My Code

- Linked List
  - Reverse The Linked List Iterative

    ```

    ```

  - Reverse The Linked List Recursive

    ```

    ```

  - Linked List
- Queue
  -
- Sorting
  - Bubble Sort

    ```
    #include<iostream>
    using namespace std;

    void bubbleSort(int arr[],int n){
        for(int i=0;i<n-1;i++){
            for(int j=0;j<n-i-1;j++){
                if(arr[j]>arr[j+1]){
                    int temp=arr[j];
                    arr[j]=arr[j+1];
                    arr[j+1]=temp;
                }
            }
        }
    }

    int main(){
        int n;
        cin>>n;
        int arr[n];
        for(int i=0;i<n;i++){
            cin>>arr[i];
        }
        bubbleSort(arr,n);
        for(int i=0;i<n;i++){
            cout<<arr[i]<<" ";
        }

        return 0;
    }
    ```

  - Insertion Sort

    ```
    #include<bits/stdc++.h>
    using namespace std;

    void insertionSort(int arr[],int n){
        for(int i=1;i<n;i++){
            int current=arr[i];
            int j=i-1;
            while(arr[j]>current && j>=0){
                arr[j+1]=arr[j]; // Putting the smaller value forward
                j--;
            }
            arr[j+1]=current;
        }
    }

    int main(){
    ```

```
    int arr[]={1,-1,9,4,8};
    int n=5;
    insertionSort(arr,n);
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
}
```

▼ Heap sort

```
// https://www.youtube.com/watch?v=UVW0NfG_YWA&list=PLUcsbZa0qzu3yNzzAxgvSgRobdUUJvz7p&index=32

#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
// n is the end index and i is the starting index
void heapify(vector<int> &a,int n,int i){
    int maxIdx=i;
    int leftChild=2*i+1;
    int rightChild=2*i+2;
    // if left child exists = bascilly greater than element at maxIdx
    // Than update maxIdx
    // Check same for right child
    // In order to confirm maxIdx is at which element parent, left or right
    if(leftChild<n && a[leftChild]>a[maxIdx]){
        maxIdx=leftChild;
    }
    if(rightChild<n && a[rightChild]>a[maxIdx]){
        maxIdx=rightChild;
    }
    // Now we have swap only when maxIdx is not the parent
    if(maxIdx!=i){
        swap(a[i],a[maxIdx]);
        heapify(a,n,maxIdx);
    }
}

void heapsort(vector<int> &a){
    int n=a.size();
    // Loop from last non leaf element than
    // call heapify so it will convert to maxHeap
    // The first non-leaf node is n/2-1
    for(int i=n/2-1;i>=0;i--){
        heapify(a,n,i);
    }
    // Pick last element and swap
    for(int i=n-1;i>0;i--){
        swap(a[0],a[i]);
        heapify(a,i,0);
    }
}

int main(){
    int n;
    cin>>n;
    vector<int> a(n);
    for(int i=0;i<n;i++){
        cin>>a[i];
    }
    heapsort(a);
    for(int i=0;i<n;i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
}
```

▼ DNF Sort

```
/*
    Dividing the array into 4 parts
    1. First part from starting to low : containing zeroes
    2. Second part from low to mid : for one's
    3. Third from mid to high : Unknown
    4. Fourth from high to the end : for two's
```

```
*/

#include<bits/stdc++.h>
using namespace std;

void swapFunction(int arr[],int a,int b){
    int temp=arr[a];
    arr[a]=arr[b];
    arr[b]=temp;
}

void dnfSort(int arr[],int n){
    int low=0;
    int mid=0;
    int high=n-1;
    while(high>=mid){
        if(arr[mid]==0){
            swapFunction(arr,low,mid);
            low++;
            mid++;
        }
        else if(arr[mid]==1){
            mid++;
        }
        else{
            swapFunction(arr,high,mid);
            high--;
        }
    }
}

int main(){
    int arr[5]={0,2,1,0,2};
    dnfSort(arr,4);
    for(int i=0;i<5;i++){
        cout<<arr[i]<<" ";
    }
}
```

▼ Merge Sort

```
#include<bits/stdc++.h>
using namespace std;
void mergeTheArray(int arr[],int l,int mid,int r){
    int n1=mid-l+1;
    int n2=r-mid;
    int a[n1];
    int b[n2];
    for(int i=0;i<n1;i++){
        a[i]=arr[l+i];
    }
    for(int i=0;i<n2;i++){
        b[i]=arr[mid+1+i];
    }

    int i=0;
    int j=0;
    int k=l;

    while(i<n1 && j<n2){
        if(a[i]<b[j]){
            arr[k]=a[i];
            i++;
            k++;
        }
        else{
            arr[k]=b[j];
            j++;
            k++;
        }
    }

    while(i<n1){
        arr[k]=a[i];
        i++;
        k++;
    }
    while(j<n2){
```

```
        arr[k]=b[j];
        j++;
        k++;
    }
}
void mergeSortCostom(int arr[],int l,int r){
    if(l<r){
        int mid=(l+r)/2;
        mergeSortCostom(arr,l,mid);
        mergeSortCostom(arr,mid+1,r);
        mergeTheArray(arr,l,mid,r);
    }
}
int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    mergeSortCostom(arr,0,n-1);
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}
```

▼ Selection sort

```
/*
    Find the minimum element and swap with the first element
*/
#include<bits/stdc++.h>
using namespace std;
void insertionSort(int arr[],int n){
    for(int i=0;i<n-1;i++){
        for(int j=i;j<n;j++){
            if(arr[i]>arr[j]){
                swap(arr[i],arr[j]);
            }
        }
    }
}

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    insertionSort(arr,n);
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}
```

▼ Array Problems

▼ Maximum Element In previous i elements

```
#include<bits/stdc++.h>
using namespace std;

void maxTilli(int arr[],int n){
    int currMax=INT_MIN;
    for(int i=0;i<n;i++){
        currMax=max(arr[i],currMax);
        cout<<currMax<<" ";
    }
    cout<<endl;
}

int main(){
```

```
    int arr[]={1,0,4,7,-1,9};
    int n=6;
    maxTilli(arr,n);
}
```

▼ Negative Elements before Positive

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long int
void swapFunction(int arr[],int a,int b){
    int temp=arr[a];
    arr[a]=arr[b];
    arr[b]=temp;
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int arr[9]={-12,11,-13,-5,6,-7,5,-3,-6};
    int low=0;
    int high=8;
    while(low<high){
        if(arr[low]<0){
            low++;
        }
        else if(arr[low]>=0){
            swapFunction(arr,low,high);
            high--;
        }
    }

    for(int i=0;i<9;i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

▼ Sum Of all subarrays in any array

```
#include<bits/stdc++.h>
using namespace std;
// Total subarrays with n elements = nC2 + n;
int main(){
    int arr[]={1,2,3,4,5};
    int n=5;
    for(int i=0;i<n;i++){
        int curr=0;
        for(int j=i;j<n;j++){
            curr+=arr[j];
            cout<<curr<<endl;
        }
    }
}
```

▼ Longest Contiguous Arithmetic Subarray

```
// https://codingcompetitions.withgoogle.com/kickstart/round/000000000019ff47/00000000003bf4ed
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }

    int diff=arr[1]-arr[0];
    int ans=2;
    int j=2;
```

```
    int curr=2;

    while(j<n){
        if(diff==arr[j]-arr[j-1]){
            curr++;
        }
        else{
            diff=arr[j]-arr[j-1];
            ans=2;
        }
        ans=max(ans,curr);
        j++;
    }

    cout<<ans<<endl;
}
```

▼ Record Breaking Days

```
// https://codingcompetitions.withgoogle.com/kickstart/round/000000000019ff08/0000000000387171

#include<bits/stdc++.h>
using namespace std;
int main(){
    int arr[]={1,2,0,7,2,0,2,2};
    int n=8;

    int currentMax=INT_MIN;
    int ans=0;
    if(n==1){
        cout<<1<<endl;
        return 0;
    }
    for(int i=0;i<n-1;i++){
        if(arr[i]>currentMax && arr[i]>arr[i+1]){
            ans++;
            currentMax=arr[i];
        }
        else{
            currentMax=max(currentMax,arr[i]);
        }
    }
    if(arr[n-1]>currentMax){
        ans++;
    }
    cout<<ans<<endl;

    return 0;
}
```

▼ The Index Of First Repeating Element In Array

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int arr[]={1,5,3,4,3,5,6};
    int n=7;
    int idxArr[n+1];
    for(int i=0;i<n+1;i++){
        idxArr[i]=-1;
    }
    int minIDX=INT_MAX;
    for(int i=0;i<n;i++){
        if(idxArr[arr[i]]==-1){
            idxArr[arr[i]]=1;
        }
        else{
            idxArr[arr[i]]++;
            minIDX=min(minIDX,idxArr[arr[i]]);
        }
    }
    if(minIDX==INT_MAX){
        cout<<-1<<endl;
    }
```

```
    else{
        cout<<minIDX<<endl;
    }
}
```

▼ The subarray sum equal to S

▼ Brute Force

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int arr[]={1,2,3,7,5};
    int n=5;
    int s=12;
    for(int i=0;i<n;i++){
        int sum=0;
        for(int j=i;j<n;j++){
            sum+=arr[j];
            if(sum==s){
                cout<<"From Position "<<i<<" to "<<j<<" the sum is"<<s<<endl;
            }
        }
    }
}
```

▼ Optimized Approach

```
// Maintain Two pointers st and en, and a variable currSum sum from
// st to en. Incriment en till currSum + a[en] > S
// Increment st till currSum <= S

// Time Complexity = O(N)

#include <bits/stdc++.h>
using namespace std;
int main(){
    int arr[] = {1, 2, 3, 8};
    int n = 4;
    int s = 5;

    int i=0,j=0,st=-1,en=-1,sum=0;
    while(j<n && sum+arr[j]<=s){
        sum+=arr[j];
        j++;
    }
    if(sum==s){
        cout<<i+1<<" "<<j<<endl;
        return 0;
    }

    while(j<n){
        sum+=arr[j];
        while(sum>s){
            sum-=arr[i];
            i++;
        }
        if(sum==s){
            st=i+1;
            en=j+1;
            break;
        }
        j++;
    }
    cout<<st<<" "<<en<<endl;
}
```

▼ Smallest Positive missing value

```
#include<bits/stdc++.h>
using namespace std;
```

```
int main(){
    int arr[]={0,-9,1,3,-4,5};
    int n=6;
    bool tf[n];
    for(int i=0;i<n;i++){
        tf[i]=false;
    }
    for(int i=0;i<n;i++){
        if(arr[i]>=0){
            tf[i]=true;
        }
    }
    for(int i=0;i<n;i++){
        if(tf[i]==false){
            cout<<i+1<<endl;
            break;
        }
    }
}
```

▼ Print All Subarrays

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int arr[]={1,2,3,4,5};
    int n=5;
    for(int i=0;i<n;i++){
        for(int j=i;j<n;j++){
            for(int k=i;k<=j;k++){
                cout<<arr[k]<<" ";
            }
            cout<<endl;
        }
    }
}
```

▼ Maximum Subarray Sum Using Kadanes Algorithm

```
#include<bits/stdc++.h>
using namespace std;

int maximumSubarraySum(int arr[],int n){
    int currentSum=0;
    int maxSum=INT_MIN;
    for(int i=0;i<n;i++){
        currentSum+=arr[i];
        if(maxSum<currentSum){
            maxSum=currentSum;
        }
        if(currentSum<0){
            currentSum=0;
        }
    }
    return maxSum;
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n=4;
    int arr[n]={-1,-2,-3,-4};
    cout<<maximumSubarraySum(arr,n);
    return 0;
}
```

▼ Maximum Circular Subarray Problem

```
// Max subarray sum = Total sum - non contributing elements
// Inverting the signs of the array and than applying the kedanes algorithm

#include<bits/stdc++.h>
```

```
using namespace std;

int kedane(int arr[],int n){
    int maxNum=INT_MIN;
    int currentSum=0;

    for(int i=0;i<n;i++){
        currentSum+=arr[i];
        if(currentSum<0){
            currentSum=0;
        }
        maxNum=max(currentSum,maxNum);
    }
    return maxNum;
}

int main(){
    int arr[]={4,-4,6,-6,10,-11,12};
    int n=7;
    int nonWrapSum=kedane(arr,n);
    // Contributing elements are wrapping
    // Non Contributing are nonWrapping
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=arr[i];
        arr[i]=-arr[i];
    }

    int wrapSum=sum+kedane(arr,n);
    cout<<max(wrapSum,nonWrapSum)<<endl;
    return 0;
}
```

▼ Sudoku Solver

```
/*
https://www.youtube.com/watch?v=FWAIf_EVUKE
    1.  Find out the empty place
    2.  Try all possible values and if any any value is possible than
        Recursively call the function
    3.  But if you find out at any future place than some previous step was wrong
        and we are not able to from answer than backtrack.
*/

#include <bits/stdc++.h>
using namespace std;

bool isValid(vector<vector<char>> &board, int row, int col, char c){
    for (int i = 0; i < 9; i++){
        if (board[i][col] == c)
            return false;

        if (board[row][i] == c)
            return false;

        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
            return false;
    }
    return true;
}

bool solve(vector<vector<char>> &board){
    for (int i = 0; i < board.size(); i++){
        for (int j = 0; j < board[0].size(); j++){
            if (board[i][j] == '.'){
                for (char c = '1'; c <= '9'; c++){
                    if (isValid(board, i, j, c)){
                        board[i][j] = c;

                        if (solve(board))
                            return true;
                        else
                            board[i][j] = '.';
                    }
                }

                return false;
            }
```

```
        }
    }
    return true;
}

void solveSudoku(vector<vector<char>> &board){
    solve(board);
}

int main(){
    vector<vector<char>> soduku = {
        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},
        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };
    solveSudoku(soduku);
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++){
            cout << soduku[i][j] << " ";
        }
        cout << endl;
    }
}
```

▼ Union And Intersection in two arrays

```
/*

*/
#include<bits/stdc++.h>
using namespace std;

vector<int> intersectionOfArrays(int arr[],int arr1[],int n1,int n2){
    unordered_map<int,int>mp;
    unordered_map<int,int>mp1;
    for(int i=0;i<n1;i++){
        mp[arr[i]]++;
    }
    for(int i=0;i<n2;i++){
        mp1[arr1[i]]++;
    }
    int ans=0;
    if(mp.size()>=mp1.size()){
        ans=1;
    }

    vector<int>ansArr;
    if(ans==1){
        for(auto it:mp1){
            for(auto i:mp){
                if(it.first==i.first){
                    ansArr.push_back(it.first);
                }
            }
        }
    }
    else{
        for(auto it:mp){
            for(auto i:mp1){
                if(it.first==i.first){
                    ansArr.push_back(it.first);
                }
            }
        }
    }
    return ansArr;
}

vector<int> unionOfTwoArrays(int arr[],int arr1[],int n1,int n2){
    unordered_map<int,int>mp;
    for(int i=0;i<n1;i++){
        mp[arr[i]]++;
```

```
    }
    for(int i=0;i<n2;i++){
        mp[arr1[i]]++;
    }
    vector<int>ans;
    for(auto it:mp){
        ans.push_back(it.first);
    }
    return ans;
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n1,n2;
    cin>>n1>>n2;
    int arr[n1];
    int arr1[n2];
    for(int i=0;i<n1;i++){
        cin>>arr[i];
    }
    for(int i=0;i<n2;i++){
        cin>>arr1[i];
    }
    vector<int>unionInTheArrays=unionOfTwoArrays(arr,arr1,n1,n2);
    vector<int>intersectionOF=intersectionOfArrays(arr,arr1,n1,n2);

    cout<<"The union of both the arrays is : ";
    for(auto it:unionInTheArrays){
        cout<<it<<" ";
    }
    cout<<endl;

    cout<<"The intersection of both the arrays is : ";
    for(auto it:intersectionOF){
        cout<<it<<" ";
    }
    cout<<endl;
    return 0;
}
```

▼ Cyclically Rotating The array by one

```
#include<bits/stdc++.h>
using namespace std;

void swapFunction(int arr[],int a,int b){
    int temp=arr[a];
    arr[a]=arr[b];
    arr[b]=temp;
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    for(int i=0;i<n;i++){
        swapFunction(arr,n-1,i);
    }

    for(auto it:arr){
        cout<<it<<" ";
    }
    cout<<endl;
    return 0;
}
```

▼ Merge Two Sorted arrays without extra space

```
/*
    https://www.geeksforgeeks.org/merge-two-sorted-arrays-o1-extra-space/
    https://www.youtube.com/watch?v=hVl2b3bLzBw
*/
#include<bits/stdc++.h>
using namespace std;

void merge(int arr1[], int arr2[], int n, int m){
    int i = 0, j = 0, k = n - 1;
    while (i <= k and j < m) {
        if (arr1[i] < arr2[j])
            i++;
        else {
            swap(arr2[j++], arr1[k--]);
        }
    }
    sort(arr1, arr1 + n);
    sort(arr2, arr2 + m);
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int arr1[]={1,2,3,4};
    int arr2[]={-1,0,5,9,10,15};
    int n=4;
    int m=6;
    merge(arr1,arr2,n,m);
    for(int i=0;i<n;i++){
        cout<<arr1[i]<<" ";
    }
    for(int j=0;j<m;j++){
        cout<<arr2[j]<<" ";
    }
    return 0;
}
```

▼ Count Inversion

