| Vivekanand Education Society's Institute of Technology, Chembur, Mumbai, Department Of Computer Engineering Year 2024-25 MID TERM TEST |
|---|

| | |
|---|---|
| **Class : Second Year (D7)** | **Division: A/ B/ C** |
| **Semester: IV** | **Subject: Design and Analysis of Algorithm** |
| **Date: 12th March 2025** | **Time: 9:00 am** |

**Q1. (Attempt any five of the following.)**
**a. Apply Master Theorem to derive the time complexity for given recurrence relation : T(n) = 3 T(n/2) + n n >= 1**

using the **Master Theorem**, we compare it with the standard form:

$$T(n) = aT(n/b) + f(n)$$

where:

- $a = 3$ (number of recursive calls),
- $b = 2$ (factor by which the problem size is reduced),
- $f(n) = n$ (additional work done outside the recursive calls).

the Master Theorem case that applies is **Case 1**: $T(n) = \Theta(n^{\log_2 3})$

==================================================================

**Q1 b. Derive the Time Complexity of the given code snippet**

```
for ( i = 0 ; i < n ; i++)
    for ( j = 1 ; j < n ; j = j  * 2)
        // statement
```

The outer `for` loop runs from `i = 0` to `i < n`, incrementing `i` by `1` in each iteration.
Thus, it executes **O(n) times**.

The inner `for` loop starts from `j = 1` and doubles (`j = j * 2`) in each iteration until `j < n`

- The values of `j` will be:
  $1, 2, 4, 8, 16, \ldots, n$
- This forms a geometric progression, stopping when $j \geq n$.
- The number of iterations satisfies:

$$j = 2^k \leq n$$

Taking logarithm on both sides:

$$k \leq \log_2 n$$

Thus, the inner loop runs **O(log n) times**.

Since the outer loop runs **O(n) times**, and the inner loop runs **O(log n) times** for each outer iteration, the total time complexity is: $O(n \log n)$

==================================================================

**Q1 c. Apply Quick Sort upon the following elements considering the first element as the pivot 4, 3, 8, 1, 7, 9**

Step 1: Partitioning Around Pivot (4)

- Pivot: 4
- Elements less than 4: [3, 1]
- Elements greater than 4: [8, 7, 9]
- After partitioning: [3,1] 4 [8,7,9] (Pivot is now at the correct position, index 2)

Step 2: Recursively Sort Left Subarray [3, 1]

- Pivot: 3 (first element)
- Elements less than 3: [1]
- Elements greater than 3: None
- After partitioning: [1] 3 (Pivot 3 is at the correct position, index 1)
- The subarray [1] is already sorted.

Step 3: Recursively Sort Right Subarray [8, 7, 9]

- Pivot: 8 (first element)
- Elements less than 8: [7]
- Elements greater than 8: [9]
- After partitioning: [7] 8 [9] (Pivot 8 is at the correct position, index 4)
- The subarrays [7] and [9] are already sorted.

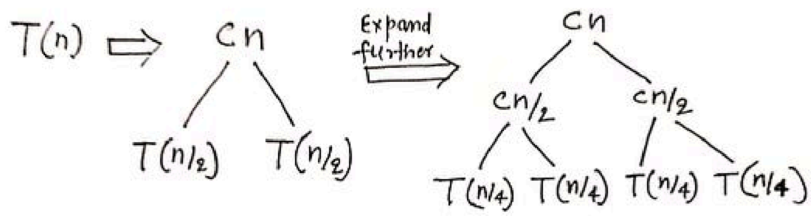
Final Sorted Array ⇒ Combining all partitions: [1,3,4,7,8,9]

==========================================================================

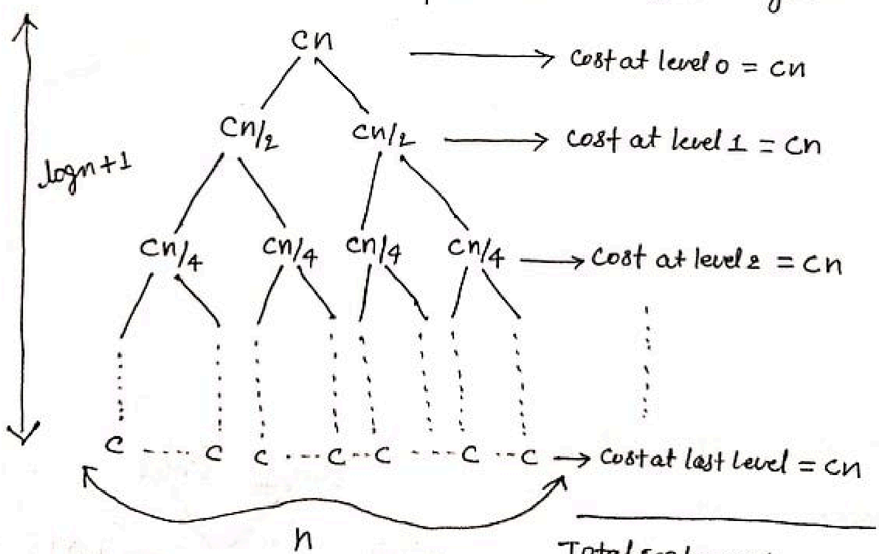
**Q1 d. Derive the Time complexity of Merge Sort using Recursive Tree method.**

