Academic year 2024-2025 (Even Sem)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Date	1st April 2025	Maximum Marks	50+10
Course Code	CD343AI	Duration	90+30
Sem	IV	CIE I	
UG	UG	SCHEME AND SOLUTION	
Design and Analysis of Algorithms (Common to CS/CD/CY/IS/AIML)			

S.No	Part A	M
1.1	Since the median is chosen in O(n) time, it ensures a balanced partition of the array into two halves. The recurrence relation for this modified Quicksort is: $T(n)=T(n/2) + O(n)$ Solving by backward substitution: $T(n)=T(n/2)+n$ $=T(n/4)+n/2+n$ $=T(n/8)+n/4+n/2+n$ $=\cdots+O(n)$ Expanding until the base case $T(1)=O(1)$, we get O(n log n) complexity. Final Answer: **O(n log n)** Can also be solved using Master Method	01
1.2	The function contains two nested loops, each running n^2 times, followed by a recursive call fun(n-3). Hence $T(n) = O(n^2) + T(n-3)$ for $n > 1$ $T(1) = 1$ Final Answer: $O(n^3)$ (OPTIONAL)	01
1.3	Bubble Sort Process: Initial Array: [7, 2, 5, 1, 9] 1st Pass: [2, 5, 1, 7, 9] (3 swaps) 2nd Pass: [2, 1, 5, 7, 9] (1 swap) 3rd Pass: [1, 2, 5, 7, 9] (1 swap) Final Answer: **1 swap** in the 3rd pass **Tracing – 1 Mark,** **Final answer – 1 Mark**	02
1.4	i) $T(n) = \Theta(n \log n)$ (Case 3) **1 Mark** ii)Does not apply ($a < 1$) **1 Mark**	02
1.5	Innermost loop enters infinite loop, hence invalid algorithm, cannot derive T(n) – **2 Marks.** **In the case students corrects the innermost loop iterator update to k=k*2, then** • **Outer loop**: i runs from n/2 to n → **O(n)** • **Middle loop**: j doubles each time (logarithmic) → **O(log n)** • **Inner loop**: k doubles each time → **O(log n)** $T(n) = O(n) \times O(\log n) \times O(\log n) =$ $\mathbf{O(n \log^2 n)}$	02

1.6	$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}.$$ Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$.	02
	Part B	

2. a | 05

Solution
to compute $\mathbf{2^n}$ for any nonnegative integer n that is based on the formula $\mathbf{2^n = 2^{n-1} + 2^{n-1}}$

Recursive Algorithm: **1 mark**

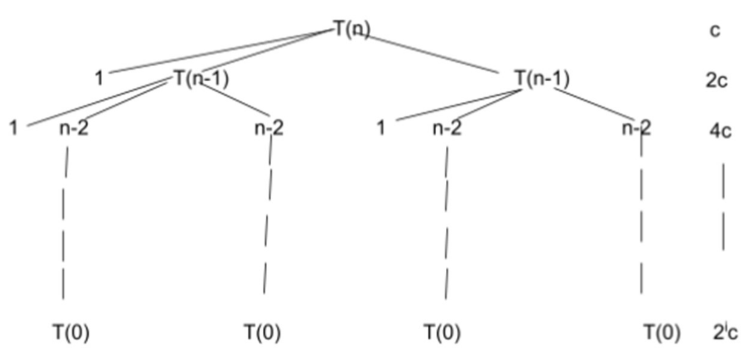
```
int compute (int n) {
   if (n == 0)
       return 1;
   return  compute (n - 1) +  compute (n - 1);
}
```

Recurrence Relation + Solution: **(1+ 1) = 2 marks**
Let T(n) be the number of additions, the $T(n)=2T(n-1) + 1$, $T(0) = 0$

Using backward substitution method:
$T(n)=2(2T(n-2) +1)+1$
$\quad =4T(n-2) + 2 + 1$

$T(n)=8T(n-3)+4+2+1$
…
$T(n)=2^nT(0) + (2^n-1)$
Since $T(0) = 0$,
$T(n)=2^n-1 = O(2^n)$. Thus, the algorithm performs exponential additions.