```
/*
    Brute force approach is to take two loops and then traverse
    the array and if (i<j && a[i] > a[j]) then we increase the counter
    but Time Complexity = O(n^2)
*/
#include <bits/stdc++.h>
using namespace std;
long long int mergeSort(long long int arr[], long long int array_size){
    long long int temp[array_size];
    return _mergeSort(arr, temp, 0, array_size - 1);
}

long long int _mergeSort(long long int arr[], long long int temp[], long long int left, long long int right){
    long long int mid, inv_count = 0;
    if (right > left) {
        mid = (right + left) / 2;
        inv_count += _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid + 1, right);
        inv_count += merge(arr, temp, left, mid + 1, right);
    }
    return inv_count;
}

long long int merge(long long int arr[], long long int temp[], long long int left, long long int mid,long long int right){
    long long int i, j, k;
    long long int inv_count = 0;
    // left half of the array is from (left,mid-1)
    // Right half from (mid,right)
    i = left; /* i is index for left subarray*/
    j = mid; /* j is index for right subarray*/
    k = left; /* k is index for resultant merged subarray*/
    while ((i <= mid - 1) && (j <= right)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        }
        else {
            temp[k++] = arr[j++];
```

```
            // Counting the pairs which will be formed when we
            // formed when right part is smaller than left part
            // But the part of left array which we have alredy traversed
            // is removed that is why mid-i
            inv_count = inv_count + (mid - i);
        }
    }

    /* Copy the remaining elements of left subarray
(if there are any) to temp*/
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    /* Copy the remaining elements of right subarray
       (if there are any) to temp*/
    while (j <= right)
        temp[k++] = arr[j++];

    /*Copy back the merged elements to original array*/
    for (i = left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

// Driver code
int main()
{
    long long int arr[] = { 1, 20, 6, 4, 5 };
    long long int n = sizeof(arr) / sizeof(arr[0]);
    long long int ans = mergeSort(arr, n);
    cout << " Number of inversions are " << ans;
    return 0;
}
```

▼ Best Time To Buy And Sell Stocks

```
#include<bits/stdc++.h>
using namespace std;

class Solution{
    public:
    int maxProfit(vector<int>&prices){
        int profit=INT_MIN;
        for(int i=0;i<prices.size()-1;i++){
            for(int j=i;j<prices.size();j++){
                profit=max(profit,-1*(prices[i]-prices[j]));
            }
        }
        if(profit<0){
            return 0;
        }
        else{
            return profit;
        }
    }

    int maxProfitOpti(vector<int>&prices){
        int mini=INT_MAX;
        int profit=0;
        for(int i=0;i<prices.size();i++){
            mini=min(mini,prices[i]);
            profit=max(profit,prices[i]-mini);
        }
        return profit;
    }
};

int main(){
    Solution obj;
    int n;
    cin>>n;
    vector<int>prices(n,0);
    for(int i=0;i<n;i++){
        cin>>prices[i];
    }
    int x=obj.maxProfit(prices);
    int y=obj.maxProfitOpti(prices);
```

```
    cout<<x<<" "<<y<<endl;
}
```

▼ Count Pairs Sum

```
#include<bits/stdc++.h>
using namespace std;

class Solution{
    public:
    int getPairsCount(int arr[],int n,int k){
        int cnt=0;
        for(int i=0;i<n;i++){
            for(int j=i+1;j<n;j++){
                if(arr[i]+arr[j]==k){
                    cnt++;
                }
            }
        }
        return cnt;
    }

    // Here Every pair will be counted twice because
    // (1,2) and (2,1) will be treated with sum = 3
    // Hence We need to divide the result by two
    // And if arr[i] = k/2 than (1,1) and (1,1) will
    // be added twice again for set (1,1)
    // Hence one of them needed to be subtracted

    int getPairCountOpti(int arr[],int n,int k){
        unordered_map<int,int>mp;
        for(int i=0;i<n;i++){
            mp[arr[i]]++;
        }
        int cnt=0;
        for(int i=0;i<n;i++){
            cnt+=mp[k-arr[i]];
            if(k-arr[i]==arr[i]){
                cnt--;
            }
        }
        return cnt/2;
    }
};

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    int k;
    cin>>k;
    Solution obj;
    int x=obj.getPairsCount(arr,n,k);
    int y=obj.getPairCountOpti(arr,n,k);
    cout<<x<<" "<<y<<endl;
}
```

▼ Common Elements in Three Arrays

```
#include<bits/stdc++.h>
using namespace std;

class Solution{
    public:
    vector<int>commonElements(int arr1[],int arr2[],int arr3[],int n1,int n2,int n3){
        vector<int>ans;
        unordered_map<int,int>m1;
        for(int i=0;i<n1;i++){
            m1[arr1[i]]++;
        }
        for(int i=0;i<n2;i++){
            m1[arr2[i]]++;
```

```
        }
        for(int i=0;i<n3;i++){
            m1[arr3[i]]++;
        }

        for(auto it:m1){
            if(it.second==3){
                ans.push_back(it.first);
            }
        }
        return ans;
    }
};

int main(){
    int n1,n2,n3;
    cin>>n1>>n2>>n3;
    int arr1[n1],arr2[n2],arr3[n3];
    for(int i=0;i<n1;i++){
        cin>>arr1[i];
    }
    for(int i=0;i<n2;i++){
        cin>>arr2[i];
    }
    for(int i=0;i<n3;i++){
        cin>>arr3[i];
    }
    Solution o;
    vector<int> x=o.commonElements(arr1,arr2,arr3,n1,n2,n3);
    for(auto it:x){
        cout<<it<<" ";
    }
    cout<<endl;
}
```

▼ Rearrange array in alternating positive & negative items

```
#include<bits/stdc++.h>
using namespace std;
void answerFunction(int arr[],int n){
  sort(arr,arr+n);
  int i=1,j=1;
  while(j<n){
    if(arr[j]>0){
      break;
    }
    j+=1;
  }

  while(arr[i]<0 && j<n){
    swap(arr[i],arr[j]);
    i+=2;
    j+=1;
  }
}
int main(){
  int n;
  cin>>n;
  int arr[n];
  for(int i=0;i<n;i++){
    cin>>arr[i];
  }
  answerFunction(arr,n);
  for(int i=0;i<n;i++){
    cout<<arr[i]<<" ";
  }
  cout<<endl;
}
```

▼ Check If There exists any subarray with sum zero

```
#include<bits/stdc++.h>
using namespace std;
// The Logic which is being used here is that if there is some
// subarray which has sum zero than the previous sum will repreat itself
```

```
bool SubarrayWithZerosum(int arr[],int n){
  unordered_set<int>sumSet;
  int sum=0;
  for(int i=0;i<n;i++){
    sum+=arr[i];
    if(sum==0 || sumSet.find(sum)!=sumSet.end()){
      return true;
    }
    sumSet.insert(sum);
  }
  return false;
}

bool subarrayWithGivenSum(int arr[],int n,int k){
  unordered_set<int>sumSet;
  int sum=0;
  for(int i=0;i<n;i++){
    sum+=arr[i];
    if(sum==k || sumSet.find(sum-k)!=sumSet.end()){
      return true;
    }
  }
  return false;
}

int main(){
  int n,k;
  cin>>n>>k;
  int arr[n];
  for(int i=0;i<n;i++){
    cin>>arr[i];
  }
  cout<<SubarrayWithZerosum(arr,n)<<" ";
  cout<<subarrayWithGivenSum(arr,n,k);
}
```

▼ Factorial Of A Large Number

```
#include<bits/stdc++.h>
using namespace std;
#define MAX 500
int multiply(int res[],int res_size,int x);
void factorial(int n){
    int res[MAX];
    res[0]=1;
    int res_size=1;
    for(int x=2;x<=n;x++){
        res_size=multiply(res,res_size,x);
    }
    for(int i=res_size-1;i>=0;i--){
        cout<<res[i];
    }
    cout<<endl;
}

int multiply(int res[],int res_size,int x){
    int carry=0;
    for(int i=0;i<res_size;i++){
        int prod=res[i]*x+carry;
        res[i]=prod%10;
        carry=prod/10;
    }
    while(carry){
        res[res_size]=carry%10;
        carry=carry/10;
        res_size++;
    }
    return res_size;
}

int main(){
    factorial(6);
}
```

▼ Maximum Subarray Product

```
#include<bits/stdc++.h>
using namespace std;

int maxSubarrayProd(int arr[],int n){
    if(n==0){
        return -1;
    }
    int minProd=arr[0];
    int maxprod=arr[0];
    int ans=arr[0];
    int choice1,choice2;
    for(int i=1;i<n;i++){
        choice1=minProd*arr[i];
        choice2=maxprod*arr[i];
        minProd=min(arr[i],min(choice1,choice2));
        maxprod=max(arr[i],max(choice1,choice2));
        ans=max(ans,maxprod);
    }
    return ans;
}

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int j=0;j<n;j++){
        cin>>arr[j];
    }
    cout<<maxSubarrayProd(arr,n)<<endl;
}
```

▼ Length Of Longest Consecutive Subsequence

```
/*
    The constraints are taken from the GFG question
*/
#include<bits/stdc++.h>
using namespace std;
int findLongestConseqSubseq(int arr[],int n){
    unordered_set<int>s;
    for(int i=0;i<n;i++){
        s.insert(arr[i]);
    }
    int count=0;
    int maxCount=0;
    for(int i=0;i<=100000;i++){
        if(s.find(i)!=s.end()){
            count++;
        }
        else{
            count=0;
        }
        maxCount=max(maxCount,count);
    }
    return maxCount;
}
int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    cout<<findLongestConseqSubseq(arr,n);
}
```

▼ Count More than n/k Occurrences

```
#include<bits/stdc++.h>
using namespace std;

int countOccurence(int arr[], int n, int k){
    unordered_map<int,int>m;
    for(int i=0;i<n;i++){
```

```
        m[arr[i]]++;
    }
    int fre=n/k;
    int count=0;
    for(auto it:m){
        if(it.second>fre){
            count++;
        }
    }
    return count;
}

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    cout<<countOccurence(arr,n,3);
    return 0;
}
```

▼ STL

▼ Heaps : Priority Queue

```
/*
    1. MAXHEAP : priority-queue<int,vector<int>>
    2. MINHEAP : priority-queue<int,vector<int>,greater<int>>

    push = O(log(n))
    pop = O(log(n))
    top = O(1)
    size = O(1)
*/

#include<iostream>
#include<bits/stdc++.h>
#include<queue>
using namespace std;
int main(){
    priority_queue<int,vector<int>,greater<int>>pq;
    pq.push(2);
    pq.push(3);
    pq.push(1);
    cout<<pq.top()<<endl;
}
```

▼ Hashing : Maps

```
/*
    Standard template library
    --> Maps
        Insertion : O(log(n))
        Accessing : O(log(n))
        Deletion : O(log(n))
        implemented using red black trees
        Key, value pair
        map<int,int>mp

    --> Unordered Maps
        Insertion : O(1)
        Accessing : O(1)
        Deletion : O(n)
        Hash tables (basically array of buckets)
        unorder_map<int,int>mp;
*/

#include<iostream>
#include<map>
using namespace std;
int main(){
    map<int,int> m;
    m[8]=2;
```

```
    cout<<m[8]<<endl;
}
```

▼ Algorithm : Next Permutation

```

```

▼ Heaps

▼ Median Of Running stream

```
// Numbers are coming and we have to tell median after each input
/*
    arr = [10, 15 , 21 , 30 , 18 , 19]
    After first input :
        Sorted : 10
        Median : 10

    Second input :
        Sorted : 10 , 15
        Median : (10 + 15)/2

    Third input :
        Sorted : 10 ,15 , 21
        Median : 15

    Fourth Input :
        Sorted : 10 ,15 , 21 , 30
        Median : (15 + 21)/2

    Fifth input :
        Sorted : 10 , 15 , 18 , 21 , 30
        Median : 18

    Sixth Input :
        Sorted : 10 , 15 , 18 , 19 , 21 , 30
        Median : (18 + 19)/2


    --> Optimal approach : Using Heaps
        1. Keep one maxHeap and one MinHeap
        2. Partition the array into two parts
        3. During insertion if(sizeof(maxHead)==sizeof(minHeap)),
           then add into maxHeap otherwise minHeap and always maintain
           sizeof | MaxHeap - MinHeap | = 1
        4. For median if(sizeof(maxHead)==sizeof(minHeap)), then average
           of top of minHead + MaxHeap else top of maxHeap
    --> pqmax contains small elements
    --> pqmin contains maximum elements
*/

#include <bits/stdc++.h>
using namespace std;

priority_queue<int, vector<int>> pqmax;
priority_queue<int, vector<int>, greater<int>> pqmin;

// If sizes are not equal than top of larger size of heap
// If equal than average of the top of both heaps
void insertEle(int x){
    if (pqmin.size() == pqmax.size()){
        // base case if no element is inserted
        if (pqmax.size() == 0){
            pqmax.push(x);
            return;
        }

        // If incoming element is smaller than top of pqmax than push to pqmax
        // because maxheap stores all smaller elements
        if (x < pqmax.top()){
            pqmax.push(x);
        }
        else{
            pqmin.push(x);
        }
    }
```

```
    else{
        // Size of maxheap is greater than minheap
        // than if incoming element is greater than pqmax.top than push to pqmin
        // else swap pqmax.top and the incoming and push the swapped x to pqmin
        if(pqmax.size()>pqmin.size()){
            if(x>=pqmax.top()){
                pqmin.push(x);
            }
            else{
                int temp=pqmax.top();
                pqmax.pop();
                pqmax.push(x);
                pqmin.push(temp);
            }
        }

        // if size of minheap is greater
        // if x<pqmin.top() than push to pqmax
        // else swap with pqmax.top and x and push x to pqmin
        else{
            if(x<pqmin.top()){
                pqmax.push(x);
            }
            else{
                int temp=pqmin.top();
                pqmin.pop();
                pqmin.push(x);
                pqmax.push(temp);
            }
        }
    }
}

double findMedian(){
    if(pqmin.size()==pqmax.size()){
        return (pqmax.top()+pqmin.top())/2.0;
    }
    else if(pqmax.size()>pqmin.size()){
        return pqmax.top();
    }
    else{
        return pqmin.top();
    }
}

int main(){
    int arr[]={10, 15 , 21 , 30 , 18 , 19};
    for(int i=0;i<6;i++){
        insertEle(arr[i]);
        cout<<findMedian()<<" ";
    }
    cout<<endl;
}
```

▼ Merge K sorted arrays

```
/*
    Brute force : Take 2 arrays and merge until only one array is left
    --> Efficient solution :
    1. Create a mini heap of pairs.
        Pair --> {value , array nunmber}
    2. Insert (first element ,array number) of all the sorted
       array into minHeap.
    3. Main Idea : We will pop element from the Min Heap and store
       into the answer array. Insert the next element of the Sorted
       Array into MinHeap.
    4. We also need to keep track of the indices of the elements
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int k;
    cin>>k;
    vector<vector<int>>a(k);
    for(int i=0;i<k;i++){
        int size;
        cin>>size;
```

```
        a[i]=vector<int> (size);
        for(int j=0;j<size;j++){
            cin>>a[i][j];
        }
    }
    // vector of size k and initialised with 0
    // This will contain first element of each array
    vector<int> idx(k,0);

    // Creating Min Heap of pairs {value,array number}
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
    for(int i=0;i<k;i++){
        pq.push({a[i][0],i});
    }
    vector<int> ans;
    while(!pq.empty()){
        pair<int,int> x=pq.top();
        pq.pop();
        ans.push_back(x.first);

        // Checking if the current array became empty or not
        if(idx[x.second]+1<a[x.second].size()){
            pq.push({a[x.second][idx[x.second]+1],x.second});
        }
        idx[x.second]+=1;
    }
    for(int i=0;i<ans.size();i++){
        cout<<ans[i]<<endl;
    }
}
```

▼ Smallest Subsequence with sum K

```
// We aaplied greedy approach here
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[]={1,1,3,2,8};
    int k=13;
    int n=5;
    priority_queue<int,vector<int>>pq;
    for(int i=0;i<n;i++){
        pq.push(a[i]);
    }
    int sum=0;
    int count=0;
    while(!pq.empty()){
        sum+=pq.top();
        pq.pop();
        count++;
        if(sum>=k){
            break;
        }
    }
    if(sum<k){
        cout<<"-1"<<endl;
    }
    else{
        cout<<count<<endl;
    }
}
```

▼ Hashing

▼ Count Frequency of elements

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int arr[]={2,2,2,1,1,4,3,4};
    map<int,int>mp;
    for(int i=0;i<8;i++){
        mp[arr[i]]++;
    }
    for(auto it:mp){
```

```
        cout<<it.first<<" "<<it.second<<endl;
    }
}
```

▼ Print the vertical order of the binary tree

```
/*
    Given a binary tree in array representation
            10 7 4 3 11 14 6
                  10
                 /  \
                /    \
               7      4
              / \    / \
            3   11 /  6
                14
            Parent Index = i
            Left son = 2 * i + 1
            Right son = 2 * i + 2
    1. Stating from root node
    2. Recursively call left and right with (HD -1) and (HD +1)
       as arguments
    Base case : When current node = NULL (return)
    3. Pushh the value into vector corresponding to the horizontal
       distance (HD).

    Map
    0   --> {10,11,14}
    -1  --> {7}
    -2  --> {3}
    1   --> {4}
    1   --> {6}
*/

#include<iostream>
#include<map>
#include<vector>
using namespace std;
class node{
    public:
    int key;
    node* left;
    node* right;
    node(int val){
        key=val;
        left=NULL;
        right=NULL;
    }
};

void getVerticalOrder(node* root,int hdis,map<int,vector<int>>&m){
    if(root==NULL){
        return ;
    }
    m[hdis].push_back(root->key);
    getVerticalOrder(root->left,hdis-1,m);
    getVerticalOrder(root->right,hdis+1,m);
}

int main(){
    node* root=new node(10);
    root->left=new node(7);
    root->right=new node(4);
    root->left->left=new node(3);
    root->left->right=new node(11);
    root->right->left=new node(14);
    root->right->right=new node(6);
    map<int,vector<int>> m;
    int hdis=0;
    getVerticalOrder(root,hdis,m);

    map<int,vector<int>>::iterator it;
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first<<" ";
        for(int i=0;i<it->second.size();i++){
            cout<<(it->second)[i]<<" ";
        }
        cout<<endl;
```

```
    }
    return 0;
}
```

