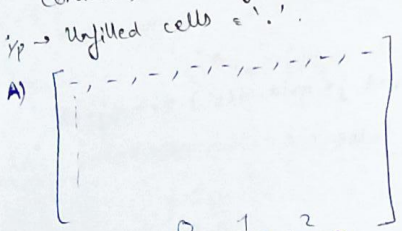
BACKTRACKING-2

Q) Consider a 9*9 2D array grid that is partially filled with numbers from 1 to 9. The Sudoku problem is to fill remaining blocks with no.s from 1 to 9 so that every row, column & subgrid(3*3) contains exactly one instance of digits (1 to 9)

i/p → unfilled cells = '.'.

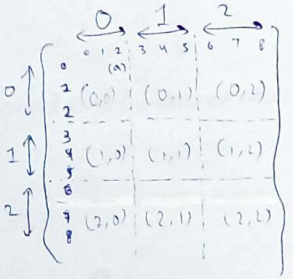
A)



Here also we apply similar approach for 'n' queens'.

We go through each column & write no.s 1-9, then move to next row.

Now:



Now if we multiply (i*3, j*3), then we get the topmost left ele. of our subgrid

Now if we want to check 3x3 wise, lets divide 3 rows/columns into single!

$\left(\frac{i}{3}\right)\times 3 \quad , \quad \left(\frac{j}{3}\right)\times 3$

(0) → 0,0 → grid

intersection of both gives us rig. grid in which ele. is present

Code:

```
bool canWePlace(vector<vector<char>>& board, int row, int col, char num){
    for(int j=0; j<9; j++){
        if(board[row][j]==num) return false;
    }
    for(int i=0; i<9; i++){
        if(board[i][col]==num) return false;
    }
    int r = (row/3)*3;
    int c = (col/3)*3;
    for(int i=r; i<(r+3); i++){
        for(int j=c; j<(c+3); j++){
            if(board[i][j]==num) return false;
        }
    }
    return true;
}
```

```
bool sudoku(vector<vector<char>>& board, int row, int col){
    if(col==9) return sudoku(board, row+1, 0);
    if(row==9) return true;
    if(board[row][col]=='.'){
        for(int num=1; num<=9; num++){
            if(canWePlace(board, row, col, '0'+num)){
                board[row][col] = '0'+num;   converting to ASCII form
                bool res = sudoku(board, row, col+1);
                if(res) return true;
                board[row][col] = '.';
            }
        }
        return false;
    }
    else
        return sudoku(board, row, col+1);
}

void printBoard(vector<vector<char>>& board){
    for(auto row: board){
        for(auto col: row){
            cout<<col<<" ";
        } cout<<endl;
    }
}

int main(){
    vector<vector<char>> board;
    board = {{'.','.','.','.','2','6','.','7','.','1'},
             {'6','8','.','.','.','7','.','.','9','.'},
             {'1','9','.','.','.','.','4','5','.','.'},
             {'8','2','.','1','.','.','.','.','4','.'},
             {'.','.','4','6','.','2','9','.','.','.'},
             {'.','5','.','.','.','3','.','2','8','.'},
             {'.','.','9','3','.','.','.','7','4','.'},
             {'.','4','.','.','.','5','.','.','.','6'},
             {'7','.','3','.','1','8','.','.','.','.'}};
    cout<<"Original Sudoku Puzzle:"<<endl;
    printBoard(board);
    if(sudoku(board, 0, 0)){
        cout<<"\nSolved Sudoku Puzzle"<<endl;
        printboard(board);
    }else cout<<"\nNo Solution exists"<<endl;
    return 0;
}
```

o/p:

```
Original Sudoku Puzzle:
. . 2 6 . 7 . 1
6 8 . . 7 . . 9 .
1 9 . . . 4 5 . .
8 2 . 1 . . . 4 .
. . 4 6 . 2 9 . .
. 5 . . . 3 . 2 8
. . 9 3 . . . 7 4
. 4 . . 5 . . 3 6
7 . 3 . 1 8 . . .

Solved Sudoku Puzzle:
4 3 5 2 6 9 7 8 1
6 8 2 5 7 1 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 1 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 2 8
5 1 9 3 2 6 8 7 4
2 4 8 9 5 7 1 3 6
7 6 3 4 1 8 2 5 9
```

Q) Given integers M, N & K, the task is to place K knights on an M*N chessboard such that they don't attack each other. The knights are expected to be placed on different squares on the board. A knight can move 2 squares vertically & one square horizontally or two squares horizontally & one square vertically. The knights attack each other if one of them can reach the other in single move. There are multiple ways of placing K knights on an M*N board or sometimes, no way of placing them, we are expected to list out all the possible solutions.

ex: i/p: M=3, N=3, K=5

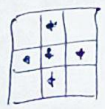
o/p: KA KA KAKA KA KA K K KA KA , total no. of solns: 2

A) (Similar to nqueen) Knight ~ (i-2,j-1) (i-2,j+1) (i-1,j-2) (i,j) (i+1,j+2) we will go top→bottom, so need to check if attack coming from bottom

, we check each cell → if attacked ↓ revert back

if not attacked ↳ put it

ex: K=5, m=3, n=3 ↳



,

attack will always come from above as we are going (top→bottom)

Code:

```
bool canWePlace(vector<vector<char>>& grid, int i, int j){
    if(i-1>=0 and j-2>=0 and grid[i-1][j-2]=='K') return false;
    if(i-2>=0 and j-1>=0 and grid[i-2][j-1]=='K') return false;
    if(i-1>=0 and j+2>=0 and grid[i-1][j+2]=='K') return false;
    if(i-2>=0 and j+1>=0 and grid[i-2][j+1]=='K') return false;
    return true;
}
void knights(int m, int n, vector<vector<char>>&grid, int k, int i, int j){
    if(k==0){
        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                cout<<grid[i][j];
            } cout<<endl;
        } cout<<"****"<<endl;
        return;
    }
    if(j==n) return knights(m,n,grid,k,i+1,0);
    if(i==m) return;
    for(int row=i; row<m; row++){
        for(int col=(row==i?j:0); col<n; col++){
            if(canWePlace(grid,row,col)){
                grid[row][col]='K';
                knights(m,n,grid,k-1,row,col+2);
                grid[row][col]='_';
            }
        }
    }
}
int main(){
    int m=3, n=3, k=5;
    vector<vector<char>> grid(n, vector<char>(m,'_'));
    knights(m,n,grid,k,0,0);
    return 0;
}
```

if we are on the same row, then 'j', if we moved to next row, start from '0'

↳ op:

```
K _ K
_ K _
K _ K
****
_ K _
K K K
_ K _
****
```

Q) Consider an N*N chessboard. The Knight's Tour problem is to print order when the Knight visits that block of the chessboard. Initially, the Knight will be placed at the first block of the chessboard. The rule is that the Knights visits each block exactly once.

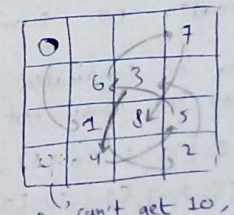
ex: N=8; op:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

↳ suppose knight starts at topleft, we need to see that → Can it move in board such a way that it can cover complete board?

A) (i-2,j-1) (i-2,j+1) (i-1,j-2) (i-1,j+2) (i,j) (i+1,j-2) (i+1,j+2) (i+2,j-1) (i+2,j+1)

, ex:



can't get 10, so we revert to 8, check another possibility.

if all possibilities failed, then we revert back to 7 & place 8 at other possibilities

like that

Code:

```
bool f(vector<vector<int>> &grid, int i, int j, int n, int count){
    if(i<0 or j<0 or i>=n or j>=n or grid[i][j]>=0) return false;   // visited cell
    grid[i][j] = count;
    if(count == n*n-1){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                cout<<grid[i][j]<<"\t";
            } cout<<endl;
        } return true;
    }
    if(f(grid, i-1, j-2, n, count+1)) return true;
    if(f(grid, i-1, j+2, n, count+1)) return true;
    if(f(grid, i-2, j+1, n, count+1)) return true;
    if(f(grid, i-2, j-1, n, count+1)) return true;
    if(f(grid, i+1, j+2, n, count+1)) return true;
    if(f(grid, i+1, j-2, n, count+1)) return true;
    if(f(grid, i+2, j-1, n, count+1)) return true;
    if(f(grid, i+2, j+1, n, count+1)) return true;
    grid[i][j] = -1;
    return false;
}
int main(){
    int n=5;
    vector<vector<int>> grid(n, vector<int>(n,-1));
    f(grid,0,0,n,0);
    return 0;
}
```

op:

0	17	10	5	2
9	4	1	16	11
18	21	12	3	6
13	8	23	20	15
22	19	14	7	24

Q) Given collection of candidate no.'s (candidates) & a target no., Find all unique combinations in candidates where the candidates no.'s sum to target.

a) Each no. in candidates may only be used once in combination
→ Solution set must not contain duplicate combinations.

ex: ip1: c = [10,1,2,7,6,1,5], target = 8
op: [[1,1,6],[1,2,5],[1,7],[2,6]]

ip2: c = [2,5,2,1,2], target = 5
op: [[1,2,2],[5]]

Q) Same no. from candidates may be chosen unlimited no. of times. 2 combo.'s are unique if frequency of atleast one of the chosen no.'s is different. → (Here array has distinct integers)

ip1: c = [2,3,5], target = 8
op: [[2,2,2,2],[2,3,3],[3,5]]

ip2: c = [2], target = 1
op: []

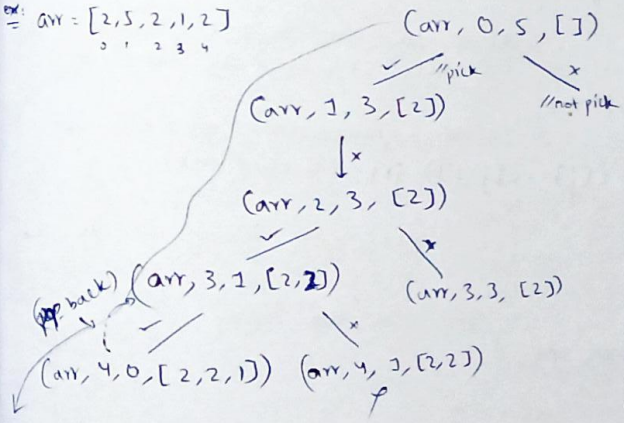
ip3: c = [2,3,6,7], target = 7
op: [[2,2,3],[7]]
→ ('2' can be used multiple times)

A) a) Basically we need all subsets of ele., so ex: [1,2,2,2,5]
[1,2,2] → can't use these(2,2) → so we remove all same ele. once a subset is formed

If any ele. > target → we exclude all subsets made by that ele.

f(arr, idx, t, v) = { f(arr, idx+1, t, v) → if ele. excluded ; v.push_back, f(arr, idx+1, t-arr[idx], v) → if ele. included. }
(subset that sums to target)

ex: arr = [2,5,2,1,2]
0 1 2 3 4



arr → (sort) [1,2,2,2,5]
2 2*
(explored)
Once we have explored 2*, then we discard all upcoming 2's as we will get repeated subsets.

Code:

```
vector<vector<int>> result;
void f(vector<int> &c, int idx, int t, vector<int> &v){
    if(t==0){
        result.push_back(v);
        return;
    }
    if(idx == c.size()) return;
    if(c[idx] <= t){
        v.push_back(c[idx]);
        f(c, idx+1, t-c[idx], v);
        v.pop_back();
    }
    int j = idx+1;
    while(j<c.size() and c[j]==c[j-1]) j++; // moving to next idx, if same element
    f(c, j, t, v);
}
```

(a) (b) (c)

```
vector<vector<int>> combinationSum2(vector<int> &c, int target){
    result.clear();
    sort(c.begin(), c.end());
    vector<int> v;
    f(c, 0, target, v);
    return result;
}
int main(){
    vector<int> candidates = {2,1,2,5,2};
    int target = 5;
    vector<vector<int>> combinations = combinationSum2(candidates, target);
    for(auto comb: combinations){
        cout<<"[";
        for(int num: comb){
            cout<<num<<" ";
        }
        cout<<"]\n";
    }
    return 0;
}
```

↳ op: [1 2 2]
[5]

b) Since we have distinct ele.s in array; so now we can repeat 1 ele. → many times!

Code:

(a)
```
f(c, idx, t-c[idx], v);
```
→ Here we can repeat elements, so we keep checking at same idx!

(b)
```
// while(j<c.size() and c[j]==c[j-1]) j++;
```
→ to remove/move on to next ele. which is not duplicate.

↳ Since our array has distinct integers, so we can comment out this line.

(c)

(d)
```
int main(){
    // similar to above one
}
```

↳ eg: i/p: [2,3,5], target = [8]

o/p: [[2,2,2,2],
[2,3,3],
[3,5]]

BINARY TREES - 1

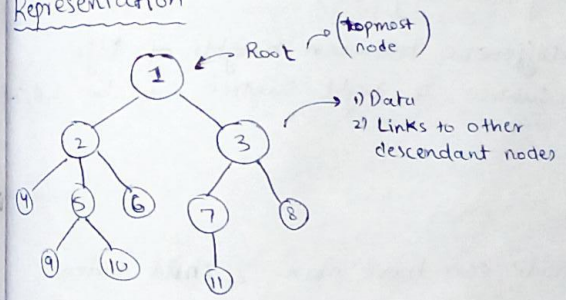
Data Structures

- Linear (Sequential)
  - array:
  - linked list: [] → [] → []
  - Stack
  - Queue
- Non-linear (Hierarchical)
  - Trees, Graphs

What is a Tree Data Structure?

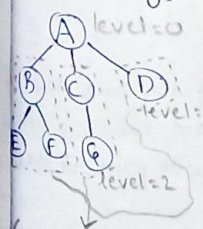
→ Non-linear data structures stores hierarchical data.
→ Elements are stored at diff. levels
→ Elements are called "nodes" - which are connected/linked together to represent hierarchy.

Representation



* A tree is represented by its root node

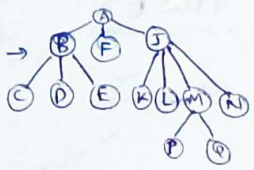
Terminology



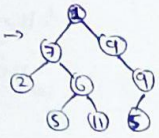
(ex. of subtree)

- Root: → Topmost node (ex: A) → Not a child node/parent node
- Child Node: → successor/descendant of any node (ex: E, F are children nodes of B)
- Parent Node: → predecessor of any node (ex: A is parent node of B, C, D)
- Sibling Nodes: → Nodes with common parent node (ex: E, F are sibling nodes)
- Leaf Node: → which have no child node (ex: E, F, G, D)
- Number of edges: → link btw 2 nodes (ex: A → E : 2 edges)
- Level: → no. of edges root nodes
- Height: → max. no. of edges btw a leaf node & root node (ex: here 2)
- Size: → no. of nodes in a tree → n = nodes = n-1 edges
- Subtree: → tree which is child of a node

Types of Trees:

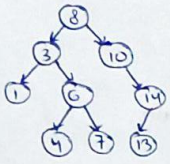
1) Generic Trees: → 

→ A node can have any no. of child nodes.

2) Binary Trees (BT): → 

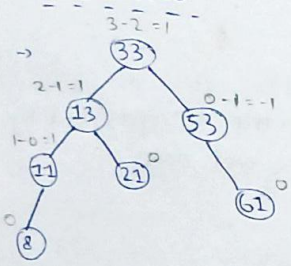
→ Tree in which a node can have max. 2 child nodes

3) Binary Search Trees: (BST)

→ 

→ BT in which all nodes in left subtree have value less than root node value & all nodes in right subtree will have value more than root node value.

