Date __/__/____

Set - 1

1. Define asymptotic notation and explain its importance in analyzing algorithm efficiency. 3 M

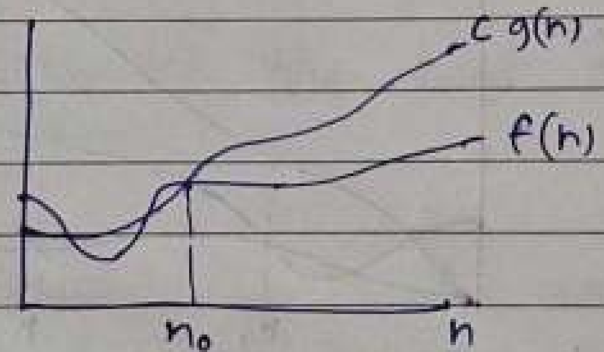
⇒ :: Asymptotic notations :-

'O' - notation (Asymp. upper bound)

⇒ A function $f(n)$ is said to big $O(g(n))$ iff there exists constant c & a constant n_0 s.t:

$$0 \le f(n) \le c\,g(n) \quad \forall\ n \ge n_0$$

zero

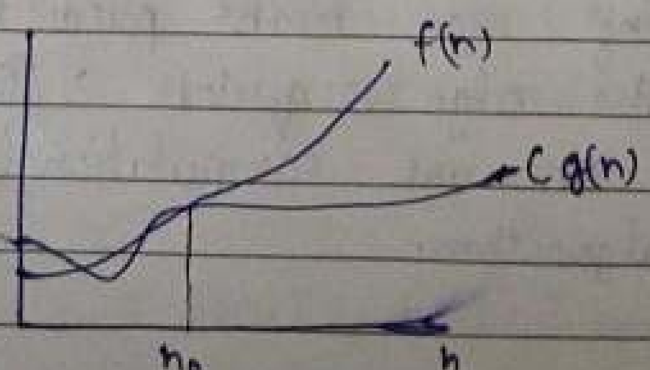


$$f(n) = O(g(n))$$

'Ω' - notation (Asymp. Lower bound)

⇒ A function $f(n)$ is said to big $\Omega(g(n))$ iff there exists constants c and n_0 s.t:

$$0 \le c\,g(n) \le f(n) \quad \forall\ n \ge n_0$$



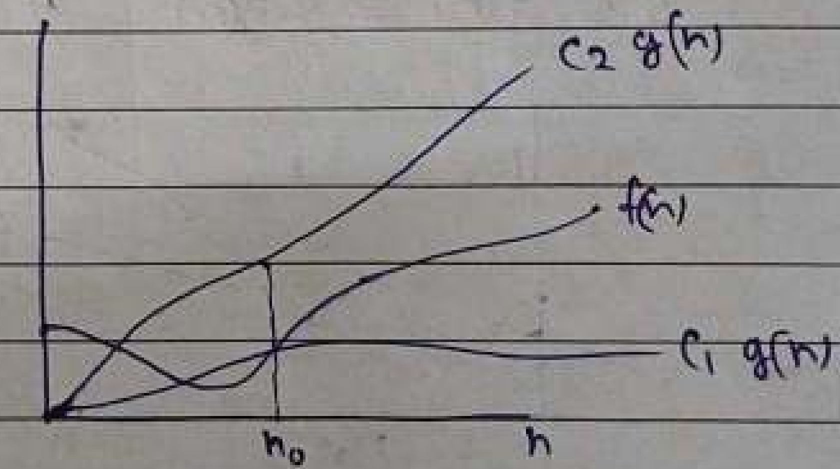
$$f(n) = \Omega(g(n))$$

'θ' notation (Asymp. tight bound)

⇒ A function f(n) is said to be $\theta(g(n))$ iff there exists constant c_1, c_2 & a constant n_0 s.t:

$$0 \le f(n) \le c_2\, g(n)$$
$$0 \le c_1\, g(n) \le f(n)$$

$$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \forall\, n \ge n_0$$



$$f(n) = \theta(g(n))$$

⇒ $f(n) = O(g(n))$ Worst case

⇒ $f(n) = \Omega(g(n))$ Best case

⇒ $f(n) = \theta(g(n))$ Avg case.

Its importance in Analyzing Algo efficiency.

⇒ Analyzing algo means predicting the resources that the algo requires. The primary concern is to measure computational time for an algorithm.

Date ___/___/______

→ The worst case: It gaurantees that the algorithm will never slower than this.