Recursive Call Tree: **1 Mark**



Is it a good algorithm? 1 Mark
No, it is highly inefficient (**O(2^n)**). Instead, an **iterative approach** or using bitwise shifting (1<<n) achieves **O(1)** time complexity.

2. b	**Solution:** Optimal Algorithm: **4 marks** 1. Label the jars 1 to 5. 2. Take: 1 pill from jar 1 2 pills from jar 2 3 pills from jar 3 4 pills from jar 4 5 pills from jar 5 3. Weigh the total pills (**say W)**. 4. The expected weight (if all were 10g) is: =1(10) + 2(10) + 3(10) + 4(10) + 5(10) = 150g The actual weight will be: X = **150** − W where X (the weight difference) identifies the contaminated jar. Efficiency Analysis: **1 mark** Single weighing operation → **O(1)** Best algorithm for the problem as it minimizes scale usage to just one weighing.	05

3. a	Algorithm: **2 marks**
	
	ALGORITHM *SelectionSort*($A[0..n-1]$)
	//Sorts a given array by selection sort
	//Input: An array $A[0..n-1]$ of orderable elements
	//Output: Array $A[0..n-1]$ sorted in nondecreasing order
	for $i \leftarrow 0$ **to** $n-2$ **do**
	$min \leftarrow i$
	for $j \leftarrow i+1$ **to** $n-1$ **do**
	if $A[j] < A[min]$ $min \leftarrow j$
	swap $A[i]$ and $A[min]$
	
	Time complexity analysis as per the mathematical framework : 3 Marks
	Best = worst = average case = $O(n^2)$ or $\Theta(n^2)$
	
	Time complexity: $O(n^2)$
	Space complexity: O(1)
	In-place/out-of-place? In-place
	Stability? Unstable
	Comparison Sort? Comparison
		05

3. b	**Strassen's algorithm: 2 marks** $$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$ $$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$ where: $$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$ $$m_2 = (a_{10} + a_{11}) * b_{00}$$ $$m_3 = a_{00} * (b_{01} - b_{11})$$ $$m_4 = a_{11} * (b_{10} - b_{00})$$ $$m_5 = (a_{00} + a_{01}) * b_{11}$$ $$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$ $$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$ Time complexity : **2 marks** • Input size – N (matrix order) • Basic operation – Multiplication • Number of multiplications M (n) will be: **M(n) = 7M(n/2) for n > 1,** **M(1) = 1** Solving it by backward substitutions for $n = 2^k$: $M(2^k) = 7M(2^{k-1})$ $= 7[7M(2^{k-2})] = 7^2M(2^{k-2})$ $= \ldots$ $= 7^iM(2^{k-i})$ $= \ldots$ $= 7^kM(2^{k-k}) = 7^k$ Since $k = \log_2 n$ $$\boxed{M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}}$$ Comparision: **1 mark** Strassen's algorithm ($O(n^{2.81})$ is asymptotically faster than conventional matrix multiplication ($O(n^3)$, but its high constant factors limit practicality to large matrices.	05

4	• Merge sort algorithm: **2 marks** • Merge algorithm: **3 marks** • Deriving recurrence relation + solution**: (2 + 3) = 5 marks**	10

```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)
```

```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    i ← 0; j ← 0; k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j]
            A[k] ← B[i]; i ← i + 1
        else A[k] ← C[j]; j ← j + 1
        k ← k + 1
    if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
    else copy B[i..p − 1] to A[k..p + q − 1]
```

1. input's size: n – number of elements to be sorted. (Assuming for simplicity that n is a power of 2)
2. basic operation: comparison
3. No worst, average, and best cases
4. Let T(n) = number of times the basic operation is executed.
   $T(n) = 2T(n/2) + T_{divide\_merge}(n)$ for $n > 1$, $T(1) = 0$

Solve using back substitution:

$$T(n) = 2T(n/2) + O(n)$$

$$= 4T(n/4) + 2O(n/2) + O(n)$$

$$= 8T(n/8) + 4O(n/4) + 2O(n/2) + O(n)$$

General Form:

$$T(n) = 2^k T(n/2^k) + kO(n)$$

Stopping Condition:

When $n/2^k = 1$, then $k = \log_2 n$.

