CS-700 Algorithms and Complexity

Assignment-2

Abhijith C
Roll Number: 242CS003
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:

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

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

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.

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

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:

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

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

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

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)
```

Time Complexity Analysis:

The algorithm to find the k-th smallest element in a sorted matrix involves binary search on the range of matrix values and counting elements in each iteration. Below is a detailed breakdown:

- **Range of Values**:
  - The matrix has its smallest element at matrix[0][0] and the largest element at matrix$[m-1][n-1]$.
  - The value range, V, is given by:

$$V = \text{matrix}[m-1][n-1] - \text{matrix}[0][0]$$

- **Binary Search**:
  - A binary search is performed over the value range V, where each step halves the search space.
  - The binary search takes $O(\log V)$ iterations.

- **Counting Elements**:
  - For each midpoint in the binary search, the algorithm counts how many elements in the matrix are less than or equal to the midpoint.
  - This counting is done using a two-pointer technique, starting from the bottom-left of the matrix, and takes $O(m + n)$, where m is the number of rows and n is the number of columns.
- Each binary search iteration requires $O(m+n)$ time for counting elements.
- With $O(\log V)$ iterations, the total time complexity becomes:

  $$T(n) = O((m + n) \log V)$$

  where:
  - m is the number of rows in the matrix.
  - n is the number of columns in the matrix.
  - V is the range of values in the matrix.

Thus, the overall complexity $O((m + n) \log V)$ is efficient.

Question 6: Count of Smaller Elements to the Right

Problem Statement: Given an integer array of length n, for each element in the array, determine how many elements to its right are smaller than that element. You are required to solve this problem using a divide and conquer approach with a time complexity of $O(n \log n)$.

```
Function countAndMerge(array, output, index, low, mid, high):
    n = high - low + 1                    // Total elements in segment
    Create buffer[n]                       // Temp array for sorted elements
    bufferCount = 0                        // Counter for buffer
    left = low                             // Left pointer for left subarray
    right = mid + 1                        // Right pointer for right subarray

    // Merging process
    While left <= mid AND right <= high do:
        If array[left] > array[right] then:
            // Count smaller elements than current left element
            output[index[array[left]]] += (high - right + 1)
            buffer[bufferCount++] = array[left]      // Add left element
            left++
        Else:
            buffer[bufferCount++] = array[right]    // Add right element
            right++

    // Copy remaining elements from left subarray to buffer
    While left <= mid do:
        buffer[bufferCount++] = array[left++]

    // Copy remaining elements from right subarray to buffer
    While right <= high do:
        buffer[bufferCount++] = array[right++]

    // Copy sorted buffer back to original array
    For i from 0 to n - 1 do:
        array[low + i] = buffer[i]      // Copy sorted buffer back


Function countSmallestElementsInRight(array, output, index, low, high):
    If high > low then:
        mid = (low + high) / 2      // Calculate midpoint
        // Recursively sort and count for left half
        countSmallestElementsInRight(array, output, index, low, mid)
        // Recursively sort and count for right half
        countSmallestElementsInRight(array, output, index, mid + 1, high)
        // Merge two halves and count smaller elements
        countAndMerge(array, output, index, low, mid, high)


Function main():
    size = Read user input           // Read array size
    Create array[size], output[size]      // Dynamically allocate memory
    Create index[100]  // Assumption: elements are <= 100

    // Input elements into array and initialize output counts
    For i from 0 to size - 1 do:
        array[i] = Read user input     // Read each element
        output[i] = 0                        // Initialize output counts
        index[array[i]] = i                 // Store original index

    // Call function to count smallest elements on the right
    countSmallestElementsInRight(array, output, index, 0, size - 1)
```

Time Complexity Analysis:

The algorithm uses a modified merge sort to count the number of smaller elements to the right of each element in the array. The algorithm recursively divides the array into halves until single elements remain, resulting in a recursion tree height of:

$$h = \log_2 n$$

The merge operation has a time complexity of $O(n)$ since it traverses both subarrays. This leads to the recurrence relation:

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

Using the Master Theorem, we find: $a = 2$ $b = 2$ $f(n) = O(n)$
We check:

$$n^{\log_b a} = n^{\log_2 2} = n$$

Since $f(n)$ matches $n^{\log_b a}$, we apply case 2 of the Master Theorem:

$$T(n) = O(n \log n)$$

Thus, the time complexity for the algorithm is:

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

Question 7: Majority Element Problem

Problem Statement: An array $A[1...n]$ has a majority element if more than half of its entries are the same. Given an array, design an efficient algorithm to determine whether the array has a majority element, and, if so, to find that element. The elements are not necessarily from an ordered domain, so comparisons like $A[i] > A[j]$ are not allowed. However, you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

Part (i)

Show how to solve this problem in $O(n \log n)$ time. (Hint: Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

Pseudocode:

```
FUNCTION countOccurrences(arr, size, candidate):
    count = 0  // Initialize count to zero
    FOR i FROM 0 TO size - 1 DO:
        IF arr[i] == candidate THEN:
  // Increment count if candidate matches current element
            count = count + 1
        END IF
    END FOR
    RETURN count  // Return the total count of occurrences
END FUNCTION

FUNCTION majorityElement(arr, left, right):
    // Base case: if the subarray has one element, return that element
    IF left == right THEN:
        RETURN arr[left]
    END IF

    // Find the mid-point of the current subarray
    mid = (left + right) / 2

    // Recursively find the majority elements in the left and right halves
    m1 = majorityElement(arr, left, mid)
    m2 = majorityElement(arr, mid + 1, right)

    // If both halves return the same majority element, return it
    IF m1 == m2 THEN:
        RETURN m1
    END IF

    // Count occurrences of m1 and m2 in the whole array
    count1 = countOccurrences(arr, right - left + 1, m1)
    count2 = countOccurrences(arr, right - left + 1, m2)

    // Check if m1 is the majority element
    IF count1 > (right - left + 1) / 2 THEN:
        RETURN m1
    END IF

    // Check if m2 is the majority element
    IF count2 > (right - left + 1) / 2 THEN:
        RETURN m2
    END IF

    RETURN -1  // No majority element found
END FUNCTION
```

Time Complexity Analysis:

The time complexity of the `majorityElement` function can be determined by analyzing the two key parts of the algorithm:

1. Recursive Division: The function recursively divides the array into two halves, similar to the divide-and-conquer approach used in Merge Sort.

   - At each recursive step, the array is split into two halves.
   - This process continues until the subarrays contain only one element.

   Since the array is halved at each step, the depth of the recursion tree is $O(\log n)$, where n is the size of the array.

2. Counting Occurrences: After solving the problem for the two halves, the function counts the occurrences of the two possible majority elements in the entire array segment.

   - This counting operation takes $O(n)$ time at each level of the recursion.

The total time complexity is the combination of dividing the array recursively and counting the occurrences of potential majority elements. The recurrence relation for this process is:

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

This recurrence relation is a standard divide-and-conquer form, which solves to:

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

The time complexity of the `majorityElement` function is $O(n \log n)$, where n is the size of the input array.

Part (ii)

Can you provide a linear-time algorithm? (Hint: Use another divide-and-conquer approach:

- Pair up the elements of A arbitrarily to form $n/2$ pairs.

- For each pair: if the elements are different, discard both; if they are the same, keep just one. Show that after this procedure, at most $n/2$ elements remain, and they have a majority element if and only if A does.)

Pseudocode:

```
FUNCTION countOccurrences(arr, size, candidate):
    count = 0  // Initialize count to zero
    FOR i FROM 0 TO size - 1 DO:
        IF arr[i] == candidate THEN:
  // Increment count if candidate matches current element
            count = count + 1
        END IF
    END FOR
    RETURN count  // Return the total count of occurrences
END FUNCTION

FUNCTION majorityElementLinear(arr, size):
    candidate = -1  // Variable to store potential majority candidate
    count = 0  // Count to keep track of the current candidate's occurrences

    // First pass: Find a candidate by pairing up elements
    FOR i FROM 0 TO size - 1 DO:
        IF count == 0 THEN:
            candidate = arr[i]  // Set current element as candidate
            count = 1           // Start counting occurrences of this candidate
        ELSE IF arr[i] == candidate THEN:
   // Increment count if current element matches candidate
            count = count + 1
        ELSE:
            count = count - 1 // Decrement count if current element is different
        END IF
    END FOR

    // Second pass: Verify if the candidate is the majority element
    count = 0  // Reset count for verification
    FOR i FROM 0 TO size - 1 DO:
        IF arr[i] == candidate THEN:
  // Increment count if current element matches candidate
            count = count + 1
        END IF
    END FOR