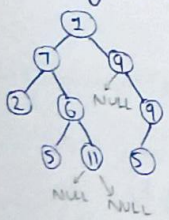
4) AVL Trees:

→ 

→ Self balancing Trees
→ Difference between heights of left subtree & right subtree can be -1, 0, 1

Binary Tree:

→ Every node can have max. 2 child nodes
→ Every node will contain → data
→ link to left child
→ link to right child



→ Implementation of Node class:

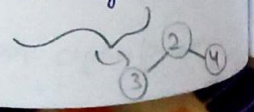
Node: | Left | data | Right |
(stores the address of left & right child node respectively)
* if any child does not exist, pointer will point to NULL

Code:

```
class Node{
    public:
    int val;
    Node* left;
    Node* right;
    Node(int v){
        value = v;
        left = right = NULL;
    }
};
```

```
int main(){
    Node* root = new Node(2);
    root->left = new Node(3);
    root->right = new Node(4);
    cout<<"Root Node "<<root->value<<endl;
    cout<<"Left Child "<<root->left->value<<endl;
    cout<<"Right Child "<<root->right->value<<endl;
    return 0;
}
```

→ Root Node 2
Left Child 3
Right Child 4



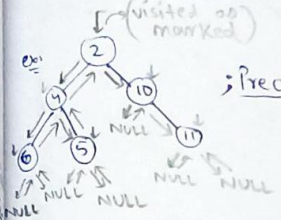
Traversals:

→ DFS {Depth first traversal}
→ BFS {Breadth first traversal}

→ DFS: → travel along the height

→ DFS
Pre-order | Inorder | Postorder

→ Preorder:
1) Visit the root node
2) Left subtree
3) Right subtree
} recursively

ex: 

; Preorder: 2 4 6 5 10 11

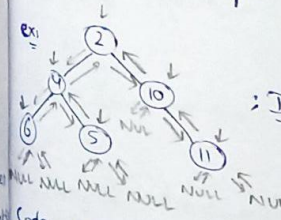
Code:

```
void preorderTraversal(Node* rootNode){
    if(rootNode == NULL) return; // base case
    cout<<rootNode->value<<" ";
    preorderTraversal(rootNode->left);
    preorderTraversal(rootNode->right);
}
```
} recursive call

```
int main(){
    Node* rN = new Node(2);
    rN->left = new Node(4);
    rN->right = new Node(10);
    rN->left->left = new Node(6);
    rN->left->right = new Node(5);
    rN->right->right = new Node(11);
    preorderTraversal(rN);
    return 0;
}
```

o/p: 2 4 6 5 10 11

→ Inorder:
1) Visit left subtree
2) Root Node
3) Right Subtree
} recursively

ex: 

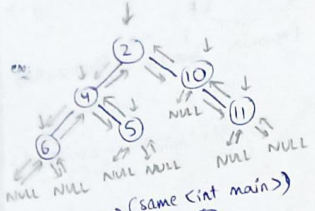
; Inorder: 6 4 5 2 10 11

Code: → {same <int main()>}

```
void inorderTraversal(Node* rootNode){
    if(rootNode == NULL) return;
    inorderTraversal(rootNode->left);
    cout<<rootNode->value<<" ";
    inorderTraversal(rootNode->right);
}
```

o/p: 6 4 5 2 10 11

→ Postorder: 1) Visit left subtree
2) Right subtree
3) Root Node
} recursively



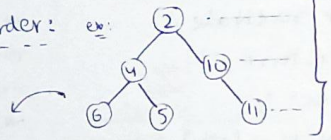
ex: → Postorder: 6 5 4 11 10 2

Code: → (same int main())

```
void postorderTraversal(Node* rootNode){
    if(rootNode==NULL) return;
    postorderTraversal(rootNode->left);
    postorderTraversal(rootNode->right);
    cout<<rootNode->value<<" ";
}
```

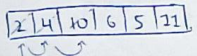
o/p:

→ BFS: ⇒ Levelorder: ex:



→ goes level wise
↳ o/p: 2 4 10 6 5 11
we use "Queue" to get this conclusion.

explanation



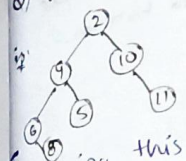
- We insert root node → pop it → insert it's child nodes → pop '4' & inserting it's child node → pop '10' & inserting it's child nodes ...
- We can print: qu.size() → to get no. of nodes at particular level

Code: → {same <int main>}

```
void levelorderTraversal(Node* rootNode){
    if(rootNode==NULL) return;
    queue<Node*>q;
    q.push(rootNode);
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel--){
            Node* currNode = q.front();
            q.pop();
            cout<<currNode->value<<" ";
            if(currNode->left!=NULL) q.push(currNode->left);
            if(currNode->right!=NULL) q.push(currNode->right);
        } cout<<endl;
    }
}
```

↳ o/p:
2
4 10
6 5 11

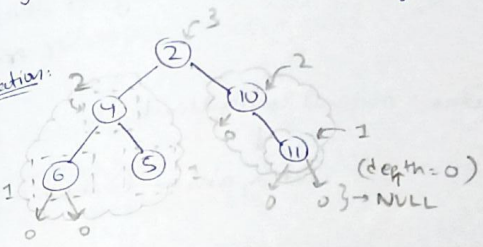
Q) Given root of binary tree, return it's maximum depth.
(no. of nodes along the longest path from root node to farthest leaf node.)



ip: , o/p: 3

A) Solving this recursively → max depth of tree = max(left subtree max depth, right subtree max depth) + 1

↳ explanation:



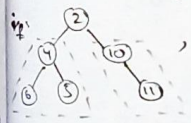
Code:

```
int maxDepth(Node* root){
    if(root==NULL) return 0;
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    return max(leftDepth, rightDepth)+1;
}
```

```
int main(){
    :
    cout<<maxDepth(rootNode);
}
```

↳ o/p: 3 (for given i/p in que.)

Q) Given root of binary tree, return no. of leaf nodes present in it
(node which doesn't have any left and right child)



ip: , o/p: 3

A) Solving recursively → we need ⇒ leafNodes(left subtree) + leafNodes(right subtree)

base case → root==NULL → 0

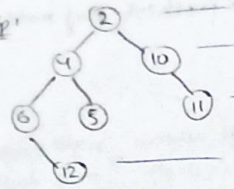
Code:

```
int leafNodes(Node* root){
    if(root==NULL) return 0;
    if(root->left==NULL and root->right==NULL) return 1;
    int leftSubtreeLeafNodes = leafNodes(root->left);
    int rightSubtreeLeafNodes = rightNodes(root->right);
    return leftSubtree LeafNodes + rightSubtree LeafNodes;
}
```

```
int main(){
    :
    cout<<leafNodes(rootNode);
}
```

→ o/p: 3

Q) Given root of binary tree, imagine yourself standing right side of it, return the value of nodes you can see top to bottom.

i/p: 

o/p: [2, 10, 11, 12]

↳ He can see rightmost node of all levels (i.e. BFS)

A) Level Order Traversal

ex: 

→ when: nodes at currentlevel = 1

↓

Put in ans → [2, 10, 11, 12]

Code:

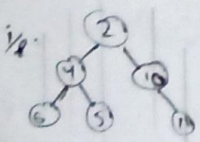
```
vector<int> rightViewBinaryTree(Node* root){   // Returns vector of integers
    vector<int> ans;
    if(root==NULL) return ans; // if tree is empty
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel){
            Node* currNode = q.front();
            q.pop();
            if(nodesAtCurrentLevel==1){
                ans.push_back(currNode->value);
            }
            if(currNode->left!=NULL){
                q.push(currNode->left);
            }
            if(currNode->right!=NULL){
                q.push(currNode->right);
            }
            nodesAtCurrentLevel--;
        }
    }
    return ans;
}
```

```
int main(){
    |
    |
    vector<int> ans =
        rightViewBinaryTree(root);
    for(auto i: ans){
        cout<<i<<" ";
    } cout<<endl;
}
```

o/p: 2 10 11 12

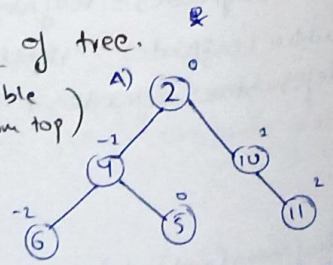
Q) Given binary tree. Print top view of tree.

(set of nodes visible when viewed from top)

i/p: 

(vertically level order)

o/p: [6, 4, 2, 10, 11]

A) 

Column of left child = column of parent child - 1

Column of right child = column of parent child + 1

Level order → queue - Pair(Node, column)

Map - column → value

Whenever can ele. comes, we store it

ex: {2, 5} → in i/p

stored ↳ in map → at same column present (so ignored)

Code:

```
vector<int> topView(Node* root){
    vector<int> ans;
    if(root==NULL) return ans;
    queue<pair<Node*, int>> q;
    q.push(make_pair(root, 0));
    map<int, int> m;
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel--){
            pair<Node*, int> p = q.front();
            Node* currNode = p.first;
            int currCol = p.second;
            q.pop();
            if(m.find(currCol)==m.end()){
                m[currCol] = currNode->value;
            }
            if(currNode->left!=NULL){
                q.push(make_pair(currNode->left, currCol-1));
            }
            if(currNode->right!=NULL){
                q.push(make_pair(currNode->right, currCol+1));
            }
        }
    }
    for(auto it: m){
        ans.push_back(it.second);
    }
    return ans;
}
```

o/p: 6 4 2 10 11

Explanation:

① Initialization:

- 'q': Initially contains root node (2,0) where '0' is initial column position
- 'm', 'ans': initially empty
  - 'm' → map
  - 'ans' → vector

② First Iteration: • nodesAtCurrentLevel = 1

- Extract '(2,0)' from 'q'
  - ↳ currNode = 2, currCol = 0
- q.pop()
- m.find(currCol): Check if 'm' contains key '0'
  - → Since 'm' is initially empty, 'm.find(currCol)' will return 'm.end()' because key '0' is not found.

- m[currCol] = currNode → value ; Adds '2' to the map with
- Enqueues left & right children of '2' with adjusted column positions : '(4, -1)' & '(10, 1)'.

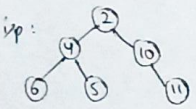
- Similarly ----

Q) WAP to see bottom view of binary tree ?

A) Before if we are receiving first value, we are inserting into map.
Now if we receive value in same col, we will just update it.

So code: Same as prev. one ! except: m → m[currCol] = currNode (changes to)

Q) Given a binary tree, print the level order in reverse manner ie. print the last row first & then the rows before.



i/p: [6,5,11,4,10,2]

A) Level order : | 2 | 10 | 4 | 11 | 5 | 6 | → This time, we will insert right child & then left child & we will store all these in stack

→ o/p LIFO → 6, 5, 11, 4, 10, 2

Code:

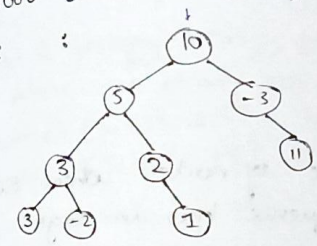
```
vector<int> reverseLevelOrder(Node* root){
    vector<int> ans;
    if(root==NULL) return ans;
    queue<Node*> q;
    stack<Node*> s;
    q.push(root);
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel--){
            Node* currNode = q.front();
            q.pop();
            s.push(currNode);
            if(currNode->right) q.push(currNode->right);   // != NULL } → by default
            if(currNode->left) q.push(currNode->left);
        }
    }
    while(!s.empty()){                                     // it stores node. Since we
        ans.push_back(s.top()->value);                     // value, so
        s.pop();
    }
    return ans;
}
```

INTERVIEW PROBLEMS ON BINARY TREES

Q) Given root of binary tree & an integer targetSum, return the no. of paths where the sum of the values along the path equals targetSum. The path doesn't need to start or end at the root or a leaf, but it must go downwards (i.e. traveling only from parent nodes to child nodes)

i/p: root = [10,5,-3,3,2,null,11,3,-2,null,1] , targetSum = 8

o/p: 3



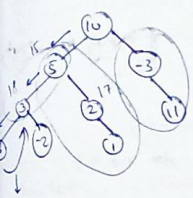
A) We can solve this by prefix sum approach

ex: 10 5 3 3

10 15 18 21

target sum → : checking if [currSum - targetSum] is present in our tree.

Now, we also use map!



Sum	freq.
0	1
10	1
15	1
18	1
21	1
16	1
13	1
18	1
7	1

currSum = 15, 18, 16

16 - 8 = 8 ✓

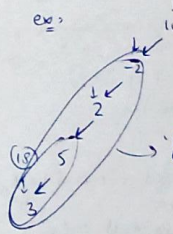
10 - 8 = 2 → not present in tree, so we add in map

∴ ans = 3

(while going back, we need to subtract)

reason of storing frequency:

ex:



10 2

8 1

8 → we got twice so we are storing frequency

∴ $T_c = O(n)$ (linearly traversing all nodes)

$S_c = O(n)$

Code:

```
int pathSumHelper(Node* root, int targetSum, long int currSum, unordered_map<long int, int> pathCount){
    if(root==NULL) return 0;
    currSum += root->value;
    int ansCount = pathCount[currSum - targetSum];   // if not present, then ansCount = 0 (default)
    pathCount[currSum]++;                            // else freq. of that will come to ansCount
    ansCount += pathSumHelper(root->left, targetSum, currSum, pathCount);
    ansCount += pathSumHelper(root->right, targetSum, currSum, pathCount);
    pathCount[currSum]--;                            // while backtracking ie. while going back
    return ansCount;
}
```

```
int pathSum(Node*, root, int targetSum){
    unordered_map<long int, int> pathCount;
    pathCount[0] = 1;  → Sum=0, freq.=1
    return pathSumHelper(root, targetSum, 0, pathCount);
}

int main(){
    Node* rootNode = new Node(10);
    rootNode->left = new Node(5);

    int targetSum = 8;                    → o/p: 3
    cout << pathSum(rootNode, targetSum);
    return 0;
}
```

Q) A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence atmost once. Note that path doesn't need to pass via root. Path sum of a path is the sum of the nodes values in the path.

Given root of binary tree, return max. path sum of any non-empty path.

i/p: root = [1,2,3] , o/p: 6

Ai ex: → max ✓

Cases: (where max sum can be)
1) Left child (if right has (-)ve nos)
2) Right child
3) Both left & right child
4) only root node (if both child are (-)ve)

maxSum = max(maxSum, leftmaxSum + rightmax Sum + root->val)
(if >0) (if >0)

* we will do postorder traversal

Now → return ⇒ root->val + max(leftmaxSum, rightmaxSum);

ex:

(it needs to tell prev. node (here 10) that what ans we will get.)

maxSum will get updated everytime, so we are passing by reference

Code: ④

```
int maxPathSumHelper(Node* root, int &maxSum){
    if(root==NULL) return 0;
    int leftMaxSum = max(0, maxPathSumHelper(root->left, maxSum));
    int rightMaxSum = max(0, maxPathSumHelper(root->right, maxSum));
    maxSum = max(maxSum, root->value + leftMaxSum + rightMaxSum);
    return root->value + max(leftMaxSum, rightMaxSum);
}
```

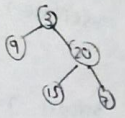
```
int maxPathSum(Node* root){
    int maxSum = INT_MIN;  → incase our tree has all -ve nos
    maxPathSumHelper(root, maxSum);
    return maxSum;
}
int main(){
    cout << maxPathSum(rootNode);
}
```

- $T_c = O(n)$ (traversing nodes once)
- $S = O(n)$ (during recursion)

Q) Given 2 integer arrays preorder & inorder (binary tree traversal) of same tree, construct & return the binary tree.

i/p: preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]

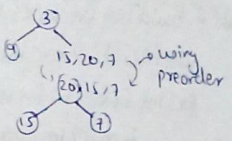
o/p: [3,9,20,null,null,15,7]



Ai Preorder → root, left, right & Inorder → left, root, right

pre: [3,9,20,15,7] , in: [9,3,15,20,7]
(prestart) (preend) (instart) (inend)
root (root)

(checking so that we know how many nodes in left subtree & right subtree)



using preorder

	Left Subtree	Right Subtree
Inorder:	instart, rootindex - 1	rootindex + 1, inend
Preorder:	prestart + 1, prestart + leftsubtree elements	prestart + leftsubtree elements + 1, pre-end

leftsubtree elements = rootindex - instart

$T_c = O(n)$, $S_c = O(n)$ (storing in map)

Like above, we will call it recursively.

