National Institute of Technology
Karnataka, Surathkal

Assignment -2

CS700

Algorithms And Complexity

Submitted To:
Dr. Vani M
Associate Professor
Department Of Computer
Science And Engineering

Submitted By:
Dileep Reddy G
242CS017
S Vishnu Sai
242CS035

Problems

1. Given a set of strings, find the longest common prefix
   (a) Divide the array into 2 halves until there is only 1 string in each half.
   (b) In the conquering step, compare the prefix of both the strings and return the common substring.
   (c) Once the iterations are completed the function returns the longest common prefix in the given list of strings.

   Time complexity:

$$T(m) = \begin{cases} O(1) & \text{if } m \leq 1 \\ 2T\left(\frac{n}{2}\right) + O(m) & \text{if } m > 1 \end{cases}$$

   Since we divide the array into 2 equal halves and iterate both the strings in the half, the recurrence relation is:

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

   Where 'm' is the size of the longest string in the list.

   Solution using back substitution:

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + O(m)\right) + O(m)$$
$$T(n) = 2^2T\left(\frac{n}{4}\right) + 2O(m) + O(m)$$
$$T(n) = 2^2T\left(\frac{n}{4}\right) + 3O(m)$$
$$T(n) = 2^2\left(2T\left(\frac{n}{8}\right) + O(m)\right) + 3O(m)$$
$$T(n) = 2^3T\left(\frac{n}{8}\right) + 2^2O(m) + 3O(m)$$
$$T(n) = 2^3T\left(\frac{n}{8}\right) + 7O(m)$$

   After k substitutions,

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + (2^k - 1)O(m)$$

   The base case occurs when $\frac{n}{2^k} = 1$, which gives $k = \log_2 n$.

$$T(1) = O(1)$$

   Substituting $k = \log_2 n$ into the general equation:

$$T(n) = 2^{\log_2 n}T(1) + \left(2^{\log_2 n} - 1\right)O(m)$$
$$T(n) = nO(1) + (n-1)O(m)$$
$$T(n) = O(n) + O(nm) = O(nm)$$

   The time complexity of the recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + O(m)$ is **O(nm) .**

2. Give an $O(n^2)$ algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

   (a) Create a lis array of size n, where each element is initialized to 1. This array will store the length of the longest increasing subsequence ending at each index.

   (b) For each element in the array , we compare it with all previous elements:

      i. If the current element arr[i] is greater than a previous element arr[prev], and the longest increasing subsequence ending at arr[i] is smaller than the subsequence ending at arr[prev] plus 1, update the value of lis[i].

   (c) After filling the lis array, the length of the longest increasing subsequence will be the maximum value in the lis array.

   Time complexity:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ \max\limits_{0 \leq j < i} (T(j) + O(1)) & \text{if } n > 1 \end{cases}$$

   Proof:

   1.Base Case:

$$T(1) = O(1)$$

   This is because if the array has one element, the LIS is the element itself, which takes constant time.

   2.Recursive Case:

$$T(n) = \max_{0 \leq j < i} (T(j) + O(1))$$

   For each element $arr[i]$ (where $1 \leq i < n$), we compare it with all previous elements $arr[j]$ (where $0 \leq j < i$) to find the longest subsequence that can be extended by $arr[i]$. This comparison and possible update operation for $LIS[i]$ is $O(1)$.

   3.Combining the Recurrence:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} O(1)$$

   The nested loops result in a time complexity of $O(n^2)$.

3. Implement an algorithm to count the number of inversions in an array using the divide and conquer approach.

   (a) We can use merge sort to count the inversions in an array, First, we divide the array into two halves: a left half and a right half until we get only one number in each half.

   (b) During the merging process, count the number of elements from the left half are greater than the right half which are the inversions required.

   (c) We sum the inversions present in each recursion step, in the end we get the total number of inversions in the array.

