# Set - 1

Q1. Define asymptotic notation & explain its importance in analyzing algorithm efficiency.

=> Asymptotic notation is a mathematical tool used to describe the efficiency of an algorithm by focusing on how its requirement time or space changes as the input size ($n$) grows very large (approaches infinity).

Asymptotic notation is crucial coz it provides a standard, machine-independent way to compare algorithms & predict their behavior, especially with large datasets.

1) Focuses on Scaling (Growth Rate):
   e.g. $O(n^2)$ is much slower than $O(n \log n)$

2) Predicts Performance (Scalability): By using notation like $O(n)$, $O(n^2)$, we can confidently predict which algo will scale better.

3) Define Bounds (Worst/Best Case): The three primary notations give a formal description of an algo performance limits:

   - Big O ($O$) : The Upper Bound (Worst Case).
   - Omega ($\Omega$) : The lower Bound (Best Case)
   - Theta ($\Theta$) : The tight Bound (Avg-Case)

Q.2. Tail Recursion

→ If the recursive call is the last statement in the fun. its called a tail recursion.
→ No computation is alone after the recursion call returns.
→ It can be easily optimized by the compiler to save memory.

Head Recursion

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

Q.3. Derive the idx formula for accessing ele. in a 2D array stored in row-major order.

⇒ In RMO, elements are stored row by row. For an element A[i][j].

$$\text{Address}(A[i][j]) = \text{Base}(A) + [(i * n) + j] * sz.$$

Base address. | no. of rows. | size.

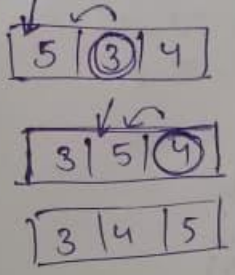
Q.4.

| Linear Search | Binary Search |
|---|---|
| → Works on sorted or unsorted data both. | → Works only on sorted data. |
| → Scans / traverse each ele. one by one. | → Repeatedly divides array into halves. |
| → slower for large data | → much faster |
| → T.C. → O(n) | → T.C. → O(log n) |

Q.5 Insertion Sort

Algorithm

```
for (i = 1 to n-1)
    curr : arr[i].
    j : i-1
    while (j >= 0 && arr[j] > curr) {
        arr[j+1] = arr[j]
        j--;
    }
    arr[j+1] = curr;
```




Working

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

T.C. → $O(n^2)$

if sorted arr → $O(n)$.

Q.6. Sparse Matrix

A matrix in which most elements are zero. Storing all elements is waster of memory. So special representations are used.

Representation method.

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

Triplets (coordinate) Representation

Stores as triplets i
(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)

Saves memory

Q.7. 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.

steps

i) Initialize three pointers:
* previous = null
* current = head
* next = null

input :- 10 → 20 → 30 → null

Output :- 30 → 20 → 10 → null.

ii) Repeat until curr == null;
* next = curr->next.
* curr->next = prev.
* prev = curr
* curr = next.

iii) Finally set head = prev.

// code.

```
void reverse (Node* head) {
    Node* curr = head;
    Node* prev = NULL;
    tail = &head;
    while (curr != NULL) {
        Node* next = curr->next;
        curr->next = prev;

        prev = curr
        curr = next;
    }
    head = prev;
}
```

Q.8: a) Tower of Hanoi problem (recursion).

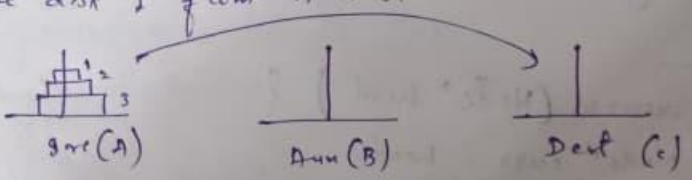
```
TOH (n, src, Aux, Dest)
if (n == 1)
    return "Move disk 1 from src to Dest"
else {
    TOH (n-1, src, Dest, Aux)
    print "Move disk n from src to Dest"
    TOH (n-1, Aux, src, Dest)
}
```

Conditions

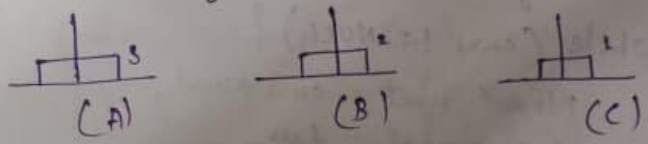
Move n disks from Src to Dest using Aux - 1 Only 1 disk can move at a time & A larger disk can't be placed on a smaller disk.

