Geeks for geeks solutions

ARRAY

Left:-
Array-8

1) Subarray with given sum:-
   Given an unsorted array A of size N that contains only non-negative integers, find a continuous sub-array which adds to a given number S.

   Soln:-
   We traverse from left to right and initialize two variables L and Sum to 0. L is left index pointer and sum is the current sum ie sum till present index. Now if at any moment, the sum becomes greater than target sum, we subtract the array starting from left pointer till it is less or equal and when sum is equal to target we update right pointer to “i” and break out.

```
vector<int> subarraySum(int arr[], int n, int s){
    int l=0;
    int r=0;
    vector<int>v;
    long long sum=0;
    int flag=0;
    for(int i=0;i<n;i++)
    {
        sum+=arr[i];
        while(sum>s && l<i)
        {
            sum-=arr[l];
            l++;
        }
        if(sum==s)
        {
            flag=1;
            r=i;
            break;
        }
    }
    if(!flag)
```

```
        {
            v.push_back(-1);
        }
        else
        {
            v.push_back(l+1);
            v.push_back(r+1);
        }
        return v;
    }
```

2) Count the triplets:-

Given an array of distinct integers. The task is to count all the triplets such that sum of two elements equals the third element.

Soln:- We sort it in decreasing order and then traverse left to right and in each step take the current element as target element and find pair of element which sum upto the target in O(N) but taking left and right pointer and checking the sum. If sum is equal we inc left and dec right , if sum is greater than target then we inc left else dec right for each current element.

```
int countTriplet(int arr[], int n)
    {
        int count=0;
        sort(arr,arr+n);
        int l=0;
        int r=0;
        for(int i=n-1;i>=0;i--)
        {
            int l=i-1;
            int r=0;
            while(l>r)
            {
                if(arr[l]+arr[r]==arr[i])
                {
                    count++;
                    l--;
                    r++;
```

```
                }
                else if(arr[l]+arr[r] > arr[i])
                {
                    l--;
                }
                else
                {
                    r++;
                }
            }
        }
        return count;
    }
```

3) Kadane’s Algorithm
Given an array arr of N integers. Find the contiguous sub-array with maximum sum.

Soln:- We take two variables max_till_now to INT_MIN and max_ending to 0. Then we traverse the array and add current element to max_ending. If max_ending is greater than max_till_now then we update max_till_now with max_ending and if max_ending is negative we make it 0.

```
int maxSubarraySum(int arr[], int n){
    int max_so_far = INT_MIN, max_ending_here = 0;

    for (int i = 0; i < n; i++)
    {
        max_ending_here = max_ending_here + arr[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}
```

4) Missing number in array
Given an array of size N-1 such that it can only contain distinct integers in the range of 1 to N. Find the missing element.

Soln:-
Sum all the array elements and find value of n*(n+1)/2. Their difference gives us the element.

```
int MissingNumber(vector<int>& array, int n) {
    long long sum=0;
    for(int i=0;i<n-1;i++)
    {
        sum+=array[i];
    }
    return (n*(n+1))/2 - sum;
}
```

5) Merge two sorted arrays
Given two sorted arrays arr1[] and arr2[] of sizes N and M in non-decreasing order. Merge them in sorted order without using any extra space. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.

Soln:-
Compare from starting element and push to vector depending upon which one is smaller and then increment the pointer till both arrays are now in the merged array.

```
void merge(int arr1[], int arr2[], int n, int m)
{
    vector<int> v;
    int i,j;
    for(i=0,j=0; i<n && j<m ;)
    {
        if(arr1[i]<arr2[j])
        {
            v.push_back(arr1[i]);
            i++;
        }
        else
```

```
        {
            v.push_back(arr2[j]);
            j++;
        }
    }
    if(i<n)
    {
        while(i<n)
        {
            v.push_back(arr1[i]);
            i++;
        }
    }
    else
    {
        while(j<m)
        {
            v.push_back(arr2[j]);
            j++;
        }
    }
    int k=0;
    for(int i=0;i<n;i++)
    {
        arr1[i]=v[k++];
    }
    for(int i=0;i<m;i++)
    {
        arr2[i]=v[k++];
    }
}
```

6) Rearrange array alternatively

Given a sorted array of positive integers. Your task is to rearrange the array elements alternatively i.e first element should be max value, second should be min value, third should be second max, fourth should be second min and so on.