▼ Number of Subarrays with sum zero

```
/*
    Compute prefix sum
    prefix sum is the sum till ith element
    if lets suppose some value is x (index i) in the array and again that vale comes at index (j)
    now this means that there was some chemical locha in between due to which the sum
    became zero

    now the sum of i+1 th element to j will be zero
    If we compute how many times x has incurred in the array
    and choose two at a time in MC2
    and if the key is zero than we also have to include it because zero is what we want as answer

    1. Map prefix sum to a map
    2. For every key, choose 2 values from all the occurances of
       particular prefsum (MC2)
    3. Speacial Case : for prefsum 0, we have to also include them
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n=4;
    int arr[n]={1,-1,1,-1};
    map<int,int>cnt;

    int prefSum=0;
    for(int i=0;i<n;i++){
        prefSum+=arr[i];
        cnt[prefSum]++;
    }
    for(auto it:cnt){
        cout<<it.first<<" "<<it.second<<endl;
    }

    int ans=0;
    map<int,int>::iterator it;
    for(it=cnt.begin();it!=cnt.end();it++){
        int c=it->second;
        ans+=c*(c-1)/2;
        if(it->first==0){
            ans+=it->second;
        }

    }
    cout<<ans<<endl;
}
```

▼ Sliding Window For Minimum subarray sum

```
/*
    Brute force
    1.  Iterate from i=0 to i=n-k-1 in outer loop
    2.  starting from every j=i compute sum of k elements and maintain
        the minimum

    Time complexity - O(n*k)

    optimal solution (sliding window)
    1.  Compute the sum of first k elements (i=0 to k)
    2.  while increasing i, subtact a[i-1] and a[i-k-1] in the previous
        sum, while will became current sum.

    Time complexity - O(n)
*/
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n=8;
    int k=3;
```

```
    int arr[n]={-2,10,1,3,2,-1,4,5};
    int s=0,ans=INT_MAX;
    for(int i=0;i<k;i++){
        s+=arr[i];
    }
    ans=min(ans,s);
    for(int i=1;i<n-k+1;i++){
        s-=arr[i-1];
        s+=arr[i+k-1];
        ans=min(ans,s);
    }
    cout<<ans<<endl;
}
```

▼ Top K most frequent elements in the stream

```
/*
    Given an array : 1, 2, 2, 2, 3, 4
    and k=2;
    We have to output elements in decreasing frequency till we reach
    (K+1)th distinct elements.

    Approach (Hashing)
    1.  Create a map
    2.  While travelling, keep trach of elements and when we find
        (K+1)th without element = break
*/

#include <bits/stdc++.h>
using namespace std;
int main(){
    int n=6;
    int k=2;
    int a[n]={1,2,2,2,3,4};
    map<int,int> freq;
    for(int i=0;i<n;i++){
        int currentSize=freq.size();
        if(freq[a[i]]==0 && currentSize==k){
            break;
        }
        freq[a[i]]++;
    }

    vector<pair<int,int>> ans;
    map<int,int>::iterator it;
    for(it=freq.begin();it!=freq.end();it++){

        // This next condition is important because when we checked if
        // freq[a[i]]==0 && currentSize==k than we increased the size by one
        // In order to prevent that from inserting we put this if condition
        if(it->second!=0){
            pair<int,int> p;
            // interchanged the first and second
            // because we want to sort on the basis of frequency
            p.first=it->second;
            p.second=it->first;
            ans.push_back(p);
        }
    }

    // descending sort
    sort(ans.begin(),ans.end(),greater<pair<int,int>>());
    vector<pair<int,int>> ::iterator it1;
    for(it1=ans.begin();it1!=ans.end();it1++){
        cout<<it1->second<<" "<<it1->first<<endl;
    }

    return 0;
}
```

▼ Greedy Problems

▼ Minimum number of notes to make an amount

```
/*
    We have given an array of denominations and a value X. We have to find
    the minimum number of coins required to make value X
*/
#include<bits/stdc++.h>
using namespace std;
int main(){
    int arr[]={2000,500,200,100,50,20,10,5,2,1};
    int n=2000;
    int ans=0;
    for(int i=0;i<10;i++){
        ans+=(n/arr[i]);
        n-=(n/arr[i])*(arr[i]);
    }
    cout<<ans<<endl;
}
```

▼ Activity Selection Problem

```
/*
    custom comparators : https://www.youtube.com/watch?v=zBhVZzi5RdU 40:00
    In this question used a different way to impliment custom comparators
    We are given with tasks ans we want to fins how many tasks we can
    do if we are given with starting and ending time for each task and
    we can do only one task at a time

    Greedy
    if you are at the ith activity
    What should be your next step?
    Take the next activity which ends first
*/
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<pair<int,int>>p;
    for(int i=0;i<3;i++){
        int a,b;
        cin>>a>>b;
        p.push_back({a,b});
    }
    sort(p.begin(),p.end(),[&](pair<int,int>&a,pair<int,int>&b){
        return a.second<b.second;
    });
    int ans=0;
    int end=0;
    for(auto it:p){
        if(end<=it.first){
            ans++;
            end=it.second;
        }
    }
    cout<<ans<<endl;
}
```

▼ Fractional Knapsack

```
/*
    We are given 'n' items with {weight,value} of each item and
    the capacity of knapsack (Sack) "W". We need to put these items in
    knapsack such that final value of items in knapsack is maximum.
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    float capacitity;
    cin>>capacitity;
    vector<pair<float,float>>p;
    for(int i=0;i<n;i++){
        float weight,value;
```

```
        cin>>weight>>value;
        p.push_back({weight,value});
    }

    vector<pair<float,float>>q;
    for(int i=0;i<n;i++){
        q.push_back({p[i].first,(p[i].second/p[i].first)});
    }
    sort(q.begin(),q.end(),[&](pair<float,float>&a,pair<float,float>&b){
        return a.second>b.second;
    });

    for(auto it:q){
        cout<<it.first<<" "<<it.second<<endl;
    }

    float filledSpace=0;
    float valueTaken=0;
    int i=0;
    while(filledSpace<=capacitity && i<q.size()){
        if(filledSpace+q[i].first<=capacitity){
            filledSpace+=q[i].first;
            valueTaken+=q[i].first*q[i].second;
        }
        else{
            float spaceLEft=capacitity-filledSpace;
            filledSpace+=spaceLEft;
            valueTaken+=spaceLEft*q[i].second;
        }
        i++;
    }

    cout<<"Stolen Weight : "<<filledSpace<<endl;
    cout<<"Stolen value : "<<valueTaken<<endl;
    return 0;
}
```

▼ Minimum and Maximum Sum Of Difference

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<int>arr(n);
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    sort(arr.begin(),arr.end());
    long long int mn=0,mx=0;
    for(int i=0;i<n/2;i++){
        mx+=abs(arr[i+n/2]-arr[i]);
        mn+=abs(arr[2*i+1]-arr[2*i]);
    }

    cout<<mx<<" "<<mn<<endl;
    return 0;
}
```

▼ Graph Theory

▼ Adjacency Matrix

```
/*
    Components of a graph
    1.  Nodes - These are the stages or vertices.
        For instance, users in case of facebook
    2.  Edges - Links between states in a graph.
        For instance, connections between users.

          \--> Undirected - Two way edges
          \--> Directed - One way edges

    Representation of graphs ---> Adjacency matrix
                             ---> Adjacency List
```

```
    1. Adjacency matrix : 2D array where a[i][j] = 1 if there is an
       edge from i to j else a[i][j] = 0
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;

    //declare the adjacent matrix
    int adj[n+1][n+1];

    // take edges as input
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u][v]=1;
        adj[u][v]=1;
    }
    return 0;

    /*
        Disadvantages of adjacency matrix
        1.  Can be used for smaller values of n, you cannot create a
            n*n matrix for n equal to 10^5, in this case we use adjacency
            list
    */
}
```

▼ Adjacency List

```
/*
            1---2--|
            |   |  4
            5---3--|
        1 is the adjacency list of 2
        2 is an adjacent node of 1
        1 is an adjacent node of 2
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    // if it is a weighted graph than we just add a pair instead of int
    // and take the input of weight
    vector<vector<int>> adj(n+1);
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;

        // If the graph is directed graph one of these two lines will be deleted
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
}
```

▼ BFS Traversal

```
/*
    The basic rule is first visit the adjacent nodes and then
    carry on with other nodes

    Now there can be multiple components of a tree so, inorder
    to ensure safety we use a for loop to traverse through each node
    and make a visited array

    for bfs funcion we make a queue and insert first element and
    continue to traverse until the queue does not become empty
    and look up to adjacency list and push and mark visited
*/
```

```
#include<bits/stdc++.h>
using namespace std;

void bfs(vector<int> &visited,int i,vector<int>&bfsArr, vector<vector<int>>&a){
    queue<int> q;
    q.push(i);
    visited[i]=1;
    while(!q.empty()){
        int top=q.front();
        q.pop();
        bfsArr.push_back(top);
        for(int i=0;i<a[top].size();i++){
            if(visited[a[top][i]]!=1){
                visited[a[top][i]]=1;
                q.push(a[top][i]);
            }
        }
    }
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<vector<int>> a(n+1);
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        a[v].push_back(u);
        a[u].push_back(v);
    }
    vector<int> visited(n+1,0);
    vector<int> bfsArr;

    for(int i=1;i<=n;i++){
        if(visited[i]==0){
            bfs(visited,i,bfsArr,a);
        }
    }
    for(auto it:bfsArr){
        cout<<it<<" ";
    }
    cout<<endl;
}
```

▼ DFS Traversal

```
#include<bits/stdc++.h>
using namespace std;
class solution{
void dfs(int node,vector<int>& vis,vector<int>adj[],vector<int>& storeDfs){
        storeDfs.push_back(node);
        vis[node]=1;
        for(auto it:adj[node]){
            if(!vis[it]){
                dfs(it,vis,adj,storeDfs);
            }
        }
    }
    public:
    vector<int>dfsOfGraph(int V,vector<int>adj[]){
        vector<int>storeDfs;
        vector<int>vis(V+1,0);
        for(int i=1;i<=V;i++){
            if(!vis[i]){
                dfs(i,vis,adj,storeDfs);
            }
        }
        return storeDfs;
    }
};

int main(){
    solution* x=new solution();
    int n,m;
    cin>>n>>m;
    vector<int> a[n+1];
    for(int i=0;i<m;i++){
        int u,v;
```

```
        cin>>u>>v;
        a[v].push_back(u);
        a[u].push_back(v);
    }
    vector<int>dfsArr=x->dfsOfGraph(n,a);
    for(auto it:dfsArr){
        cout<<it<<" ";
    }
    cout<<endl;
}
```

▼ Detect Cycle Using BFS Method 1

```
/*
    everything else remains the same just there is a small change and
    that is if adjacent node at any point is already visited than we return true
*/
#include<bits/stdc++.h>
using namespace std;
bool bfs(vector<int> &visited,int i,vector<int>&bfsArr,vector<vector<int>>&a){
    queue<int> q;
    q.push(i);
    visited[i]=1;
    while(!q.empty()){
        int top=q.front();
        q.pop();
        bfsArr.push_back(top);
        for(int i=0;i<a[top].size();i++){
            if(visited[a[top][i]]==1){
                return true;
            }
            else if(visited[a[top][i]]!=1){
                visited[a[top][i]]=1;
                q.push(a[top][i]);
            }
        }
    }
    return false;
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<vector<int>> a(n+1);
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        a[v].push_back(u);
        a[u].push_back(v);
    }
    vector<int> visited(n+1,0);
    vector<int> bfsArr;
    bool answer;
    for(int i=1;i<=n;i++){
        if(visited[i]==0){
            answer=bfs(visited,i,bfsArr,a);
            if(answer){
                break;
            }
        }
    }
    if(answer){
        cout<<"Cycle Detected\n";
    }
    else{
        cout<<"Cycle not detected\n";
    }
}
```

▼ Detect Cycle Using BFS Method 2

```
#include <bits/stdc++.h>
using namespace std;
bool bfs(int s, int V, vector<int> adj[], vector<int> &visited){
    vector<int> parent(V, -1);
```

```
    // Create a queue for BFS
    queue<pair<int, int>> q;

    visited[s] = true;
    q.push({s, -1});

    while (!q.empty()){
        int node = q.front().first;
        int par = q.front().second;
        q.pop();

        for (auto it : adj[node]){
            if (!visited[it]){
                visited[it] = true;
                q.push({it, node});
            }
            else if (par != it)
                return true;
        }
    }
    return false;
}
// { Driver Code Starts.
int main(){
    int V, E;
    cin >> V >> E;
    vector<int> adj[V];
    for (int i = 0; i < E; i++){
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> visited(V + 1, 0);
    bool answer;
    for (int i = 1; i <= V; i++){
        if (visited[i] == 0){
            answer = bfs(i, V, adj, visited);
            if (answer){
                break;
            }
        }
    }

    if (answer){
        cout << "Cycle Detected\n";
    }
    else{
        cout << "Cycle not detected\n";
    }

    return 0;
}
```

▼ Detect Cycle using DFS

```
#include<bits/stdc++.h>
using namespace std;
bool dfsCycle(int node, int parent, vector<int> &vis, vector<int> adj[]) {
        vis[node] = 1;
        for(auto it: adj[node]) {
            if(!vis[it]) {
                // If somewhere deeper returns a true than this must also return true
                if(dfsCycle(it, node, vis, adj))
                    return true;
            }

            // the element behid the current and next if they are not equal than somebody
            // must have visited it previiously indicating a cycle
            else if(it!=parent)
                return true;
        }

        return false;
}

int main(){
```

```
    int n,m;
    cin>>n>>m;
    vector<int>adj[n+1];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[v].push_back(u);
        adj[u].push_back(v);
    }
    vector<int> visited(n+1,0);
    vector<int> bfsArr;
    bool answer;
    for(int i=1;i<=n;i++){
        if(visited[i]==0){
            answer=dfsCycle(i,-1,visited,adj);
            if(answer){
                break;
            }
        }
    }
    if(answer){
        cout<<"Cycle Detected\n";
    }
    else{
        cout<<"Cycle not detected\n";
    }
}
```

▼ Bipartite Graph Using BFS

```
/*
    A graph is called as a bipartite graph if we can color every node such that
    two adjacent nodes do not have same color
    1. If a graph has a odd length cycle than it cannot be a bipartite graph
    2. else it is
*/
#include <bits/stdc++.h>
using namespace std;

bool bipartiteBfs(int src, vector<int> adj[], int color[]) {
    queue<int>q;
    q.push(src);
    color[src] = 1;
    while(!q.empty()) {
        int node = q.front();
        q.pop();

        for(auto it : adj[node]) {
            if(color[it] == -1) {
                color[it] = 1 - color[node];
                q.push(it);
            } else if(color[it] == color[node]) {
                return false;
            }
        }
    }
    return true;
}
bool checkBipartite(vector<int> adj[], int n) {
    int color[n];
    memset(color, -1, sizeof color);
    for(int i = 0;i<n;i++) {
        if(color[i] == -1) {
            if(!bipartiteBfs(i, adj, color)) {
                return false;
            }
        }
    }
    return true;
}
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n+1];
    for(int i = 0;i<m;i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
```

```
        adj[v].push_back(u);
    }

    if(checkBipartite(adj, n)) {
        cout << "yes";
    } else {
        cout << "No";
    }
    return 0;
}
```

▼ Bipartite Graph Using DFS

```
#include <bits/stdc++.h>
using namespace std;

bool bipartiteDfs(int node, vector<int> adj[], int color[]) {
    for(auto it : adj[node]) {
        if(color[it] == -1) {
            color[it] = 1 - color[node];
            if(!bipartiteDfs(it, adj, color)) {
                return false;
            }
        } else if(color[it] == color[node]) return false;
    }
    return true;
}
bool checkBipartite(vector<int> adj[], int n) {
    int color[n];
    memset(color, -1, sizeof color);
    for(int i = 0;i<n;i++) {
        if(color[i] == -1) {
            color[i] = 1;
            if(!bipartiteDfs(i, adj, color)) {
                return false;
            }
        }
    }
    return true;
}
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
    for(int i = 0;i<m;i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    if(checkBipartite(adj, n)) {
        cout << "yes";
    } else {
        cout << "No";
    }
    return 0;
}
```

▼ Cycle In Directed Graph DFS

```
/*
    Refer 2:00 to 3:07 for, Why the previous algorithm fails which was for undirected graph
*/
#include<bits/stdc++.h>
using namespace std;
class Solution {
private:
    bool checkCycle(int node,vector<int>adj[],int vis[],int dfsVis[]){
        vis[node] = 1;
        dfsVis[node] = 1;
        for(auto it : adj[node]) {
            if(!vis[it]) {
                if(checkCycle(it, adj, vis, dfsVis)) return true;
            } else if(dfsVis[it]) {
```

```
                return true;
            }
        }
        dfsVis[node] = 0;
        return false;
    }
public:
    bool isCyclic(int N, vector<int> adj[]) {
       int vis[N], dfsVis[N];
       memset(vis, 0, sizeof vis);
       memset(dfsVis, 0, sizeof dfsVis);
       for(int i = 0;i<N;i++) {
           if(!vis[i]) {
               if(checkCycle(i, adj, vis, dfsVis)) {
                   return true;
               }
           }
       }
       return false;
    }
};
int main(){
        int V, E;
        cin >> V >> E;
        vector<int> adj[V];
        for(int i = 0; i < E; i++){
            int u, v;
            cin >> u >> v;
            adj[u].push_back(v);
        }
        Solution obj;
        cout << obj.isCyclic(V, adj) << "\n";
    return 0;
}
```

▼ Topological Sort DFS

```
/*
    In topological sort, the order in which the links are present in the
    graph must be followed in the answer also
    meaning like if there is a link like
                  1 ---------> 2, The link is towards 2
    So in the answer 1 must come before 2.
    There can be multiple topological sort for a given graph
    It should be a directed graph and acyclic graph