Base case → prestart > pre-end (or) instart > inend

& we will store values with index in a map → (of inorder) → So that searching = O(1)

∴ $T_c = O(n)$ ↝ Traversing all elements

$S = O(n)$ ↝ as we are using map for space!

Code (*)

```
Node* buildTreeHelper(vector<int> &preorder, int preStart, int preEnd, vector<int> &inorder
        , int inStart, int inEnd, unordered_map<int,int> & inmap){
    if(preStart > preEnd or inStart > inEnd) return NULL; //handling leaf cases
    Node* rootNode = new Node(preorder[preStart]); // first ele. in preorder is the root
    int rootInorderIndex = inmap[rootNode->value];
    int leftSubtreeSize = rootInorderIndex - inStart;
(A) rootNode->left = buildTreeHelper(preorder, preStart+1, preStart+leftSubtreeSize,
                inorder, inStart, rootInorderIndex-1, inmap);
    rootNode->right = buildTreeHelper(preorder, preStart+leftSubtreeSize+1, preEnd,
                inorder, rootInorderIndex+1, inEnd, inmap);
    return rootNode;
}
Node* buildTree(vector<int> &preorder, vector<int> &inorder){
    unordered_map<int,int> inmap;
    for(int i=0; i<inorder.size(); i++){
        inmap[inorder[i]] = i;
    }
    return buildTreeHelper(preOrder, 0, preorder.size()-1, inorder, 0, inorder.size()-1
                , inmap);
}
void printInorder(Node* root){
    if(root){
        printInorder(root->left);
        cout<<root->value<<" ";
        printInorder(root->right);
    }
    else{
        cout<<"NULL ";
    }
}
int main(){
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};
    Node* rootNode = buildTree(preorder, inorder);
    cout<<"Inorder Traversal is: ";
    printInorder(rootNode);
    return 0;
}
```

o/p:

NULL 9 NULL 3 NULL 15 NULL 20 NULL 7

NOTE: If we want to build tree from postorder & inorder:

Node* buildTreeHelper(—), Node* buildTree(—), int main(){—}

postOrder ⇐ preOrder
postStart ⇐ preStart } change
postEnd ⇐ preEnd

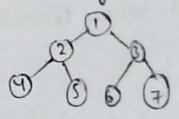
In (A):

```
if(postStart > postEnd or inStart > inEnd) return NULL; //handling leaf case
Node* rootNode = new Node(postOrder[postEnd]); //last element in postorder is the root
int rootInorderIndex = inmap[rootNode->value];
int rightSubtreeSize = inEnd - rootInorderIndex;
rootNode->left = buildTreeHelper(postorder, postStart, postEnd-rightSubtreeSize-1,
                inorder, inStart, rootInorderIndex-1, inmap);
rootNode->right = buildTreeHelper(postorder, postEnd-rightSubtreeSize,
                postEnd-1, inorder, rootInorderIndex+1, inEnd, inmap);
```

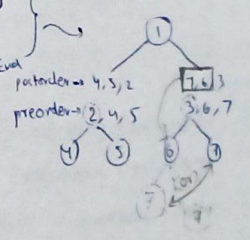
Q) Given 2 int. arrays, preorder & postorder where preorder is the preorder traversal of a binary tree & postorder is the postorder traversal of the same tree, reconstruct & return binary tree

i/p: preorder = [1,2,4,5,3,6,7] ; o/p: [1,2,3,4,5,6,7]
postorder = [4,5,2,6,7,3,1]

If there exists multiple answers, return any one of them.



A) Preorder → root, left, right → [1,2,4,5,3,6,7]
Postorder → left, right, root → [4,5,2,6,7,3,1]



Postorder	Left Subtree: postStart, leftChildIndex	Right Subtree: leftChildIndex+1, postEnd-1
Preorder	preStart+1, preStart+leftSubtreeElements	preStart+leftSubtreeElements+1, preEnd

Assuming preorder ke next → left subtree ele

leftSubtreeElements = leftChildIndex - postStart + 1

$T_c = O(N)$

$S_c = O(N)$ → for map of postorder!

Code: ⊛

```
Node* constructFromPrePostHelper(vector<int> &preorder, int prestart, int preend,
        vector<int> &postorder, int poststart, int postend, unordered_map<int,int> &postmap){
    if(poststart > postend or prestart > preend) return NULL;
    Node* rootNode = new Node(preorder[prestart]);
    if(prestart == preend) return rootNode;
    int leftChildVal = preorder[prestart+1];
    int leftChildIndex = postmap[leftChildVal];
    int leftSubtreeSize = leftChildIndex - poststart + 1;
    rootNode->left = constructFromPrePostHelper(preorder, prestart+1, prestart+leftSubtreeSize,
                        postorder, poststart, leftChildIndex, postmap);
    rootNode->right = constructFromPrePostHelper(preorder, prestart+leftSubtreeSize+1,
                        preend, postorder, leftChildIndex+1, postend-1, postmap);
    return rootNode;
}
Node* constructFromPrePost(vector<int> &preorder, vector<int> &postorder){
    unordered_map<int,int> postmap;
    for(int i=0; i<postorder.size(); i++){
        postmap[postorder[i]] = i;
    }
    return constructFromPrePostHelper(preorder, 0, preorder.size()-1, postorder,
                        postorder.size()-1, postmap);
}
```

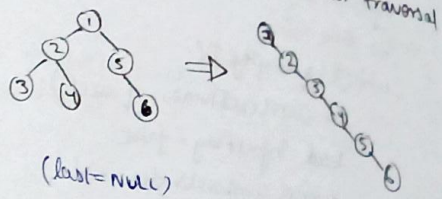
⊖

```
int main(){
    vector<int> preorder = {1,2,4,5,3,6,7};
    vector<int> postorder = {4,5,2,6,7,3,1};
    Node* rootNode = constructFromPrePost(preorder, postorder);
    printInorder(rootNode);
    return 0;
}
```

o/p: NULL 4 NULL 2 NULL 5 NULL 1 NULL 6 NULL 3 NULL 7 NULL

Q) Given root of binary tree, flatten the tree into a "linked list"

- the LL should use the same Node class where the right child pointer points to the next node in the list & the left child pointer is always null.
- the LL should be in the same order as a pre-order traversal of the binary tree



ip: root = [1,2,5,3,4,null,6]

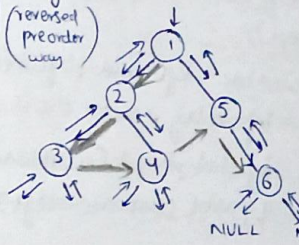
o/p: 1->2->3->4->5->6

Algo:

```
if(root == NULL) return
flatten(root->right)
flatten(root->left)
root->right = last
root->left = NULL
last = root
return
```

(going in reversed preorder way)

(last = NULL)



$T_c = O(n)$

$S_c = O(n)$ (n = height of binary tree → recursion stack)

Code: ⊛

```
Node* lastNode = NULL;
Node* flatten(Node* root){
    if(root == NULL) return NULL;
    flatten(root->right);
    flatten(root->left);
    root->right = lastNode;
    root->left = NULL;
    lastNode = root;
    return root;
}
```

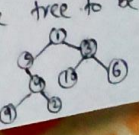
```
void printLL(Node* head){
    Node* current = head;
    while(current != NULL){
        cout<< current->value << " ";
        current = current->right;
    }
}
int main(){
    Node* rootNode = new Node(1);
    printLL(flatten(rootNode));
}
```

o/p: 1 2 3 4 5 6

Q) You are given the root of a binary tree with unique values, and an integer start. At minute 0, an infection starts from the node with value start. Each minute, a node becomes infected if:

- The node is currently uninfected
- The node is adjacent to an infected node.

Return no. of minutes needed for the entire tree to be infected.



ip: root = [1,2,5,3,4,null,6]

o/p: [1,null,2,null,3,null,4,null,5,null,6]

A) Code, then explanation:

(*)

```
int calculateTime(Node * startNode, unordered_map<Node*, Node*>& parent){
    unordered_set<Node*> infected;
    queue<Node*> q; // required for BFS -> Level order traversal
    q.push(startNode);
    infected.insert(startNode);
    int time = 0;
    while(!q.empty()){
        int currLevelNodes = q.size();
        bool infectFlag = false;
        while(currLevelNodes--){
            Node* currNode = q.front();
            q.pop();
            if(currNode->left and !infected.count(currNode->left)){
                infectFlag = true;
                infected.insert(currNode->left);
                q.push(currNode->left);
            }
            if(currNode->right and !infected.count(currNode->right)){
                infectFlag = true;
                infected.insert(currNode->right);
                q.push(currNode->right);
            }
            if(parent[currNode] and !infected.count(parent[currNode])){
                infectFlag = true;
                infected.insert(parent[currNode]);
                q.push(parent[currNode]);
            }
        }
        if(infectFlag) time++;
    }
    return time;
}

Node* makeParent(Node * root, unordered_map<Node*,Node*>& parent, int start){
    queue<Node*> q;
    q.push(root);
    Node* startNode;
    while(!q.empty()){
        Node* currNode = q.front();
        q.pop();
        if(currNode->value == start) startNode = currNode;
        if(currNode->left != NULL){
            parent[currNode->left] = currNode;
            q.push(currNode->left);
        }
        if(currNode->right){
            parent[currNode->right] = currNode;
            q.push(currNode->right);
        }
    }
    return startNode;
}

int amountOfTime(Node* root, int start){
    unordered_map<Node*, Node*> parent;
    Node* startNode = makeParent(root, parent, start);
    return calculateTime(startNode, parent);
}

int main(){
    Node* root = new Node(1);
    ...
    int startTime = amountOfTime(root, 3);
    cout<<"Time for infection \n";
    cout<<startTime;
}
```

(making child-parent pair)

(value of node from where infection is starting)

→ stores every child-parent node pair

(return startNode & parentmap)

↳ Time for infection: 4

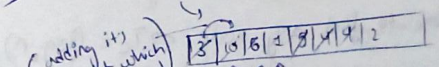
Eg:

We will do BFS → level order

map<child node, parent node>

& we will maintain a queue

(adding its adjacent which are uninfected)



infected - 3, 10, 6, 1, 5, 4, 2

uninfected + adjacent → time++

Now,

T_c: O(N)

(time to go through all nodes in BFS manner)

S: O(N) → queue, set, map

Q) How to take binary tree as user input?

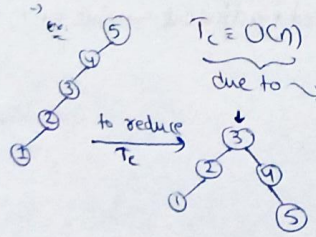
A)
```
#include <iostream>
using namespace std;
class Node{
    public:
    int value;
    Node* left;
    Node* right;
    Node(int data){
        value = data;
        left = right = NULL;
    }
};

int main(){
    Node* root = takeInput();
    cout<<"Inorder Traversal of Constructed Tree: ";
    printInorder(root);
    return 0;
}
```

```
Node* takeInput(){
    cout<<"Enter value of node (Enter -1 to represent NULL): ";
    int value;
    cin>>value;
    if(value==-1) return NULL;
    Node* root = new Node(value);
    cout<<"Enter left child of "<<value<<": ";
    root->left = takeInput();
    cout<<"Enter right child of "<<value<<": ";
    root->right = takeInput();
    return root;
}
void printInorder(Node* root){
    if(root){
        printInorder(root->left);
        cout<<root->value<<" ";
        printInorder(root->right);
    }
}
```

✡ BINARY SEARCH TREES

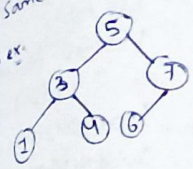
- It is a type of binary tree (max 2 child nodes) where all values in left subtree of node < value of node & all values in right subtree of node > value of node
- This is true for root node of all subtrees in a BST
- Advantages: → Insertion, Deletion, Search } $T_c = O(\log_2 n)$ ↳ avg. case
  → Sorted order: easier to find min. & max. elements
- Disadvantages:



$T_c = O(n)$ → worst case, due to ↳ unbalanced tree (skewed in 1 direction)

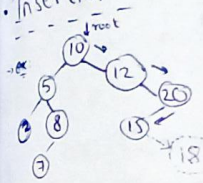
- Applications: Sets, Maps & Priority Queues

- Traversals: Same like in binary trees



Preorder: (root, left, right): 5 3 1 4 7 6
Inorder: (left, root, right): 1 3 4 5 6 7
Postorder: (left, right, root): 1 4 3 6 7 5

- Insertion:



Add → (18) ⇒ Code:

T_c → Avg $= O(\log_2 n)$
↳ Worst $= O(n)$
(passing by reference as we making changes in orig. root)

```
class Node{
    public:
    int value;
    Node* left;
    Node* right;
    Node(int v){
        value = v;
        left = right = NULL;
    }
};
class BST{
    public:
    Node* root;
    BST(){
        root = NULL;
    }
};
```

// iterative approach

```
void insertBST(Node* &root, int val){
    Node* newNode = new Node(val);
    if(root==NULL){
        root = newNode;
        return;
    }
    Node* curr = root;
    while(true){
        if(curr->value > val){
            if(curr->left==NULL){
                curr->left = newNode;
                return;
            }
            curr = curr->left;
        }
        else{
            if(curr->right==NULL){
                curr->right = newNode;
                return;
            }
            curr = curr->right;
        }
    }
}
```

```
void inorderTraversal(Node* root){
    if(root==NULL) return;
    inorderTraversal(root->left);
    cout<<root->value<<" ";
    inorderTraversal(root->right);
}
```

```
int main(){
    BST bst1;
    insertBST(bst1.root, 3);
    insertBST(bst1.root, 1);
    insertBST(bst1.root, 4);
    inorderTraversal(bst1.root);
    return 0;
}
```

↳ op: 1 3 4

// recursive approach

```
Node * insertBST(Node* root, int val){
    if(root == NULL){
        Node* newNode = new Node(val);
        return newNode;
    }
    if (root -> value > val){
        root -> left = insertBST(root->left, val);
    }
    else if(root -> value < val){
        root -> right = insertBST(root -> right, val);
    }
}
```

```
int main(){
    BST bst1;
    bst1.root = insertBST
                (bst1.root,
    insertBST(bst1.root,
    insertBST(bst1.root,
    inorderTraversal(bst1.
    return 0;
}
```

o/p: 1 3 4

- Search: $T_c = O(\log n)$ → Avg. case

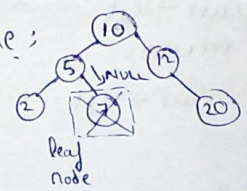
Code:

```
bool searchBST(Node * root, int key){
    if(root == NULL) return false;
    if (root -> value == key) return true;
    if(root -> value > key) return searchBST (root->left, key);
    if(root -> value < key) return searchBST(root->right, key);
}
int main(){
    :
    cout<< searchBST(bst1.root ,8) ;
}
```

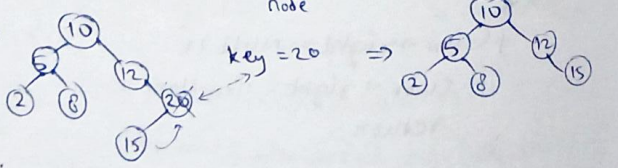
o/p: 0

- Deletion:

→ Node has 0 child ie. leaf node;

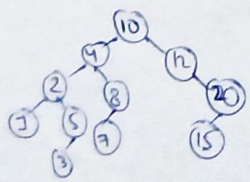


→ Node has 1 child;



T_c = O(log n) → avg.
O(n) → worst

→ Node has 2 child;



S1: Find just larger / just smaller value
⇒ Largest value smaller than 5
⇒ Smallest value larger than 5
S2: Node → value = just smaller value
S3: Delete justSmallerValue in root → left

Code:

```
Node * deleteBST(Node* root, int key){
    if(root == NULL) return root;
    if(root -> value < key) root->right = deleteBST (root -> right, key);
    else if (root -> value > key) root -> left = deleteBST(root -> left, key);
    else{ // root is node to be deleted
        if(root -> left == NULL and root -> right == NULL){ // 0 child
            free(root);
            return NULL;
        }
        else if(root -> left == NULL){ // 1 child
            Node* temp = root -> right; //
            free(root);
            return temp;
        }
        else if (root -> right == NULL){ // 1 child
            Node* temp = root -> left; //
            free(root);
            return temp;
        }
        else{
            Node* justSmallerNode = largestNodeBST(root -> left);
            root -> value = justSmallerNode -> value;
            root -> left = deleteBST (root -> left, justSmallerNode -> value);
        }
    }
    return root;
}
```

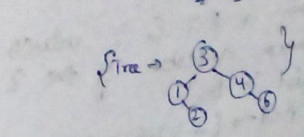
pointing to non-null child

{before "deleteBST" func.}

```
Node* largestNodeBST (Node* root){
    Node* curr = root;
    while (curr and curr -> right){
        curr = curr -> right;
    }
    return curr;
}
```

```
int main(){
    :
    bst1.root = deleteBST(bst1.root,
    inorderTraversal(bst1.root);
}
```

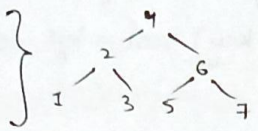
o/p: 1 2 3 6



Q) Given sorted array. Create a Balanced Binary Search Tree out of it. A Balanced BST is height-balanced ie. the difference btw the height of the left subtree & right subtree is not more than 1. Print preorder traversal of BST.