Final Complexity:

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

5	**Counting Inversions Algorithm (Using Modified Merge Sort): (3+4) = 7 marks** **Time analysis: 3 marks**	10

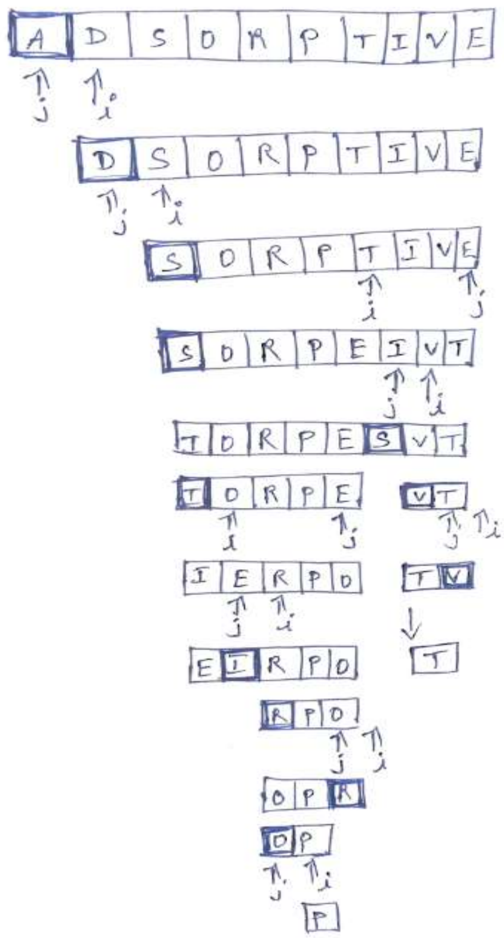
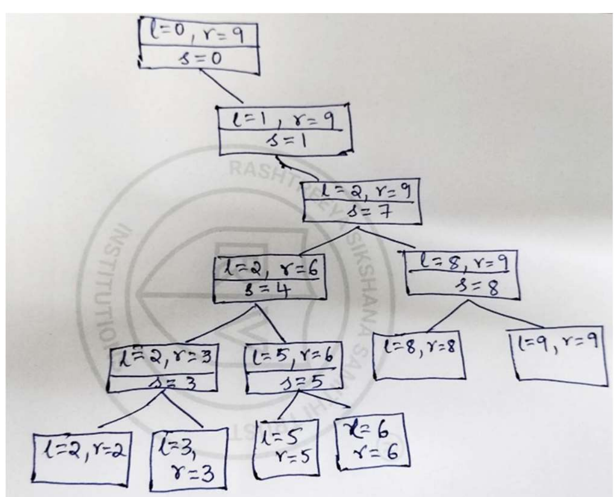
```
ALGORITHM CountInversions(A, left, right)
    IF left >= right THEN
        RETURN 0
    END IF
    mid ← (left + right) / 2
    invLeft ← CountInversions(A, left, mid)
    invRight ← CountInversions(A, mid + 1, right)
    invMerge ← MergeAndCount(A, left, mid, right)
    RETURN (invLeft + invRight + invMerge)
```

```
ALGORITHM MergeAndCount(A, left, mid, right)
    i ← left, j ← mid + 1, k ← 0
    invCount ← 0
    CREATE Temp[right - left + 1]

    WHILE i ≤ mid AND j ≤ right DO
        IF A[i] ≤ A[j] THEN
            Temp[k] ← A[i]
            i ← i + 1
        ELSE
            Temp[k] ← A[j]
            invCount ← invCount + (mid - i + 1)
            j ← j + 1
        END IF
        k ← k + 1
    END WHILE

    COPY remaining elements from A[left..mid] to Temp[]
    COPY remaining elements from A[mid+1..right] to Temp[]
    COPY Temp[] back to A[left..right]

    RETURN invCount
```

Time Complexity Analysis: This approach modifies Merge Sort to count inversions while sorting. The recurrence is: T(n)=2T(n/2)+O(n) which solves to: O(n log n)

Thus, the algorithm efficiently counts inversions in O(nlogn), much better than the naive O(n2) approach.

Note: Marks will only be awarded for efficient strategies; inefficient approaches may not be considered

5	Quicksort algorithm/s: **(2 + 3) = 5 marks** Tracing**: 3 marks** Recursion tree**: 2 marks**  Recursion call tree 	10