    Firstly run a for loop as usual from 0 to number of nodes in the graph
    We would also require a stack
    and also a visited array

    Start iterating over N and then mark the current element as visited
    And if there are no adjacent node for any node than we push that into stack

    if they have some nodes than call recursively if we find any visited node as adjacent
    than we move to next adjacent node and push and leave

    This way we are traversing in such a way that the tail of the arrow get inserted at the last
    so when we pop the the element than it comes first and that is what we wanted
*/

#include<bits/stdc++.h>
using namespace std;
class solution{
    void findTopoSort(int node,vector<int>&vis,stack<int>&st,vector<int>adj[]){
        vis[node]=1;
        for(auto it:adj[node]){
            if(!vis[it]){
                findTopoSort(it,vis,st,adj);
            }
        }
        st.push(node);
    }
    public:
    vector<int> topoSort(int N,vector<int>adj[]){
        stack<int>st;
        vector<int>vis(N,0);
        // Here the nodes are zero based so use i=0 we can change it according to problem
        for(int i=0;i<N;i++){
```

```
            if(!vis[i]){
                findTopoSort(i,vis,st,adj);
            }
        }
        vector<int>topo;
        while(!st.empty()){
            topo.push_back(st.top());
            st.pop();
        }
        return topo;
    }
};
int main(){
    int n,m;
    cin>>n>>m;
    vector<int>adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
    }
    solution x;
    vector<int>sol=x.topoSort(n,adj);
    for(auto it:sol){
        cout<<it<<" ";
    }
    cout<<endl;
}
```

▼ Topological Sort BFS (Kahn's Algorithm)

```
/*
    Firstly create an array with indegree
    create a queue
    Push all the elements with indegree 0 into the queue
    Than pop the front of queue and add into ans and check
    adjacent nodes and reduce their indegree by 1
    again check if is there any node with indegree 0
    repeat this until the queue becomes empty

    Known as : Kahn's Algorithm
*/

#include<bits/stdc++.h>
using namespace std;
class solution{
    public:
    vector<int> topoSort(int n,vector<int> adj[]){
        queue<int> q;

        // This will be a zero index based graph
        vector<int> indegree(n,0);
        for(int i=0;i<n;i++){
            for(auto it:adj[i]){
                indegree[it]++;
            }
        }

        for(int i=0;i<n;i++){
            if(indegree[i]==0){
                q.push(i);
            }
        }

        vector<int> topo;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            topo.push_back(node);
            for(auto it:adj[node]){
                indegree[it]--;
                if(indegree[it]==0){
                    q.push(it);
                }
            }
        }
        return topo;
    }
```

```
};

int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
    }
    solution x;
    vector<int> ans=x.topoSort(n,adj);
    for(auto it:ans){
        cout<<it<<" ";
    }
    cout<<"\n";

    return 0;
}
```

▼ Kahn For Cycle Detection (Reverse of Kahn)

```
/*
    Here we will be using the reverse logic of kahn's algorithm because
    it works only for directed acyclic graph and if are not able to generate
    topological sort than that means we have a cycle else not
*/

#include<bits/stdc++.h>
using namespace std;
class solution{
    public:
    bool isCyclic(int n,vector<int>adj[]){
        queue<int> q;
        vector<int> indegree(n,0);
        for(int i=0;i<n;i++){
            for(auto it:adj[i]){
                indegree[i]++;
            }
        }

        for(int i=0;i<n;i++){
            if(indegree[i]==0){
                q.push(i);
            }
        }

        int cnt=0;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            cnt++;
            for(auto it:adj[node]){
                indegree[it]--;
                if(indegree[it]==0){
                    q.push(it);
                }
            }

        }
        if(cnt==n){
            return false;
        }
        else{
            return true;
        }
    }
};

int main(){
    int n,m;
    cin>>n>>m;
    vector<int>adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
```

```
    }
    solution x;
    bool ans=x.isCyclic(n,adj);
    if(ans){
        cout<<"YES\n";
    }
    else{
        cout<<"NO\n";
    }

    return 0;
}
```

▼ Shortest Path In Undirected Graph Unit Weight

```
/*
    Declare a distance array similar to number of nodes
    mark at as INT_MAX
    make queue and insert the source and mark dis[node]=0
    now pop the element from queue and check adj nodes do this for all nodes
    and keep increasing distance and also compare and keep the shortest distance

    we are using BFS because it visits the nodes in sequential manner
*/

#include<bits/stdc++.h>
using namespace std;

void bfs(vector<int> adj[],int n,int src){
    int dist[n];
    for(int i=0;i<n;i++){
        dist[i]=INT_MAX;
    }
    queue<int>q;
    dist[src]=0;
    q.push(src);
    while(!q.empty()){
        int node=q.front();
        q.pop();
        for(auto it:adj[node]){
            if(dist[node]+1<dist[it]){
                dist[it]=dist[node]+1;
                q.push(it);
            }
        }
    }
    for(int i=0;i<n;i++){
        cout<<dist[i]<<" ";
    }
    cout<<endl;
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    bfs(adj,n,0);
}
```

▼ Shortest Path Directed Graph Acyclic

```
#include<bits/stdc++.h>
#define INF INT_MAX
using namespace std;
void findTopoSort(int node,int vis[],stack<int>&st,vector<pair<int,int> >adj[]){
    vis[node]=1;
    for(auto it:adj[node]){
        if(!vis[it.first]){
            findTopoSort(it.first,vis,st,adj);
        }
```

```
    }
    st.push(node);
}

void shortestPath(int src,int n,vector<pair<int,int>>adj[]){
    int vis[n]={0};
    stack<int>st;
    for(int i=0;i<n;i++){
        if(!vis[i]){
            findTopoSort(i,vis,st,adj);
        }
    }

    int dist[n];
    for(int i=0;i<n;i++){
        dist[i]=INF;
    }
    dist[src]=0;
    while(!st.empty()){
        int node=st.top();
        st.pop();
        if(dist[node]!=INF){
            for(auto it:adj[node]){
                if(dist[node]+it.second<dist[it.first]){
                    dist[it.first]=dist[node]+it.second;
                }
            }
        }
    }
    for(int i=0;i<n;i++){
        cout<<dist[i]<<" ";
    }
    cout<<endl;
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<pair<int,int>>adj[n];
    for(int i=0;i<m;i++){
        int u,v,wt;
        cin>>u>>v>>wt;
        adj[u].push_back({v,wt});
    }
    shortestPath(0,n,adj);
}
```

▼ Shortest Path In UnDirected Graph Dijkstra's Algorithm

```
/*
    Shortest Path In Undirected Graphs
    declare a minimum priority queue (distance,node) such that node
    with minimum distance stays at the top
    define a distance array make as INT_MAX
    mark distance[src]=0 and insert it into the priority queue
    pick up first node in the queue and look at the adjacency list (node,weight)
    now calculate the distance for each node and keep updating it if it is better
    and if that is better push it into the queue

    repeat all the steps until queue is empty
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m,src;
    cin>>n>>m;
    vector<pair<int,int>> g[n+1];

    int a,b,wt;
    for(int i=0;i<m;i++){
        cin>>a>>b>>wt;
        g[a].push_back({b,wt});
        g[b].push_back({a,wt});
    }

    cin>>src;

    // priority_queue(distance,from where it came)
```

```
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
    vector<int> distTo(n+1,INT_MAX);

    distTo[src]=0;
    pq.push({0,src});  // (distance,from)
    while(!pq.empty()){
        int dist=pq.top().first;
        int prev=pq.top().second;
        pq.pop();
        for(auto it:g[prev]){
            int next=it.first;
            int nextDist=it.second;
            if(distTo[next]>distTo[prev]+nextDist){
                distTo[next]=distTo[prev]+nextDist;
                pq.push({distTo[next],next});
            }
        }
    }
    for(auto it:distTo){
        cout<<it<<" ";
    }
    cout<<"\n";
}
```

▼ Prims Algorithm Minimum Spanning tree

```
/*
    When you can convert a graph into a tree with exaclty N nodes
    and n-1 edges, every node must be reachable with every other node

    Take any one node and find the least weighted edge among adjacents edges
    Now than check the adjacents edges of of the both connected nodes
    After finding the minimum one than connect it
    Again check of these three nodes and keep repeating until all nodes are visited

    One more thing there must not be any cycle formed

    ----> Implimentaion
    Three arrays :
        1. the key array : initialised with infinity and 0th index as 0
        2. the mst array : initialised with false
        3. the parent array : initialised with -1

    Now in starting mark mst of 0 = true and look for adjacent nodes
    than take the weights of adjacents weights and and mark them
    according to the node's index in key array,
    now iterate over key array anf find the least weight mark mst for that
    node as true and also put the previous node into parent of that node

    repeat until all mst becomes true than go to parent array and iterate from
    1 to 4
*/

#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    int a,b,wt;
    vector<pair<int,int>> adj[n];
    for(int i=0;i<m;i++){
        cin>>a>>b>>wt;
        adj[a].push_back({b,wt});
        adj[b].push_back({a,wt});
    }
    int parent[n];
    int key[n];
    bool mstSet[n];
    for(int i=0;i<n;i++){
        key[i]=INT_MAX,mstSet[i]=false,parent[i]=-1;
    }
    key[0]=0;
    for(int count=0;count<n-1;count++){
        int mini=INT_MAX,u;

        // Finding the index with minimum possible key value
        for(int v=0;v<n;v++){
            if(mstSet[v]==false && key[v]<mini){
```

```
                mini=key[v],u=v;
            }
        }
        mstSet[u]=true;

        for(auto it:adj[u]){
            int v=it.first;
            int weight=it.second;

            // if is it not a part of MST and also the current weight
            // is grater than previous key than put that weight into key[index]
            // Also marking the parent with previous node
            if(mstSet[v]==false && weight<key[v]){
                parent[v]=u,key[v]=weight;
            }
        }
    }
    for(int i=1;i<n;i++){
        cout<<parent[i]<<" - "<<i<<"\n";
    }
    return 0;
}

// Time Complexity : greater than O(n^2)
// Space Complexity : parent,key,mstSet,Adjlist
```

▼ Optimized Prims Algorithm

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    int a,b,wt;
    vector<pair<int,int>> adj[n];
    for(int i=0;i<m;i++){
        cin>>a>>b>>wt;
        adj[a].push_back({b,wt});
        adj[b].push_back({a,wt});
    }
    int parent[n];
    int key[n];
    bool mstSet[n];
    for(int i=0;i<n;i++){
        key[i]=INT_MAX,mstSet[i]=false,parent[i]=-1;
    }
    key[0]=0;
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
    pq.push({0,0});
    for(int count=0;count<n-1;count++){
        int mini=INT_MAX,u;

        // Finding the index with minimum possible key value
        int u=pq.top().second;
        pq.pop();
        mstSet[u]=true;

        for(auto it:adj[u]){
            int v=it.first;
            int weight=it.second;

            // if is it not a part of MST and also the current weight
            // is grater than previous key than put that weight into key[index]
            // Also marking the parent with previous node
            if(mstSet[v]==false && weight<key[v]){
                parent[v]=u;
                pq.push({key[v],v});
                key[v]=weight;
            }
        }
    }
    for(int i=1;i<n;i++){
        cout<<parent[i]<<" - "<<i<<"\n";
    }
    return 0;
}

// time completity : N logN
```

▼ Disjoint Set

```
/*
    Every node will be parent by himself in starting
    There will be a rank array whivh will store the ranks of nodes
    in the array, and initially everything will be zero

    When we attach two similar kind of rank nodes than make sure to increasing the parent by 1
    The lesser rank one becomes the child of higher rank
    When the ranks are different than there is no need to incerase the rank
*/

#include<bits/stdc++.h>
using namespace std;

int parent[100000];
int rankArr[100000];

void makeSet(int n){
    for(int i=0;i<=n;i++){
        parent[i]=i;
        rankArr[i]=0;
    }
}

int findPar(int node){
    // Time complexity = O(4*a) --> constant time
    if(node==parent[node]){
        return node;
    }

    // This step is the path compresion step
    return parent[node]=findPar(parent[node]);
}

void unionOper(int u,int v){
    u=findPar(u);
    v=findPar(v);
    if(rankArr[u]<rankArr[v]){
        parent[u]=v;
    }
    else if(rankArr[v]<rankArr[u]){
        parent[v]=u;
    }
    else{
        parent[v]=u;
        rankArr[u]++;
    }
}

int main(){
    int n;
    cin>>n;
    makeSet(n);
    while(n--){
        int u,v;
        cin>>u>>v;
        unionOper(u,v);
    }
    // if u ans v are belonging to same component or not
    int u,v;
    cin>>u>>v;
    // Basically finding the main parent
    if(findPar(u)!=findPar(v)){
        cout<<"Different Component\n";
    }
    else{
        cout<<"Same Parent\n";
    }
}
```

▼ Kruskal's Algorithm for minimum spanning tree

```
/*
    Sort all the edges according to weight
    Take first one and check if they both belong to same component
    and if they do not than connect them else leave them

    at the end we will have a minimum spanning tree

    Time Complexity : O(M logM) + O(M X 4a)
                          ^           ^
                        Sorting     For iteration
          Finally   : O(M logM)
    Space Complexity : O(N)
*/

#include <bits/stdc++.h>
using namespace std;
struct node
{
    int u;
    int v;
    int wt;
    node(int first, int second, int weight)
    {
        u = first;
        v = second;
        wt = weight;
    }
};

bool comp(node a, node b)
{
    return a.wt < b.wt;
}

int findPar(int u, vector<int> &parentArr)
{
    if (u == parentArr[u])
    {
        return u;
    }
    return parentArr[u] = findPar(parentArr[u], parentArr);
}

void unionn(int u, int v, vector<int> &parentArr, vector<int> &rankArr)
{
    u = findPar(u, parentArr);
    v = findPar(v, parentArr);
    if (rankArr[u] < rankArr[v])
    {
        parentArr[u] = v;
    }
    else if (rankArr[v] < rankArr[u])
    {
        parentArr[v] = u;
    }
    else
    {
        parentArr[v] = u;
        rankArr[u]++;
    }
}
int main()
{
    int N, m;
    cin >> N >> m;
    vector<node> edges;
    for (int i = 0; i < m; i++)
    {
        int u, v, wt;
        cin >> u >> v >> wt;
        edges.push_back(node(u, v, wt));
    }
    sort(edges.begin(), edges.end(), comp);

    vector<int> parentArr(N);
    for (int i = 0; i < N; i++)
    {
        parentArr[i] = i;
    }
```

```
    vector<int> rankArr(N, 0);

    int cost = 0;
    vector<pair<int, int>> mst;
    for (auto it : edges)
    {
        if (findPar(it.v, parentArr) != findPar(it.u, parentArr))
        {
            cost += it.wt;
            mst.push_back({it.u, it.v});
            unionn(it.u, it.v, parentArr, rankArr);
        }
    }
    cout << cost << endl;
    for (auto it : mst)
    {
        cout << it.first << " - " << it.second << endl;
    }
    return 0;
}
```

▼ Bridges In Graphs

```
/*
    Bridges when removed from the graph separate the graph into two parts
    Two arrays
        time of insertion : stores the time during the insertion during DFS
        lowest time : Lowest insertion time among all nodes
        whenever we find a backedge then we replace the low time of that node
        with insertion time of its ancestor

                            (node) ----------(node)
                            /      \       |   /
                           /        \    > |  /
                       (node)       \   (node)
                       /    | <      \
                      /     |         \
                  (node)--(node)      \
                                      (node)----(node)
                                        /   \   |
                                       /     \  |
                                   (node)   (node)
                                       \
                                      (node)
  Back edges can never be bridges
*/

// Time complexity = O ( Nodes + Edges )
// Space Complexity = O(2N)
#include <bits/stdc++.h>
using namespace std;
void dfs(int node, int parent,vector<int>&vis,vector<int>&tin,vector<int> &low, int &timer, vector<int> adj[]) {
    vis[node] = 1;
    tin[node] = low[node] = timer++;
    for(auto it: adj[node]) {
        if(it == parent) continue;

        if(!vis[it]) {
            dfs(it, node, vis, tin, low, timer, adj);
            low[node] = min(low[node], low[it]);
            if(low[it] > tin[node]) {
                cout << node << " " << it << endl;
            }
            // if it is inserted than low of node will be time of insertion
            // Because time of insertion will be the lowest in that case
        } else {
            low[node] = min(low[node], tin[it]);
        }
    }
}
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];
    for(int i = 0;i<m;i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
```

```
        adj[v].push_back(u);
    }

    vector<int> tin(n, -1);
    vector<int> low(n, -1);
    vector<int> vis(n, 0);
    int timer = 0;
    for(int i = 0;i<n;i++) {
        if(!vis[i]) {
            dfs(i, -1, vis, tin, low, timer, adj);
        }
    }

    return 0;
}
```

▼ Points Of Articulation

```
/*
    The articulation points are the nodes which when taken out leads to
    two or more components
    condition : low [it] >= time [node] && parent != -1
    There is greater than equal to because we must reach the other end of
    the node
*/
#include<bits/stdc++.h>
using namespace std;
void dfs(int node,int parent,vector<int>&vis,vector<int>&time,vector<int>&low
  ,int &timer,vector<int>adj[],vector<int>&isArti){
    vis[node]=1;
    time[node]=low[node]=timer++;
    int child=0;
    for(auto it:adj[node]){
        if(it==parent){
            continue;
        }
        if(!vis[it]){
            dfs(it,node,vis,time,low,timer,adj,isArti);
            low[node]=min(low[node],low[it]);
            child++;
            if(low[it]>=time[node] && parent!=-1){
                isArti[node]=1;
            }
        }
        else{
            low[node]=min(low[node],time[it]);
        }
    }

    // This case is for the starting parent
    if(parent==-1 && child>1){
        isArti[node]=1;
    }
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<int>adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int>time(n,-1);
    vector<int>low(n,-1);
    vector<int>vis(n,0);
    vector<int>isArti(n,0);
    int timer=0;
    for(int i=0;i<n;i++){
        if(!vis[i]){
            dfs(i,-1,vis,time,low,timer,adj,isArti);
        }
    }
    for(int i=0;i<n;i++){
        if(isArti[i]==1){
            cout<<i<<" ";
```

```
        }
    }
    cout<<endl;
}
```