→ The best case: It is an Ideal case. It gaurantees that the algorithm will never faster than this.

→ The avg case: It shows the typical performance of algorithm : "not the best not worst" but what usually happens.

2. Explain the difference b/w head and tail recursion with example: 3M

⇒ Head Recursion: In head recursion, the recursive call happens first - "before any other statements in the function".

⇒ That means the function calls itself before doing any work.

e.g.

```
void headRec (int n) {
    if (n == 0) {
        return; }

    headRec (n-1);
    cout << n << " ";
}

// headRec ( 5); will give
// 1 2 3 4 5
```

Date __/__/____

Tail Recursion: In tail recursion, the recursive call is the last statement in a function.

e.g.

```
void tailRec (int n) {
    if(n == 0){
        return;
    }

    cout << n << " ";
    tailRec (n-1);
}

// tailRec (5) will print
// 5 4 3 2 1
```

3. Derive the index formula for accessing elements in a 2-D array sorted in row major order. 4 M

⇒ Suppose we have a 2-D array,

A [m] [n] And we want find A[i][j] ;

where

m = no of rows

n = no of cols.

⇒ In memory elems are stored like

A [0] [1] , A[0][2] , A[0][3] ---- A[0] [n-1] , A[1] [0] ,→

↳ A[1] [1] - - - A[1][n-1] , A[2] [0] ---- A[2][n-1]

Date __/__/____

→ Count elems before A[i][j]

→ Each row has m elems.
→ Rows before row i = i rows
→ So we have to skip n×i elems.
→ Then with in row we have to move j columns.

So total elems before A[i][j]

$$= i*m+j$$

→ let each elms occupies w bytes, so

$$\boxed{\text{Address of } A[i][j] = B + w*(i*m+j);}$$

where B = Base address
w = element size (e.g int 4 bytes)
i, j = Actual row and column whose address is to be found.
m = number of elems in a row or no of column in matrix.

if start column and row is not from zero then

$$\boxed{\text{Add } A[i][j] = B + w*(m*(i-LR)+(j-LC))}$$

where LR an LC are starting index.

4. Explain the difference b/w linear search and binary search with their time complexities. 4M

→ Linear Search: In Linear search, we check each element one by one until the target found or reched at the end.

Algorithm:
⇒ Start from first element
⇒ compare each element with the key.
⇒ if found, return index or value.
⇒ if not found till end - element not present.

code

```
for (inte i =0; i<n; i++){
        if (arr[i] = = key){
            return i;
        }
    }
return -1;
```

TC: i) Best case O(1) // element (Target) present at
// starting

ii) Worst case O(n) // element present at
// last index bec then
// we have itr loop n times.

iii) Avg case O(n/2) ≈ O(n)

$\boxed{\text{In L.S no need Sorting}}$

Date ___/___/______

Binary Search: In binary search array must be sorted. We repeatedly divide the search range in half until the target is found or range becomes empty.

Algorith:
⇒ Find the middle element
⇒ if middle == target return middle.
⇒ if middle < target search in right half.
⇒ if middle > target search in left half.
⇒ Repeat until target found or range becomes empty.

code:
```
int BS(int arr[], int n, int t){

    int first = 0;  int last = n-1;
      while (first <= last) {
     int mid = first + (last-first)/2;

     if(arr[mid] == t){ return mid; }

     else if (arr[mid] < t){
          first = mid+1;
       }
    else {
        last = mid-1;
      }
   }
        return -1;

}
```

Date ___/___/______

TC ⇒ i) $O(1)$ Best case // ~~elems~~ target
// present at mid

⇒ 1st iteration n/2 elms remain
2nd " n/4 " "
3rd " n/8 " "
:
until only 1 elems remain.
let k = no of iteration.

So $\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = \log_2 2^k$

$\log n = k \log_2 2$

$k = \log n$ times.

So

for worst and avg case

$$\boxed{TC = O(\log_2 n)}$$

$\boxed{\text{Space complexity for both is } O(1)}$

↓

L.S and B.S

5. Write the algorithm and explain working of insertion sort with example! 5 marks

let $A[\,] = \{3, 1, 2, 5, 4\}$

j i

3	1	2	5	4

0 1 2 3 4

1	3	2	5	4

1	2	3	5	4

1	2	3	5	4

1	2	3	4	5

sorted

Algorithm!

⇒ Start from second element.

⇒ Compare with elements before it.

⇒ Shift all elements that are greater than the key, one position to right.

⇒ Insert the key at its correct position.

⇒ Repeat all elems from 1 to n-1.

