# Practice Questions - MTE



Date
Page No.

Set 1

1. Asymptotic notation - a mathematical way to describe the running time or space requirement of an algorithm as the input size grows very large.
It helps express the efficiency and growth rate of an algorithm independently of machine or implementation details.
Types → 1. Big O - describes upper bound - Worst case
2. Omega Ω - describes lower bound - best case
3. Theta Θ - describes tight bound - when both upper & lower bounds are the same.

2. Tail Recursion
→ If the recursive call is the last statement in the function, its called a tail recursion.
→ No computation is done after the recursion call returns.
→ It can be easily optimized by the compiler to save memory.

(WHILE)

Head Recursion
→ If the recursive call happens before any computation in the function.
→ Work is done after the recursive call returns.
→ More memory usage, can't be optimized easily.

3. Derive the index formula for accessing elements in a 2D array stored in row-major order. In RMO, elements are stored row by row. For an element A[i][j]

☆ Address(A[i][j]) = Base(A) + [(i x n) + j] x w

Base(A) ⇒ Base Address
m ⇒ number of rows
n ⇒ number of columns
w ⇒ size

| 4. Linear Search | Binary Search |
|---|---|
| Data Type → Works on unsorted or sorted data both. | → Works only on sorted data. |
| Method → Scans each element one by one. | → Repeatedly divides array into halves |
| Efficiency → Slower for large data | → Much faster. |
| T.Cs → Best Case → O(1) | → Best case → O(1) |
| → Worst case → O(n) | → Worst case → O(log n) |

5. Insertion Sort Algorithm

(i) for i = 1 —— n-1
(ii) key = A[i]
(iii) j = i-1
(iv) while j >=0 and A[j] > key
(v) A[j+1] = A[j]
(vi) j = j-1
(vii) A[j+1] = key

Working:-
Pick each element from the unsorted part & insert it at the correct position in the sorted part.

Example -
Array = [5, 2, 4, 6]

| Step | Array State |
|---|---|
| i = 1 | [2, 5, 4, 6] |
| i = 2 | [2, 4, 5, 6] |
| i = 3 | [2, 4, 5, 6] → sorted array |

Best case ⇒ O(n)
Worst case ⇒ $O(n^2)$

6. Sparse Matrix

A matrix in which most elements are zero. Storing all elements wastes memory, so special representations are used.

Representation methods

(i) Triplets (or Coordinate)
(ii) Compressed Row Storage (CRS)
(iii) Compressed Column storage (CCS)

Triplets (Coordinate) Representation

Stores as triplets :-

(rows, columns, value).

Ex-

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 2 & 0 \\ 0 & 1 & 4 \end{bmatrix}$$

3 → (0, 2, 3)
2 → (1, 1, 2)
4 → (2, 2, 4)
1 → (2, 1, 1)

Note → Saves memory.

2. The process of reversing a singly linked list with an example.
Change the direction of links so that the next of each node points to its previous node.

i/p => 10 → 20 → 30 → null
o/p => 30 → 20 → 10 → null

Steps
1. Initialize three pointers:
* previous = null
* current = head
* next = null
2. Repeat until current == null;
* next = current.next
* current.next = previous
* previous = current
* current = next
3. Finally, set head = previous

Time & Space Complexity
Time: O(n)
Space: O(1)

```
class Node {
  int data;
  Node next;
  Node (int d) { data = d; next = null; }
}
class ReverseLinkedList {
    Node reverse (Node head) {
    Node previous = null, current = head, next = null;
    while (current != null) {
    next = current.next;
    current.next = previous;
    previous = current;
    current = next;
    }
    head = previous;
    return head;
}
```

(8)(a) Tower of Hanoi problem (recursive)

TOH(n, Src, Aux, Desti)

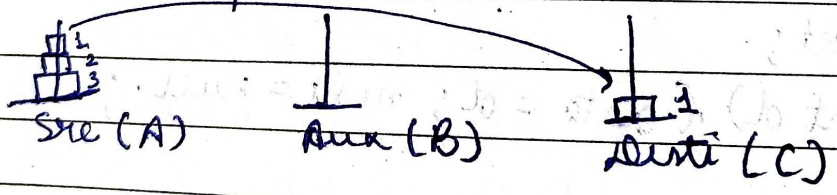
1. if n==1
2. return "Move disk 1 from Src to Desti"
3. else
4. TOH(n-1, Src, Desti, Aux)
5. print "Move disk n from Src to Desti"
6. TOH(n-1, Aux, Src, Desti)

Conditions

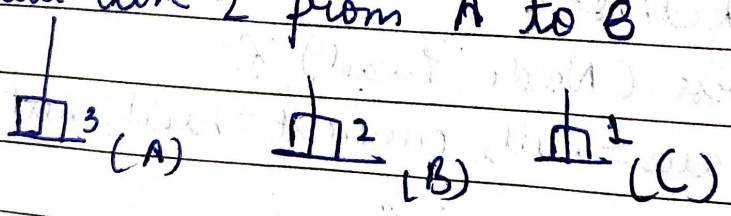
Move n disks from Src to Desti using Aux — 1. Only 1 disk can move at a time.
2. A larger disk can't be placed on a smaller disk.