I/P: n = 7
[1, 2, 3, 4, 5, 6, 7]

O/P: preorder = 4 2 1 3 6 5 7



A) Mid ele. = root & left ele = left subtree
right ele = right subtree

traversing all ele
$T_c = O(n)$
$S_c = O(h)$
↓
height of recursion
=
height of tree

(Since balanced tree, so $h = \log n$)

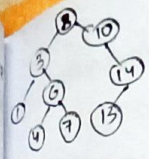
Code:

```
void preorderTraversal(Node* root){
    if(root==NULL) return;
    cout<<root->value<<" ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
Node* sortedArrayToBST(vector<int> v, int start, int end){
    if(start > end) return NULL;
    int mid = (start+end)/2;
    Node* root = new Node(v[mid]);
    root->left = sortedArrayToBST(v, start, mid-1);
    root->right = sortedArrayToBST(v, mid+1, end);
    return root;
}
int main(){
    int n;
    cin>>n;
    vector<int> v(n);
    for(int i=0; i<n; i++) cin>>v[i];
    BST bst;
    bst.root = sortedArrayToBST(v, 0, n-1);
    preorderTraversal(bst.root);
    return 0;
}
```

Q) Given a BST & 2 values. You need to find LCA ie. lowest common ancestor of the 2 nodes provided both the nodes exist in BST.

I/P: n = 9 → [8, 3, 1, 6, 4, 7, 10, 14, 13]
node-1 = 3
node-2 = 13

O/P: lowest common ancestor = 8



A) LCA – shared ancestor farthest from root node

* Every node is it's own ancestor

ex: 3, 10 → now LCA lies btw them → ans 8
4, 3 → ans: 3

Code:

```
Node* lowestCommonAncestor(Node* root, Node* node1, Node* node2){
    if(root == NULL) return NULL; // base case
    if(root->value > node1->value and root->value > node2->value){
        // LCA lies in left subtree
        return lowestCommonAncestor(root->left, node1, node2);
    }
    if(root->value < node1->value and root->value < node2->value){
        // LCA lies in right subtree
        return lowestCommonAncestor(root->right, node1, node2);
    }
    return root;
}
int main(){
    ...
    Node* node1 = new Node(2);
    Node* node2 = new Node(6);
    Node* temp = lowestCommonAncestor(bst.root, node1, node2);
    cout<<temp->value;
}
```

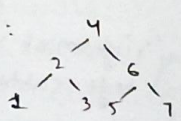
(we can also pass direct values)

→ if root value lies btw node1 & node2 value
→ if root value = any of nodes value)

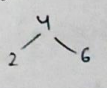
Q) Remove all leaf nodes from BST

I/P: no. of nodes in BST, followed by the node values
O/P: preorder traversal of BST before & after removing the leaf nodes

before:



after:



A) Leaf nodes → nodes with no child nodes → root->left == NULL, root->right == NULL

Max depth :- 'h'; $T_c = O(n)$ → traversing all ele-s

$S_c = O(h)$ → height

```
Node* removeLeafNodes(Node* root){
    if(root==NULL) return NULL; //base case
    if(root->left==NULL and root->right==NULL){
        return NULL; //when root is leaf node
    } //recursive case
    root->left = removeLeafNodes(root->left);
    root->right = removeLeafNodes(root->right);
    return root;
}
```

→ handles the case if node has only 1 child

```
int main(){
    BST bst;
    : //input of binary search tree
    preorderTraversal(bst.root);
    bst.root = removeLeafNodes(bst.root);
    cout<<endl<<preorderTraversal(bst.root);
}
```

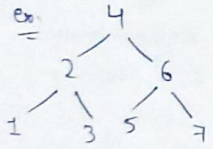
```
void preorderTraversal(){
}
```

Q) Find inorder predecessor & successor for a given key in BST

i/p: I/p to program is the key for which we want to find the inorder predecessor & successor in the BST.

o/p: O/p of program is the inorder predecessor & successor of the given key in the BST

A) ex:



→ inorder: 1 2 3 4 5 6 7

brute-force: store above order in vector & print the front & back of key.

(Here we need extra space)

Instead; (On observing)

pre - greatest value smaller than 'k' → (rightmost value in left subtree)

succ - Smallest value greater than 'k' → (leftmost value in right subtree)

if -> root > key

succ = root

root = root->left

if -> root < key

pre = root

root = root->right

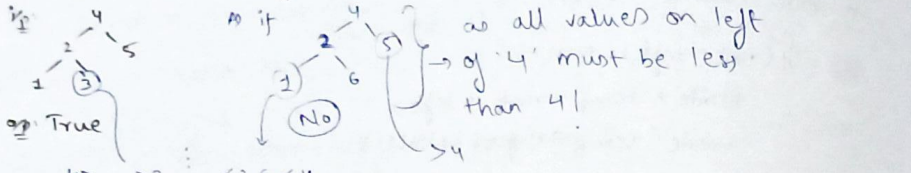
```
void inorderPreSuccBST(Node* root, Node* &pre, Node* &succ, int key){
    if(root==NULL) return;
    if(root->value==key){
        //pre - rightmost value in left subtree
        if(root->left!=NULL){
            Node* temp = root->left;
            while(temp->right!=NULL){
                temp = temp->right;
            }
            pre = temp;
        }
        if(root->right!=NULL){
            Node* temp = root->right;
            while(temp->left!=NULL){
                temp = temp->left;
            }
            succ = temp;
        }
        return;
    }
    if(root->value > key){
        succ = root;
        inorderPreSuccBST(root->left, pre, succ, key);
    }
    else if(root->value<key){
        pre = root;
        inorderPreSuccBST(root->right, pre, succ, key);
    }
}
```

if one of the child is NULL so to handle that case

```
int main(){
    : //Tree input
    Node* pre = NULL;
    Node* succ = NULL;
    inorderPreSuccBST(bst.root, pre, succ, 4);   (key you can set)
    if(pre!=NULL) cout<<"pre-"<<pre->value<<endl;
    else cout<<"pre-null"<<endl;
    if(succ!=NULL) cout<<"succ-"<<succ->value<<endl;
    else cout<<"succ-null"<<endl;
}
```

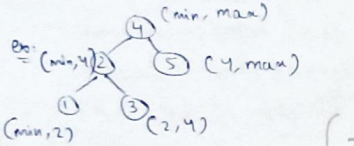
Interview Problems On BST

Q) Given a binary tree, check if it's a BST or not?



We can solve this by making range & doing it recursively!

if → node is defined within range (lies in range) & its left + right child are BST (or tree) } Hence the node / tree is a BST



For 4
↳ min = INT_MIN
max = INT_MAX

(taking long because node val. can be = to int min/max, so min (or max should be < or) than that)

Code:

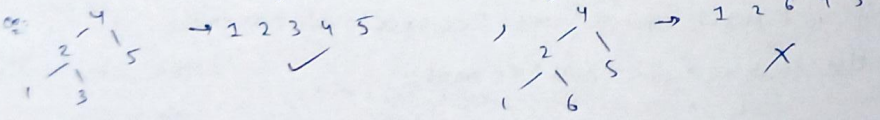
```
bool isValidBSTHelper(Node* root, long minVal, long maxVal){
    if(root==NULL) return true;   // when BST is empty or we are at leaf nodes
    if(root->value <= minVal or root->value >= maxVal) return false;   // to check if parent = child
    return isValidBSTHelper(root->left, minVal, root->value);
    return isValidBSTHelper(root->right, root->value, maxVal);
}
bool isValidBST(Node* root){
    return isValidBSTHelper(root, LONG_MIN, LONG_MAX);
}
```

(left/right subtree must be BST)

- $T_c = O(n)$ ← no. of nodes
- $S_c = O(h)$ → in skewed tree: $S_c = O(n)$

Method 2

Inorder traversal of a BST is 'always' sorted.



We can store this order in vector → to avoid, we can compare current value with prev. value & hence we can get our

↳ there $T_c = O(n)$, $S_c = O(n)$

```
bool isValidBSTHelper(Node* root, Node* &prev){
    if(root==NULL) return true;
    if(!isValidBSTHelper(root->left, prev)){ // left subtree
        return false;
    }
    if(prev!=NULL and root->value <= prev->value){ // root node
        return false;
    }
    prev = root;
    return isValidBSTHelper(root->right, prev); // right subtree
}
bool isValidBST(Node* root){
    Node* prev = NULL;
    return isValidBSTHelper(root, prev);
}
```

Q) Given 2 vectors that represent a sequence of keys. Imagine we make a BST from each array. We need to tell whether 2 BST's will be identical or not without actually constructing the tree.

ex: arr1 = {4,2,5,1,3} Ans: BST's are identical
arr2 = {4,5,2,3,1}

A) 1) Find the first elements within range in both arrays.
2) If both elements are leaf nodes - return true
3) If one elements is leaf nodes - return false.
4) If elements not same - return false
5) Recursively check for left & right subtree.

- $T_c = O(n^2)$ ↳ [for 1 ele. in arr1, we are looking in whole arr2]
- $S_c = O(h)$ ↳ $O(n)$ worst case

(checking for left & right subtree)

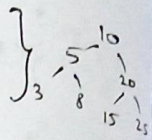
```
bool checkIdenticalBSTHelper(vector<int>&v1, vector<int>&v2, int a1, int a2, int minVal, int maxVal){
    int i; // find the first ele. in v1 within range
    for(i=a1; i<v1.size(); i++){
        if(v1[i] > minVal and v1[i] < maxVal) break;
    }
    int j; // find the first ele. in v2 within range
    for(j=a2; j<v2.size(); j++){
        if(v2[j] > minVal and v2[j] < maxVal) break;
    }
    // if no ele. found within range (leaf nodes)
    if(i==v1.size() and j==v2.size()) return true;
    if((i==v1.size() and j!=v2.size()) and (i!=v1.size() and j==v2.size())){ // if only 1 vector doesn't have ele. within range
        return false;
    }
    if(v1[i]!=v2[j]) return false; // checking both ele. are equal or not
    return checkId(v1, v2, a1+1, a2+1, v1[i], maxVal) and checkId(v1, v2, a1+1, a2+1, minVal, v1[i]);
}
```

```
bool checkIdenticalBSTs(vector<int>& v1, vector<int> &v2){
    return checkIdenticalBSTHelper(v1, v2, 0, 0, INT_MIN, INT_MAX);
}
```

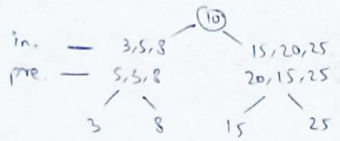
Q) Given preorder traversal of BST, construct the BST.

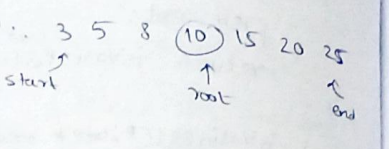
If given preorder: [10, 5, 3, 8, 20, 15, 25]

If inorder traversal of BST: [3, 5, 8, 10, 15, 20, 25]



A) We can inorder by sorting preorder!





Code:

```
Node* bstFromPreorderHelper(vector<int> &preorder, int prestart, int preend,
        vector<int>& inorder, int instart, int inend, unordered_map<int,int> inmap){
    if(prestart > preend or instart > inend) return NULL;
    Node* root = new Node(preorder[prestart]);
    int inroot_idx = inmap[preorder[prestart]];   // returns index at which that ele. is present
    int leftsubtree_ele = inroot_idx - instart;
    root->left = bstFromPreorderHelper(preorder, prestart+1, prestart+leftsubtree_ele,
                    inorder, instart, inroot_idx-1, inmap);
    root->right = bstFromPreorderHelper(preorder, prestart+leftsubtree_ele+1,
                    preend, inorder, inroot_idx+1, inend, inmap);
    return root;
}
Node* bstFromPreorder(vector<int> &preorder){
    vector<int> inorder = preorder;
    sort(inorder.begin(), inorder.end());   // now from preorder we need to find inorder value, so instead of traversing always, we can create map to access
    unordered_map<int,int> inmap;
    for(int i=0; i<inorder.size(); i++){
        inmap[inorder[i]] = i; //storing value & index pairs
    }
    return bstFromPreorderHelper(preorder, 0, preorder.size()-1, inorder,
                    0, inorder.size()-1, inmap);
}
int main(){
    BST bst;
    vector<int> v = {10,5,3,8,20,15,25};
    bst.root = bstFromPreorder(v);
    inorderTraversal(bst.root);
    return 0;
}
```

$T_c = O(n\log n) + O(n)$; $S_c = O(n) + O(n) + O(h)$

(for sorting) (for traversing vector ele.s) ; (vector) (map) (recursive)

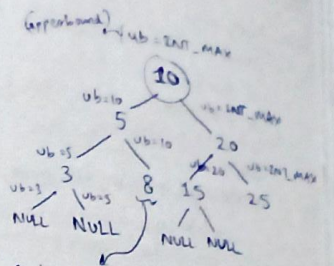
Method-2:

In preorder → [10, 5, 3, 8, 20, 15, 25]

(root, left, right)

root

- value < root → left subtree
- value > root → right subtree



(if value can't be at left, so it's obviously at right → so we check if it's >10 or <(10) only)

$T_c = O(n)$ → for traversing

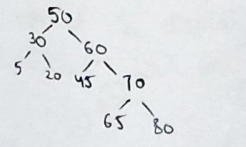
$S_c = O(h)$ → for traversing the recursion stack

Code:

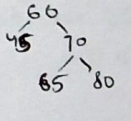
```
Node* bstFromPreorderHelper(vector<int> &preorder, int &index, int upperbound){
    if(index >= preorder.size()) return NULL;
    if(preorder[index] >= upperbound) return NULL;
    Node* root = new Node(preorder[index]);
    index++;
    root->left = bstFromPreorderHelper(preorder, index, root->value);
    root->right = bstFromPreorderHelper(preorder, index, upperbound);
    return root;
}
Node* bstFromPreorder(vector<int> &preorder){
    int index = 0;
    return bstFromPreorderHelper(preorder, index, INT_MAX);
}
```

Q) Given a Binary Tree, write a func. that returns the sum of the largest subtree which is also a BST. If the complete binary tree is BST, then return the sum of the whole tree.