Soln:-
Take min_index as 0 , max_index as n-1 and max_element as last element+1. And perform
if(i is even)
{
	a[i]+=(a[max_index]%max_element)*max_element;
	max_ind--;
}
else
{
	a[i]+=(a[min_index]%max_element)*max_element;
	min_ind++;
}

```
void rearrange(long long *arr, int n)
{
   int min_index=0;
   int max_index=n-1;
   int max_ele=arr[max_index]+1;
        for(int i=0;i<n;i++)
        {
           if(i%2==0)
           {
              arr[i]+=(arr[max_index]%max_ele)*max_ele;
              max_index--;
           }
           else
           {
              arr[i]+=(arr[min_index]%max_ele)*max_ele;
              min_index++;
           }
        }
        for(int i=0;i<n;i++)
        {
           arr[i]/=max_ele;
        }
}
```

7) Sort 0s,1s,2s

Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order.

Soln:-
Use counting sort

```
void sort012(int a[], int n)
{
    int count[3]={0};
    for(int i=0;i<n;i++)
    {
        count[a[i]]++;
    }
    int ind=0;
    for(int i=0;i<3;i++)
    {
        while(count[i]--)
        {
            a[ind++]=i;
        }
    }
}
```

8) Number of pairs
Given two arrays X and Y of positive integers, find the number of pairs such that xy > yx (raised to power of) where x is an element from X and y is an element from Y.

Soln:-
We sort array Y and maintain a count array which tells how many 1's 2's 3's 4's are there in Y kyuki in general agar y > x then x^y > y^x but kuch exceptions h elements ki to wo alag se handle kri h hamne check krte hue ki x ki value kya h. Now we found lowest ind in y such that y is greater than x and add the no of remaining elements to global count and also add and subtract exception cases.

```
long long countEle(int countY[],int x,int n,int Y[])
{
    if(x==1)
    {
        return 0;
```

```
    }
    int* idx = upper_bound(Y, Y + n, x);
    int ans=(Y+n)-idx;
    ans+=countY[1];
    if(x==2)
    {
        ans-=(countY[4]+countY[3]);
    }
    if(x==3)
    {
        ans+=countY[2];
    }
    return ans;
}
long long countPairs(int X[], int Y[], int m, int n)
{
   int countY[5]={0};
   sort(Y,Y+n);
   for(int i=0;i<n;i++)
   {
       if(Y[i]<5)
       {
           countY[Y[i]]++;
       }
   }
   long long count=0;
   for(int i=0;i<m;i++)
   {
       count+=countEle(countY,X[i],n,Y);
   }
   return count;
}
```

9) Rain water harvesting

Given an array arr[] of N non-negative integers representing the height of blocks. If width of each block is 1, compute how much water can be trapped between the blocks during the rainy season.

Soln:-

We take 2 arrays and store max height till now ie till the present index both from left and right to know the max height of the building in right of present building and their min - present stores the amount of water and we add up to get total water.

```
int trappingWater(int arr[], int n){
    long long sum=0;
    int left[n];
    int right[n];
    left[0]=arr[0];
    for(int i=1;i<n;i++)
    {
        left[i]=max(left[i-1],arr[i]);
    }
    right[n-1]=arr[n-1];
    for(int i=n-2;i>=0;i--)
    {
        right[i]=max(right[i+1],arr[i]);
    }
    for(int i=0;i<n;i++)
    {
        sum+=min(left[i],right[i])-arr[i];
    }
    return sum;
}
```

10) Equilibrium Point

Given an array A of N positive numbers. The task is to find the first Equilibium Point in the array.

Equilibrium Point in an array is a position such that the sum of elements before it is equal to the sum of elements after it.

Soln:- We maintain left and right sum to check at each element and when they are equal return the index of that element.

```
int equilibriumPoint(long long a[], int n) {
    long long totSum=0;
    long long leftSum=0;
    long long rightSum=0;
    for(int i=0;i<n;i++)
    {
        totSum+=a[i];
```

```
    }
    rightSum=totSum;
    int ind=-2;
    for(int i=1;i<n;i++)
    {
        leftSum+=a[i-1];
        rightSum-=a[i-1];
        if(leftSum==(rightSum-a[i]))
        {
            ind=i;
            break;
        }
    }
    if(n==1)
    {
        return 1;
    }
    return ind+1;
}
```

11) Leaders in an array

Given an array A of positive integers. Your task is to find the leaders in the array. An element of array is leader if it is greater than or equal to all the elements to its right side. The rightmost element is always a leader.