   Time complexity:
   Since we divide the array into 2 equal halves and iterate the halves only once, the recurrence relation can be written as:

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

   According to master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

   where:

   - $a = 2$
   - $b = 2$
   - $f(n) = 1$

$$\log_b a = \log_2 2 = 1$$

   Cases in master theorem:

   - If $f(n) = O(n^{\log_b a - E})$, the time complexity is $O(n^{\log_b a})$
   - If $f(n) = \Theta(n^{\log_b a})$, the time complexity is $\Theta(logn * n^{(}\log_b a)$
   - If $f(n) = \Omega(n^{\log_b a + E})$, the time complexity is $\Theta(f(n)$

   In our case:

$$f(n) = 1 \quad \text{and} \quad \log_b a \ = 1$$

   Since $f(n) = \log_b a$, the time complexity is:

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

   The time complexity is **O(nlogn)**.

4. 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.

   (a) Divide the array into groups of size 5. Each group will have 5 elements, except the last group which may have less than 5 elements.

   (b) Create a function to sort the array and find the median, use this function to find the median of all groups and recursively call this function to find the median of medians.

   (c) Partition around the median of medians and assume its position = pos.

   (d) If pos == k return element at pos.

   (e) If pos >k check the left sub array

   (f) If pos <k check the right sub array with k = k - pos + low + 1.

   Time complexity:
   We divide the array into groups of 5 and at least half of the elements in each group i.e 3 are greater than the median of medians except the last groups which has less than 5 elements. This can we written as:

$$3\left(\left\lfloor\frac{1}{2}\left\lfloor\frac{n}{5}\right\rfloor-2\right\rfloor\right)=\frac{3n}{10}-6$$

   In the worst case this can be:

$$n-\left(\frac{3n}{10}-6\right)=\frac{7n}{10}+6$$

   The recurrence relation can be written as:

$$T\left(\frac{n}{5}\right)+T\left(\frac{7n}{10}+6\right)+O(n)$$

   Solution using induction Assume that there exists a constant $c > 0$ such that for some base case (e.g., $n \leq n_0$):

$$T(n) \leq cn$$

   Substitute the inductive assumption into the recurrence relation:

$$T(n)=T\left(\frac{n}{5}\right)+T\left(\frac{7n}{10}+6\right)+O(n)$$

$$T(n) \leq c\frac{n}{5}+c\left(\frac{7n}{10}+6\right)+O(n)$$

$$T(n) \leq c\frac{n}{5}+c\left(\frac{7n}{10}\right)+6c+O(n)$$

$$T(n) \leq \frac{c}{5}n+\frac{7c}{10}n+6c+O(n)$$

$$T(n) \leq \left(\frac{c}{5} + \frac{7c}{10}\right) n + 6c + O(n)$$

$$T(n) \leq \left(\frac{2c}{2}\right) n + 6c + O(n)$$

$$T(n) \leq cn + 6c + O(n)$$

$$T(n) \leq cn$$

The time complexity of the algorithm is **O(n)**

5. Given an m×n matrix where each row and each column is sorted in ascending order. Find the k-th smallest element in this matrix with running time strictly less than O(n2).

   (a) Divide the matrix range into two halves.

   (b) Perform a binary search on the possible range of values in the matrix (between $Mat[0][0]$ and $Mat[n-1][n-1]$).

   (c) Recursively solve the problem by splitting the matrix value range during binary search.Each time, calculate how many elements are smaller than or equal to the midpoint of the current range.

   (d) During the midpoint calculation step (just like in merge sort), count how many elements are smaller than or equal to the midpoint by using binary search within each row.

   (e) While counting, if the midpoint value is smaller than an element in the current row, we reduce the search space for that row, and when the midpoint is greater than a row's element, we count all the smaller elements to the left in that row (because the row is sorted).

Time complexity:
Since we divide the matrix value range into two halves (binary search) and for each midpoint, we perform a binary search within each row, the recurrence relation can be written as:

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

Where $T(n)$ is the time complexity for counting elements smaller than or equal to a given number, and $O(n \log n)$ is the time for the counting process in each step.