Ip:



Op: 320



A) Brute force: Going to each node & checking if its subtrees are BST or not → $T_c = O(n \times n) = O(n^2)$

Now for a node to be BST

(left subtree max value < Node < right subtree min value)

For leaf node → min.value = INT_MAX, max.value = INT_MIN } → 'NULL' case

Code. {we are calculating max size & sum}

```
class NodeInfo{
    public:
    int minValue;
    int maxValue;
    int maxCurrentSum;
    int size;
    NodeInfo(int min, int max, int sum, int sz){
        minValue = min;
        maxValue = max;
        maxCurrentSum = sum;
        size = sz;
    }
};
int ans = 0;
NodeInfo maxSumBSTHelper(Node* root){
    if(!root) return NodeInfo(INT_MAX, INT_MIN, 0, 0); // leaf node
    NodeInfo leftsubtree = maxSumBSTHelper(root->left);
    NodeInfo rightsubtree = maxSumBSTHelper(root->right);
    if(root->value > leftsubtree.maxValue and root->value < rightsubtree.minValue){
        // root node forms a BST
        int currentSum = leftsubtree.maxCurrentSum + rightsubtree.maxCurrentSum
                         + root->value;
        ans = max(ans, currentSum); // {if we have (-)ve value in node, to avoid counting it we use (as it will ↓ maxSum)}
        int currentSize = leftsubtree.size + rightsubtree.size + 1;
        return NodeInfo(min(leftsubtree.minValue, root->value),
                        max(rightsubtree.maxValue, root->value), currentSum, currentSize);
    } // root node doesn't form a BST
    return NodeInfo(INT_MIN, INT_MAX, max(leftsubtree.maxCurrentSum, rightsubtree.maxCurrentSum),
                    max(leftsubtree.size, rightsubtree.size));
}
vector<int> maxSumAndSizeBST(Node* root){
    NodeInfo result = maxSumBSTHelper(root);
    return {ans, result.size};
}
int main(){
    Node* root = new Node(50);
    root->left = new Node(30);

    vector<int> result = maxSumAndSizeBST(root);
    cout << "Max Sum" << result[0] << "\n Max Size" << result[1];
    return 0;
}
```

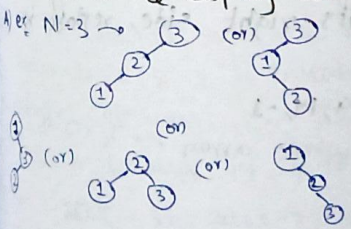
- $T_c = O(n)$, $S_c = O(n)$

Dry run: (for above ex. input in que.)

1) We start at root node (50). Since it's not a leaf node, we recursively call maxSumBSTHelper on its left child (30) & right child (60).
2) For node 30, we again recursively call maxSumBSTHelper on its left child (5) & right child (20).
3) Node 5 is a leaf node, so it returns {INT_MAX, INT_MIN, 0, 0}
4) Node 20 is also a leaf node, so it returns {INT_MAX, INT_MIN, 0, 0}
5) Back to node 30, since 30 is not greater than its right child (20), it doesn't form a BST. So, it returns {INT_MIN, INT_MAX, maxCurrentSum=0, size=0}
6) Similarly for node 60, we recursively call maxSumBSTHelper on its left child (45) & right child (70).
7) Node 45 is a leaf node, so it returns {INT_MAX, INT_MIN, 0, 0}
8) For node 70, we recursively call maxSumBSTHelper on its left child (65) and right child (80).
9) Both nodes 65 & 80 are leaf nodes, so they return {INT_MAX, INT_MIN, 0, 0}
10) Back to node 70, since 70 > maxValue of left child (65) and 70 < minValue of right child (80), it forms a BST. So, it returns {minValue = 65, maxValue = 80, maxCurrentSum = 215, size = 3}
11) Back to node 60, since 60 > maxValue of left child (45) and 60 < minValue of right child (65), it forms a BST. So, it returns {minValue = 45, maxValue = 80, maxCurrentSum = 320, size = 5}.
12) Finally, back to the root node (50), since it doesn't form a BST (50 < minValue of right child = 45), it returns {INT_MIN, INT_MAX, maxCurrentSum = 320, size = 5}.

9) WAP to construct all unique BST's for keys ranging from 1 to N. The program should prompt the user to enter the value of N, & then construct & display all possible BST's for the given range of keys.

Ex: N=3 →



∴ If N=5 → 1 as root, 2 as root, 3 as root, 4 as root, 5 as root] → left subtree = 1,2 ; right subtree = 4,5 (similarly)

N=1

f(1,0)
f(2,1)
NULL NULL

base case
if (start > end)
break;

∴ Logic: from i = 1 to N
i = rootNode
left subtrees = f(1, i-1)
right subtrees = f(i+1, N)
& match every left subtree with every right subtree with 'i' as root node.

Code:

```
vector<Node*> generateTreesHelper(int start, int end){
    vector<Node*> treeList;
    if(start > end){ // base case
        treeList.push_back(NULL);
        return treeList;
    } // recursive case
    for(int i = start; i <= end; i++){
        vector<Node*> leftsubtrees = generateTreesHelper(start, i-1);
        vector<Node*> rightsubtrees = generateTreesHelper(i+1, end);
        for(Node* leftsubtre : leftsubtrees){
            for(Node* rightsubtre : rightsubtrees){
                Node* rootNode = new Node(i);
                rootNode -> left = leftsubtre;
                rootNode -> right = rightsubtre;
                treeList.push_back(rootNode);
            }
        }
    }
}
vector<Node*> generateTrees(int n){
    return generateTreesHelper(1, n);
}
int main(){
    int n; cin>>n;
    vector<Node*> ans = generateTrees(n);
    int count = 0;
    for(Node* root : ans){
        count++;
        cout<<"BST: "<<inorderTraversal(root)<<endl;
    }
    cout<<"Total trees "<<count;
    return 0;
}
```

Q) Given an array of integers, replace every element with the least greater element on it's right side in the array. If there are no greater elements on it's right side, replace it with -1.

ip: 8, 3, 10, 1, 6, 9, 14 op: 9, 6, 14, 6, 9, 14, -1

A) Brute force: checking 1 ele & comparing whole array → $T_c = O(n^2)$

Soln: (using BST) → $T_c = O(n \log n)$

8, 3, 10, 1, 6, 9, 14

→ Traverse from right to left

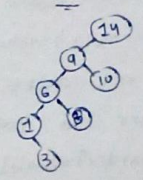
→ If ele. in Left subtree → Least greatest ele. is its prev. element

→ If ele. in right subtree → then ignore!

Code:

```
Node* insertNode(Node* root, int element, int successor){
    if(root == NULL) return new Node(element);
    if(element < root -> value){
        successor = root -> value;
        root -> left = insertNode(root -> left, element, successor);
    }
    else if(element > root -> value){
        root -> right = insertNode(root -> right, element, successor);
    }
    return root;
}
void replaceLeastGreaterElement(vector<int> &v){
    Node* root = NULL;
    for(int i = v.size()-1; i >= 0; i--){
        int successor = -1;
        root = insertNode(root, v[i], successor); // returns tree after inserting v[i] in our BST
        v[i] = successor;
    }
}
int main(){
    int n; cin>>n; vector<int> v(n);
    for(int i = 0; i < n; i++) cin>>v[i];
    replaceLeastGreaterElement(v);
    for(int i = 0; i < n; i++) cout<<v[i]<<" ";
    return 0;
}
```

For que. ip tree:



Dry run

1) for i=6 (ele.=14) → successor = -1, root = NULL, so new node with value 14 is made
vector → 8 3 10 1 6 9 -1

2) for i=5 (ele.=9) → successor = -1, root ≠ NULL & 9<14 → ∴ successor = 14 & 9 is inserted to the left of 14. ∴ vector → 8 3 10 1 6 14 -1

3) for i=4 (ele.=6), successor = -1, root ≠ NULL & 6<14 & 6<9, successor becomes 9 & 6 is inserted to the left of 9 ∴ vector → 8 3 10 1 9 14 -1

4) for i=3 (ele.=1), succ. = -1, root ≠ NULL & since 1<14, 1<9 & 1<6, succ. becomes 6 & 1 is inserted to left of 6. ∴ vector → 8 3 10 6 9 14 -1

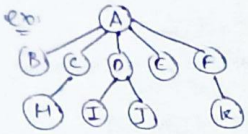
5) for i=2 (ele.=10), succ. = -1, root ≠ NULL & since 10<14 & 10>9, successor becomes 14, & 10 inserted right of 9. ∴ vector → 8 3 14 6 9 14 -1

6) for i=1 (ele.=3), succ = -1, root ≠ NULL & 3<14, 3<10, 3<9 & 3<6 & 3>1
successor → 6 → ∴ vector → 8 6 14 6 9 14 -1

7) for i=0 (ele.=8), succ = -1, root ≠ NULL & 8<14, 8<10, 8<9 & 8>6,
successor → 9 → ∴ vector → 9 6 14 6 9 14 -1.

GENERIC TREES

- Every node has data & list of references to its children nodes.
- ex:



- We also call "generic tree" as "N-array tree"

Implementation:

- Creating a Node class:

[if → children.size() == 0 ↳ Leaf Node]

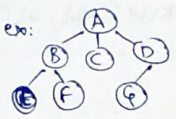
```
class Node{
    int data;
    vector<Node*> children;
    Node(char data){
        this->data = data;
    }
};
```

- works fine for integers!
- if char. is input at node: ↳ char data;

Traversal: {$T_c = O(n)$}

- Preorder:
  1) Print root
  2) Recursively call for child nodes (L-R)

ex:



→ A B E F C D G

Code:

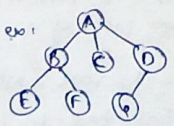
```
Void preorderTraversal(Node* root){
    if(root == NULL) return;
    cout<< root -> data << " ";
    for(Node* child : root -> children){
        preorderTraversal (child);
    }
}
```

o/p

```
int main(){
    Node* root = new Node('A');
    root -> children.push_back (new Node('B'));
                "                ('C'));
                "                ('D'));
    root -> children[0] -> children.push_back (new Node('E'));
                                "              ('F'));
    root -> children[2] -> children.push_back (new Node('G'));
    preorderTraversal (root);
    return 0;
}
```

- Inorder:
  1) Recursively visit all child nodes except last
  2) Print root node
  3) Recursively visit last child node

ex:



→ E B F C A D G

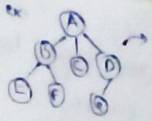
{A: B C D}
↳ {D: G}

Code:

```
void inorderTraversal (Node* root){
    if (root == NULL) return;
    int childNodes = root -> children.size();
    for(int i=0; i<childNodes-1; i++){
        inorderTraversal (root -> children[i]);
    } cout << root -> data << " ";
    if (childNodes > 0) inorderTraversal (root -> children[childNodes-1]);
}
```

- Postorder:
  1) Recursively visit all child nodes
  2) Print root

ex:



→ E F B C G D A

Code:

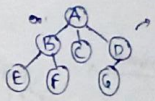
```
void postorderTraversal (Node* root){
    if(root == NULL) return;
    for(Node* child : root -> children){
        postorderTraversal (child);
    }
    cout << root -> data << " ";
}
```

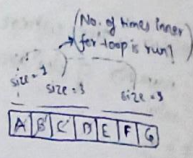
- Levelorder: {we use 'queue' → FIFO}

Code:

```
void levelorderTraversal (Node* root){
    if(root == NULL) return;
    queue <Node*> q;
    q.push(root);
    while(!q.empty()){
        int currentLevelNodes = q.size();
        while (currentLevelNodes--){
            Node* curr = q.front();
            q.pop();
            cout << curr -> data << " ";
            for(Node* child : curr -> children){
                q.push(child);
            }
        } cout << endl;
    }
```

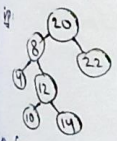
ex:





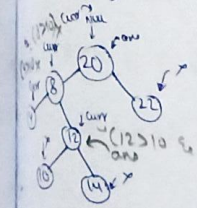
Q) Given a generic tree & an integer x. Find & return the node with the next larger element in the tree i.e find a node just greater than x. Return NULL if no node is present with value greater than x

ex:



x = 10 o/p: 12

for every node → check: if currnode->data > x and currnode->data < ansnode->data
update ansnode



$\therefore T_c = O(n)$ → Traversing each node

$S_c = O(h)$ ↳ height of tree as we call it recursively for child nodes

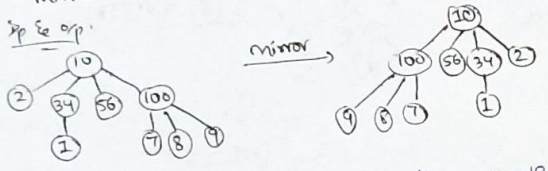
Code:

```
void justGreaterNode(Node* root, int x, Node* &ans){
    if(root==NULL) return;
    if(root->data > x and (ans==NULL or root->data < ans->data)){
        ans = root; //update ans
    }
    for(Node* child : root->children){
        justGreaterNode(child, x, ans);
    }
}
```

```
int main(){
    int x=10;
    Node* ans=NULL;
    justGreaterNode(root, x, ans);
    if(ans==NULL){
        cout<<"NULL";
    }else cout<<ans->data;
    return 0;
}
```

Q) Given a tree where every node contains variable number of children, convert the tree to its mirror.

ip & op:



A) We are going level-wise & then reverse it; 10 : 2 34 56 100 → 10 : 100 56 34 2

So we can call parent → then reverse it's child nodes & then recursively call for its subsequent child nodes

$T_c = O(n)$, $S_c = O(h) + O(m)$ (recursive stack space)

(h: height of tree) (m: max. no. of nodes at a level)

Code:

```
void mirrorTree(Node* root){
    if(root==NULL) return;
    if(root->children.size()<2){
        return;
    }
    reverse(root->children.begin(), root->children.end());
    for(Node* child : root->children){
        mirrorTree(child);
    }
}
int main(){
    :
    levelorderTraversal(root);
    mirrorTree(root);
    levelorderTraversal(root);
}
```

// if 0, 1 child nodes are present, no need to reverse

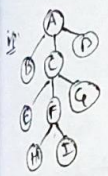
Q) Serialize & Deserialize an N-ary tree.

Serialization = process of converting an object into a format that can be stored / transmitted.