Ex n=3

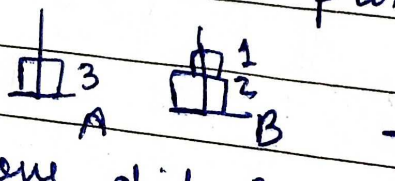
1. Move disk 1 from A to C



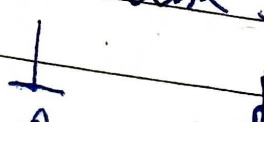

2. Move disk 2 from A to B




3. Move disk 1 from C to B

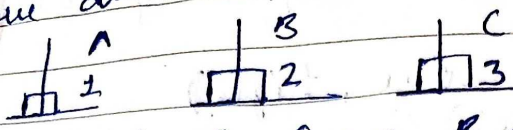



4. Move disk 3 from A to C

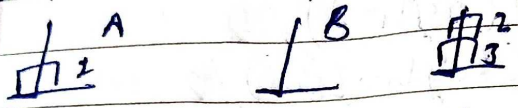


Date
Page No

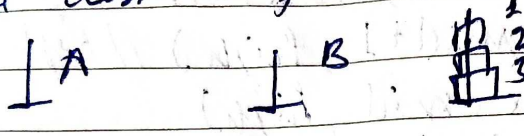
5. Move disk 1 from B to A




6. Move disk 2 from B to C




7. Move disk 1 from A to C




Total moves = $2^n - 1 = 7$

T.C = $O(2^n)$

S.C = $O(n)$

(b)

| Recursion | Iteration |
|---|---|
| ① Function calls itself | ① Uses loops like (for / while) |
| ② High memory usage | ② Low |
| ③ Slower (due to overhead) | ③ Faster |
| ④ Easier for divide & conquer | ④ Easier for simple repetition |
| Ex → Tower of Hanoi | Ex → Loop - based summati |

9. Case Study – Sort an array using merge

```
MergeSort (arr, low, high)
    if (low < high)
        mid = low + (high - low) / 2
    MergeSort (arr, low, mid) // left
    MergeSort (arr, mid+1, high) // Right
    Merge ← midMS (arr, low, mid, high)
```

☆ Take two pointer i & j such that
i point at low at first i = 0
j point at mid + 1 at first j = mid + 1

☆ Also make a temporary array with a pointer k = 0

```
while (i <= mid && j <= high) {
    if (arr[i] < arr[j]) {
        temp[k] = arr[i];
        k++, i++
    else (temp[k] = arr[j]
        k++, j++
}
while (i <= mid) {
    temp[k++] = arr[i++];
}
while (j <= high) {
    temp[k++] = arr[j++]
}
```

putting into original array.

```
for (int i=0; k=0; i<=high; i++, k++)
    arr[i] = temp[k];
}
```

Time Complexity for all cases = O(n log n)
Space Complexity → O(n)

Advantages over Bubble Sort

| Merge Sort | Bubble Sort |
|---|---|
| ★ T.C ⇒ O(n log n) | ★ T.C ⇒ O(n²) |
| ★ More efficient for larger data sets | ★ Slower for large data set |
| → Stable | → Stable |
| → Divide & conquer | → Simple comparison-based |

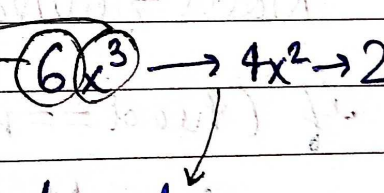
OR

Application of Doubly Linked Lists in Polynomial Representation.

A DLL can efficiently represent a polynomial like $6x^3 + 4x^2 + 2x + 1$

Each node stores

★ Coefficient
★ Exponent
★ Pointer to previous & next nodes




⇨ Algo for Insertion (Sorted by Power)

1) Create a new node with coefficient & exponent
2) If the list is empty make it head.
3) Traverse the list to find correct position
4) Insert the node before or after accordingly
5) Adjust the next & prev pointers.

Algo for Deletion (By Power)

1) Traverse the list until the node with given exponent is found.
2) If found:
   (a) Adjust pointers of prev & next
   (b) Delete the target node
3) If not found, print "Not found"

```
Class Node {
int coef, pow;
Node next, prev;
Node (int c, int p {
  coeff = c;
  pow = p;
}
class Polynomial {
  Node head;
void insert (int c, int p) {
Node newNode = new Node (c, p);

if (head == null || head.pow < p) {
    newNode.next = head;
if (head != null) head.prev = newNode;
head = newNode;
    return;
}

Node temp = head;
while (temp.next != null && temp.next.po
  temp = temp.next;
newNode.next = temp.next;
if (temp.next != null) temp.next.prev = null
    temp.next = newNode;
}   newNode.prev = temp;
```

```
void delete (int p){
  Node temp = head;
while (temp != null && temp.power != p)
    temp = temp.next;
if (temp == null) return;
if (temp.prev != null){temp.prev.next =
                        temp.next;}
else head = temp.next;
if (temp.next != null) { temp.next.prev =
                          temp.prev;
                        }

void display (){
  Node temp = head;
  while (temp != null){
    SOPL ( temp.coeff + "x^" + temp.power);
    temp = temp.next;
if (temp != null){
  SOPL (" + ");
    }
  }

public class Main{
  PSVM (String [] args){
Polynomial poly = new Polynomial();
  poly.insert (6,3);
  _____ (4,2);
  _____ (2,1);
SOP ("Polynomial : ");
  poly.display();
poly delete (2);
  SOPL (" After deletion");
poly.display();
  }
}
```