▼ Strongly Connected Components Kosaraju's Algorithm

```
/*
    1. Find the topological sort --> O(N)
    2. Transpose the graph - reverse the edges --> O(N+E)
    3. DFS according to stack data  --> O(N+E)

    space complexity = O(N+E)+O(N)+O(N)
*/

#include<bits/stdc++.h>
using namespace std;
void dfs(stack<int>&st,int node,vector<int>&vis,vector<int>adj[]){
    vis[node]=1;
    for(auto it:adj[node]){
        if(!vis[it]){
            dfs(st,it,vis,adj);
        }
    }
    st.push(node);
}

void revDFS(int node,vector<int>&vis,vector<int>transpose[]){
    cout<<node<<" ";
    vis[node]=1;
    for(auto it:transpose[node]){
        if(!vis[it]){
            revDFS(it,vis,transpose);
        }
    }
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<int>adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
    }

    stack<int>st;
    vector<int> vis(n,0);
    for(int i=0;i<n;i++){
        if(!vis[i]){
            dfs(st,i,vis,adj);
        }
    }

    vector<int>transpose[n];
    for(int i=0;i<n;i++){
        vis[i]=0;

        // Previously there was an edge from i to it now from it to i
        for(auto it:adj[i]){
            transpose[it].push_back(i);
        }
    }

    while(!st.empty()){
        int node=st.top();
        st.pop();
        if(!vis[node]){
            cout<<"SEC : ";
            revDFS(node,vis,transpose);
            cout<<endl;
        }
    }
}
```

▼ Bellman Ford's Algorithm

```
/*
    For finding shortest path in negative weighted graph
    Dosn't work for cyclic graph with negative edges because the distance
    will keep on decreasing whenever we iterate over the graph

    Also can be used to detect if there is a negative cycle
    works for only directed graphs

    1. Take input
    2. dist[src]=0 all other INT_MAX
    3. Relax all the edges N-1 times
        Relaxatation means if(dis[u]+wt<dist[v]){
                                dist[v]=dist[u]+wt;
                            }
    if the dustance reduces after doing one more relaxation after N-1 than it has cycle

    Time Complexity = O(N-1)*O(Edges)
    Space Complexity = O(N)
*/

#include<bits/stdc++.h>
using namespace std;
class node{
    public:
    int u;
    int v;
    int wt;
    node(int first,int second,int weight){
        u=first;
        v=second;
        wt=weight;
    }
};

int main(){
    int n,m;
    cin>>n>>m;
    vector<node>edges;
    for(int i=0;i<m;i++){
        int u,v,wt;
        cin>>u>>v>>wt;
        edges.push_back(node(u,v,wt));
    }
    int src;
    cin>>src;
    int inf=10000000;
    vector<int>dist(n,inf);
    for(int i=0;i<=n-1;i++){
        for(auto it:edges){
            if(dist[it.u]+it.wt<dist[it.v]){
                dist[it.v]=dist[it.u]+it.wt;
            }
        }
    }

    int fl=0;
    for(auto it:edges){
        if(dist[it.u]+it.wt<dist[it.v]){
            cout<<"Negative Cycle\n";
            fl=1;
            break;
        }
    }

    if(!fl){
        for(int i=0;i<n;i++){
            cout<<i<<" "<<dist[i]<<endl;
        }
    }

    return 0;
}
```

▼ Trees

▼ Binary Tree Node structure

```
struct node{
    int data;
    struct node* left;
    struct node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};
```

▼ Preorder

```
void preorder(struct node* root){
    if(root==NULL){
        return;
    }
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
}
```

▼ Inorder

```
void inorder(struct node* root){
    if(root==NULL){
        return;
    }
    inorder(root->left);
    cout<<root->data<<" ";
    inorder(root->right);
}
```

▼ Postorder

```
void postorder(struct node* root){
    if(root==NULL){
        return;
    }
    postorder(root->left);
    postorder(root->right);
    cout<<root->data<<" ";
}
```

▼ Basic Implementation

```
#include"bits/stdc++.h"
using namespace std;
struct node{
    int data;
    struct node* left;
    struct node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

int main(){
    struct node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    /*
        1
       / \
```

```
      2   3

    */

    root->left->left=new node(4);
    root->left->right=new node(5);
    /*
            1
           / \
          2   3
         / \
        4   5

    */

   return 0;
}
```

▼ Binary Tree from postorder and inorder

```
#include<iostream>
using namespace std;

/*

------> Algorithm
1. Start from the end of postorder and Pick element to create a node
2. Decrement postorder index
3. Search for picked element's pos in inorder
4. Call to build right subtree from inorder's pos+1 to n
5. Call to build left subtree from inorder's 0 to pos-1
6. Return the node

*/
// From postorder and preorder we cannot build one tree
class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

int search(int inorder[],int start,int end,int curr){
    for(int i=start;i<=end;i++){
        if(inorder[i]==curr){
            return i;
        }
    }
    return -1;
}

node* builtTree(int postorder[],int inorder[],int start,int end){
    static int idx=end;
    if(start>end){
        return NULL;
    }
    int curr=postorder[idx];
    idx--;
    node* node1=new node(curr);
    if(start==end){
        return node1;
    }
    int pos=search(inorder,start,end,curr);
    node1->right=builtTree(postorder,inorder,pos+1,end);
    node1->left=builtTree(postorder,inorder,start,pos-1);

    return node1;
}

void inoderPrint(node* root){
    if(root==NULL){
        return;
    }
```

```
    inoderPrint(root->left);
    cout<<root->data<<" ";
    inoderPrint(root->right);
}

int main(){
    int postorder[]={4,5,2,6,7,3,1};
    int inorder[]={4,2,5,1,6,3,7};
    node* root=builtTree(postorder,inorder,0,6);
    inoderPrint(root);
}
```

▼ Binary tree from preorder and inorder

```
#include<iostream>
using namespace std;
/*

------> Algorithm
1. Pick element from preorder and create a node
2. Increment preorder index
3. Search for picked element's pos in inorder
4. Call to build left subtree from inorder's 0 to pos-1
5. Call to build right subtree from inorder's pos+1 to n
6. Return the node

*/
class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// Start and end represent the starting and ending positions of the indorder sequence
int search(int inorder[],int start,int end,int curr){
    for(int i=start;i<=end;i++){
        if(inorder[i]==curr){
            return i;
        }
    }
    return -1;
}

node* builtTree(int preorder[],int inorder[],int start,int end){
    // Since we have recursion and the value will get destroyed once another function is called so to prevent that we use static va
    static int idx=0;
    if(start>end){
        return NULL;
    }
    int curr=preorder[idx];
    idx++;
    node* node1=new node(curr);
    if(start==end){
        return node1;
    }
    int pos=search(inorder,start,end,curr);
    node1->left=builtTree(preorder,inorder,start,pos-1);
    node1->right=builtTree(preorder,inorder,pos+1,end);

    return node1;
}

void inoderPrint(node* root){
    if(root==NULL){
        return;
    }
    inoderPrint(root->left);
    cout<<root->data<<" ";
    inoderPrint(root->right);
}

int main(){
```

```
    int preorder[]={1,2,4,3,5};
    int inorder[]={4,2,1,5,3};
    node* root=builtTree(preorder,inorder,0,4);
    inoderPrint(root);
    return 0;
}
```

▼ Level order Traversal

```
#include"bits/stdc++.h"
using namespace std;

struct node{
    int data;
    struct node* left;
    struct node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

void printLevelOrder(node* root){
    if(root==NULL){
        return;
    }
    queue<node*> q;
    q.push(root);
    q.push(NULL);
    while(!q.empty()){
        node* node1=q.front();
        q.pop();
        if(node1!=NULL){
            cout<<node1->data<<" ";
            if(node1->left){
                q.push(node1->left);
            }
            if(node1->right){
                q.push(node1->right);
            }
        }
        else if(!q.empty()){
            q.push(NULL);
        }
    }
}

int main(){
    struct node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);
    printLevelOrder(root);
}
```

▼ Sum Ok Kth Level nodes

```
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};
```

```
/*

        1
      /   \
     2     3
    / \   / \
   4   5 6   7

*/

int sumAtk(node* root,int k){
    if(root==NULL){
        return -1;
    }
    queue<node*> q;
    q.push(root);
    q.push(NULL);
    int level=0;
    int sum=0;
    while(!q.empty()){
        node* node1=q.front();
        q.pop();
        if(node1 != NULL){
            if(level == k){
                sum+=node1->data;
            }
            if(node1 -> left){
                q.push(node1 -> left);
            }
            if(node1 -> right){
                q.push(node1 -> right);
            }
        }
        else if(!q.empty()){
            q.push(NULL);
            level++;
        }
    }

    return sum;
}

int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);
    int k=2;
    cout<<sumAtk(root,k)<<endl;
}
```

▼ Number of nodes and sum of all nodes

```
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// int countNode(node* root){
//     if(root==NULL){
//         return 0;
//     }
//     return countNode(root->left)+countNode(root->right)+1;
// }
```

```
int sumOfAllNodes(node* root){
    static int sum=0;
    if(root==NULL){
        return 0;
    }
    sum+=root->data;
    sumOfAllNodes(root->left);
    sumOfAllNodes(root->right);

    return sum;
}

// int sumOfAllNodes(node* root){
//     if(root==NULL){
//         return 0;
//     }
//     return sumOfAllNodes(root->left)+sumOfAllNodes(root->right)+root->data;
// }

int countNode(node* root){
    static int sum=0;
    if(root==NULL){
        return 0;
    }
    sum++;
    countNode(root->left);
    countNode(root->right);

    return sum;
}
int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);
    cout<<countNode(root)<<endl;
    cout<<sumOfAllNodes(root)<<endl;
}
```

▼ Height and diameter of a binary tree

```
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// Time complexity = O(n) n=number of elements int the tree
int calcHeight(node* root){
    if(root==NULL){
        return 0;
    }
    int lHeight=calcHeight(root->left);
    int rHeight=calcHeight(root->right);

    return max(lHeight,rHeight)+1;
}

// Time complexity = O(n^2) n=number of nodes
int calcDiameter(node* root){
    if(root==NULL){
        return 0;
    }
    int lHeight=calcHeight(root->left);
    int rHeight=calcHeight(root->right);
```

```
    int currDiameter=lHeight+rHeight+1;

    int lDiameter=calcDiameter(root->left);
    int rDiameter=calcDiameter(root->right);

    return max(currDiameter,max(lDiameter,rDiameter));
}

// Time Complexity = O(n) n=number of nodes
int clacDiameterOpti(node* root,int &height){
    if(root==NULL){
        height=0;
        return 0;
    }
    int lh=0;
    int rh=0;
    int lDiameter=clacDiameterOpti(root->left,lh);
    int rDiamater=clacDiameterOpti(root->right,rh);
    int currDiameter=lh+rh+1;
    height=max(lh,rh)+1;

    return max(currDiameter,max(lDiameter,rDiamater));
}

int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);

    cout<<calcHeight(root)<<endl;
    cout<<calcDiameter(root)<<endl;
    int height=0;
    cout<<clacDiameterOpti(root,height)<<endl;
}
```

▼ Sum Replacement Problem

```
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// the time complexity of O(n) where n is the number of elements
void sumReplace(node* root){
    if(root==NULL){
        return;
    }
    sumReplace(root->left);
    sumReplace(root->right);
    if(root->left!=NULL){
        root->data+=root->left->data;
    }
    if(root->right!=NULL){
        root->data+=root->right->data;
    }
}

void preorder(node* root){
    if(root==NULL){
        return;
    }
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
```

```
}

int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);
    preorder(root);cout<<endl;
    sumReplace(root);preorder(root);cout<<endl;
}
```

▼ Balanced Binary tree

```
// For each node,
// the difference between the heights if the left subtree and right subtree<=1
// mod(leftHeight-rightHeight)<=1

/*
    Balanced Tree
        (node)
         /  \
     (node) (node)
       /
   (node)

*/

/*
    Imbalanced Tree
        (node)
         /  \
     (node) (node)
       /
   (node)
      \
       (node)
*/

#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

int height(node* root){
    if(root==NULL){
        return 0;
    }
    int lh=height(root->left);
    int rh=height(root->right);
    return max(lh,rh)+1;
}
// Total time complexity O(n^2)
bool isBalanced(node* root){
    if(root==NULL){
        return true;
    }
    if(isBalanced(root->left)==false){
        return false;
    }
    if(isBalanced(root->right)==false){
        return false;
    }
    int lh=height(root->left);
    int rh=height(root->right);
    if(abs(lh-rh)<=1){
```

```
        return true;
    }
    else{
        return false;
    }
}

// time complexity = O(n)
bool optimisedBalancedTree(node* root,int &height){
    if(root==NULL){
        height=0;
        return true;
    }
    int lh=0,rh=0;
    if(optimisedBalancedTree(root->left,lh)==false){
        return false;
    }
    if(optimisedBalancedTree(root->right,rh)==false){
        return false;
    }
    height = max(lh,rh)+1;
    if(abs(lh-rh)<=1){
        return true;
    }
    else{
        return false;
    }
}

int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);

    cout<<isBalanced(root)<<endl;
    int height=0;
    cout<<optimisedBalancedTree(root,height)<<endl;
}
```

▼ Right View Of A Binary Tree

```
#include"bits/stdc++.h"
using namespace std;
/*

        1(ans)
       / \
      2   3(ans)
     / \ / \
    4  5 6  7(ans)

*/

/*
-----> Algorithm
1. Travesing the tree level wise
2. Storing the rightmst most node of every level
*/

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// Since we have inserted each element only once into our queue
// so that is why we have trivesed the whole tree
// Time complexity = O(n) n=number of nodes
```

```
void rightView(node* root){
    if(root==NULL){
        return ;
    }
    queue<node*> q;
    q.push(root);
    while(!q.empty()){
        int n=q.size();
        for(int i=0;i<n;i++){
            node* curr=q.front();
            q.pop();
            if(i==n-1){
                cout<<curr->data<<" ";
            }
            if(curr->left){
                q.push(curr->left);
            }
            if(curr->right){
                q.push(curr->right);
            }
        }
    }
    cout<<endl;
}
int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->left->right=new node(5);
    root->right->left=new node(6);
    root->right->right=new node(7);
    rightView(root);
}
```

▼ Left View Of A Binary Tree

```
#include "bits/stdc++.h" // 27.11
using namespace std;
/*

    (ans)1
        / \
  (ans)2   3
      / \ / \
(ans)4  5 6  7

*/

/*
-----> Algorithm
1. Travesing the tree level wise
2. Storing the leftmost most node of every level
*/

class node
{
public:
    int data;
    node *left;
    node *right;
    node(int val)
    {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Since we have inserted each element only once into our queue
// so that is why we have trivesed the whole tree
// Time complexity = O(n) n=number of nodes
void leftView(node *root)
{
    if (root == NULL)
    {
        return;
    }
```

```
    queue<node *> q;
    q.push(root);
    while (!q.empty())
    {
        int n = q.size();
        for (int i = 0; i < n; i++)
        {
            node *curr = q.front();
            q.pop();
            if (i == 0)
            {
                cout << curr->data << " ";
            }
            if (curr->left)
            {
                q.push(curr->left);
            }
            if (curr->right)
            {
                q.push(curr->right);
            }
        }
    }
    cout << endl;
}
int main()
{
    node *root = new node(1);
    root->left = new node(2);
    root->right = new node(3);
    root->left->left = new node(4);
    root->left->right = new node(5);
    root->right->left = new node(6);
    root->right->right = new node(7);
    leftView(root);
}
```

▼ Flattening the binary tree

```
// flatteing the tree into linked without creating another linkedlist
// After flatenning left of each node should become NULL
// right should contain next node in preorder sequence
/*
        1             1
       / \             \
      2   3 ------>     2
         /               \
        4                 3
                           \
                            4
-------> Algorithm
1. Recursively , flattern the left and right subtrees
      1                 1
     / \               / \
    2   3    ------>   2   3
       / \                \
      4   5                4
                            \
                             5
2. Store the left tail and right tail
3. Store right subtree in 'temp' and make left subtree as right subtree

            (tree)          (temp)
               1               3
                \               \
                 2               4
                                  \
                                   5
4. Join right subtree with left tail
5. Return right tail
*/
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
```

```
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

void flattern(node* root){
   if(root==NULL || (root->left==NULL && root->right==NULL)){
      return;
   }
   if(root->left){
      flattern(root->left);
      // Swapping left to right and left to NULL
      node* temp=root->right;
      root->right=root->left;
      root->left=NULL;

      // Now we have to calculate the tail of root's new right to that we can append the temp
      node* t =root->right;
      while(t->right!=NULL){
         t=t->right;
      }
      t->right=temp;
   }

   // Now if there is any branching in right subtree
   flattern(root->right);
}

void preOrderPrint(node* root){
   if(root==NULL){
      return;
   }

   preOrderPrint(root->left);
   cout<<root->data<<" ";
   preOrderPrint(root->right);
}

int main(){
   node* root=new node(4);
   root->left=new node(9);
   root->left->left =new node(1);
   root->left->right=new node(3);
   root->right=new node(5);
   root->right->right=new node(6);
   preOrderPrint(root);cout<<endl;
   flattern(root);
   preOrderPrint(root);
}
```

