Date
Page No.

SET-5

(Q.1) **Definition of Complexity** - Complexity of An algorithm refers to a measure of the amount of resources it consumes when solving a problem. Resources can be time or space. Complexity helps in evaluating the efficiency of an algorithm.

Types of Complexities -

(1) Time Complexity - Measures the time taken by an algorithm to run as a function of input size n.

(2) Space Complexity - Measures the amount of memory used by an algorithm during execution.

(Q.2) Types of Recursion

Direct Recursion → function calls itself directly

Indirect Recursion → function calls another function which eventually calls the first function.

Tail Recursion → Recursive call is the last statement in the function.

Non-tail recursion → Recursive call is not the last statement; more operations follow after recursion.

(Q.3) Index Formula for a 2D array (ROW)

Rows stored one after another in memory.

Address $A[i][j] = B(A) + ((i \times n) + j) \times w$

Where

$B(A)$ = Base address

i = row index (0-based)

j = column — (——)

n = number of columns

w = size of each element in memory.

Explanation → ★ Each row has n elements
★ To reach row i, skip (i x n) elements
★ Then move j elements into the row.

Q.4) Search Techniques
Searching is the process of finding the location or presence of a specific element in a data structure like an array or list.
Types → (1) Linear Search
★ Checks each element one by one until the desired element is found.
★ Works on both sorted & unsorted arrays.
★ T.C = O(n)
(2) Binary Search
★ Works only on sorted arrays
★ Divides the array into halves and repeatedly checks the middle element.
★ T.C = O(log n)
⇨ Binary Search Algorithm.
(1) Find the middle element of the array.
(2) If middle element = target → element found.
(3) If target < middle → search in left half.
(4) If target > middle → search in right half.
(5) Repeat steps 1-4 until the element is found or the subarray size becomes 0.

(Q.5) Quick Sort Algorithm

(1) Choose a pivot element from the array
(2) Partition the array into 2 subarrays:
* Elements less than pivot.
* ——— greater than ——
(3) Recursively apply quick sort to the subarray
Combine the results.

T.C ⇒ Best / Avg case : $O(n \log n)$
Worst case : $O(n^2)$

[redacted]

Real-life Example →
Quick sort is used in database systems to sort large datasets efficiently e.g sorting a list of students by marks before generating a rank list.

(Q.6) Iterative Algo for Fibonacci series.

(1) Input n
(2) Initialize a=0, b=1
(3) if n=0, return a,
(4) if n=1, return b

(5) for i = 2 to n:
(6) fib = a+b;
(7) a = b;
(8) b = fib
(9) Return b as the nth fibonacci number.

Explanation -
Starts from the first two fibonacci numbers (0 & 1)
Iteratively calculates the next number by adding the previous two.
Avoids recursion, so it is faster & uses less memory than recursive methods.

(Q.7) Circular Linked List (CLL)

* last node points back to the first node, forming a circle.
* Unlike a singly linked list, there is no NULL at the end.

Applications -

* Efficient for repeatedly traversing the list
* Used in buffered queues and round-robin scheduling.
* Useful in systems that require cyclic traversal, like multiplayer games or operating system process scheduling.

Deletion in CLL :-

(1) Input - Head pointer of CLL, key to delete
(2) If the list is empty, return NULL
(3) If the node to delete is the only node:
→ * Set head = NULL ; return head.
(4) If deleting the head node:

→ ☆ Traverse to last node
→ ★ Set last node's next to head's next
→ ★ Update head to head's next
→ ★ Free the deleted node.

(5) If deleting other nodes
→ ★ Traverse the list to find the node with k
★ keeping track of previous node
→ ★ Set previous.next = current.next

(6) → Free the current node

(7) Return the updated head

Explanation
Maintains circular nature after deletion
Works for head node, middle node, & single-node list.

(8) (a) Recursive Approach	Iterative Approach
Function calls itself to solve a smaller problem	Uses loops to repeat operations without function calling itself
Each Recursive call uses system call stack memory	Uses constant stack space. No risk of stack overflow for loops.
Can lead to stack overflow for large inputs	Lower, only variables in current iteration are stored.
Higher memory consumption, because each call stores local variables, return address, etc.	
Easier implementation	May require more code/ extra variables to mimic recursion.
Slower for deep recursion due to call overhead	Faster, no extra call overhead.

(b) Recursive program to compute factorial

```
class Factorial {
    static int factorial (int n) {
        if (n==0 || n==1) {
            return 1;
        }
        return n * factorial(n-1); }
    public S V M (String[] args) {
        int num = 5;
        SOPL ("Factorial:" + factorial (num);
    }
}
```

(Q.9) Linked Lists for Polynomial Representation

Application

* Polynomials can have variable numbers of terms, so dynamic memory allocation is helpful.
* Linked Lists allow storing each term as a node containing :— coefficient, exponent, pointer to next node.