Soln:-

We traverse from right and check if current element is max till now and if yes then push it in vector.

```
vector<int> leaders(int a[], int n){
    vector<int> v;
    int max_ele=a[n-1];
    v.push_back(a[n-1]);
    for(int i=n-2;i>=0;i--)
    {
        if(a[i]>=max_ele)
        {
            v.push_back(a[i]);
            max_ele=a[i];
        }
    }
```

```
    reverse(v.begin(),v.end());
    return v;
}
```

12) Minimum Platforms

Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.

Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms.

Soln:- Sort both arrival and departure in ascending order and check if current arrival is greater than departure then we dont need platform else we need it if arrival is less than departure time so we increment the platform numbers and keep track of max no till now.

```
int findPlatform(int arr[], int dep[], int n)
{
        sort(arr,arr+n);
        sort(dep,dep+n);
        int result=1;
        int curr=1;
        int i=1;
        int j=0;
        while(i<n && j<n)
        {
            if(arr[i]<=dep[j])
            {
                curr++;
                i++;
            }
            else
            {
                curr--;
                j++;
            }
            result=max(result,curr);
        }
        return result;
```

```
}
```

13) Reverse array in groups
Given an array arr[] of positive integers of size N. Reverse every sub-array group of size K.
Soln:- We use sliding window type solution where we first reverse first k , then next k and so on.

```
void helper(vector<long long>& arr,int l,int r)
    {
        for(int i=l;i<=(r+l)/2;i++)
        {
            swap(arr[i],arr[r-i+l]);
        }
    }
    void reverseInGroups(vector<long long>& arr, int n, int k){
        int lowerind=0;
        int upperind=lowerind+k-1;
        if(upperind>=n)
        {
            helper(arr,lowerind,n-1);
        }
        else
        {
            helper(arr,lowerind,upperind);
        }
        while(upperind<n)
        {
            //cout<<upperind<<" "<<lowerind<<endl;
            lowerind=upperind+1;
            upperind=lowerind+k-1;
            //cout<<upperind<<" "<<lowerind<<endl;
            if(upperind>=n)
            {
                //cout<<"HI";
                helper(arr,lowerind,n-1);
            }
            else
            {
```

```
            helper(arr,lowerind,upperind);
        }
    }
}
```

14) Kth smallest element
Given an array arr[] and a number K where K is smaller than size of array, the task is to find the Kth smallest element in the given array. It is given that all array elements are distinct.
Soln:- We use quicksort approach i.e. we keep taking pivot element till we are at the k-1 position and its element is at its right place.
So basically, we want to bring the correct element at this place using quicksort approach.

**Other approach is to sort the array and find arr[k-1]

15) Pythagorean Triplet
Given an array arr of N integers, write a function that returns true if there is a triplet (a, b, c) that satisfies a2 + b2 = c2, otherwise false.
Soln:- Since range of elements is less, we use counting sort approach to store the occurrence of all elements till maxele here. Next we iterate in max*max time complexity for each combination of elements meanwhile checking their occurrence is not 0 . Now we find square root of (i*i + j*j) , store it in int and compare its square with original no, then check if the value of this square is their in counting array ie it is less than max element and its occurrence is more than 0 ie it exist so return true else return false.

```
bool checkTriplet(int arr[], int n) {
        int maxele=arr[0];
        for(int i=1;i<n;i++)
        {
            maxele=max(maxele,arr[i]);
        }
        int a[maxele+1]={0};
        for(int i=0;i<n;i++)
        {
            a[arr[i]]++;
        }
```

```
    for(int i=0;i<maxele+1;i++)
    {
        for(int j=0;j<maxele+1;j++)
        {
            if(i!=j && a[i]!=0 && a[j]!=0)
            {
                int val=sqrt(i*i + j*j);
                if((val*val)==(i*i + j*j) && val<=maxele && a[val]!=0)
                {
                    return true;
                }
            }
        }
    }
    return false;
}
```

16) Chocolate distribution
Given an array A of positive integers of size N, where each value represents number of chocolates in a packet. Each packet can have variable number of chocolates. There are M students, the task is to distribute chocolate packets such that :

1. Each student gets one packet.

2. The difference between the number of chocolates given to the students having packet with maximum chocolates and student having packet with minimum chocolates is minimum.