Deserialization = process of converting serialized string back into an object

Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

Ex: Sample way of serializing : node_val : num_children child_1_val, ... , child_n_val



A : 3 : B, C, D
B : 0
C : 3 : E, F, G
E : 0
F : 2 : H, I
H : 0
I : 0
G : 0
D : 0

→ serialised data

A)

```
void serialiseTree(Node* root, string &ans){
    if(root==NULL) return;
    ans += to_string(root->data) + ":" + to_string(root->children.size()) + ":";
    for(Node* child : root->children){
        ans += to_string(child->data) + ",";
    }
    if(root->children.size()!=0) ans.pop_back();
    ans += "\n";
    for(Node* child : root->children){
        serialiseTree(child);
    }
}
```

issue: after leaf node also it will add ","

removes last ',' & when root->children.size()==0, it won't remove ';'

→ next line

```
int main(){
    :
    string serializedTree = "";
    serialiseTree(root, serializedTree);
    cout << serializedTree << endl;
}
```

Now we have to convert serialised string back to normal;

A : 3 : B, C, D \n

We will break our string until we find ':' → no. of child nodes
':' → child is present
'\n' → new Node is next

Read code for better understanding;

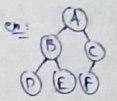
Code:

```
Node* deserialiseTreeHelper(int nodeValue, unordered_map<int,string> mp){
    Node* node = new Node(nodeValue);
    string nodeStr = mp[nodeValue];
    if(nodeStr[0]=='0') return node;
    int breakPos2 = nodeStr.find(':');
    int childNodesNumber = stoi(nodeStr.substr(0,breakPos2));
    string childNodesStr = nodeStr.substr(breakPos2+1);
    int start = 0;
    for(int i=0; i<childNodesNumber; i++){
        int end = childNodesStr.find(',', start);
        if(end==string::npos) end = childNodesStr.size();
        int childNodeValue = stoi(childNodesStr.substr(start, end));
        node->children.push_back(deserialiseTreeHelper(childNodeValue, mp));
        start = end+1;
    }
    return node;
}
Node* deserialiseTree(string serialisedStr){
    if(serialisedStr=="") return NULL;
    unordered_map<int, string> mp;
    int start = 0;
    for(int i=0; i<serialisedStr.size(); i++){
        if(serialisedStr[i]=='\n'){
            string nodeStr = serialisedStr.substr(start, i-start);
            int breakPos1 = nodeStr.find(':');
            int nodeValue = stoi(nodeStr.substr(0, breakPos1));
            mp[10]= nodeStr.substr(breakPos1+1);
            start = i+1;
        }
    }
    int rootNodeValue = stoi(serialisedStr.substr(0, serialisedStr.find(
    return deserialiseTreeHelper(rootNodeValue, mp);
}
int main(){
    Node* deserialise = deserialiseTree(serialisedTree);
    levelOrderTraversal(deserialise);
}
```

HEAPS

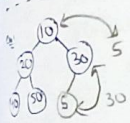
- **Binary Heap:** → is a complete binary tree → all levels will be filled completely except maybe last level & last level has keys as left as possible.
  - → 2 types:
    - **Min Heap:** parent node has smaller value than child nodes
    - **Max Heap:** parent node has larger value than child nodes

- **Representation of Heap Using Arrays:**
  - last Node will be at index 1
  - for node at index 'i'; ⇒ left child = 2*i
    ⇒ right child = 2*i+1
  - for node at index 'i' ⇒ parent node = i/2

eg: 

= | A | B | C | D | E | F |
1 2 3 4 5 6

- **Insertion in Min Heap:**



x = 5

→ Add '5' to last node; | 10 | 20 | 30 | 40 | 50 | 5 |
1 2 3 4 5 6

→ Compare '5' with its parent → p(5) = arr[6/2] = arr[3] = 30 (if less → swap)

→ p(5) = arr[3/2] = arr[1] = 10

(Height of complete bin. tree) ∴ $T_c = O(\log n)$ → worst case → for insertion & deletion of both type of heaps
$\Omega(1)$ → best case

size will get changed, so we passed by ref.

```
const int N = 1e3
void insertMinHeap(int minHeap[], int &size, int val){
    size++;
    minHeap[size] = val;
    int curr = size;
    while(curr/2 >0 and minHeap[curr/2]> minHeap[curr]){   // checks if 'curr' has reached root node!
        swap(minHeap[curr/2], minHeap[curr]);
        curr = curr/2;
    }
}
```

```
int main(){
    int minHeap[N] = {-1,10,20,30,40,50};
    int size = 5, val = 7;
    insertMinHeap(minHeap, size, val);
    for(int i=0; i<size; i++){
        cout<<minHeap[i]<<" ";
    }
    return 0;
}
```

op: -1 7 20 10 40 50 30

- **Insertion in max Heap:**

x = 100 → checks parent node val. → if '>', then swap!

```
void insertMaxHeap(int maxHeap[], int &size, int val){
    size++;
    maxHeap[size] = val;
    int curr = size;
    while(curr/2 >0 and maxHeap[curr/2]<maxHeap[curr]){
        swap(maxHeap[curr/2], maxHeap[curr]);
        curr = curr/2;
    }
}
```

• Deletion in a min Heap:

Here, we can only delete our root node!

Steps:
1) Swap last node ↔ root node
2) Compare new root node with it's left/right child, find min child & swap!

Code:

```
void removeMinHeap(int minHeap[], int &size){
    minHeap[1] = minHeap[size];
    size--;
    int curr = 1;                          // point root node
    while(2*curr <= size){                 // 2*curr <= size → has minimum left child
        int leftchild = 2*curr;            // but we are not sure if we have right
        int rightchild = 2*curr+1;         // child or not, so we explicitly check again
        int minchild = leftchild;
        if(rightchild <= size and minHeap[leftchild] > minHeap[rightchild]){
            minchild = rightchild;
        }
        if(minHeap[minchild] >= minHeap[curr]){
            return;
        }
        else{
            swap(minHeap[minchild], minHeap[curr]);
            curr = minchild;
        }
    }                                      // looping till curr node reaches leaf node
}
```

• Deletion in a max Heap:

Here also, we can only delete our root node!

Steps:
1) Swap last node ↔ root node
2) Compare new root with left/right child, find max child & swap!

Code:

```
void removeMaxHeap(int maxHeap[], int &size){
        :
    if( ---- and maxHeap[rightchild] > maxHeap[leftchild]) maxchild = rightchild
    if(maxHeap[maxchild] >= maxHeap[curr]){
        swap(maxHeap[maxchild], maxHeap[curr]);
        curr = maxchild;
    }
    else{
        return;
    }
}
```

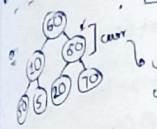
• Heapify:

properties of minHeap/maxHeap

• Using Heapify to generate a min Heap:

60	10	80	50	5	20	70

curr



→ we check from last before nodes which are not leaf nodes

→ Now we check & swap positions!

60	10	20	50	5	80	70

curr ↓

60	5	20	50	10	80	70

curr ↓

5	60	20	50	10	80	70

curr ↓

5	10	20	50	60	80	70

curr-- since it didn't become leaf yet, so we checked again & swapped

{ leaf nodes start from $(n/2+1)$,
last non-leaf node $= (n/2)$ }

Code:

```
void heapify(int arr[], int &size, int curr){
    while(2*curr <= size){
        int leftchild = 2*curr;
        int rightchild = 2*curr+1;
        int minchild = leftchild;
        if(rightchild <= size and arr[rightchild] < arr[minchild]){
            minchild = rightchild;
        }
        if(arr[minchild] >= arr[curr]){
            return;
        }
        else{
            swap(arr[minchild], arr[curr]);
            curr = minchild;
        }
    }
}
```

```
int const N = 1e3;
int main(){
    int arr[N] = {-1,60,10,80,50,5,20,70};
    int size = 7;
    for(int i = size/2; i > 0; i--){
        heapify(arr, size, i);
    }
    for(int i = 1; i <= size; i++){
        cout << arr[i] << " ";
    }
    return 0;
}
```

Op: 5 10 20 50 60 80 70

Similarly we can use heapify to create max heap!

→ Deleting element from minheap: smallest value
→ Deleting element from maxheap: largest value

Heap Sort:

1) Convert the array into heap data structure using heapify.
2) One by one delete the root node of the heap & replace it with the last node in the heap & then heapify the root of the heap. Repeat the process till the size of heap is greater than 1.

60, 10, 80, 50, 5, 20, 70 → Heapify it → : 80, 50, 70, 10, 5, 20, 60

↓

Now delete root node : 60, 50, 70, 10, 5, 20 | 80 (heapify)

Time: O(nlogn)

Space saved ✓ — Similarly delete root node & place it end of array of heap!

Already sorted

Code

```
Void heapify(int arr[], int &size, int curr){
    while(2*curr <= size){
        :
        if(rightchild<=size and arr[r-]>arr[max-]){
            maxchild = rightchild;
        }
        if(arr[max-] <= arr[curr]){
            return;
        }
        else{
            swap(arr[maxchild], arr[curr]);
            curr = maxchild;
        }
    }
}
```

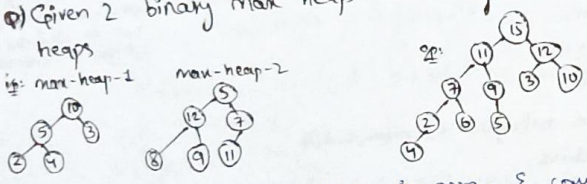
```
void remove(int arr[], int &size){
    swap(arr[1], arr[size]);
    size--;
    heapify(arr, size, 1);
}
```

(root)

```
void heapSort(int arr[], int n){
    // a) heapify parent node
    for(int i = n/2; i>0; i--){
        heapify(arr, n, i);
    }
    // deleting ele.s from max heap
    while(n>0){ until size -> 0
        remove(arr, n);
    }
}
```

In int main() → array, size } → heapify(arr, n) → print 1 to <=n !

Q) Given 2 binary max heaps as arrays, the task is to merge the given heaps



o/p:

A) Approach: → Make both i/p heaps in array & combine them into new array
→ Now apply heapify on new array!

Code → $T_c = O(n+m)$

```
void heapify(vector<int> &v, int size, int curr){
    while(2*curr+1 < size){
        int leftchild = 2*curr+1;
        int rightchild = 2*curr+2;        } → for '0' based indexing
        int maxchild = leftchild;
        if(rightchild < size and v[rightchild] > v[leftchild]){
            maxchild = rightchild;
        }
        if(v[maxchild] <= v[curr]) return;
        else{
            swap(v[maxchild], v[curr]);
            curr = maxchild;
        }
    }
}
```

```
vector<int> mergeHeaps(vector<int> &a, vector<int> &b, int n, int m){
    vector<int> mergeHeap;
    for(auto ele : a) mergeHeap.push_back(ele);
    for(auto ele : b) mergeHeap.push_back(ele);
    int lastParentNode = (n+m)/2 - 1;   } → looping all parent nodes
    for(int i = lastParentNode; i>=0; i--){
        heapify(mergeHeap, n+m, i);
    }
    return mergeHeap;
}
```

Q) Given arr[] & int 'k', k < size of array. Find k^{th} smallest ele. in array, given all array ele.'s are distinct.

Easy Soln: sort array → arr[k-1] is answer.

Code Soln:

(a) Heapify array → 1) Create minHeap 2) Remove (k-1) elements

Sir → Heapify

```
int kthSmallestElement(int arr[], int size, int k){
    for(int i = size/2 - 1; i>=0; i--){ //create min heap
        heapify(arr, size, i);          '0' based indexing
    }
    k = k-1; → remove k-1 ele
    while(k--){
        remove(arr, size);
    }
    return arr[0];
}
```

```
void remove(int arr[], int &size){
    swap(arr[0], arr[size-1]);
    size--;
    heapify(arr, size, 0);   0 based indexing
}
```

```
void heapify( - - - ){
    L = 2*curr+1
    R = 2*curr+2
    minchild = L
    :
}
```

Q) arr[N], k1, k2 = given. Find sum of ele. btw $k1^{th}$ & $k2^{th}$ smallest ele. of array.
It may be assumed that: (1 <= k1 & k2 <= n).

eg: N=7
{20, 8, 22, 4, 12, 10, 14}
k1=3, k2=6

eg: 26

(1 based indexing)

A) ① either sort array & find!

② → a) Min Heap
b) k1 ele.'s remove
c) remove ele.'s 1 by 1 & add to sum : k2-k1-1 times

A) Code:

```
void heapify( - - ){
    // same as prev
}
void remove(int arr[], int &size){
    int removedVal = arr[0];
    swap(arr[0], arr[size-1]);   } → 0 based indexing
    size--;
    heapify(arr, size, 0);
    return removedVal;
}
```

```
int sumOfSmallest(int arr[], int size, int k1, int k2){
    for(int i = size/2 - 1; i>=0; i--){
        heapify(arr, size, i);
    }
    int sum = 0
    int elements = k2 - k1 - 1;
    // remove k1 elements
    while(k1--){
        remove(arr, size);
    }
    // calculate sum by removing ele.'s
    while(elements--){
        sum += remove(arr, size);
    }
    return sum;
}
```

```
int main(){
    int arr[7] = {. . . . . };
    size = 7;
    k1 = 3;
    k2 = 6;
    cout << sumOfSmallest(arr, size, k1, k2);
    return 0;
}
```

PRIORITY QUEUES

- STL container - implement heap data structure
- Every element will have some priority
- Queue where front element will have the highest priority (smallest, largest)
- Syntax: (#include<queue>)
  - → priority_queue <datatype> pq;
  - priority_queue <int> pq1; ⟹ Max Heap by default
  - priority_queue <int, vector<int>, greater<int>> pq2; ⟹ Min Heap

- Insertion: pq.push(1), pq.push(2), pq.push(3) → $T_c = O(\log N)$ → 3 | 1 | 2 (max heap)
- Deletion: pq.pop() removes the front element. ex: 3 | 1 | 2 → 2 | 1
- → pq.top(): returns the top element
- → pq.size(): returns the size of priority queue
- → pq.empty(): returns boolean value
  - → 1 = if empty
  - → 0 = if not

Q) arr[], int k, return kth largest ele. in array. (in sorted order)

ip: [3,7,2,9,5], k=3, ans=5

A) ① Sort array & find kth largest ele.
② Make max heap, remove (k-1) elements & kth ele. is ans.
③ Make min heap, remove (n-k) smallest ele.s → we will get kth largest ele.
→ $T_c = O(N \log k)$ (min heap, k: size of queue)

Code:

```
int kthLargestEle(vector<int> a, int n, int k){
    priority_queue<int, vector<int>, greater<int>> pq; → min heap
    for(int i=0; i<n; i++){
        pq.push(a[i]);
        if(pq.size() > k) pq.pop(); → removing smallest ele.s out of k+1 ele.s
    }
    return pq.top();
}
```

Q) You're given set of point in X-Y plane. Your final, 'k' closest points to origin using manhattan distance. {No 2 points same distance from origin}

ip: n=6, k=3

5 3 = 5+3 = 8
1 2 = 1+2 = 3
3 4 = 3+4 = 7
5 6 = 5+6 = 11
7 8 = 7+8 = 15
9 10 = 9+10 = 19

op: 1 2, 3 4, 5 3

A) 1) Store in min heap → starting ele.s are ans.
2) Store in max heap & we remove higher ele.s until 'k' ele.s left!

pair<distance, coordinates>

```
vector<pair<int,int>> kClosestPoints(vector<pair<int,int>>&pts, int n, int k){
    priority_queue<pair<int, pair<int,int>>> pq; // max heap
    for(auto &pt: pts){
        int distance = pt.first + pt.second;
        pq.push(make_pair(distance, pt));
        if(pq.size() > k){
            pq.pop(); // remove largest distance pt in (k+1) pts
        }
    }
    vector<pair<int,int>> ans(k);
    while(!pq.empty()){
        ans[pq.size()-1] = pq.top().second;
        pq.pop();
    }
    return ans;
}
```

heap = T2B, filling vector = R2L, as topmost = largest val

```
int main(){
    int n,k;
    cin>>n>>k;
    vector<pair<int,int>> pts(n);
    vector<pair<int,int>> anspts;
    for(auto&pt:pts){
        cin>> pt.first >>pt.second;
    }
    anspts = kClosestPoints(pts, n, k);
    for(auto pt: anspts){
        cout<<pt.first<<" "<<pt.second<<endl;
    }
    return 0;
}
```

Q) Operation: remove 2 smallest ele. from array, multiply & return product. We can do 'k' operations on it. Return largest ele. after 'k' operations!

ip: n=5, k=3, {2 4 3 1 5}, op: 20

A) → Min Heap
→ Pop 2 ele.s, push their product
→ Pop until 1 ele. is left

```
int largestPoint(vector<int> a, int n, int k){
    priority_queue<int, ve - - - > pq; → Min heap
    for(auto i: a) pq.push(i);
    while(k--){
        int fE = pq.top();
        pq.pop();
        int sE = pq.top();
        pq.pop();
        int prod = fE * sE;
        pq.push(prod);
    }
    while(pq.size() > 1) pq.pop(); → finding largest element in pq
    return pq.top();
}
```

Q) Given character array tasks, representing the tasks CPU needs to do, where each letter represents a different task. Task could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either 1 task or just be idle.
However, there's non(-)ve integer n that represents the cooldown period btw 2 same tasks (same letter in array), so there must be atleast 'n' units of time btw any 2 same tasks.
Return least no. of units of times that the CPU will take to finish all given tasks.

ip: ['A','A','A','B','B','B'], n:2, op: 8

A → B → idle → A → B → idle → A → B

There is atleast 2 units of time btw any 2 same tasks.