▼ Nodes at A Distance K

```
/*
Lecture number 27.14
            1             Given node = 5
      1--> / \ <--2           K = 3
          5   2           Ans = {8,9,3,4}
    1--> /   / \ <--3
        6   3   4
    2--> \   ^<--3
          7
     3-->/ \ <--3
        8   9

1.  First case : Going through subtree
    and decrease the K value whenever at another node
2.  Find the distance of all ancestors from target node and name it dth node
    Than Find (K-d)th node in other subtree

*/
// Reckeck
#include"bits/stdc++.h"
using namespace std;

class node{
```

```
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// FirstCase
void printSubtreeNodes(node* root,int k){
    if(root==NULL || k<0){
        return;
    }
    if(k==0){
        cout<<root->data<<" ";
        return;
    }
    printSubtreeNodes(root->left,k-1);
    printSubtreeNodes(root->right,k-1);
}
// Case 2
// The return represents the distance from current node to ancestors
int printNodesAtK(node* root,node* target,int k){
    if(root==NULL){
        // havn't got the target node
        return -1;
    }
    if(root==target){
        printSubtreeNodes(root,k);
        return 0;
    }

    // not doing k-1 because we haven't found our node
    int dl=printNodesAtK(root->left,target,k);
    if(dl!=-1){
        if(dl+1==k){
            cout<<root->data<<" ";
        }
        else{
            printSubtreeNodes(root->right,k-dl-2);
        }
        return 1+dl;
    }
    int dr=printNodesAtK(root->right,target,k);
    if(dr!=-1){
        if(dr+1==k){
            // The current node is at kth distance
            cout<<root->data<<" ";
        }
        else{
            printSubtreeNodes(root->left,k-dr-2);
        }
        return 1+dr;
    }

    return -1;
}

int main(){
   node* root=new node(1);
   root->left=new node(2);
   root->left->left =new node(4);
   root->right=new node(3);
   printNodesAtK(root,root->left,1);
}
```

▼ Maximum Path Length root to leaf

```
/*
        1
       / \
     -12  3
     /   / \
    4   5  -6
```

```
    Here the maximum sum is 1+3+5

-----> Strategy

For each node, compute:
1. Node val
2. Max path through left child + node val
3. Max path through right child + node val
4. Max Path through left child + Max path through right child + node val

Than comparing the path sum with a global variable
than reporting the maximum value
*/

#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

int maxPathSumUtil(node* root,int &ans){
    if(root==NULL){
        return 0;
    }
    int left=maxPathSumUtil(root->left,ans);
    int right=maxPathSumUtil(root->right,ans);

    int nodeMax=max(max(root->data,root->data+left+right),max(root->data+left,root->data+right));
    ans=max(ans,nodeMax);

    // Now we have to return the maxVal form root
    // Now we wont include root->data+left+right in the value because that would make it complete path sum
    int singlePathSum=max(root->data,max(root->data+left,root->data+right));

    return singlePathSum;
}

int maxPathSum(node* root){
    int ans=INT_MIN;
    // This maxPathSumUtil funcion will be a recursive
    // Will calcuate all the 4 values and return the maximun of them
    maxPathSumUtil(root,ans);
    return ans;
}

int main(){
    /*
                1
               / \
              2   3
             /     \
            4       5
    */
   node* root=new node(1);
   root->left=new node(2);
   root->left->left=new node(4);
   root->right=new node(3);
   root->right->right=new node(5);

   cout<<maxPathSum(root);
}
```

▼ Binary Search Trees Implementation

```
// In normal trees the searching takes O(n)
// But in binary search trees it is reduced
// to log(n)
/*
-----> Three rules
```

```
1.  In binary search trees, the left subtree of
    a node contains only nodes with keys lesser
    than the node's key
2.  The right subtree of a node contains only
    nodes with keys hreater than the node's key
3.  The left and right subtree each must also be
    a binary search tree. There must be no
    duplicate nodes


           3
          / \
         2   7
        /   / \
       1   5   8
          / \
         4   6
*/
// How to build a binary search tree from array
// Inorder traversal of a binary search tree gives sorted material
#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

node* insertAtBST(node *root,int val){
    if(root==NULL){
        // Now this becomes out root
        return new node(val);
    }
    if(val<root->data){
        root->left=insertAtBST(root->left,val);
    }
    else{
        root->right=insertAtBST(root->right,val);
    }
    return root;
}

void inorderPrint(node* root){
    if(root==NULL){
        return;
    }
    inorderPrint(root->left);
    cout<<root->data<<" ";
    inorderPrint(root->right);
}

int main(){
    node* root=NULL;
    root=insertAtBST(root,5);
    root =insertAtBST(root,1);
    root=insertAtBST(root,3);
    root=insertAtBST(root,4);
    root=insertAtBST(root,2);
    root=insertAtBST(root,7);
    inorderPrint(root);
}
```

▼ searching In A Binary Tree

```
/*
        Levels      Nodes
          0    --->   n
          1    --->  n/2
          2    --->  n/4
          3    --->  n/8
          .           .
```

```
          .           .
          .           .
          nodes = n, Height = h
          1 + 2 + 2^2 + ..... + 2^(h-1)=n
                  2^(h)-1 = n
                h = log(n+1)
            Time Complexity = O(logn)
*/

#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// Time complexity becomes O(logn)
node* searchInBST(node* root,int key){
    if(root==NULL){
        return NULL;
    }
    if(root->data==key){
        return root;
    }
    if(root->data>key){
        return searchInBST(root->left,key);
    }
    return searchInBST(root->right,key);
}

int main(){
    /*
        ---> Binary search tree
                4
               / \
              2   5
             / \   \
            1   3   6
    */
   node* root=new node(4);
   root->left=new node(2);
   root->left->left=new node(1);
   root->left->right=new node(3);
   root->right=new node(5);
   root->right->right=new node(6);
   if(searchInBST(root,5)==NULL){
       cout<<"does not exists\n";
   }
   else{
       cout<<"exists\n";
   }
}
```

▼ distance Between Two Nodes

```
#include"bits/stdc++.h"
using namespace std;
class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

/*
```

```
-----> Algorithm
1. Find the LCA(Lowest common ancestor)
2. Find distance of n1(d1) and n2(d2) from LCA
3. return (d1+d2)
*/

node* LCA(node* root,int n1,int n2){
    if(root==NULL){
        return NULL;
    }
    if(root->data==n1 || root->data==n2){
        return root;
    }
    node* leftLCA=LCA(root->left,n1,n2);
    node* rightLCA=LCA(root->right,n1,n2);

    // Checking if we got actual LCA's
    if(leftLCA && rightLCA){
        return root;
    }
    // Checking if we didn't got the LCA
    if(!leftLCA && !rightLCA){
        return NULL;
    }
    //
    if(leftLCA){
        return LCA(root->left,n1,n2);
    }
    return LCA(root->right,n1,n2);
}

int findDist(node* root,int k,int dist){
    if(root==NULL){
        return -1;
    }
    if(root->data==k){
        return dist;
    }
    int left=findDist(root->left,k,dist+1);
    if(left!=-1){
        return left;
    }
    return findDist(root->right,k,dist+1);
}

int distBWNodes(node* root,int n1,int n2){
    node* lca=LCA(root,n1,n2);
    int d1=findDist(lca,n1,0);
    int d2=findDist(lca,n2,0);
    return d1+d2;
}

int main(){
    /*
            1
           / \
          2   3
         /     \
        4       5
    */
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    root->left->left=new node(4);
    root->right->right=new node(5);
    cout<<distBWNodes(root,4,5);
}
```

▼ Deletion from BST

```
/*
Case 1 : node is a leaf
        Delete it directly
Case 2 : node has one child
        Replace the node with a child and delete the node
Case 3 : when the node has 2 childs
        Find the inorder successor of the node
        Replace the node with inorder successor
```

```
        Than simply delete
*/
#include"bits/stdc++.h"
using namespace std;
class node{
    public:
    int data;
    node* right;
    node* left;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// For returning the successor node
node* inorderSucc(node* root){
        node* curr=root;
        while(curr && curr->left){
                curr=curr->left;
        }
        return curr;
}


node* deleteInBST(node* root,int key){
        if(root->data>key){
                root->left=deleteInBST(root->left,key);
        }
        else if(root->data<key){
                root->right=deleteInBST(root->right,key);
        }
        // Till here we have found the node and we have to delete it
        else{
                // Case 1
                if(root->left==NULL){
                        node* temp=root->right;
                        free(root);
                        return temp;
                }  // Case 2
                else if(root->right==NULL){
                        node* temp=root->left;
                        free(root);
                        return temp;
                } // Case 3
                node* temp= inorderSucc(root->right);
                // Swapping the values of successor and node to be deleted
                root->data=temp->data;
                root->right=deleteInBST(root->right,temp->data);
                // than finally deleting the swapped node
        }
        return root;
}

void inorderPrint(node* root){
        if(root==NULL){
                return;
        }
        inorderPrint(root->left);
        cout<<root->data<<" ";
        inorderPrint(root->right);
}

int main(){
        node* root=new node(4);
        root->left=new node(2);
        root->left->left=new node(1);
        root->left->right=new node(3);
        root->right=new node(5);
        root->right->right=new node(6);
        /*
                        4
                       / \
                      2   5
                     / \   \
                    1   3   6
        */
       inorderPrint(root);cout<<endl;
       root=deleteInBST(root,5);
```

```
       inorderPrint(root);
}
```

▼ BST from preorder

```
// Setting a minimum and maximum value of each element of the array
/* here the numbers in the brackets are of the form
               (min,max)
               N --> NULL
                    |
                    |
           This is preorder sequence
array ----->  7  ,  5  ,  4  ,  6  ,  8  ,  9
          (N,N) (N,7) (N,5) (5,7) (7,N) (8,N)


                     (7,N)
             (N,7) 7 <-| 8 added here  (8,N)
                  /                      \
            (N,5) 5 (5,7) <--- 6 added here   9
                 /
         (N,4) 4 (4,5)
*/

#include<iostream>
#include<climits>
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

node* constructBST(int preorder[],int* preorderIdx,int key,int min,int max,int n){
    if(*preorderIdx>=n){
        return NULL;
    }
    node* root=NULL;
    // If this condition does not satisfy than we return NULL
    if(key>min && key<max){
        root=new node(key);
        *preorderIdx=*preorderIdx+1;
        if(*preorderIdx<n){
            // This will built entire left subtree
            root->left=constructBST(preorder,preorderIdx,preorder[*preorderIdx],min,key,n);
        }
        if(*preorderIdx<n){
            // This will built entire right subtree
            root->right=constructBST(preorder,preorderIdx,preorder[*preorderIdx],key,max,n);
        }
    }

    return root;
}

void inorderPrint(node* root){
    if(root==NULL){
        return;
    }
    inorderPrint(root->left);
    cout<<root->data<<" ";
    inorderPrint(root->right);
}

int main(){
    /*
            10
           /  \
          2    13
         /    /
        1    11
```

```
    */
    int preorder[]={10,2,1,13,11};
    int n=5;
    int preorderIdx=0;
    node* root=constructBST(preorder,&preorderIdx,preorder[0],INT_MIN,INT_MAX,n);
    inorderPrint(root);
    cout<<endl;
}
```

▼ Valid BST Check

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

/*
-----> Approach - 1
1. find out max of leftSubtree (maxL)
2. find out min of rightSubtree (minR)
3. if(minR>maxL) than true

-----> Approach - 2
1. two variables min allowed and max allowed
2. min allowed < node && max allowed > node

               node(min,max)
               /           \
        left(min,node)   right(node,right)
*/

bool isBST(node* root,node* min=NULL,node* max=NULL){
    if(root==NULL){
        return true;
    }
    if(min && root->data<=min->data){
        return false;
    }
    if(max && root->data>=max->data){
        return false;
    }
    bool leftValid=isBST(root->left,min,root);
    bool rightValid=isBST(root->right,root,max);

    return (leftValid && rightValid);
}

int main(){
    node* root=new node(1);
    root->left=new node(2);
    root->right=new node(3);
    /*
            1
           / \
          2   3
    */
    cout<<isBST(root)<<endl;

    node* root2=new node(5);
    root2->left=new node(2);
    root2->right=new node(7);
    /*
            5
           / \
          2   7
    */
    cout<<isBST(root2)<<endl;
}
```

▼ Balanced BST from sorted array

```
/*
----> Algorithm
1. MAke the middle element the root
2. Recursively, do the same for subtrees
    a. start to mid-1 for left subtree
    b. mid+1 to end for right subtree
*/

#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

node* sortedArrayToBST(int arr[],int start,int end){
    if(start>end){
        return NULL;
    }
    int mid=(start+end)/2;
    node* root=new node(arr[mid]);
    root->left=sortedArrayToBST(arr,start,mid-1);
    root->right=sortedArrayToBST(arr,mid+1,end);
    return root;
}

void inorderPrint(node* root){
    if(root==NULL){
        return;
    }
    inorderPrint(root->left);
    cout<<root->data<<" ";
    inorderPrint(root->right);
}

int main(){
    int arr[]={10,20,30,40,50};
    node* root=sortedArrayToBST(arr,0,4);
    inorderPrint(root);
    cout<<endl;
}
```

▼ Catalan Numbers

```
/*

These numbers are a sequence of natural numbers that occur
in various counting problems, often involving recursively
defines objects
their closed form is in terms of binomial coefficients

        C = 2n(C)n / n+1   =  Sum(C(i)C(n-i)), i=0 to n-1

*/
/*

----> Application of catalan numbers
1. Possible Bst's
2. Parenthesis / Bracket combinations
3. Possible forests
4. Ways of triangulations
5. Possible paths in matrix
6. Divide a circle using N chords

*/
```

```
#include<iostream>
using namespace std;

int catalan(int n){
    if(n<=1){
        return 1;
    }
    int res=0;
    for(int i=0;i<=n-1;i++){
        res+=catalan(i)*catalan(n-i-1);
        // here we have used n-i-1 because n is not 0 based value
        // It is a 1 based value so it adds 1 if we dont subtract
    }

    return res;
}

int main(){
    for(int i=0;i<11;i++){
        cout<<catalan(i)<<" ";
    }
}
```

▼ Print All BST For N

```
#include<bits/stdc++.h>
using namespace std;

class node{
    public:
    int data;
    node* right;
    node* left;
    node(int val){
        data=val;
        right=NULL;
        left=NULL;
    }
};

vector<node*> constructTrees(int start,int end){
    vector<node*> trees;
    if(start>end){
        trees.push_back(NULL);
        return trees;
    }
    for(int i=start;i<=end;i++){
        vector<node*> leftSubtree=constructTrees(start,i-1);
        vector<node*> rightSubtree=constructTrees(i+1,end);
        // Now combining them
        for(int j=0;j<leftSubtree.size();j++){
            node* leftN=leftSubtree[j];
            for(int k=0;k<rightSubtree.size();k++){
                node* rightN=rightSubtree[k];
                node* node1=new node(i);
                node1->left=leftN;
                node1->right=rightN;
                trees.push_back(node1);
            }
        }
    }
    return trees;
}

void preorder(node* root){
    if(root==NULL){
        return;
    }
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
}

int main(){
    vector<node*> totalTrees=constructTrees(1,3);
    for(int i=0;i<totalTrees.size();i++){
        cout<<(i+1)<<" : ";
        preorder(totalTrees[i]);
```

```
        cout<<endl;
    }
}
```

▼ Zigzag Traversal in BST

```
// If we print left to right on first level.
// Than we have to print right to left on second level and so on
/*
----->
1. Use 2 stacks - currentLevel & nextLevel
2. variable (bool type) -leftToRight(true initially)
3. if leftToRight, push left child then right
    else, push right child then left
*/

#include<bits/stdc++.h>
using namespace std;

class node{
    public:
    int data;
    node* right;
    node* left;
    node(int val){
        data=val;
        right=NULL;
        left=NULL;
    }
};

void zigzagTreversal(node* root){
    if(root==NULL){
        return;
    }

    stack<node*> currLevel;
    stack<node*> nextLevel;
    bool leftToRight=true;
    currLevel.push(root);
    while(!currLevel.empty()){
        node* temp=currLevel.top();
        currLevel.pop();

        // If temp is not NULL than only we would we able to check left and right
        if(temp){
            cout<<temp->data<<" ";

            // When we went from leftToRight in previous level
            if(leftToRight){
                if(temp->left){
                    // First we are pushing left because we want
                    // right to pop first
                    nextLevel.push(temp->left);
                }
                if(temp->right){
                    nextLevel.push(temp->right);
                }
            }
            // When we went rightToLeft in previous level
            else{
                if(temp->right){
                    nextLevel.push(temp->right);
                }
                if(temp->left){
                    nextLevel.push(temp->left);
                }
            }
        }

        if(currLevel.empty()){
            leftToRight=!leftToRight;
            swap(currLevel,nextLevel);
        }
    }
}

int main(){
```

```
    /*
            12
           /  \
          9    15
         / \
        5   10
    */

    node* root=new node(12);
    root->left=new node(9);
    root->left->left=new node(5);
    root->left->right=new node(10);
    root->right=new node(15);
    zigzagTreversal(root);
    cout<<endl;
}
```

▼ Identical BST Check

```
/*
--------> Algorithm
Given is the root of each tree
1. If both are empty are the same time, return true
2. If both are non-empty
    a. Check that the data at nodes is equal
    b. Check if left subtrees are same
    c. Check if right subtrees are same
3. If(a,b,c) are ture, return true
    else, return false
*/

#include"bits/stdc++.h"
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};

// bool isIdentical(node* root1,node* root2){
//     if(root1==NULL && root2==NULL){
//         return true;
//     }
//     if(root1==NULL && root2!=NULL){
//         return false;
//     }
//     if(root2==NULL && root1!=NULL){
//         return false;
//     }
//     bool a=false;
//     if(root1->data==root2->data){
//         a=true;
//     }
//     if(isIdentical(root1->left,root2->left) && isIdentical(root1->right,root2->right) && a){
//         return true;
//     }
//     return false;
// }

// Second way of writing
bool isIdentical(node* root1,node* root2){
    if(root1==NULL && root2==NULL){
        return true;
    }
    else if(root1==NULL || root2==NULL){
        return false;
    }
    else{
        bool cond1=root1->data==root2->data;
        bool cond2=isIdentical(root1->left,root2->left);
        bool cond3=isIdentical(root1->right,root2->right);
```