Ex. $n = 3$

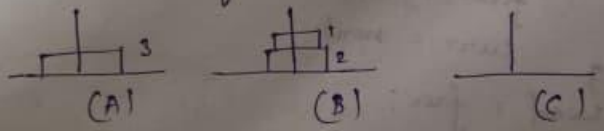
(1) Move disk 1 from A to C.



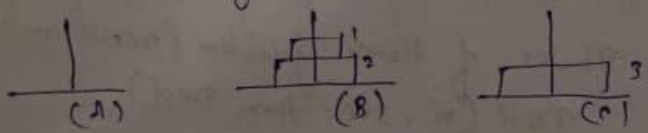

(2) Move disk 2 from A to B.



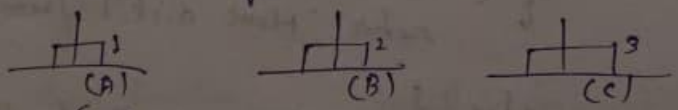

(3) Move disk 1 from C to B.



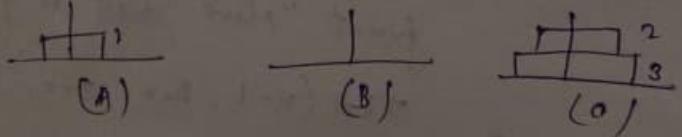

(4) Move disk 3 from A to C.



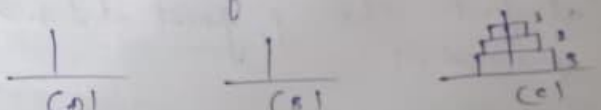

(5) Move disk 1 from B to A




(6) Move disk 2 from B to C.



④ Move disk 3 from A to C




Total moves = $2^n - 1$

$2^3 - 1 = 8 - 1 = 7$

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. |
| ⑤ Ex :- Tower of Hanoi | ⑤ Loop - based s |

Q.9

Case Study - Sort an array using merge sort.

```
Merge Sort (arr, low, high)
    if (low < high)
        mid = low + (high - low)/2
        Merge Sort (arr, low, mid)    // left
        Merge Sort (arr, mid+1, high) // right.
        merge (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 arr 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++]
}
for (int i=0, k=0; i<=high; i++, k++)
    arr[i] = temp[k];
}
```

T.C. → O(n log n)

S.C → O(n)

Advantages over Bubble sort

| Merge Sort | Bubble Sort |
| --- | --- |
| → T.C. → O(nlogn) | → T.C. → O(n²) |
| → more efficient for larger data set. | → slower for large data set |
| → Stable | → stable |
| → Divide & Conquer | → simple comparision-based. |

OR

⇒ Application of Doubly L.L. in Polynomial Representation.

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

Each Node stores.

① Coefficient
② Exponent
③ pointer to prev & next nodes.

Algo → Insertion

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

Algo → Deletion

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".

//code

```
class Node {
    int coef, pow;
    Node* next, prev;
    Node (int c, int p) {
        coef = c;
        pow = p;
    }
};

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;
    }
}
```

```
Node* temp = head;
while (temp->next != NULL && temp->pow > pow)
    temp = temp->next;

void insertion(Node* &head, int coef, int pow){
  Node* newNode = new Node(coef, pow, NULL, NULL);
  if (!head){
     head = newNode;
     return;
  }
  Node* temp = head;
  while (temp->next && temp->pow > pow)
      temp = temp->next;

  if (temp->pow < pow)
      newNode->next = temp;
      newNode->prev = temp->prev;
      if (temp->prev) temp->prev->next = newNode;
      temp->prev = newNode;
      if (temp == head) head = newNode;
  } else {
      newNode->next = temp->next;
      newNode->prev = temp;
      if (temp->next) temp->next->prev = newNode;
      temp->next = newNode;
  }
}
```

Deletion

```
void delete(Node* &head, int pow)
    Node* temp = head;
    while (temp && temp->pow != pow)
        temp = temp->next;
    if (!temp) return;
    if (temp->prev) temp->prev->next = temp->next;
    else head = temp->next;
    if (temp->next) temp->next->prev = temp->prev;
    delete temp;
}
```