A) ex: n=2 ; A B X
t=1 2 3
∴ (n+1) time → same task can't be executed!

ex: A=2, B=1, n=2 → BA*A ✓ → t=5; → A B * A ✓ → t=4
∴ Always must start with high freq.

So → we will store freq. of each task
→ freq. inserted to max heap → priority wise!
→ loop till one time frame & repeat

Code:

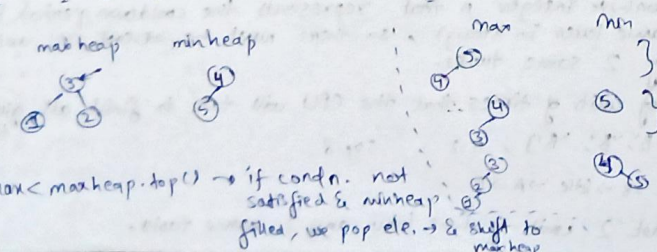
```
int leastInterval(vector<char> tasks, int n){        // n = cooldown time
    unordered_map<char,int> taskFreq; // count freq. of tasks
    for(auto t: tasks) taskFreq[t]++;
    priority_queue<int> pq; // insert freq. into max heap
    for(auto e: taskFreq) pq.push(e.second);
    int totalTime=0 // finding time until pq is empty
    while(!pq.empty()){
        vector<int> temp;
        for(int i=0; i<=n; i++){ // looping 1 time frame => n+1 units of time
            if(!pq.empty()){
                int freq = pq.top();
                pq.pop();
                if(freq>1) temp.push_back(freq-1);   // adding remaining tasks in temp. vector
            }
            total time++;
            if(pq.empty() and temp.empty()){
                return totalTime; // all tasks executed
            }
        }
        for(auto t: temp) pq.push(t); // adding back remaining tasks from temp vector to pq
    }
    return totalTime;
}
```

Q) Given stream of integers, find median of stream?

A) ex: 1 2 3 4 5 6 → we must sort ele.'s & add middle

else: 1) Convert first half to maxheap } → maxheap size can be atmost greater than 1
2) Second half to minheap }

ex: 1 2 3 4 5 ; 5 4 3 2 1



max < maxheap.top() → if condn. not satisfied & minheap filled, we pop ele. → & shift to maxheap

∴ T_c = median = O(1), adding in heap = O(log n), adding n ele. in stream = O(nlogn)

```
priority_queue<int> maxpq; // stores lower half of stream
priority_queue<int, vector<int>, greater<int>> minpq; // stores greater half of stream
void addNum(int num){
    if(maxpq.empty() or num<=maxpq.top()){
        maxpq.push(num);
    }
    else minpq.push(num);
    //rebalance the pq
    if(maxpq.size()>minpq.size()+1){
        minpq.push(maxpq.top());
        maxpq.pop();
    }
    else if(minpq.size()>maxpq.size()){
        maxpq.push(minpq.top());
        minpq.pop();
    }
}
```

```
double findMedian(){
    if(minpq.size()==maxpq.size()){
        return (minpq.top()+maxpq.top())/2.0;
    }
    else{
        return maxpq.top(); // so that float can also be represented
    }
}
int main(){
    addNum(1);addNum(2);addNum(3);addNum(4);addNum(5);addNum(6);
    cout<<findMedian();
}
```

GREEDY - 1

- They build soln. piece by piece | we select the smallest value & build answers
- 4 approach in general;
  - Divide and Conquer
  - Brute force
  - DP
  - Greedy

Q) Fractional Knapsack

Given wt's & profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get max. total profit in the knapsack. In fractional knapsack, we can break items for maximizing the total value of the knapsack.

ex: arr[] = {{60,10},{100,20},{120,30}}, W=50 ; op: 240

sol: By taking item of weight 10 & 20 kg & 2/3 fraction of 30 kg. Hence total price = 60+100+(2/3)(120) = 240

A) Knapsack ~ bag!

a) Now we can check all possibilities → brute force

b) We need min weight & max value, so we find; value/weight

ex: 60/10 = 6, 100/20 = 5, 120/30 = 4 → (v/w) = high → consider it!

∴ We sort (v/w) in dec. order → T_c = O(nlogn)

S_c = O(sorting)

Code:

```
struct Item {
    int value;
    int weight;
};
bool cmp(Item i1, Item i2){
    // custom comparator for sorting
    double vw_i1 = static_cast<double>(i1.value)/i1.weight;
    double vw_i2 = static_cast<double>(i2.value)/i2.weight;
    return vw_i1 > vw_i2;
}
double fractional(int W, vector<Item> &items){
    double ans = 0;
    sort(items.begin(), items.end(), cmp);
    for(const auto &item: items){
        if(item.weight <= W){ // pick whole item in knapsack
            ans += item.value;
            W -= item.weight;
        }
        else{
            double fraction = static_cast<double>(W)/item.weight;
            ans += fraction * item.value;
            W = 0;
        }
    }
    return ans;
}
```

(all public ← (by default), in class, variables : private)

converts integer → double

in general, used to convert one type to another

```
int main(){
    int n, W;
    cin>>n>>W;
    vector<Item> items;
    for(int i=0; i<n; i++){
        int v, w;
        cin>>v>>w;
        Item it;
        it.value = v;
        it.weight = w;
        items.push_back(it);
    }
    cout<<fractional(W, items);
    return 0;
}
```

Q) Maximum meetings in one room

There is 1 meeting room in a firm. There are N meetings in the form of (S[i], F[i]) ; S[i] = start time of meeting 'i', F[i] = finish time of meeting 'i'. Find max. no. of meetings that can be taken in room. Print all meeting no.s

Ex: S[] = {1,3,0,5,8,5}
F[] = {2,4,6,7,9,9}

Ans: 1 2 4 5

Ans: eg: sort as per 2nd val → (1,2) (3,4) (0,6) (5,7) (8,9) (5,9)

check if next meet start time > last meet end time ✓

$T_C = O(n \log n)$

Code:

```
struct meeting{
    int start;
    int end;
    int idx;
};
bool cmp(meeting m1, meeting m2){
    return m1.end < m2.end;
}
void maxMeet(vector<meeting> &arr){
    sort(arr.begin(), arr.end(), cmp);
    cout<<arr[0]<<" ";
    meeting last = arr[0];
    for(int i=1; i<arr.size(); i++){
        if(arr[i].start > last.end){
            cout<<arr[i].idx<<" ";
            last = arr[i];
        }
    }
}
int main(){
    int n; cin>>n; int i=0; vector<meeting> arr;
    while(n--){
        int s,e; cin>>s>>e; i++; meeting m;
        m.start = s; m.end = e; m.idx = i; arr.push_back(m);
    }
    maxMeet(arr);
    return 0;
}
```

Q) Activity Selection Problem:

Given N activities with their start & finish day given in array start[] & end[]. Select max no. of activities that can be performed by a single person, assuming that a person can only work on a single activity at a given day.

Note: Duration of activity includes both starting & ending day.

Ex: N=4
start[] = {1,3,2,5}
end[] = {2,4,3,6}

Ans: 3

Ex: A person can perform activities 1, 2, 4

A) Similar as prev. prob. ! here instead of printing idx's, we do count++ & print count in end.

Q) Survive on Island:

You are person in an island. There is only one shop in this island, this shop is open on all days of week except Sunday. Consider

S = no. of days req. to survive
M = food unit req. each day to survive

⇒ N = max. food unit you can buy each day

find min. no. of days you need to buy food from shop so I can survive next 'S' days or determine it isn't possible to survive.

A) Food we can buy in one week = 6N
Food consumption in a week = 7M

$\left(S - \frac{S}{7}\right) * N \geq S * M$

$\frac{S}{7}$: No. of Sundays

this gives us total unit of food we can purchase

No. of days we can purchase food

if → $\left(S - \frac{S}{7}\right) < S * M$ return -1;

else return: ceil$\left(\frac{S*M}{N}\right)$

Greedy approach:

If you can survive 1 week, you can survive multiple weeks & we stock the food as much as we can, so that food doesn't run out!

Q) Given N (very large), the task is to print the largest palindromic num. obtained by permutting the digits of N. If not possible, print appro. message

Ex: 313551 Ans: 531135

A) Brute force → check all possibilities → $T_C = O(n!)$

① Store freq. of each digit & arrange them high to low wise!

Ex: 3→2, 2→2, 1→2 } 321123 ✓ , 2→2, 3→1 } 232

→ if more than 1 odd, not possible

Code: → $T_c = O(n)$ → string length traversal

```
bool isPossible(unordered_map<int,int> &mp){
    int count=0;
    for(int i=0; i<=9; i++){
        if(mp.count(i)){
            if(mp[i]%2!=0) count++;
            if(count>1) return false;
        }
    }
    return true;
}
```

→ Iterates through digit 0-9 & checks if the count of each digit = even. If its odd, count++ → If more than 1 digit odd count, returns false

```
int main(){
    cout<< max_palindrome("____");
    return 0;
}
```

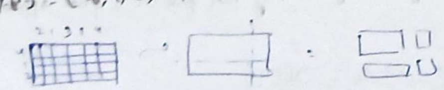
```
string max_palindrome(string s){
    int l= s.size();
    unordered_map<int,int> mp;
    for(int i=0; i<l; i++){
        mp[s[i]-'0']++;      } stores freq of each digit
    }
    if(!isPossible(mp)){
        return "NP";
    }
    vector<char> v(l);
    int front=0;
    for(int i=9; i>=0; i--){
        if(mp[i]%2!=0){          count of digit is odd → placing at centre
            v[l/2] = char(i+48);
            mp[i]--;
        }
        while(mp[i]>0){
            v[front]= char(i+48);
            v[l-front-1] = char(i+48);
            mp[i] -=2;
            front++;
        }
    }
    string res ="";
    for(int i=0; i<v.size(); i++){
        res+ = v[i];
    }
    return res;
}
```

GREEDY - 2

Q) A board : length M, width N. Break this board into M*N squares such that cost of breaking is minimum. Cutting cost for each edge will be given for board in 2 arrays x[] & y[]. Choose such a sequence of cutting such that cost is minimized. Return minimized cost.

I/p: M=6, N=4
x[] = {2,1,3,1,4}
y[] = {4,1,2}

O/p: 42

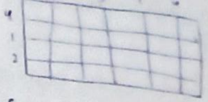
A) ex: 

} When we cut from one side, then whole figure gets divided into 2. So if we try to cut in same direction again, we have to cut twice as pieces = 2.

→ If we have a cut later will be done on multiple rectangle, then cost of cut should be considered on each rectangle

→ vertical cuts increase horizontal blocks.
horizontal cuts increase vertical blocks

→ Order will be high to low, as high cost into low pieces = min. price



hori cuts on blocks [hz = x2
vz = 1 (vertical cut on blocks)

x = 4 3 2 1 1
y = 4 2 1
} use 2 pointer approach to traverse

cost = 4x1 + 4x2 + ... (similarly)
(then hz++) (then vz++)

$S_c = O(1)$ & $T_c = O(n\log n) + O(n\log n)$ (for sorting)

Code:

```
#define ll long long int
bool cmp(int x, int y){
    return x > y;
}
ll minCost(int n,int m, vector<ll>&vertical, vector<ll>&horizontal){
    sort(vertical.begin(), vertical.end(), cmp);
    sort(horizontal.begin(), horizontal.end(), cmp);
    int hz = 1, vz = 1;
    int h=0, v=0;
    ll ans= 0;
    while(h< horizontal.size() and v<vertical.size()){
        if(vertical[v] > horizontal[h]){
            ans += vertical[v] * vz;
            hz++; v++;
        }
        else{
            ans += horizontal[h] * hz;
            vz++; h++;
        }
    }
    while(h< horizontal.size()){
        ans+ = horizontal[h] * hz;
        vz++; h++;
    }
    while(v< vertical.size()){
        ans+= vertical[v] * vz;
        hz++; v++;
    }
    return ans;
}
```

```
int main(){
    //input -> n, m, vectors
    //          horizontal [0,n-1]   vertical [0,m-1]
    cout<< minCost(n,m,vertical,horizontal);
}
```

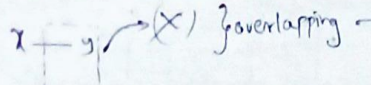
Q) Given array of 'intervals', where intervals[i] = [start_i, end_i], return min. no. of intervals you need to remove to make rest of intervals non-overlapping

I/p: intervals = [[1,2],[2,3],[3,4],[1,3]]
O/p: 1

I/p2: in = [[1,2],[1,2],[1,2]]
O/p2: 2

I/p3: in = [[1,2],[2,3]]
O/p3: 0

A)



(X) } overlapping → check: end > curr.start

; we sort our interval as per end wise;

[(1,2),(1,3),(2,3),(3,4)]

✓ ✗ ✓ ✓

Code:

```
bool comp(vector<int>&i1, vector<int>&i2){   // passing by reference to avoid space & time in making copies of vector!
    return i1[1]< i2[1];
}
int eraseOverlapIntervals(vector<vector<int>>& intervals){
    sort(intervals.begin(), intervals.end(), comp);
    int lastPicked=0, count=0;
    for(int i=1; i<intervals.size(); i++){
        if(intervals[lastPicked][1] > intervals[i][0]){
            count++; //overlap detected, remove current
        }
        else{
            lastPicked=i; //no overlap, pick current one
        }
    }
    return count;
}
```

B) Smallest Number:

Find smallest no. with given sum of digits as 'S' & no. of digits as 'D'

ex: S=9, D=2 ex: 18

A) Filling LSB with highest digit & MSB with lowest digit } _ _ _ _ _ _ _ (MSB ↑, LSB ↑)

Code:

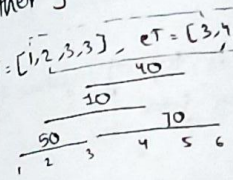
```
int main(){
    int d,s,i;
    cin>>d>>s;
    vector<char> v(d,'0');   // size d, initialised with '0'
    if(9*d < s) return 0;
    else{s--;
        for(i=d-1; i>=0; i--){
            if(s<9) break;
            v[i]='9';
            s-=9;
        }
        v[0]='1';   // MSB
        v[i]=char(s+'0');
    }
    else if(9*d==s){
        for(int i=d-1; i>=0; i--){
            v[i]='9';
            s-=9;
        }
    }
    for(int i=0; i<v.size(); i++){
        cout<<v[i];
    }
    return 0;
}
```

Q) We have 'n' jobs, every job scheduled from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given startTime, endTime & profit arrays, return the max. profit you can take such that there are no 2 jobs in the subset with overlapping time range.

If you choose a job that ends at time x, you will be able to start another job that starts at time x.

ex: ST=[1,2,3,3], ET=[3,4,5,6], profit=[50,10,40,70] } o/p: 120



Subset chosen = 1st & 4th job!