    // Check if the candidate appears more than half the time
    IF count > size / 2 THEN:
        RETURN candidate  // Return candidate if it is the majority element
    ELSE:
        RETURN -1  // No majority element found
    END IF
END FUNCTION
```

Time Complexity Analysis:

The time complexity of the linear-time majority element algorithm can be analyzed as follows:

Finding a Candidate:

- In the first pass through the array, we iterate through all n elements.
- The operations inside the loop take constant time, $O(1)$.
- Therefore, the time complexity for this part is $O(n)$.

Verifying the Candidate:

- In the second pass, we again iterate through all n elements to count the occurrences of the candidate.
- Each operation takes constant time, $O(1)$.
- Thus, the time complexity for this verification step is also $O(n)$.

Overall Time Complexity: The total time complexity of the algorithm is the sum of the time complexities of both passes:

$$O(n) + O(n) = O(n)$$

The overall time complexity of the majority element algorithm using the pairing method is $O(n)$, indicating that it runs in linear time relative to the size of the input array.

Question 8: Strassen's Algorithm for Matrix Multiplication

Implement Strassen's algorithm to multiply two square matrices. Test your implementation by comparing the results with those from the standard matrix multiplication method. Measure and compare the execution times of both methods for matrices of different sizes.

Pseudocode : Standard Matrix Multiplication

```
FUNCTION matrix_multiply(A, B, result, n):
    // Initialize result matrix to zero
    FOR i FROM 0 TO n - 1 DO:
        FOR j FROM 0 TO n - 1 DO:
            result[i][j] = 0  // Set the current element of result to zero

    // Perform the multiplication
    FOR i FROM 0 TO n - 1 DO:
        FOR j FROM 0 TO n - 1 DO:
            FOR k FROM 0 TO n - 1 DO:
   // Accumulate the product
                result[i][j] = result[i][j] + A[i][k] * B[k][j]

END FUNCTION
```

Time Complexity Analysis

- **Outer Loop**: The outer loop runs n times.
- **Middle Loop**: For each iteration of the outer loop, the middle loop runs n times.
- **Innermost Loop**: The innermost loop runs n times for each combination of i and j.

The total number of operations can be calculated as:

$$n \times n \times n = n^3$$

Overall Time Complexity: The overall time complexity of the standard matrix multiplication algorithm is:

$$O(n^3)$$

Pseudocode for Strassen's Algorithm

```
FUNCTION strassen_matrix_multiply(A, B, size, result):
    IF size == 1 THEN
        result[0][0] = A[0][0] * B[0][0]
        RETURN

    // Divide matrices into quadrants
    newSize = size / 2
    A11, A12, A21, A22 = split(A, newSize)
    B11, B12, B21, B22 = split(B, newSize)

    // Calculate M1 to M7 using Strassen's formulas
    M1 = strassen_matrix_multiply(A11 + A22, B11 + B22, newSize)
    M2 = strassen_matrix_multiply(A21 + A22, B11, newSize)
    M3 = strassen_matrix_multiply(A11, B12 - B22, newSize)
    M4 = strassen_matrix_multiply(A22, B21 - B11, newSize)
    M5 = strassen_matrix_multiply(A11 + A12, B22, newSize)
    M6 = strassen_matrix_multiply(A12 - A22, B21 + B22, newSize)
    M7 = strassen_matrix_multiply(A11 - A21, B11 + B12, newSize)

    // Combine results into the final result matrix
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    // Fill the result matrix
    result = combine(C11, C12, C21, C22, newSize)

END FUNCTION

FUNCTION split(matrix, size):
    // Splits a matrix into four equal sub-matrices (quadrants)
    RETURN A11, A12, A21, A22 // Sub-matrices of size `size`

FUNCTION combine(C11, C12, C21, C22, size):
    // Combines four quadrants into a single matrix of size `2 * size`
    RETURN combined_matrix
```

Strassen's algorithm improves the naive $O(n^3)$ complexity of standard matrix multiplication.

Overview of Strassen's Algorithm

Strassen's algorithm multiplies two $n \times n$ matrices by dividing each matrix into four submatrices and recursively computing products of these submatrices.

Steps Involved

1. **Divide**: Split each matrix A and B into four $n/2 \times n/2$ submatrices.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

2. **Conquer**: Compute the following 7 products:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

3. **Combine**: Use the products to form the result matrix C:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 + M_3 - M_2 - M_6 \end{aligned}$$

Recurrence Relation

Let $T(n)$ denote the time complexity:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Solving the Recurrence

Using the Master Theorem:

- $a = 7$, $b = 2$, $k = 2$
- Calculate $\log_b a$:

$$\log_2 7 \approx 2.81 \implies n^{\log_2 7} = n^{2.81}$$

Since $n^{\log_b a} > n^k$, we use case 1 of the Master Theorem:

$$\boxed{T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})}$$

Question 9

Problem:

Given a chessboard with dimensions $n \times n$, where $n = 2^k$ and $k > 0$. On this chessboard, exactly one square is missing. Develop an algorithm to cover the entire chessboard (excluding the missing square) using L-shaped triominoes.

A L-shaped triomino is a 2×2 block with one cell of size 1×1 missing (See Figure 1).

Constraints:

1. Two triominoes should not overlap.
2. Triominoes should not cover the defective (missing) square.
3. Triominoes should cover all other squares.

Pseudocode

We will solve this problem using a divide-and-conquer approach. The algorithm recursively divides the board into quadrants and fills them using L-shaped triominoes while ensuring that the missing square is never covered.

Global Variable Setup:

```
t_count = 0  // Global counter for L-shaped triominoes placed
```

Function: placeT(board, sr, sc, size, mr, mc):

```
function placeT(board, sr, sc, size, mr, mc):
    // Base case: If size is 2x2, place one triomino in available cells
    if size == 2:
        t_count += 1
        for each cell in 2x2 grid:
            if (cell is not the missing one):
                board[sr + i][sc + j] = t_count  // Place triomino
        return  // End base case

    // Recursive case: Divide board into four quadrants
    h_size = size / 2
    t_num = t_count++  // Increment triomino count

    // Top-left quadrant
    if mr < sr + h_size and mc < sc + h_size:
        placeT(board, sr, sc, h_size, mr, mc)
    else:
        board[sr + h_size - 1][sc + h_size - 1] = t_num
        placeT(board, sr, sc, h_size, sr + h_size - 1, sc + h_size - 1)

    // Top-right quadrant
    if mr < sr + h_size and mc >= sc + h_size:
        placeT(board, sr, sc + h_size, h_size, mr, mc)
    else:
        board[sr + h_size - 1][sc + h_size] = t_num
        placeT(board, sr, sc + h_size, h_size, sr + h_size - 1, sc + h_size)

    // Bottom-left quadrant
    if mr >= sr + h_size and mc < sc + h_size:
        placeT(board, sr + h_size, sc, h_size, mr, mc)
    else:
        board[sr + h_size][sc + h_size - 1] = t_num
        placeT(board, sr + h_size, sc, h_size, sr + h_size, sc + h_size - 1)

    // Bottom-right quadrant
    if mr >= sr + h_size and mc >= sc + h_size:
        placeT(board, sr + h_size, sc + h_size, h_size, mr, mc)
    else:
        board[sr + h_size][sc + h_size] = t_num
        placeT(board, sr + h_size, sc + h_size, h_size, sr + h_size, sc + h_size)
```

Main Function:

```
function main():
    // Get board size and missing square
    read b_size
    read mr, mc

    // Allocate the chessboard
    board = allocate 2D array of size b_size x b_size
    initialize all cells to 0

    // Mark the missing square
    board[mr][mc] = -1

    // Call recursive function to place triominoes
    placeT(board, 0, 0, b_size, mr, mc)

    // Print the board
    for each row in board:
        print row  // Output filled board

    // Free allocated memory
    free(board)
```

Time Complexity Analysis

The algorithm follows a divide-and-conquer approach. Here's a step-by-step breakdown of its time complexity:

Key Observations:

- The board is divided into four quadrants of equal size, with one L-shaped triomino placed in each recursive step to ensure that the missing square is not covered.
- The base case occurs when the board is of size 2×2, which is constant time $O(1)$ because we directly place one triomino in the three available squares.
- At each step of recursion, the algorithm processes 4 quadrants, each of size $\frac{n}{2} \times \frac{n}{2}$.

Recursive Relation:

- At each level of recursion, there are four subproblems, each of size $\frac{n}{2} \times \frac{n}{2}$.
- Constant work is done to place one L-shaped triomino, which takes $O(1)$ time.

Thus, the recurrence relation for this algorithm can be written as:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(1)$$

Solving the Recurrence: This is a standard divide-and-conquer recurrence, which can be solved using the **master theorem**. The recurrence relation is:

$$T(n) = 4T\left(\frac{n}{2}\right) + f(n)$$

Where:

$$f(n) = O(1)$$

We now calculate $n^{\log_b a}$, which is the growth of the recursive work:

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

Next, we compare $f(n)$ with n^2.

Since $f(n) = O(1)$, it grows slower than n^2. Therefore, the time complexity is dominated by the recursive work:

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

This means that the time complexity of the triomino placement algorithm is $O(n^2)$.