According to master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a = 1$
- $b = 2$
- $f(n) = O(n \log n)$

$\log_b a$:

$$\log_b a = \log_2 1 = 0$$

Cases in master theorem:

- If $f(n) = O(n^{\log_b a - E})$, the time complexity is $O(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, the time complexity is $\Theta(logn * n^{(}\log_b a)$
- If $f(n) = \Omega(n^{\log_b a + E})$, the time complexity is $\Theta(f(n)$

In our case: The function $f(n) = O(n \log n)$ has an extra logarithmic factor compared to $O(n)$.

Since $f(n)$ is larger than $n^{\log_b a}$, we apply Case 3 of the Master Theorem, where $f(n)$ dominates. In this case, the time complexity is driven by $f(n) = O(n \log n)$, and since the binary search has $O(\log y)$ iterations (where y is the range of values between $Mat[0][0]$ and $Mat[n-1][n-1]$), the total time complexity becomes:

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

Thus, the overall time complexity is:

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

Where y represents the range of values in the matrix. Since y is typically $O(n^2)$, this simplifies to:

$$T(n) = (O(n \log^2 n))$$

The time complexity is $O(n \log^2 n)$.

6. 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 need to solve this problem using divide and conquer approach with time complexity of O(n log n).

   (a) Divide the array into two halves.

   (b) Recursively solve the problem for both halves.

   (c) During the merge step (just like in merge sort), we can count how many smaller elements are on the right side of each element in the left half by comparing it to elements in the right half.

   (d) While merging, if an element from the left half is greater than an element from the right half, it means that all remaining elements in the right half (since the right half is sorted) are smaller than the current element in the left half. We increment the count for the left element by the number of remaining elements in the right half.

   Time complexity:
   Since we divide the array into 2 equal halves and iterate the halves only once, during the merge step, we count the smaller elements on the right. the recurrence relation can be written as:

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

   According to master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

   where:

   - $a = 2$
   - $b = 2$
   - $f(n) = 1$

   $\log_b a$:

$$\log_b a = \log_2 2 = 1$$

   Cases in master theorem:

   - If $f(n) = O(n^{\log_b a - E})$, the time complexity is $O(n^{\log_b a})$
   - If $f(n) = \Theta(n^{\log_b a})$, the time complexity is $\Theta(logn * n^{(}\log_b a)$
   - If $f(n) = \Omega(n^{\log_b a + E})$, the time complexity is $\Theta(f(n)$

   In our case:

$$f(n) = 1 \quad \text{and} \quad \log_b a \ = 1$$

   Since $f(n) = \log_b a$, the time complexity is:

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

   The time complexity is **O(nlogn)**.

7. An array $A[1 \ldots n]$is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is A[i] ¿ A[j]?". (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is A[i] = A[j]?" in constant time.

   i. Show how to solve this problem in O(nlogn) time. (Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

   (a) Divide the array into two halves. Split the array A into two subarrays A_1 and A_2, each of size $n/2$, recursively solving the problem for both halves.

   (b) Recursively solve the problem for both halves. Recursively find the majority element in A_1 and A_2. If both halves have the same majority element, then that element is the majority element for the whole array.

   (c) Combine the results in the merge step. If the majority elements in A_1 and A_2 are the same, return that element. If they are different, count the occurrences of both majority candidates in the full array and check if any of them occurs more than $n/2$ times. If none of them does, then no majority element exists.

   (d) While merging, count how often the two candidate elements from the subarrays occur in the entire array. If one of the candidates occurs more than $n/2$ times, it is the majority element.

   Time complexity:
   Since we divide the array into two halves and count the occurrences of the majority elements during the merge step, the recurrence relation can be written as:

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

   According to master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

   where:

   - $a = 2$
   - $b = 2$
   - $f(n) = 1$

   $\log_b a$:

$$\log_b a = \log_2 2 = 1$$

   Cases in master theorem:

   - If $f(n) = O(n^{\log_b a - E})$, the time complexity is $O(n^{\log_b a})$
   - If $f(n) = \Theta(n^{\log_b a})$, the time complexity is $\Theta(logn * n^{(}\log_b a)$
   - If $f(n) = \Omega(n^{\log_b a + E})$, the time complexity is $\Theta(f(n)$

In our case:

$$f(n) = 1 \quad \text{and} \quad \log_b a \ = 1$$

Since $f(n) = \log_b a$, the time complexity is:

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

The time complexity is **O(nlogn)**.