A) Code:

```
int jobScheduling(vector<int> &startTime, vector<int> &endTime, vector<int> &profit){
    vector<vector<int>> jobs;
    for(int i=0; i<startTime.size(); i++){
        jobs.push_back({startTime[i], endTime[i], profit[i]});
    }
    sort(jobs.begin(), jobs.end());   // sort wrt 'startTime' ascending wise
    priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
    int mp=0;   // keeps track of job with earliest ending times; mp → max profit
    for(int i=0; i<jobs.size(); i++){
        int start=jobs[i][0];
        int end=jobs[i][1];
        int profit=jobs[i][2];
        while(!pq.empty() and start>=pq.top()[0]){
            mp=max(mp, pq.top()[1]);
            pq.pop();
        }
        pq.push({end, profit+mp});
    }
    while(!pq.empty()){
        mp=max(mp, pq.top()[1]);
        pq.pop();
    }
    return mp;
}
```

Dry run:

1) jobs = {{1,3,50}, {2,4,10}, {3,5,40}, {3,6,70}}
2) Sorting → jobs = {{1,3,50}, {2,4,10}, {3,5,40}, {3,6,70}}
3) Initialize → pq={}, mp=0

4) Iterate through jobs:

→ First job '{1,3,50}' ⇒ Since prio. queue is empty, push {3,50} onto pq

→ Second job '{2,4,10}' ⇒ No jobs in 'pq' have ending time earlier/equal to start time of current job, so push {4,10+0} onto pq
↳ Queue: [{3,50}, {4,10}]

→ 3rd job {3,5,40} ⇒ Job '{3,50}' is popped & max. profit = 50. Now 3rd job added with curr. max profit added to queue
↳ [{4,10}, {5,90}]

→ 4th job {3,6,70} ⇒ start time < end time of job at top of queue {4,10}; So it's added without popping any jobs.

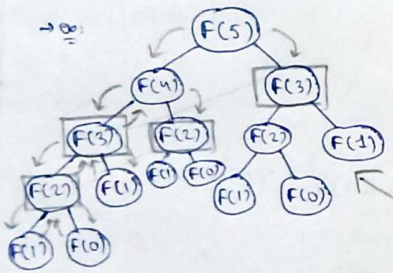
∴ Queue → [{4,10}, {5,90}, {6,120}].

→ Finding max profit → ans = 120

DP-1

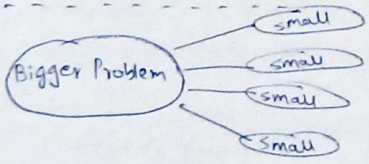
- DP: → If you already know about something, don't recompute it.
  → reuse more, repeat less
- Overlapping Subproblems: (ex: fibonacci)

→ ex:



Now here we need value of F(3) & F(2) again! We have already calculated it once though! So this is overlapping subproblems (or) repeating subproblems

- Optimal Substructure:



Here we divide bigger problem into smaller subproblems & now we find the optimal solution of these subproblems which results in optimal soln. of bigger problem.

GREEDY	DP
→ A greedy algo. builds up soln. piece by piece, always choosing the next piece that offers the most obvious & immediate benefits	→ A DP algo. builds up soln. to a prob. by solving it's subproblems recursively
→ Useful for solving prob.'s where making local optimal choices at each step leads to a global optimum	→ It's used where optimal soln. can be obtained by combining optimal soln.'s to subproblems.
→ Doesn't necessarily consider the future consequences of current choice	→ DP stores the solution to subproblems & reuses them when necessary to avoid solving the same prob. multiple times

→ Greedy approach is simpler, but may not always find the optimal solution

→ Optimal soln. but is slower & more complex

- Procedure: → State of the DP
  → How the subproblems are related/identify formula btw them
  → Storing the results once completed

ex: General fibonacci code →
```
int f(int n){
    if(n==0 or n==1) return n;
    return f(n-1)+f(n-2);
}
```

Now while using DP;

Code:
```
vector<int> dp;   → used to store ans. which came first time
int f(int n){
    if(n==0 or n==1) return n;
    if(dp[n]!=-1) return dp[n];
    return dp[n]=f(n-1)+f(n-2);
}
```
↳ $S_c = O(n)$

```
int main(){
    int n; cin>>n;
    dp.resize(n+1,-1);
    cout<<f(n);
}
```

prev. f(5) → 5 3 2 1 0 → we need to store these ans., so for num 'n', we need 'n+1' space

* refer overlapping subproblems diagram

dp[i] = '-1' → means i^th state not yet computed
dp[i] ≠ '-1' → means i^th state is already precomputed

So here we are going "top to bottom" ⇒ Top Down DP (aka "Memoisation") — start with big prob., move to small prob.'s

Similarly we have "bottom up" approach ⇒ Bottom Up DP (aka "Tabulation") — start with small prob.'s, move to big prob.'s

Code: bottom up
```
int fbu(int n){
    if(n==0 or n==1) return n;
    dp.resize(n+1,-1);
    dp[0]=0; dp[1]=1;
    for(int i=2; i<=n; i++){
        dp[i]=dp[i-1]+dp[i-2];
    }
    return dp[n];
}
```
↳ $T = O(n)$

→ optimised
```
int fbuo(int n){
    if(n==0 or n==1) return n;
    dp.resize(n+1,-1);
    int c,a=0,b=1;
    for(int i=2; i<=n; i++){
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
```
↳ $S_c = O(1)$

Generally; TB is easier than BU!
(How to build dp!)

Q) Each home has some money stashed, the only constraint stopping me from robbing each of them is that adjacent houses have security systems connected & it will automatically contact police if 2 adjacent houses were broken into on the same night.

Given int array nums representing the amount of money of each house, return the max. amount of money you can rob tonight without alerting the police.

eg1: [1,2,3,1]

eg2: 4

eg2: [2,7,9,3,1]

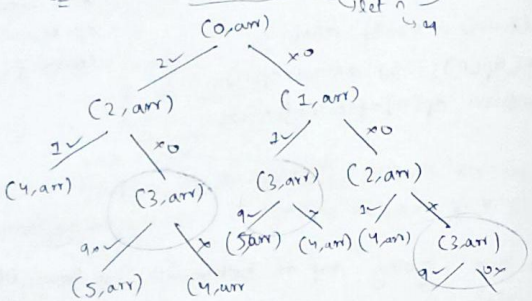
eg2: 2+9+1 = 12

A) $f(i, arr) = \max \begin{cases} arr[i] + f(i+2, arr) \\ 0 + f(i+1, arr) \end{cases}$

We can either rob a house or we can't

max profit by robbing all the houses from [i, n-1]

eg: arr: [2, 1, 1, 9] → total unique states = n+1 = 5



As you can see, we have many subproblems!

Code: (Top-down)

```
int f(int i, vector<int> &arr){
    if(i>=arr.size()) return 0;
    return max(arr[i]+f(i+2,arr), f(i+1,arr));
}
```

```
int rob(vector<int> &nums){
    return f(0, nums);
}
```

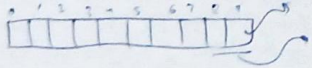
Above soln. → TLE as we do same subproblems again & again!

```
vector<int> dp;
int f(int i, vector<int> &arr){
    if(i>=arr.size()) return 0;
    if(dp[i] != -1) return dp[i];
    return dp[i] = max(arr[i]+f(i+2,arr), f(i+1,arr));
}
```

```
int rob(vector<int> &nums){
    dp.resize(nums.size()+10, -1);
    return f(0,nums);
}
```

if extra space needed

For bottom-up approach:

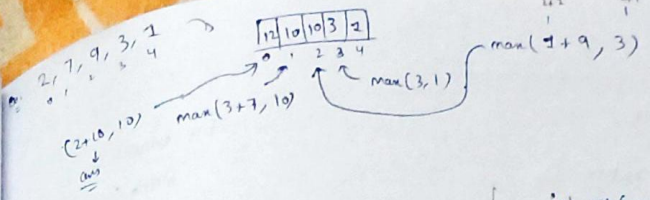


If only one house → that is ans.

If 2 houses, max of them = ans.

∴ $f(i) = \max(a[i]+f(i+2), f(i+1))$ → now we can loop.

```
for(i=n-3; i>=0; i--){
    dp[i] = max(dp[i+2]+a[i], dp[i+1])
}
```



Code:

```
vector<int> dp;
int f(int i, vector<int> &arr){
    int n = arr.size();
    if(n==1) return arr[n-1];
    dp[n-1] = arr[n-1];
    dp[n-2] = max(arr[n-1], arr[n-2]);
    for(int i=n-3; i>=0; i--){
        dp[i] = max(arr[i]+dp[i+2], dp[i+1]);
    }
    return dp[0];
}
```

```
int rob(vector<int> &nums){
    dp.resize(nums.size()+10, -1);
    return f(0,nums);
}
```

DP-2

Q) DECODE WAYS

A message contains letters from A-Z encoded to no.s: 'A'→1, 'B'→2, ... 'Z'→26

To decode an encoded message, all digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways).

eg: "11106" can be mapped: "AAJF": (1 1 10 6)
"KJF": (11 10 6)

Now grouping (1 11 06) is invalid because "06" can't be mapped into 'f' since '6' is different from '06'.

Given a string s containing only digits, return the no. of ways to do?

eg: s="12" ans: 2

A) eg: 1123 → No. of decoding of 123 → x=3 (12,3 / 1,2,3 / 1,23) — we add '1' at start
→ No. of decoding of 23 → y=2 (11,2,3 / 11,23)
ans = x+y

eg: 2734 → No. of decoding of 734 → x=1 : 2, 7, 3, 4
→ No. of decoding of 34 → (y=1 : 27, 3, 4) ✗
→ ~~26~~

Logic

$f(str, idx) = \begin{cases} f(str, idx+1) \\ + \\ \text{if } (str[idx] \times 10 + str[idx+1] \le 26) \\ f(str, idx+2) \end{cases}$

returns count of decoding of string str from idx to n-1

ans = f(str, 0)

Base case:
if (idx == n-1)
↳ decoding → str[n-1]
if (idx >= n)
↳ return 0

Code:

```
vector<int> dp;
int f(string s, int idx){ // s,0
    // base case
    if(s[idx]=='0') return 0;

    if(idx >= s.size()) return 0;

    if(idx == s.size()-1){ // last index
        if(s[idx]=='0') return 0; //handling '0' case
        else return 1;
    }
    bool can_form_2_digit = stoi(s.substr(idx,2))<=26;
    if(idx == s.size()-2){
        return can_form_2_digits + f(s,idx+1);
    }
    if(dp[idx]!=-1) return dp[idx];
    return f(s,idx+1) + ((can_form_2_digits)? f(s,idx+2):0);
}
return dp[idx] = ;
```

int main(){
    string s;
    cin>>s;
    dp.clear();
    dp.resize(105,-1);
    cout<<f(s,0);

// Now this will give TLE, so we use dp-topdown approach → (see with pencil)
(because we are doing same sub-probs. repeatedly)

dry run for s="226";

For bottom up approach, we need to know the ordering!

① dp[idx] = dp[idx+1] + (calc.) ? dp[idx+2] : 0 ;

Code:

```
vector<int> dp;
int fbu(string str){
    dp.clear();
    dp.resize(105,0);
② int n = str.size;
    dp[n-1] = (str[n-1]=='0') ? 0 : 1;
    bool can_form = stoi(str.substr(n-2,2))<=26
    if(str[n-2]=='0') dp[n-2] = 0;
    else dp[n-2] = dp[n-1] + (can_form);
    for(int i = n-3; i>=0; i--){ → depends on 'i' of 'str'
        if(str[i]=='0'){
            dp[i]=0;
            continue;
        }
        bool can_form_2_digits = stoi(str.substr(i,2))<=26;
        dp[i] = dp[i+1] + ((can_form_2_digits)?dp[i+2]:0);
    }
    return dp[0];
}
```

so we have written loop for 'i'

```
int main(){
    string s; cin>>s;
    int n = s.size();
    if(n==1){
        if(s[n-1]=='0') return 0;
        else return 1;
    }
    return fbu(s);
}
```

Q) N stones

There are N stones numbered 1,2,...N. Height of ith stone = h_i. There is a frog who is initially on stone 1. He will repeat an action some no. of times to reach stone N. The action is that if the frog is currently on stone i, it jumps to one of the following: stone i+1, i+2, ...i+k. Here, a cost of $|h_i - h_j|$ is incurred, where j is the stone to land on.

Find min. possible total cost incurred before frog reaches stone N.

Ex: n=5, k=3
10 30 40 50 20
(1) (2) (3) (4) (N)

Ans: 30
↳ |10-30| + |30-20| = 30

Ex: 10 20 30 40 50

1 → 4 → N → 30 ; 1 →(20) 2 →(20) N → (30)
1 →(i+3) 3 → N → 20 ; 1 →(20) 3 →(20) N → 50
1 →(i+2) 2 → N → 10 ; 1 →(40) 4 →(20) N → 70

f(height, i, k) = min(|h_i - h_{i+1}| + f(height, i+1, k), |h_i - h_{i+2}| + f(height, i+2, k), ..., |h_i - h_{i+k}| + f(height, i+k, k))

min. cost to reach (n-1)th stone from ith stone with max k jump allowed

Recursive logic → f(heights, i, k) = min(|h_i - h_j| + f(heights, j, k)) ∀ j ∈ [i+1, i+k] and j ≤ n-1

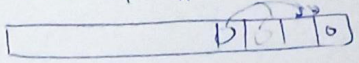
```
for(j=1; j<=k; j++){
    if(i+j <= n-1)
        ans = min(ans, (|h_i - h_{i+j}|) + f(heights, i+j, k));
```

Code:

```
vector<int> dp;
int f(vector<int>& heights, int i, int k){
    if(i == heights.size()-1) return 0;
    if(dp[i] != -1) return dp[i];
    int ans = INT_MAX;
    for(int j=1; j<=k; j++){
        if(i+j >= heights.size()) break;
        ans = min(ans, abs(heights[i] - heights[i+j]) + f(heights, i+j, k));
    }
    return dp[i] = ans;
}
```

```
int main(){
    int n; cin>>n;
    dp.clear(); dp.resize(100005,-1);
    int k; cin>>k;
    vector<int> v(n,0);
    for(int i=0; i<n; i++) cin>>v[i];
    cout<<f(v,0,k);
    return 0;
}
```

Now bottom-up approach;




```
for(i = n-2; i>=0; i--){ → order of subproblem →
    for(j=1; j<=k; j++){
        if(i+j >= n) break;
        dp[i] = min(dp[i], |h_i - h_{i+j}| + dp[i+j])
    }
}
```

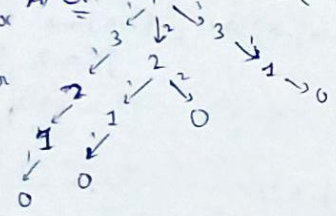
Code:

```
int fbu(vector<int>&heights, int k){
    int n = heights.size();
    dp[n-1] = 0;
    for(int i = n-2; i>=0; i--){
      for(int j = 1; j<=k; j++){
        if(i+j>=n) break;
        dp[i] = min(dp[i], abs(heights[i]-heights[i+j])) + dp[i+j]);
      }
    }
    return dp[0];
}
```

//dp resize(100005, INT_MAX)

Q) Given integer array[] of 'coins', size N representing different types of currency & integer sum. The task is to find no. of ways to make sum by using different combinations from coins[]. Assume you have ∞ supply of each type of coin → print no. of ways modulo 10^9+7

Ex: 4 = sum, coins = {1,2,3}



Exp: 4 → {1,1,1,1}, {1,1,2}, {2,2}, {1,3}

- $1 \le n \le 100$
- $1 \le x \le 10^6$
- $1 \le C_i \le 10^6$ → value of each coin

$$f(i, coins) = \sum_{j=0}^{n-1} f(i - coins(j), coins)$$

↓ no. of ways to create a value i from given coins

Code: → $T_C = O(nx)$

```
int fbu(vector<int>&coins, int x, int n){
    vector<int> dp(100005, 0);
    int MOD = 1000000007;
    dp[0] = 1;
    for(int j=0; j<n; j++){
        //go to each coin
        for(int i=1; i<=x; i++){

            if(i-coins[j]<0) continue;

            dp[i] = (dp[i]%MOD + dp[i-coins[j]]%MOD)%MOD;
        }
    }
    return dp[x];
}

int main(){
    int n, x; cin>>n>>x;
    vector<int> coins(n);
    for(int i=0; i<n; i++) cin>>coins[i];
    return 0;
}
```