```
        if(cond1 && cond2 && cond3){
            return true;
        }
        return false;
    }
}

int main(){
    /*  Case 1
            1                1
           / \              / \
          2   3    and     2   3
         / \              / \
        4   5            4   5

        case 2
            1                1
           / \              / \
          2   3    and     2   3
         / \              / \
        4   5            4   7
    */
    node* root1=new node(1);
    root1->left=new node(2);
    root1->left->left=new node(4);
    root1->left->right=new node(5);
    root1->right=new node(3);

    node* root2=new node(1);
    root2->left=new node(2);
    root2->left->left=new node(4);
    root2->left->right=new node(5);
    root2->right=new node(3);

    node* root3=new node(1);
    root3->left=new node(2);
    root3->left->left=new node(4);
    root3->left->right=new node(7);
    root3->right=new node(3);

    node* root4=new node(1);
    root4->left=new node(2);
    root4->left->left=new node(4);
    root4->left->right=new node(5);

    // 1 and 2 are identical
    // 1 and 3 has one value different
    // 1 and 4 has structural difference
    cout<<isIdentical(root1,root2)<<endl;
    cout<<isIdentical(root1,root3)<<endl;
    cout<<isIdentical(root1,root4)<<endl;
}
```

▼ largest BST in Binary tree

```

/*
----> For each node store the following information
1. min in subtree
2. max in subtree
3. subtree size
4. size of largest BST
5. isBST

Than recursively traverse in a bottom-up manner
and find out the size of largest BST
*/

#include<iostream>
#include<climits>
using namespace std;

class node{
    public:
    int data;
    node* right;
    node* left;
    node(int val){
        data=val;
```

```
        right=NULL;
        left=NULL;
    }
};

class info{
    public:
    int size;
    int max;
    int min;
    int ans;
    bool isBST;
};

info largestBSTinBT(node* root){
    if(root==NULL){
        return {0,INT_MIN,INT_MAX,0,true};
    }

    // Checking if there is a single node
    if(root->left==NULL && root->right==NULL){
        return {1,root->data,root->data,1,true};
    }
    info leftInfo=largestBSTinBT(root->left);
    info rightInfo=largestBSTinBT(root->right);

    info curr;
    curr.size=(1+leftInfo.size+rightInfo.size);
    if(leftInfo.isBST && rightInfo.isBST && leftInfo.max<root->data && rightInfo.min>root->data){
        curr.min=min(leftInfo.min,min(rightInfo.min,root->data));
        curr.max=max(leftInfo.min,max(rightInfo.max,root->data));
        curr.ans=curr.size;
        curr.isBST=true;
        return curr;
    }
    // If at current node BST is not formed
    curr.ans=max(leftInfo.ans,rightInfo.ans);
    curr.isBST=false;

    return curr;
}

int main(){
    /*
            15
           /  \
          20   30
         /
        5
     */
    node* root=new node(15);
    root->left=new node(20);
    root->left->left=new node(5);
    root->right=new node(30);

    cout<<"Largest BST in BT : "<<largestBSTinBT(root).ans<<endl;
}
```

▼ Restore Distorted BST

```
/*
Inorder of a BST is sorted
2 Elements in a sorted array are swapped

Case 1:
    Swapped elements are not adjacent to each other
    original = {1,2,3,4,5,6,7,8}
    after swapping = {1,8,3,4,5,6,7,2}
Case 2:
    Swapped elements are adjacent to each other
    original = {1,2,3,4,5,6,7,8}
    after swapping = {1,2,4,3,5,6,7,8}

Maintain 3 pointers - first, last and mid
Case 1:
    first : previous node where 1st number < previous [8]
    Mid : number where 1st number < previous [3]
    Last : 2nd node where number < previous[2]
```

```
    swap first with last
Case 2:
    first : previous node where 1st number < previous [4]
    Mid : number where 1st number < previous [3]
    Last : NULL ( Since we know both are adjacent)
    swap first and mid
*/

#include<iostream>
using namespace std;

class node{
    public:
    int data;
    node* left;
    node* right;
    node(int val){
        data=val;
        right=NULL;
        left=NULL;
    }
};

void swapFunction(int &n1,int &n2){
    int temp=n1;
    n1=n2;
    n2=temp;
}

// Traversing the tree in inorder fashion and calculating the pointers
void calcPointers(node* &root,node* &first,node* &mid,node* &last,node* &prev){
    if(root==NULL){
        return;
    }
    // Like in inorder we first traverse for root->left
    calcPointers(root->left,first,mid,last,prev);
    // Checking if the property is voilated or not
    // If violation is happening than check is it first time or second
    if(prev && root->data<prev->data){
        // if the property would has violated for the first time time than
        // first pointer will be NULL
        if(!first){
            first=prev;
            mid=root;
        }
        // After that root will point to last node
        else{
            last=root;
        }
    }
    prev=root;
    calcPointers(root->right, first,mid,last,prev);
}

void restoreBST(node* &root){
    node* first=NULL;
    node* mid=NULL;
    node* last=NULL;
    node* prev=NULL;
    calcPointers(root,first,mid,last,prev);
    if(first && last){
        swapFunction(first->data,last->data);
    }
    else if(first && mid){
        swapFunction(first->data,mid->data);
    }
}

void inorder(node* root){
    if(root==NULL){
        return;
    }
    inorder(root->left);
    cout<<root->data<<" ";
    inorder(root->right);
}

int main(){
    /*
            6
          /  \
```

```
         9   3
       / \   \
      1  4   13
    */
    node* root=new node(6);
    root->left=new node(9);
    root->left->left=new node(1);
    root->left->right=new node(4);
    root->right=new node(3);
    root->right->right=new node(13);
    inorder(root);cout<<endl;
    restoreBST(root);
    inorder(root);cout<<endl;
    return 0;
}
```

▼ Binary Search challenges

▼ Find K Elements such that the minimum distance between any 2 elements is the maximum possible

```
/*
    Given is an array with n elements that represents n
    positions along a straignt line. Find K elements such
    that the minimum distance between any 2 elements is
    the maximum possible

    -----> Approach 1
    Find all possible subsets of size K & compare the
    minimum diatance between all of them

    -----> Approach 2 (Binary Search)
    1. Sort the array for bnary search technique
    2. Pick middle element as result and check for its feasibility
    3. If feasible, search the right half of the array with larger
       minimum distance
    4. Else search the left half of the array

    Example :
    1 2 7 5 1 12    K=3
    1 2 5 7 11 12    Sorted array
        Initialising the range left = 1, right = 12
        now calculating the mid of 12 ie 6
        now we checked the feasiblity of mid =6 and it is not
        so we update right as 6 now calculating mid = 3 (1, 5, 11)
        now mid = 3 comes as feasible solution so we update our left as 3
        left = 3, right = 6 mid = 4  ----> feasible (1, 5, 11)
        left = 4, right= 6 mid = 5 ---> (1, 7, 12)
        now if we update left as 5 than we have alredy calculated ans for mid=5
        and mid=6 is not feasible
*/

#include<iostream>
#include<algorithm>
using namespace std;

bool isFeasible(int mid,int arr[],int n,int k){
    int pos=arr[0];
    int elements=1;
    for(int i=0;i<n;i++){
        if(arr[i]-pos>=mid){
            pos=arr[i];
            elements++;
            if(elements==k){
                return true;
            }
        }
    }
    return false;
}

int largestMinimumdist(int arr[],int n,int k){
    sort(arr,arr+n);
    int result=-1;
    int left=1;
    int right=arr[n-1];
    while(left<right){
        int mid=(left+right)/2;
        if(isFeasible(mid,arr,n,k)){
```

```
            result=max(result,mid);
            left=mid+1;
        }
        else{
            right=mid;
        }
    }
    return result;
}

int main(){
    int arr[]={1 ,2 ,7 ,5 ,1 ,12};
    int n=6;
    int k=3;
    cout<<largestMinimumdist(arr,n,k);
    return 0;
}
```

▼ Maximum Pages give to each student is minimum

```
/*
    n - number of in n different books [ascending ordrer]
    m - number of students
    All the books have to be didided among m students, consequetively.
    Allocate the pages in such a way that maximum pages allocates to a
    student is minimum

    Example = [ 12, 34, 67, 90]
    Students (m) = 2
    Possible allocations   (max allocated)
    [12][34, 67, 90]   ---->    191
    [12, 34] [67 , 90] ---->    157
    [12, 34, 67] [90]  ---->    113  <--- Minimum allocated


    -----> Strategy
    Apply binary search for minimum and maximum values of max pages
    where initially min would be 0
    and max would be sum of all pages
    Now check the feasiblity of this
    Assign pages to each student in a sequential manner, while the current
    number of allocate pages don't exceed the value se by binary search

    If during allocation the number of students dont exceed the limit of 'm'
    then the solution is feasible. Else, it isn't
*/

#include<iostream>
#include<climits>
using namespace std;

bool isFeasible(int arr[],int n,int m,int min){
    int studentsRequired=1;
    // Sum represent the pages which we are giving to current student
    int sum=0;
    for(int i=0;i<n;i++){
        // Check if the page we want to distribute
        // is grater than minimum possible maximum pages
        if(arr[i]>min){ // Allocation not possible
            return false;
        }
        if(sum+arr[i]>min){
            studentsRequired++;
            sum=arr[i];
            if(studentsRequired>m){
                return false;
            }
        }
        else{
            sum+=arr[i];
            // here we are still giving pages to current student
        }
    }
    return true;
}

int allocateMinimum(int arr[],int n,int m){
    int sum=0;
```

```
    if(n<m){
        return -1;
    }
    for(int i=0;i<n;i++){
        sum+=arr[i];
    }
    int start=0;
    int end=sum;
    int ans=INT_MAX;
    while(start<end){
        int mid=(start+end)/2;
        if(isFeasible(arr,n,m,mid)){
            ans=min(ans,mid);
            // Since her we want the minimum so we decrease the mid
            end=mid-1;
        }
        else{
            start=mid+1;
        }
    }
    return ans;
}

int main(){
    int arr[]={12, 34, 67, 90};
    int n=4;
    int m=2;
    cout<<allocateMinimum(arr,n,m);
}
```

▼ Painters Partition problem

```
/*
    n = length of n different boards
    m = painters available

    A painter paints 1 unit of borad in 1 unit of time and each painter will paint consecutive boards.
Find the minimum time that will be required to paint all the boards

    example : borads = [10, 20, 30, 40]
            painters (m) = 2
            possible partitions
                [10][20,30,40]
                [10,20][30,40]
                [10,20,30][40]  <---- minimum (60)
    ---> Strategy :
        Apply binary search for minimum and maximum values of 'min' length of boards to be painted

        Check the feasiblity of this chosen value.
        Checking Feasibility : ASsign borads to each painter in a sequential manner, while the current
length of assigned boards don't exceed the valuew set by binary search

        if during allocation the number of painters don't exceed the limit of 'm' then the solution is
feasible. Else, it isn't
*/
#include<iostream>
using namespace std;

int findFeasible(int boards[],int n,int limit){
    int sum=0,painters=1;
    for(int i=0;i<n;i++){
        sum+=boards[i];
        if(sum>limit){
            sum=boards[i];
            painters++;
        }
    }
    return painters;
}

int paintersPartition(int boards[],int n,int m){
    int totalLength=0;
    int k=0; // Represent largest board size
    for(int i=0;i<n;i++){
        k=max(k,boards[i]);
        totalLength+=boards[i];
    }
    // k=40;
```

```
    // totalLength = 100;
    int low = k;
    int high=totalLength;
    while(low<high){
        int mid=(low+high)/2;
        int painter=findFeasible(boards,n,mid);
        if(painter<=m){
            high=mid;
        }
        else{
            low=mid-1;
            // cout<<low<<endl;
        }
    }
    return low;
}

int main(){
    int arr[]={10,20,30,40};
    int n=4;
    int m=2;
    cout<<paintersPartition(arr,n,m);
}
```

▼ Search element in rotated and sorted array

```
/*
    We are given ascending sorted array that has  been
    rotated from a pivot point (unknown to us) and an
    element X. Search for X in the array with complexity
    less than that of linear search

    original array = [10, 20, 30, 40, 50]
    rotated array = [30, 40, 50, 10, 20]
    Rotated about 50

    ---> Strategy :
    Find the pivot point
    Apply binary search in left half
    Apply binary search in right half

    ----> Finding pivot
    only element that will be grater than its next element
        arr[i] > arr[i+1] , i <= n-1
    After finding pivot point we search into left and right halves

*/

#include<iostream>
using namespace std;

// recursively checking, if our key is equal to mid or not
int searchInRotatedArray(int arr[],int key,int left,int right){
    if(left>right){
        return -1;
    }
    int mid=(left+right)/2;
    if(arr[mid]==key){
        return mid;
    }

    if(arr[left]<=arr[mid]){
        if(key>=arr[left] && key<=arr[mid]){
            return searchInRotatedArray(arr,key,left,mid-1);
        }
        return searchInRotatedArray(arr,key,mid+1,right);
    }
    else{
        if(key>=arr[mid] && key<=arr[right]){
            return searchInRotatedArray(arr,key,mid+1,right);
        }
        return searchInRotatedArray(arr,key,left,mid-1);
    }
}

int main(){
    int arr[]={6,7,8,9,10,1,2,5};
    int n=8;
```

```
    int key=8;
    cout<<searchInRotatedArray(arr,key,0,7);
}
```

▼ Finding the peek element

```
/*
    For a given array, find the peek element in the array.
    Peak element is one which is greater than both, lft and
    right neighbours of itself

    ---> Corner cases
    Ascending array : last element is peak
    Descending array : last element is peak
    For all same elements : each element is peak

    Example : arr[] = [10 ,20 ,15 ,2 ,23 ,90 ,67 ]
                          ^             ^
    Use binary search, analyze indices from 0 to n-1
    Compute mid and for each mid check if arr[mid] is peak or not

    Else if arr[mid-1] > arr[mid]
    check left for peak  ----> end = mid

    else if arr[mid] < arr[mid+1]
    check roght for peak
*/

#include<iostream>
using namespace std;

int findPeakElement(int arr[],int low,int high,int n){
    // low and high fedine the start and end of the array
    int mid = low+(high-low)/2;
    // Sometimes mid exceeds the integer due to high + low so we use
    // this as our mid
    if((mid==0 || arr[mid-1]<=arr[mid]) && (mid==n-1 || arr[mid+1]<=arr[mid])){
        // Here if we get mid as zero than we wont be able to get arr[mid-1]
        // So we check only mid == 0

        // Also if we get mid as n-1 than we we wont bw able to get arr[mid+1]
        // So we check only mid == n-1
        return mid;
    }
    else if(mid>0 && arr[mid-1]>arr[mid]){
        return findPeakElement(arr,low,mid-1,n);
    }
    else{
        return findPeakElement(arr,mid+1,high,n);
    }
}

int main(){
    int arr[]={10 ,20 ,15 ,2 ,23 ,90 ,67};
    cout<<findPeakElement(arr,0,6,7);
}
```

▼ Sliding Window Challenges

▼ Maximum subarray sum less than X

```
/*
    For a given array and integers K and X, find the maximum sum
    subarray of size K and having sum less than X.