7.2 Can you give a linear-time algorithm?

(a) Divide the array into $n/2$ pairs.
For each pair:
- If the two elements are different, discard both of them.
- If the two elements are the same, keep one of them.

(b) Recursively reduce the number of elements.After the first pass, there are at most $n/2$ elements left. Recursively apply the same procedure to the remaining elements until only one element or a few elements are left.

(c) After the elimination process, count the occurrences of the remaining candidate in the original array. If it appears more than $n/2$ times, it is the majority element. Otherwise, no majority element exists.

Time complexity:
In each step, we reduce the number of elements by half and perform $O(n)$ work to process the pairs and count the occurrences. Thus, the recurrence relation can be written as:

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

According to master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a = 1$
- $b = 2$
- $f(n) = O(n)$

$\log_b a$:

$$\log_b a = \log_2 1 = 0$$

Cases in master theorem:

- If $f(n) = O(n^{\log_b a - E})$, the time complexity is $O(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, the time complexity is $\Theta(logn * n^{(} \log_b a)$
- If $f(n) = \Omega(n^{\log_b a + E})$, the time complexity is $\Theta(f(n)$

In our case:

$$f(n) = 1 \quad \text{and} \quad \log_b a \ = 0$$

Since $f(n) = \log_b a$, the time complexity is:

$$T(n) = O(n)$$

The time complexity is **O(n)**.

8. Implement Strassen's algorithm for multiplying 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.

Strassen's algorithm

Strassen's algorithm multiplies two $n \times n$ matrices more efficiently than the standard algorithm.

(a) **Divide:** Divide the input matrix into

$$\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$$

matrices till we get 2 X 2 matrices.

(b) **Conquer:** Calculate the below values to multiply the 2X2 matrices:

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

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

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

(c) After performing the above calculations recursively, we get the multiplication of the input matrices

Time complexity:
In the algorithm mentioned above, we are calculating 7 values of the n/2 matrix and using those to find the multiplication of the matrices, which can be represented as:

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

According to master theorem we can see that, a = 7, b = 2 and the above recurrence relation comes under Case 1 of the master theorem where T(N) = f(n), so we can say that,

$$T(n) = n^{\log_2 7} = \mathbf{n^{2.81}}$$

$\therefore$ The time complexity is $\mathbf{O(n^{2.81})}$

Standard matrix multiplication

Recurrence relation:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

According to master theorem we can see that, a = 8, b = 2 and the above recurrence relation comes under Case 1 of the master theorem where T(N) = f(n), so we can say that,

$$T(n) = n^{\log_2 8} = \mathbf{n^3}$$

$\therefore$ The time complexity is $\mathbf{O(n^3)}$

Comparison between Strassen's matrix multiplication and standard multiplication

Theoretically Strassen's matrix multiplication algorithm should perform better than the standard matrix multiplication since the time complexity of strassen's matrix multiplication algorithm is less than the standard way, but this may not be the case every time due to the following reasons:

(a) **Recursion overhead:**
Strassen's algorithm involves recursive division of matrices, where matrices are split into submatrices and then combined. This recursive process can introduce significant overhead, especially for small matrices. In contrast, traditional matrix multiplication follows a straightforward loop-based approach with less overhead.

(b) **Threshold for Matrix Size:** Strassen's algorithm typically becomes faster than traditional methods only for large matrix sizes. For smaller matrices (for example, $n \leq 64$ or even larger depending on the system), the overhead outweighs the benefits, and traditional matrix multiplication is faster.

(c) Strassen's algorithm tends to outperform traditional matrix multiplication for very large matrices (often in the range of $n \geq 512$ or even larger).