Input:

The first line of input contains an integer T, denoting the number of test cases. Then T test cases follow. Each test case consists of three lines. The first line of each test case contains an integer N denoting the number of packets. Then next line contains N space separated values of the array A denoting the values of each packet. The third line of each test case contains an integer m denoting the no of students.

Soln:- Sort the array and use sliding window method to find the mindiff across the array. Find initial difference and then move forward in the array to update it if lower difference found.

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

```
using namespace std;
int main()
 {
        int t;
        cin>>t;
        while(t--)
        {
            int n;
            cin>>n;
            int a[n];
            for(int i=0;i<n;i++)
            {
                cin>>a[i];
            }
            int m;
            cin>>m;
            sort(a,a+n);
            int mindiff=a[m-1]-a[0];
            for(int i=m;i<n;i++)
            {
                mindiff=min(mindiff,a[i]-a[i-m+1]);
            }
            cout<<mindiff<<endl;
        }
        return 0;
}
```

17) Stock buy and sell

The cost of stock on each day is given in an array A[] of size N. Find all the days on which you buy and sell the stock so that in between those days your profit is maximum.

Soln:- Traverse the array and maintain min and max index and when u find a greater element, just iterate otherwise push the min-index and max-index into the vector. Take case of the edge cases, where the elements are same by taking equality sign from previous element. Basically find the closest max min pairs such that difference between both is maximum so if we r finding bigger elements, just keep lowest as min and where the chain breaks as max and store them and move forward.

```
vector<vector<int> > stockBuySell(vector<int> A, int n){
```

```
vector< vector<int> > v;
int minInd=0;
int maxInd=0;
for(int i=1;i<n;i++)
{
    if(A[i]>A[maxInd])
    {
        maxInd=i;
    }
    if(A[i]==A[maxInd] && minInd==maxInd)
    {
        maxInd=i;
        minInd=i;
        continue;
    }
    if(A[i]==A[maxInd] && minInd!=maxInd)
    {
        maxInd=i;
        continue;
    }
    if(A[i]<A[maxInd])
    {
        if(minInd<maxInd)
        {
            vector<int> temp;
            temp.push_back(minInd);
            temp.push_back(maxInd);
            v.push_back(temp);
        }
        minInd=i;
        maxInd=i;
    }
    //cout<<minInd<<" "<<maxInd<<endl;
}
if(minInd!=maxInd)
{
    vector<int> temp;
    temp.push_back(minInd);
    temp.push_back(maxInd);
    v.push_back(temp);
```

```
        }
        return v;
    }
```

18) Element with left side smaller and right side greater

Given an unsorted array of size N. Find the first element in array such that all of its left elements are smaller and all right elements to it are greater than it.

Soln:- Maintain two arrays leftMax and rightMin to find max elements till an index i from left to right in leftMax and min elements in rightMin till an index i+1 from backwards. Now traverse original array to check if an element is greater= leftMax and less= rightMin. Extreme elements should be skipped.

```
int findElement(int arr[], int n) {
    int leftMax[n];
    leftMax[0]=arr[0];
    int rightMin[n];
    rightMin[n-1]=arr[n-1];
    for(int i=1;i<n;i++)
    {
        leftMax[i]=max(arr[i],leftMax[i-1]);
    }
    for(int i=n-2;i>=0;i--)
    {
        rightMin[i]=min(arr[i],rightMin[i+1]);
    }
    int ele=-1;
    for(int i=1;i<n-1;i++)
    {
        if(arr[i]>=leftMax[i] && arr[i]<=rightMin[i])
        {
            ele=arr[i];
            break;
        }
    }
    return ele;
}
```

19) Convert array into zig-zag fashion

Given an array Arr (distinct elements) of size N. Rearrange the elements of array in zig-zag fashion. The converted array should be in form a < b > c < d > e < f. The relative order of elements is same in the output i.e you have to iterate on the original array only.