⇒ Advantages —

→ Flexible size, no need to allocate a fixed array.
→ Easy insertion or deletion of terms.
→ Efficient for sparse polynomials

Node Structure Example :-

```
class PolyNode {
    int coff;
    int exp;
    PolyNode next;
    PolyNode (int c, int e) {
        coff = c;
        exp = e;
        next = null;
```

Date
Page No.

Addition of Two Polynomials Using LL.

* Traverse both polynomial lists simultaneously.
* Compare exponents:
  → If exponents are equal → add coefficients
  → If 1 exponents is larger → copy that term to result.
* Continue until both lists are fully traversed.

Algorithm
Initialize result linked list as NULL
Traverse both polynomials (p1 & p2):
If p1.exp > p2.exp, add p1 terms to result, move p1 forward.
If p2.exp == p2.exp, sum coefficients, add to result, move both forward.
Append remaining terms of p1 or p2 if any.
Return the result list.

⇒ Explanation:-
* Each node represents a polynomial terms.
* Traversal ensures sorted exponents are handled correctly.
* Result is a new Linked list representing the sum.

⇒ Applications in real life :-
* Symbolic mathematics programs
* Computer algebra systems.
* Any system dealing with dynamic or sparse polynomial operations.

Date
Page No.

```
class PolynomialAdd {
  static PolyNode addPoly (PolyNode p1, PolyNode p2) {
    PolyNode result = null, tail = null;
    while (p1 != null && p2 != null) {
      PolyNode temp;
      if (p1.exp > p2.exp) {
        temp = new PolyNode (p1.coff, p1.exp);
        p1 = p1.next;
      } else if (p2.exp > p1.exp) {
        temp = new PolyNode (p2.coff, p2.exp);
        p2 = p2.next;
      } else {
        temp = new PolyNode (p1.coff + p2.coff,
                             p1.exp);
        p1 = p1.next;
        p2 = p2.next; }
      if (result == null) {
        result = tail = temp;
      } else {
        tail.next = temp;
        tail = temp;
      }}

    while (p1 != null) {
      PolyNode temp = new PolyNode (p1.coff, p1.exp);
      tail.next = temp;
      tail = temp;
      p1 = p1.next;
    } while (p2 != null) {
      PolyNode temp = new PolyNode (p2.coff, p2.exp);
      tail.next = temp;
```

```
        tail = temp;
        p2 = p2.next;
    } return result;
    }
    static void printPoly(PolyNode head){
    while(head != null){
        SOP(head.coeff + "x^" + head.exp);
    if (head.next != null)
        SOP("+");
    head = head.next;
    }  SOPL();
    }
    public static void main(String[] args){
    PolyNode p1 = new PolyNode(5, 2);
        p1.next = new PolyNode(7, 1);
    PolyNode p2 = new PolyNode(5, 1);
        p2.next = new PolyNode(5, 0);
        PolyNode sum = addPoly(p1, p2);
        printPoly(sum);
    }
}
```

OR

Case Study → Merge Sort Algorithm

1) Input : Array A[0...n-1]
2) Output : Sorted array A[0...n-1]

3) if array has 1 or 0 elements, return
4) Divide the array into two halves
   → left = A[0...mid]
   → right = A[mid+1...n-1]
5) Recursively call mergeSort(left) and mergeSort(right)

6) Merge the two sorted halves using a merge function
7) Compare elements from left and right.
8) Pick the smaller element and put it into the result array.
9) Continue until all elements are merged

⇒ Java Code

```
class MergeSort {
    static void MS (int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
    int L[] = new int [n1];
    int R[] = new int [n2];
        for (int i=0; i<n1; i++) L[i] = arr[l+i];
        for (int j=0; j<n2; j++) R[j] = arr[m+1+j];
    int i=0, j=0, k=l;
     while (i<n1 && j<n2) {
      if (L[i] <= R[j]) arr[k++]=L[i++];
      else arr[k++] = R[j++]; }
      while (i<n1) arr[k++]=L[i++];
      while (j<n2) arr[k++] = R[j++];
      }
      static void merge (int arr[], int l, int r) {
          if (l<r) {
              int m = l+(r-l)/2;
               merge (arr, l, m);
               merge (arr, m+1, r);
               merge (arr, l, m, r);
      }}
public static void main (String[] args) {
        int arr[] = {12, 11, 13, 5, 6, 7};
        merge (arr, 0, arr.length-1);
for (int num : arr) System.out.print (num+" ");
```

50

Date
Page No.

⇨ Significance in Large Dataset sorting.

- ★ Consistent O(n log n) complexity which is reliable for large data.
- ★ Stable and predictable performance.
- ★ Can be adapted for external sorting where data does not fit in memory.
- ★ Excellent for parallel processing because subarrays can be sorted independently