\frac{n}{2}\right)$ accounts for the recursive calls on the two halves of the array. - The term $O(n)$ is for merging the two halves and counting the inversions.

Applying the Master Theorem: - We have $a = 2$, $b = 2$, and $f(n) = O(n)$. - We need to compare $f(n)$ with $n^{\log_b a}$:

$$n^{\log_2 2} = n^1 = O(n)$$

Since $f(n) = O(n)$ is polynomially equal to $n^{\log_b a}$, we can apply case 2 of the Master Theorem. Thus, the overall time complexity is:

$$\boxed{\mathbf{T(n) = O(n \log n)}}$$

This indicates that the algorithm runs in $O(n \log n)$ time, making it efficient for counting inversions in an array.

Problem 4: Finding the k-th Smallest Element in an Unsorted Array

Problem Statement: Implement an algorithm to find the k-th smallest element in an unsorted array with a time complexity of $O(n)$. Your goal is to identify the element that would occupy the k-th position if the array were sorted, without fully sorting the array. Use the divide and conquer approach to solve the problem.

```
Function quickSelect(array[], k, low, high):
    pivot = medianOfMedians(array, low, high)  // Select optimal pivot
    mid = partition(array, pivot, low, high)   // Partition the array
    leftSize = mid - low                       // Calculate size of left part

    // Recursively find the k-th smallest element
    If k < leftSize + 1 then
        return quickSelect(array, k, low, mid - 1)  // Search in the left part
    Else if k > leftSize + 1 then
        return quickSelect(array, k - (leftSize + 1), mid + 1, high)
    Else
        return pivot  // Pivot is the k-th smallest element

Function partition(array[], pivot, low, high):
    i, j, k = low, low, high   // Initialize pointers for partitioning
    While j <= k do:
        If array[j] < pivot then
            swap(array[i], array[j])  // Move smaller elements to the left
            i++, j++
        Else if array[j] > pivot then
            swap(array[k], array[j])  // Move larger elements to the right
            k--
        Else:
            j++  // Skip elements equal to the pivot
    return i  // Return the partition index

Function medianOfMedians(array[], low, high):
    n = high - low + 1                        // Number of elements
    groupCount = ceil(n / 5)                  // Number of groups of 5 elements
    medianArray = new array[groupCount]       // Store medians of groups

    // Sort each group of 5 elements and store its median
    For each group of 5 in array do:
        quickSort(group)
        medianArray[i] = median of group

    // Handle remaining elements if any
    If remaining elements exist then
        quickSort(remaining)
        medianArray[last] = median of remaining

    If groupCount <= 5 then
     // Sort the median array if it has 5 or fewer elements
        quickSort(medianArray)
          // Return the median of the sorted array
        return median of medians
    Else:
        return quickSelect(medianArray, groupCount / 2, 0, groupCount - 1)
        // Recursively find the median of medians array
```

Time Complexity Analysis:

We are given the recurrence relation:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) \tag{9.1}$$

Where:

- $T(n)$ represents the total time to solve the problem of size n.
- The problem is divided into two subproblems: one of size $\frac{n}{5}$ and one of size $\frac{7n}{10}$.
- The partitioning and median selection takes linear time, $O(n)$.

We will prove that $T(n) = O(n)$ using the substitution method. More specifically, we will assume that $T(n) \leq cn$ for some constant $c > 0$, and show that this holds true.

Assume that the solution is $T(n) \leq cn$ for some constant c. Now, substitute this assumption into the recurrence relation:

$$T(n) \leq c\left(\frac{n}{5}\right) + c\left(\frac{7n}{10}\right) + O(n)$$

Simplify the terms on the right-hand side:

$$T(n) \leq \frac{cn}{5} + \frac{7cn}{10} + O(n)$$

Find a common denominator for the first two terms:

$$T(n) \leq \frac{2cn}{10} + \frac{7cn}{10} + O(n)$$

This simplifies to:

$$T(n) \leq \frac{9cn}{10} + O(n)$$

The $O(n)$ term can be written as dn, where d is a constant. So we now have:

$$T(n) \leq \frac{9cn}{10} + dn$$

Factor out n:

$$T(n) \leq n\left(\frac{9c}{10} + d\right)$$

To ensure that $T(n) \leq cn$, we need the constant c to satisfy the inequality:

$$c \geq \frac{9c}{10} + d$$

Subtract $\frac{9c}{10}$ from both sides:

$$\frac{c}{10} \geq d$$

Thus, we can choose $c \geq 10d$ to ensure that the inequality holds. Therefore, we conclude that $T(n) \leq cn$ for some constant c, which proves that

$$\boxed{\mathbf{T(n) = O(n)}}$$

For small values of n (specifically, $n \leq 4$), the algorithm computes the result directly without recursion. In this case, the time complexity is constant, which does not affect the overall linear time complexity.

By substitution, we have shown that the time complexity of the median of medians algorithm is $O(n)$ in the worst case.

Question 5: Find the k-th Smallest Element in a Sorted Matrix

Problem Statement:

Given an $m \times n$ matrix where each row and each column is sorted in non-decreasing order, find the k-th smallest element in this matrix in time complexity strictly less than $O(n^2)$.

Matrix Properties:

- The matrix is sorted in non-decreasing order both row-wise and column-wise.
- The size of the matrix is $m \times n$, where m and n are positive integers.
- k is a positive integer where $1 \leq k \leq m \times n$.

Pseudocode:

```
Function count_less_or_equal(matrix, mid, rows, cols):
    count, i, j = 0, rows - 1, 0  // Start at bottom-left corner

    While i >= 0 AND j < cols do:
        If matrix[i][j] <= mid:
            count += (i + 1)  // All elements in the current column are <= mid
            j += 1        // Move to next column
        Else:
            i -= 1       // Move up to the previous row

    Return count

Function find_kth_smallest(matrix, k, rows, cols):
     // Min and max values in the matrix
    low, high = matrix[0][0], matrix[rows - 1][cols - 1]

    While low < high do:
        mid = (low + high) / 2    // Calculate middle value
        If count_less_or_equal(matrix, mid, rows, cols) < k:
            low = mid + 1         // k-th smallest must be on the right side
        Else:
            high = mid      // k-th smallest might be in the left side or is mid

    Return low            // Low is the k-th smallest element

Function main(matrix, k, rows, cols):
    Return find_kth_smallest(matrix, k, rows, cols)
```