    -----> Approach 1
        Compute sum of all possible subarrays of size K
        Than comparing and giving answer

        The time complexity of this approach will be O(n*k)
    -----> Approach 2
        1. Calculate sum of first K elements
        2. Initialize ans with this sum
        3. Iterate over the rest of the array
           Keep adding one element in sum and removing one from
```

```
           start, Compare new sum with ans in each iteration
        The time compexity of this approach is O(n), actually it is
        2*O(n) but since we ignore constants so it id O(n)
*/
#include<iostream>
using namespace std;
void maxSubarraySum(int arr[],int n,int k,int x){
    int sum=0,ans=0;
    for(int i=0;i<k;i++){
        sum+=arr[i];
    }
    if(sum<x){
        ans=sum;
    }
    for(int i=k;i<n;i++){
        sum-=arr[i-k];
        sum+=arr[i];
        if(sum<x){
            ans=max(ans,sum);
        }
    }
    cout<<ans<<" is the maximum subarray sum less than "<<x<<"\n";
}

int main(){
    int arr[]={7,5,4,6,8,9};
    int k=3;
    int x=20;
    int n=6;
    maxSubarraySum(arr,n,k,x);
}
```

▼ Minimum Subarray Size whose size is less than X

```
/*
    For a given array and an integer X, find the minimum subarray size for
    which sum > X

    --> Compute sum of all subarrays and check the condition
        Time Complexity = O(n^2)

    --> Sliding window approach
        Use variables ans, sum and start
        ans represent the minimum subarray size
        sum is the current sum
        start is the starting point of the subarray

        Iterate over array and start adding elements to sum
        if sum > X, remove elements from the start

        Time Complexity = O(n)
*/

#include<iostream>
using namespace std;

int smallestSubarrayWithSum(int arr[],int n,int x){
    int sum=0,minLength=n+1,start=0,end=0;
    while(end<n){
        while(sum<=x && end<n){
            sum+=arr[end++];
        }
        while(sum>x && start<n){
            if(end-start<minLength){
                minLength=end-start;
            }
            sum-=arr[start++];
        }
    }

    return minLength;
}

int main(){
    int arr[]={1,4,45,6,10,19};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x=51;
```

```
    cout<<smallestSubarrayWithSum(arr,n,x);
}
```

▼ Check is the sum of subarray is divisible by 3

```
/*
    For a given array and an integer K, check if any subarray such that
    elements in the subarray form a number divisible by 3

    ---> Approach 1
        Compute the numbers formed by all possible subarrays of size K.
        Check for their divisibility with 3

        Time Complexity = O(n^2)
    ---> Approach 2
        Sum of initial K elements from the array
        Use sliding window technique and one by one, start subtracting
        elements from the end and adding from the end
        At each step, check if the sum is divisible by 3 or not. If it is, print
        the elements

        Time Complexity = O(n)
*/

#include<iostream>
#include<vector>
using namespace std;

void computeNumber(vector<int> arr,int k){
    pair<int,int> ans;
    int sum=0;
    for(int i=0;i<k;i++){
        sum+=arr[i];
    }
    bool found=false;
    if(sum%3==0){
        ans=make_pair(0,k-1);
        found=true;
    }

    for(int i=k;i<arr.size();i++){
        if(found){
            break;
        }
        sum=sum+arr[i]-arr[i-k];
        if(sum%3==0){
            ans=make_pair(i-k+1,i);
            found=true;
        }
    }

    if(!found){
        ans=make_pair(-1,0);
    }
    if(ans.first==-1){
        cout<<"No Such Subarray Exists"<<endl;
    }
    else{
        for(int i=ans.first;i<=ans.second;i++){
            cout<<arr[i]<<" ";
        }
        cout<<"\n";
    }
}

int main(){
    vector<int> arr={84,23,45,12,56,82};
    int k=3;
    computeNumber(arr,k);
}
```

▼ Subarray With Palindrome Concatenation

```
/*
    For a given array and an integer K, check if any array of size K
```

```
    exists in the array such that concatenation of elements form a
    palindrome

    --> Approach 1
        Generate all subarrays of size K
        Time complexity - O(n^2)

        Check if their concatenation forms a palindrome
        Time Complexity - O(n)

        Total time complexity - O(n^3)
    --> Approach 2
        1. Store concatenation of initial K elements
        2. Iterate over the array and start deleting elements from
           the start and adding elements from the end
        3. At each step, check if concatenation is a palindrome

        Time Complexity - O(n^2)
*/
#include<iostream>
#include<cmath>
#include<vector>
using namespace std;

bool isPalindrome(int n){
    int temp=n,number=0;
    while(temp>0){
        number=number*10+temp%10;
        temp=temp/10;
    }
    if(number==n){
        return true;
    }
    else{
        return false;
    }
}

int findPalindromicSubarray(vector<int> arr,int k){
    int num=0;

    for(int i=0;i<k;i++){
        num=num*10+arr[i];
    }

    if(isPalindrome(num)){
        return 0;
        // Indicating the array is from 0 to k-1
    }

    for(int j=k;j<arr.size();j++){
        num=(num%(int)pow(10,k-1))*10 + arr[j];
        if(isPalindrome(num)){
            return j-k+1;
        }
    }

    return -1;
}

int main(){
    vector<int> arr={2,3,5,1,1,5};
    int k=4;
    cout<<findPalindromicSubarray(arr,k);
}
```

▼ Maximum number of perfect numbers in a subarray of size K in an array

```
/*
    For a given array and an integer K, find the maximum perfect
    numbers in a subarray of size K.

    Perfect number is a number if it is equal to the sum of its
    proper divisors(positive divisors) except for the number itself.

    Example : number = 6, proper  divisor = [1 , 2 , 3]

    --> Approach 1
```

```
        Generate all subarrays of size K and count number of perfect
        numbers
        Time Complexity - O(n*K)
    --> Trick
        Convert the given array into a binary array with values 0 and 1
        only
    --> Approach 2
        1. Traverse the array and if the number at index i is a perfect number
           then replace value at ith index with '1', else with a '0'
        2. Initialize a variable to store sum of divisors
        3. Traverse every number less than arr[i] and add it to sum if it is a
           divisor of arr[i]
        4. If the sum of all the divisors is equal to arr[i], then only the number
           is a perfect number
        time complexity = O(n*sqrt(n))
*/

#include<iostream>
#include<cmath>
using namespace std;

bool isNumberPerfect(int n){
    int sum=1;
    // This sum will be storing the sum of perfect divisors of n
    for(int i=2;i<sqrt(n);i++){
        if((n%i)==0){
            if(i==(n/i)){
                sum+=i;
            }
            else{
                sum+=(i+(n/i));
            }
        }
    }
    if(sum==n && n!=1){
        return true;
    }
    return false;
}

int maxSum(int arr[],int n,int k){
    if(n<k){
        cout<<"Inavlid values\n";
        return -1;
    }
    int res=0;
    for(int i=0;i<k;i++){
        res+=arr[i];
    }
    int sum=res;
    for(int i=k;i<n;i++){
        sum+=arr[i]-arr[i-k];
        res=max(res,sum);
    }

    return res;
}

int maxNumberOfPercts(int arr[],int n,int k){
    for(int i=0;i<n;i++){
        if(isNumberPerfect(arr[i])){
            arr[i]=1;
        }
        else{
            arr[i]=0;
        }
    }

    return maxSum(arr,n,k);
}

int main(){
    int arr[]={28,2,3,6,496,99,8128,24};
    int k=4;
    int n=8;
    cout<<maxNumberOfPercts(arr,n,k);
}
```

▼ Dynamic Programming

▼ 0_1Knapsack

```
/*
    Connected Problems
    1.  Subset Sum
    2.  Equal sum partition
    3.  Count of subset sum
    4.  Minimum subset sum diff
    5.  Target sum
    6.  # of subset and give diff


    Here we have choice which means either we can take it or not
    Also we have to find the maximum profit
    Both conditions satisfied so it is DP problem

    Choice Diagram
            Item 1
            /   \
           /     \
       w1<=w      w1>w
        / \        \
    inc  not inc   not inc
*/

/*
    return_type function(arguments){
        Base Condition

        Choice Diagram
    }
*/

/*
    Base Condition check : Think of the smallest vaild input
    Smallest possible thing here will be if the size of the arrays
    is zero and if the capacity is equal to zero

    if(n==0 || w==0){
        return 0;
    }
*/

#include<bits/stdc++.h>
using namespace std;

int knapsack(int wt[],int val[],int w,int n){
    if(n==0||w==0){
        return 0;
    }
    if(wt[n-1]<=w){
        // Finding the maximum out of if we selected the value and if we not
        return max(val[n-1]+knapsack(wt,val,w-wt[n-1],n-1),knapsack(wt,val,w,n-1));
    }
    else{
        return knapsack(wt,val,w,n-1);
    }
}

/*
    The DP matrix will be created for the changing values only
    in this case only W and n is changing

    Matrix of size t[n+1][w+1]
    initialize everything with -1
    memset(t,-1,size of(t))
    Now before going into recursion we will check into the matrix and take if
    value is not equal to -1 there

    if(t[n][w]!=-1)[
        return t[n][w]
    ]
*/

int t[102][1002];

int knapsackMemo(int wt[],int val[],int w,int n){
    if(n==0||w==0){
        return 0;
```

```
    }
    if(t[n][w]!=-1){
        return t[n][w];
    }
    if(wt[n-1]<=w){
        // Finding the maximum out of if we selected the value and if we not
        return t[n][w]=max(val[n-1]+knapsack(wt,val,w-wt[n-1],n-1),knapsack(wt,val,w,n-1));
    }
    else{
        return t[n][w]=knapsack(wt,val,w,n-1);
    }
}

// Sometimes the stack might get full with huge number of recursive calls
// In order to prevent this we use something called as top down method


/*
    We have to follow two steps
    Step 1: Initialize
    Step 2: Change recursive code to iterative

    The base condition of the recursive function is changed into
    the initialization of top down

    for(int i=0;i<n+1;i++){
        for(int j=0;j<w+1;j++){
            if(i==0||j==0){
                t[i][j]=0;
            }
        }
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<w+1;j++){
            if(wt[i-1]<=w){
                t[i][j]=max(val[i-1]+t[i-1][j-wt[i-1]],t[i-1][j]);
            }
            else{
                t[i][j]=t[i-1][j];
            }
        }
    }
    return t[n][w];
*/

int knapsack01DP(int wt[],int val[],int n,int w){
    for(int i=0;i<n+1;i++){
        for(int j=0;j<w+1;j++){
            if(i==0||j==0){
                t[i][j]=0;
            }
        }
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<w+1;j++){
            if(wt[i-1]<=w){
                t[i][j]=max(val[i-1]+t[i-1][j-wt[i-1]],t[i-1][j]);
            }
            else{
                t[i][j]=t[i-1][j];
            }
        }
    }
    return t[n][w];
}

int main(){
    memset(t,-1,sizeof(t));
}
```

▼ Introduction

```
/*
    Dynamic Programming : Enhanced recursion
    How to check if any problem will be solved by DP
    1.  We would be given choice if to include or not
```

```
    2.  Only one function call : no DP, When two calls then consider
    3.  Minimum, Maximum would be asked

    First write the recursive function than memoize that and than top down
*/
```

▼ Subset Sum Problem

```
/*
    We are given with a sum and an integer array
    We have to say if there is any subset present in the array whose sum
    is equal to the given sum or not
*/

#include<bits/stdc++.h>
using namespace std;

bool subsetSumPresent(vector<vector<bool>>arr,int numberArr[],int n,int s){
    for(int i=0;i<n+1;i++){
        for(int j=0;j<s+1;j++){
            if(i==0){
                arr[i][j]=false;
            }
            // j is zero when the sum required is 0
            if(j==0){
                arr[i][j]=true;
            }
        }
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<s+1;j++){
            if(numberArr[i-1]<=j){
                arr[i][j]=((arr[i-1][j-numberArr[i-1]])||(arr[i-1][j]));
            }
            else{
                arr[i][j]=arr[i-1][j];
            }
        }
    }
    return arr[n][s];
}
```

▼ Equal Sum Partition

```
/*
    We need to return true or false
    if there exists two sets with sum equal to a given sum

    Now the first thing here is if array sum is odd than we would
    simply return false

    Now find if the sum of one subset is equal to sum because the
    other will automatically be the sum
*/
#include<bits/stdc++.h>
using namespace std;

bool subsetSumPresent(vector<vector<bool>>arr,int numberArr[],int n,int s){
    for(int i=0;i<n+1;i++){
        for(int j=0;j<s+1;j++){
            if(i==0){
                arr[i][j]=false;
            }
            // j is zero when the sum required is 0
            if(j==0){
                arr[i][j]=true;
            }
        }
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<s+1;j++){
            if(numberArr[i-1]<=j){
                arr[i][j]=((arr[i-1][j-numberArr[i-1]])||(arr[i-1][j]));
```

```
            }
            else{
                arr[i][j]=arr[i-1][j];
            }
        }
    }
    return arr[n][s];
}

bool subsetSum(vector<vector<bool>>t,int arr[],int n){
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=arr[i];
    }
    if(sum%2!=0){
        return false;
    }
    else{
        return subsetSumPresent(t,arr,n,sum/2);
    }
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    return 0;
}
```

▼ Number of subsets with sum s

```
#include<bits/stdc++.h>
using namespace std;
int subsetSumWithGivenSum(int arr[],int n,int s){
    int tab[n + 1][s + 1];
    tab[0][0] = 1;

    for (int i = 1; i <= s; i++){
        tab[0][i] = 0;
    }
    for (int i = 1; i <= n; i++){
        tab[i][0] = 1;
    }

    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= s; j++){
          // if the value is greater than the sum
            if (arr[i - 1] > j){
                tab[i][j] = tab[i - 1][j];
            }
            else{
                tab[i][j] = tab[i - 1][j] + tab[i - 1][j - arr[i - 1]];
            }
        }
    }


    return tab[n][s];
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin>>n;
    int  s;
    cin>>s;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    int ans=subsetSumWithGivenSum(arr,n,s);
    cout<<ans;
    return 0;
}
```

▼ Minimum Difference Between Two Subsets

```
/*
    Absolute Difference of two sets must be the minimum
    https://www.youtube.com/watch?v=-GtpxG6l_Mc&list=PL_z_8CaSLPWekqhdCPmFohncHwz8TY2Go&index=12
*/


#include<bits/stdc++.h>
using namespace std;

int minimumDifference(int n,int arr[]){
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=arr[i];
    }
    bool t[n+1][sum+1];
    for(int i=0;i<sum+1;i++){
        t[0][i]=false;
    }
    for(int i=0;i<n+1;i++){
        t[i][0]=true;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<sum+1;j++){
            if(arr[i-1]<=j){
                t[i][j]=((t[i-1][j])||(t[i-1][j-arr[i-1]]));
            }
            else{
                t[i][j]=t[i-1][j];
            }
        }
    }

    vector<int>trueCan;
    for(int i=0;i<sum/2;i++){
        if(t[n][i]==true){
            trueCan.push_back(i);
        }
    }

    for(auto it:trueCan){
        cout<<it<<" ";
    }
    cout<<endl;

    int minim=INT_MAX;
    for(int i=0;i<trueCan.size();i++){
        minim=min(minim,sum-(2*trueCan[i]));
    }

    return minim;
}

int main(){
    int n=3;
    int arr[n]={1,2,7};
    cout<<minimumDifference(n,arr);
}
```

▼ Count The Number Of subset With A Given Difference

```
/*
    The Difference of sum of two subsets must be equal to given difference

    Sum(s1) - Sum(s2) = diff
    Sum(s1) + Sum(s2) = sum(arr)

    Adding Both ---->  sum(s1) = (diff + sum(arr))/2
    Now this problem is, count of subset sum
*/

#include<bits/stdc++.h>
using namespace std;
```

```
int countSubsetsWithGivenDiff(int arr[],int n,int diff){
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=arr[i];
    }
    int findSum=(diff+sum)/2;
    int t[n+1][findSum+1];
    for(int i=0;i<=findSum;i++){
        t[0][i]=0;
    }
    for(int i=0;i<=n;i++){
        t[i][0]=1;
    }
    for(int i=1;i<n+1;i++){
        for(int j=1;j<findSum+1;j++){
            if(arr[i-1]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=t[i-1][j]+t[i-1][j-arr[i-1]];
            }
        }
    }
    return t[n][findSum];
}

int main(){
    int arr[]={1,1,2,3};
    int n=4;
    int diff=1;
    cout<<countSubsetsWithGivenDiff(arr,n,diff)<<endl;
}
```

▼ Target Sum

```
/*
    we have to assign signs on each element of the array and find by how
    many ways we can make that sum equal to the given value

    Same as Count The Number Of Subset with A Given Difference
*/
#include<bits/stdc++.h>
using namespace std;

int targetSum(int arr[],int n,int req){
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=arr[i];
    }
    int findSum=(sum+req)/2;
    int t[n+1][findSum];
    for(int i=0;i<=sum;i++){
        t[0][i]=0;
    }
    for(int i=0;i<=n;i++){
        t[i][0]=1;
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=findSum;i++){
            if(arr[i-1]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=t[i-1][j]+t[i-1][j-arr[i-1]];
            }
        }
    }
    return t[n][findSum];
}

int main(){
    int n,req;
    cin>>n>>req;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
```

```
    cout<<targetSum(arr,n,req);
}
```

▼ Unbounded Knapsack

```
/*
    The difference is we can include one thing multiple times
    like we take one flour bag and 3-4 maggie packs from mall
    depending upon our desire
*/
#include<bits/stdc++.h>
using namespace std;

int unboundedKnapsack(int wt[],int val[],int n,int w){
    int t[n+1][w+1];
    for(int i=0;i<=n;i++){
        t[0][i]=0;
    }
    for(int i=0;i<=w;i++){
        t[i][0]=0;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<=w;j++){
            if(wt[i]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=max(t[i-1][j],val[i-1]+t[i][j-wt[i-1]]);
            }
        }
    }
    return t[n][w];
}
```

▼ Rod Cutting Problem

```
/*
    Now this problem is exactly similar to the previous problem ie.
    unbounded knapsack.
    Just the names of the variables have changed
*/
#include<bits/stdc++.h>
using namespace std;

int rodCuttingForMaxProf(int price[],int n,int length1[]){
    int len=n;
    int t[n+1][len+1];
    for(int i=0;i<n+1;i++){
        t[i][0]=0;
    }
    for(int i=0;i<len+1;i++){
        t[0][i]=0;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<len+1;i++){
            if(length1[i-1]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=max(price[i-1]+t[i][j-length1[i-1]],t[i-1][j]);
            }
        }
    }
    return t[n][len];
}
```

▼ Coin change problem: Maximum number of ways

```
#include<bits/stdc++.h>
using namespace std;
```

```
int coinChangeMax(int coin[],int n,int sum){
    int t[n+1][sum+1];
    for(int i=0;i<sum+1;i++){
        t[0][i]=0;
    }
    for(int i=0;i<n+1;i++){
        t[i][0]=1;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<sum+1;j++){
            if(coin[i-1]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=t[i-1][j]+t[i][j-(coin[i-1])];
            }
        }
    }
    return t[n][sum];
}

int main(){
    int arr[3]={1,2,3};
    int n=3;
    int sum=5;
    cout<<coinChangeMax(arr,n,sum);
}
```

▼ Coin Change problem : Minimum number of coins

```
/*
    We have stored INT_MAX-1 during initialiazation because
    when we add 1 it becomes INT_MAX and does not go out of bound
*/
#include<bits/stdc++.h>
using namespace std;

int coinMinimumCoins(int coins[],int n,int sum){
    int t[n+1][sum+1];
    for(int i=0;i<=n;i++){
        t[i][0]=0;
    }
    for(int i=0;i<=sum;i++){
        t[0][i]=INT_MAX-1;
    }
    // for(int i=1;i<=sum;i++){
    //     if(i%coins[0]==0){
    //         t[1][i]=i/coins[0];
    //     }
    //     else{
    //         t[1][i]=INT_MAX-1;
    //     }
    // }

    for(int i=2;i<=n;i++){
        for(int j=1;j<=sum;j++){
            if(coins[i-1]>j){
                t[i][j]=t[i-1][j];
            }
            else{
                t[i][j]=min(t[i-1][j],1+t[i][j-coins[i-1]]);
            }
        }
    }
    return t[n][sum];
}

int main(){
    int n,sum;
    cin>>n>>sum;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    cout<<coinMinimumCoins(arr,n,sum)<<endl;
```

```
    return 0;
}
```