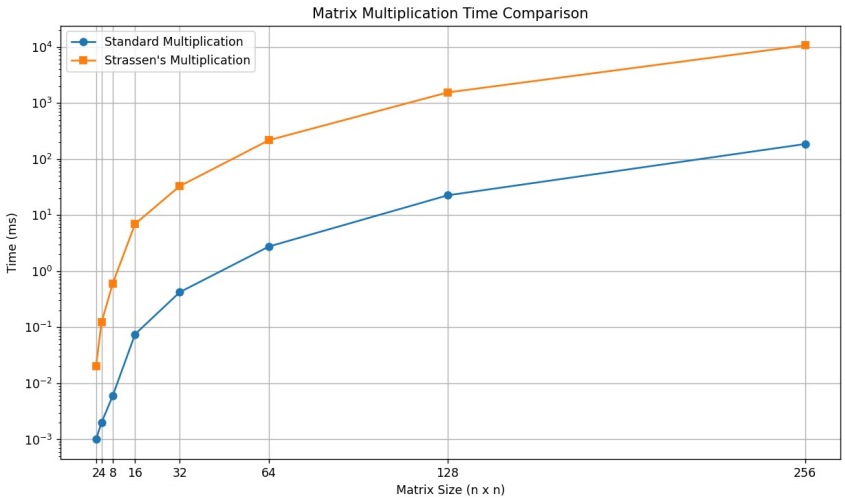

Figure 1: Comparison between strason's multiplication and standard multiplication

9. Given a chessboard with dimensions n × n, where n = 2k and k ¿ 0 . On this chessboard, exactly one square is missing (See Figure 2). 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. Constraints:
i. Two triominoes should not overlap.
ii. Triminoes should not cover defective square.
iii. Triominoes should cover all other squares.

Algorithm

(a) **Divide:**

i. Divide the given chessboard into two equal halves until the base case i.e 2X2 grid is not obtained.

ii. In the base case there are 4 cells present, so add a triomino, which leaves a single cell empty.

(b) **Conquer:**

i. Place the triomino in the center of the quadrant such that it does not cover the n/2 X n/2 subsquare that has a missing square.

ii. Since we insert a triomino in each quadrant, there exist a empty cell in each of the 4 quadrants i.e the square has 4 empty cells.

iii. Add a triomino to cover 3 empty cells from each quadrant, such leaving the defective cell as it is.

Time complexity:
In each iteration, we divide the grid into 4 quadrants and solve that sub problem. So we can write the recurrence relation as:

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

According to master theorem we can we see this belongs to case 1, so the time complexity is:

$$T(n) = n^{\log_2 4} = \mathbf{n^2}$$

$\therefore$ The time complexity is $\mathbf{O(n^3)}$

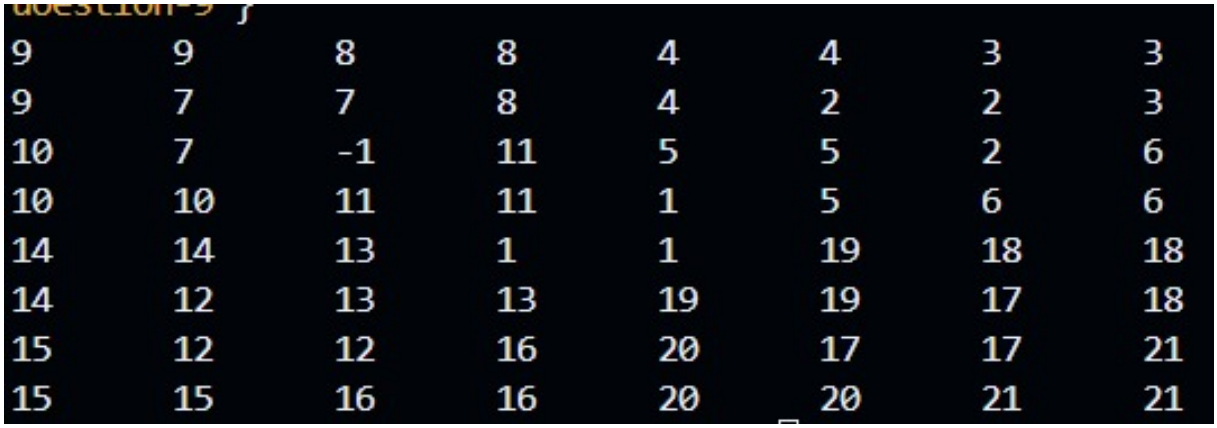

Figure 2: Code Output