Recurrence relation of the Merge sort
$$T(n) = \begin{cases} c, & \text{if } n=1 \\ 2T(n/2) + cn, & \text{if } n>1 \end{cases}$$

Let's draw the recursion tree




So on ---- here is the complete recursion tree diagram




Total cost = cnlogn + cn

**Q1 e. Find the minimum and maximum of an array using Divide and Conquer Strategy for [2, 5, 8, 1, 9, 6].**

Step 1: Divide the Array

We divide the array into two halves recursively until we get base cases.

Divide into two halves:

- Left: [2, 5, 8]
- Right: [1, 9, 6]

Step 2: Recursively Find Min & Max for Each Half

Left Half: [2, 5, 8]

Divide further:

- Left: [2] (Base Case → Min = 2, Max = 2)
- Right: [5, 8]
    - Min = 5, Max = 8

Merge:

- Min = min(2, 5) = 2
- Max = max(2, 8) = 8
- Result for [2, 5, 8]: (Min = 2, Max = 8)

Right Half: [1, 9, 6]

Divide further:

- Left: [1] (Base Case → Min = 1, Max = 1)
- Right: [9, 6]
    - Min = 6, Max = 9

Merge:

- Min = min(1, 6) = 1
- Max = max(1, 9) = 9
- Result for [1, 9, 6]: (Min = 1, Max = 9)

Step 3: Merge the Two Halves

We now merge the results from both halves:

- Left Half: (Min = 2, Max = 8)
- Right Half: (Min = 1, Max = 9)

Final result:

- Minimum = min(2, 1) = 1
- Maximum = max(8, 9) = 9

==========================================================================

**Q1 f. Explain Strassen's Matrix Multiplication Algorithm**

Strassen's algorithm is an efficient **divide-and-conquer** algorithm for **matrix multiplication** that reduces the number of multiplications compared to the standard approach. For two n×nn \times nn×n matrices, the **standard matrix multiplication** takes **O($n^3$)** time. Strassen's algorithm reduces the complexity to **O($n^{\log 7}$})** ≈ **O($n^{2.81}$)**, making it more efficient for large matrices.

Suppose you want to multiply two $2 \times 2$ matrices:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Instead of performing the traditional 8 multiplications for computing the matrix product $C = A \times B$, Strassen's algorithm reduces this to 7 multiplications by using intermediate terms.

Strassen defines 7 auxiliary products (which are combinations of sums and differences of elements of $A$ and $B$):

$$M_1 = (a+d)(e+h)$$

$$M_2 = (c+d)e$$

$$M_3 = a(f-h)$$

$$M_4 = d(g-e)$$

$$M_5 = (a+b)h$$

$$M_6 = (c-a)(e+f)$$

$$M_7 = (b-d)(g+h)$$

From these intermediate products, the final result matrix $C$ is constructed as:

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

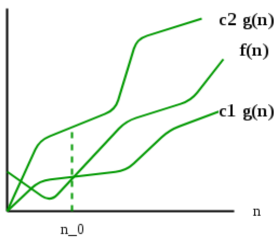
So, the algorithm replaces the usual 8 multiplications with 7, resulting in fewer computations. This trick is recursively applied to smaller and smaller submatrices until reaching the base case (typically $2 \times 2$ matrices).

==========================================================================

**Q2. a. Explain Asymptotic notation with proper graphs and examples**

**1) Θ Notation:** bounds a functions from above and below, so it defines exact asymptotic behavior.
Eg : 3n3 + 6n2 + 6000 = Θ(n3)

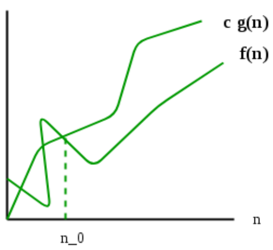
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0
such that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}




f(n) = theta(g(n))

**2) Big O Notation:** defines an upper bound of an algorithm, it bounds a function only from above.
Eg: Insertion Sort takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

O(g(n)) = { f(n): there exist positive constants c and n0
such that 0 <= f(n) <= cg(n) for all n >= n0}



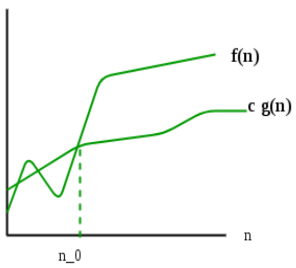

f(n) = O(g(n))

**3) Ω Notation:** provides an asymptotic lower bound.

Ω (g(n)) = {f(n): there exist positive constants c and n0
such that 0 <= cg(n) <= f(n) for all n >= n0}.