Pseudo code!

```
Insertion_sort (A, n)
1.   for i = 1 to n-1
         key = A[i]
         j = i-1)
while    A[j] > key and j>=0
         A[j+1] = A[j]
           j = j-1 ;
       A [j+1] = key
```

Date ___/___/___

⇒ Inseration sort basically works on shifting it compares an elem with previous or left elements and those elems are greter than this, shift them right by 1 position. Repeates this till n-1.

```
void inserationSort (int arr[], int n) {
    for(int i=1; i<n; i++) {
        int key= arr[i];
        int j= i-1;

        while ( j>=0 && arr[j]> key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr [j+1] = key;
    }
}
```

Dry Run!

arr = [5, 3, 4, 1, 2]

1. i = 1 :

j = 0   arr[j] = 5 greter than key = 3

→ shift 5 right

[5 5 4 1 2]   and j = -1

arr [$\underset{0}{j+1}$] = key // 3

→ [3 5 4 1 2]

(2) i = 2 :
j = 1 arr[j] = 5 which is grater than key = 4
shift 5 right
[3 5 5 1 2] and j = 0
arr[j+1] = key
1 4

3 4 5 1 2

in while loop j = 0 but arr[j] is $\not>$ key = 4,
3

(3) i = 3
key = 1 j = 2 arr[j] = 5 which is > key
shift 5 right
[3 4 5 5 2] j = 1
So arr[j+1] = key
2 1

[3 4 1 5 2]

now again key is < arr[j] becz arr[j] = 4
So it will again shift 4 at index 2 then key
putted at index 1.

Similarily repeated till i = 4.

TC : i) $O(n)$ Best case if array already sorted.
only one comparision per elem.

ii) $O(n^2)$ Worst case if array in reverse order shift all elems.

iii) $O(n^2)$ Avg case if random array.

$$\boxed{\begin{aligned} TC &= O(n^2) \\ SC &= O(1) \end{aligned}}$$

Date __/__/____

6. Discuss how sparse matrices are represented and explain any one representation mtd. 5M

⇒ If most of the elements of a matrix are zero then it is called sparse matrix.
⇒ If we store all elements including zero then it is memory inefficient.
⇒ So instead of storing all elements, we store only non-zero values and their positions.

⇒ Common Representation method
1) Array representation.
2. Linked List representation.

Array Representation method.

e.g.

$$\begin{bmatrix} 0 & 0 & 1 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	Column	Value
0	2	1
1	0	2

$$\Rightarrow \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 2 \end{bmatrix}$$

Algorithm !
⇒ Start function
⇒ initialize a 2-D vector x ~~column~~ row and 3 column // x is number of non-zero elems
// which can be counted at the
// matrix input time.

Date __/__/____

⇒ initialize k=0 // counter for sparse matrix rows.

⇒ for i=0 to m-1;

⇒ for j=0 to n-1; // two nested loops.

⇒ ~~et~~ In loops check if matrix elem is non zero then assign it in resultant matrix.

1. res [k] [0] = i
2. res [k] [1] = j
3. res [k] [2] = mat [i] [j]

⇒ then k+1 for next row.

Code.

```
vector<vector<int>> spM(vector<vector<int>>&mat,
                                              int x){
        // x is no of nonzero elems.
    int m = mat.size();
    int n = mat[0].size();
    vector<vector<int>> res(x, vector<int>(3));

    int k=0;
      for(int i=0; i<m; i++){
          for(int j=0; j<n; j++){

              if(mat[i][j] != 0){

                  res[k][0] = i;
                  res[k][1] = j;
                  res[k][2] = mat[i][j];
          k++; } }
    }   return res;
    }
```

Date ___/___/______

7. Explain the process of reversing a singly linked list with an example. 8 M

⇒ A singly list is a collection of nodes where each node contains:

i) data → the value

ii) next → address of next node.

⇒ Reversing a linked list means changing the direction of links so the first node becomes the last and the last node becomes first.

Algorithm:

⇒ Input Node Head.

⇒ Create three pointers.

prev = null

curr = pointing to head

next = null // used to temporary store next node

⇒ Repeat until curr becomes null.

→ Save next node, next = curr → next

→ Reverse the link, curr → next = prev

→ Move forward →

prev = curr

curr = next

⇒ After prev will point to the new head.

⇒ Set head = prev.

⇒ End.