Eg : Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in the worst case and sometimes in the average case.

============================================================




f(n) = Omega(g(n))

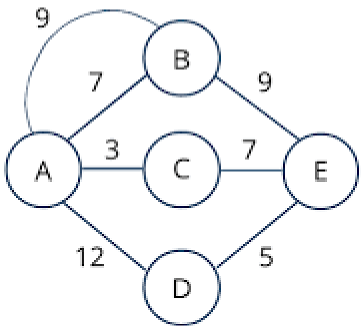
**Q2. b. Compare Insertion Sort with Selection Sort with respective the following parameters in a tabular format.**

1. **Time Complexity (Best case)**
2. **Time Complexity (Worst Case)**
3. **No. of Comparisons**
4. **Space Complexity**
5. **Adaptive (Efficiency in a nearly sorted data)**

| Parameter | Insertion Sort | Selection Sort |
|---|---|---|
| Time Complexity (Best Case) | $O(n)$ | $O(n^2)$ |
| Time Complexity (Worst Case) | $O(n^2)$ | $O(n^2)$ |
| Number of Comparisons | Best case: $O(n)$ (nearly sorted data)<br>Worst case: $O(n^2)$ | Always $O(n^2)$ |
| Space Complexity | $O(1)$ | $O(1)$ |
| Adaptive (Efficiency in Nearly Sorted Data) | Adaptive: More efficient on nearly sorted data, since it requires fewer shifts and comparisons. | Non-adaptive: Always performs the same number of comparisons regardless of how sorted the data is. |

==========================================================================

**Q3. a. Find the Minimum spanning tree using Prim's algorithm.**




| **Current vertex** | **EdgeSet** | **Selected Edge** | **VertexSet** | **Cost** |
|---|---|---|---|---|
| A | {(A,B), (A,C), (A,B), (A,D)} | (A,C) | {A,C} | 3 |
| C | {(A,B), (A,B), (A,D), (C,E)} | (C,E)<br>or<br>(A,B) | {A,C,E}<br>or<br>{A,B,C} | 10 |
| E<br>or<br>B | {(A,B), (A,B), (A,D),(E,B),(E,D)}<br>or<br>{(A,B), (A,D), (C,E),(B,E)} | (E,D)<br>or<br>(C,E) | {A,C,D,E}<br>or<br>{A,B,C,E} | 15<br>or<br>17 |
| D<br>or<br>E | {(A,B), (A,B), (A,D),(E,B)}<br>or<br>{(A,B), (A,D), (B,E), (E,D)} | (A,B)<br>or<br>(E,D) | {A,B,C,D,E}<br>or<br>{A,B,C,D,E} | **22** |

==========================================================================

**Q3. b. Fill the Knapsack (Capacity = 50) with the items given below. Mention the proportion of each item chosen**

| Profit | 60 | 100 | 120 |
|---|---|---|---|
| Weight | 10 | 20 | 30 |

### Step 1: Calculate Profit-to-Weight Ratio (Profit/Weight)

| Item | Profit | Weight | Profit/Weight Ratio |
|---|---|---|---|
| 1 | 60 | 10 | 6.0 |
| 2 | 100 | 20 | 5.0 |
| 3 | 120 | 30 | 4.0 |

### Step 2: Sort Items by Profit-to-Weight Ratio (Descending Order)

Sorted order:

**1 Item 1 (6.0) → 2 Item 2 (5.0) → 3 Item 3 (4.0)**

### Step 3: Fill the Knapsack Greedily

- **Take Item 1 completely** (Weight = 10, Capacity Left = 50 - 10 = 40)
- **Take Item 2 completely** (Weight = 20, Capacity Left = 40 - 20 = 20)
- **Take 20/30 of Item 3** (Weight = 20, Capacity Left = 20 - 20 = 0)

### Step 4: Calculate Proportions Chosen

| Item | Chosen Weight | Proportion |
|---|---|---|
| 1 | 10 | 100% |
| 2 | 20 | 100% |
| 3 | 20 | $\frac{20}{30} = 66.67\%$ |

### Total Profit Calculation

$$\text{Total Profit} = 60 + 100 + (120 \times \frac{20}{30})$$

$$= 60 + 100 + 80 = 240$$

==========================**Happy Learning**==========================