Code

```
Nod* reverseList (Node* Head) {

    Node* prev = NULL;
    Node* curr = Head;
    Node* next = NULL;

  while(curr != NULL){

      next = curr->next;
      curr -> next = prev;
      prev = curr;
      curr = next;
    }
    Head = prev;
  return Head;

}
```

Dry Run! let Lis ⇒ 10 → 20 → 30 → Null
100 200 300

Step 1

curr = 10 , prev = Null
curr != Null ✓
~~next =~~ reverse link → 10 → Null

Step 2

Curr = 20 prev = 10
curr != Null
reverse link 20 → 10

Step3. curr = 30 prev = 20
curr != null
reverse link 30 → 20 → 10
head = 30;

Date __/__/__

8.(a) Write a recursive algorithm for the Tower of Hanoi and explain 4M
[Set - 4 Q. 8(a)]

8(b) Compare the trade-offs between recursion and iteration. 4M

⇒ Recursion: A function calls itself until base condition is notmet.

⇒ Iteration: A loop repeatedly executes a block of code until a condition becomes false.

Aspect	Recursion	Iteration
Concept	function calls itself repeatedly	Uses loops for, while, do-while.
Termination	Base case	Loop condition
Memory	High (uses call stack for each call)	Low (uses single loop variables)
Speed	Slower due to funtion call overhead	Faster because no function calls.
Complexity of code	Shorter, cleaner code for complex problem	Longer code for complex logic
Risk	Stack overflow if base case missing	infinite loop if condition missing
Usefull	for problems naturally recursive (trees, fact)	suitable for simple repetitive calls.
Example:	Tower of Hanoi, factorial, fibbonacci	sum of numbers, printing list

9. Given an array of integers, design an efficient method using merge sort to sort it. Explain its time complexity and benifits over bubble sort. 10M

Merge sort algo, pseudocode

Set 4 Q no 5,

<u>code!</u>

```
void mergeSort(int arr[], int left, int right) {

    if (left < right) {
        int mid = (left + right)/2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid+1, right);
    merge (arr, left, mid, right);
    }
}

void merge(int arr[], int left, int mid, int right) {

    vector<int> L(arr+left, arr+mid+1);
    vector<int> R(arr+mid+1, arr+right+1);
    int i=0; int j=0; int k=left;

    While(i<L.size() and j<R.size()) {
        if(L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];

    }
```

```
while( i <L.size()) {
    arr[k++] = L [i++];
}
while (j < R.size()) {
    arr[k++] = R[ j++];
}
```

T.C for Best, Avg, worst = O(n log n)
S.C = O(n)

Benifits over bubble sort and time complexity.

Aspect	Merge Sort	Bubble sort
Approach	Divide and conquer	Repeated swaping
T.C	$O(n \log n)$	$O(n^2)$
Efficiency	very efficient for large databases	Very slow for large D.B.
Stable	Yes	Yes
Recusion	Yes	No
Applications	Large data storing.	Simple & small D.B.

9 OR Explain the applications of doubly linked list in polynomial representation. Demonstrate insertion and deletion operation. 10M.

⇒ Application of doubly LL in polynomial rep. :

⇒ A polynomial is a expression like:

$$P(x) = 5x^3 + 4x^2 - 2x + 7$$

Each term has two parts.

→ coefficient

→ Exponent.

⇒ To efficiently represent, modify, or compute such polynomials in memory, we use a Doubly linked List.

⇒ A DLL allows bidirectional traversal

⇒ Each node stores:

→ coefficient (coef)

→ Exponent (pow)

→ Pointer to next and previous nodes (nex, prev)

Structure of Node

```
Struct Node {
        int coef;
        int pow;
        Node* next;
        Node* prev;
};
```

Date __/__/____

⇒ Insertion operation!

→ Algo!

1. Create new node with given coef and pow.
2. if the list is empty → make new node as head.
3. else traverse to find the correct position based on exponent.
4. Insert new node!
   → Before a node with a smaller exponent.
   → Or at the end if exponent is smallest.

Code

```
void insertTerm(Node*& head, int coef, int pow) {

Node* newNode = new Node{coef, pow, NULL, NULL};

    if(!head || pow > head->pow) {
        newNode -> next = head;
    if (head) head -> prev = newNode;
        head = newNode;
        return;
    }

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

  newNode -> next = temp -> next;
   if(temp -> next) temp -> next-> prev = newNode;
      temp -> next = newNode;
      NewNode -> prev = temp;

}
```

⇒ Deletion operation!

→ Algo!

1. Search for the node with matching exponent.
2. Adjust prev and next pointers to skip the node.
3. Delete the node from memory.

Code

```
void del(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 = tem->prev;

    delete temp;

}
```