Soln:- We take a variable flag and sets it to true, which means we are expecting < and if false then we are expecting a >. So if consecutive array elements are not following the rule then we swap them.

```
void zigZag(int arr[], int n) {
        bool flag=true;
        for(int i=0;i<n-1;i++)
        {
            if(flag)
            {
                if(arr[i]>arr[i+1])
                {
                    swap(arr[i],arr[i+1]);
                }
            }
            else
            {
                if(arr[i]<arr[i+1])
                {
                    swap(arr[i],arr[i+1]);
                }
            }
            flag=!flag;
        }
    }
```

20) Last index of 1

Given a string S consisting only '0's and '1's, find the last index of the '1' present in it.

Soln:- We traverse the string and update the ind storing the occurrence of 1 each time we encounter 1

```
int lastIndex(string s)
   {
      int ind=-1;
      for(int i=0;i<s.size();i++)
      {
         if(s[i]=='1')
         {
            ind=i;
         }
      }
      return ind;
   }
```

21) Spirally traversing the array

Given a matrix of size R*C. Traverse the matrix in spiral form.

Soln:- We take variables and move to a position just less than last element of end of that row and keep shrinking the matrix.

```
vector<int> spirallyTraverse(vector<vector<int> > matrix, int r, int c)
   {
      vector<int> v;
      int m=matrix.size();
      int n=matrix[0].size();
      int counts=m*n;
      int p=0;
```

```
if(m==n)
{
    counts--;
}
while(counts>0)
{
    for(int j=p;j<n-p-1;j++)
    {
        if(counts>0)
        {
       v.push_back(matrix[p][j]);
       counts--;
        }
    }
    for(int j=p;j<m-p-1;j++)
    {
        if(counts>0)
        {
        v.push_back(matrix[j][n-p-1]);
        counts--;
        }

    }
    for(int j=0;j<n-2*p-1;j++)
    {
        if(counts>0)
        {
```

```
            v.push_back(matrix[m-p-1][n-p-1-j]);
            counts--;
            }
        }
         for(int j=0;j<m-2*p-1;j++)
        {
            if(counts>0)
            {
            v.push_back(matrix[m-p-1-j][p]);
            counts--;
            }
        }
        p++;
    }
    if(m==n && n%2==1)
    {
        v.push_back(matrix[m/2][n/2]);
    }
    else if(m==n && n%2==0)
    {
        v.push_back(matrix[m/2][n/2-1]);
    }
    return v;
}
```

22) Largest number formed from an array

Given a list of non negative integers, arrange them in such a manner that they form the largest number possible.The result is going to be very large, hence return the result in the form of a string.

Soln:- We repeat the nos till they reach a length 18 and then sort them to find the inc order and then add them to output string traversing from the back.

```
string helper(string str)
        {
            string output="";
            int num=18/str.size();
            for(int i=1;i<=num;i++)
            {
                output+=str;
            }
            int rem=18%str.size();
            for(int i=0;i<rem;i++)
            {
                output+=str[i];
            }
            return output;
        }
        string printLargest(vector<string> &arr) {
            vector< pair<string,int> > v;
            string output="";
            for(int i=0;i<arr.size();i++)
            {
                string x=helper(arr[i]);
                //cout<<x<<endl;
                v.push_back(make_pair(x,i));
```

```
    }
    sort(v.begin(),v.end());
    for(int i=arr.size()-1;i>=0;i--)
    {
        output+=arr[v[i].second];
    }
    return output;
}
```

STRINGS

1) Reverse words in a given string
   Given a String S, reverse the string without reversing its individual words. Words are separated by dots.
   Soln:- Traverse the string and keep adding characters to a temp string until a '.' is encountered. When we arrive at '.', add the temp till now to the vector of strings and reset out to empty. Now print the vector in reverse order.

```
string reverseWords(string S)
{
    vector<string> v;
    string out="";
    for(int i=0;i<S.size();i++)
    {
        if(S[i]=='.')
        {
            v.push_back(out);
            out="";
        }
        else
        {
            out+=S[i];
        }
    }
    v.push_back(out);
```

```
        out="";
        for(int i=0;i<v.size()-1;i++)
        {
            out+=v[v.size()-1-i];
            out+=".";
        }
        out+=v[0];
        return out;
    }
```

2) Permutations of given string

Given a string S. The task is to print all permutations of a given string.

Input:

The first line of input contains an integer T, denoting the number of test cases. Each test case contains a single string S in capital letter.

Output:

For each test case, print all permutations of a given string S with single space and all permutations should be in lexicographically increasing order.

Soln:- Use a helper function to swap consecutive elements and pass new string into helper function recursively. Store the strings in a vector and sort the vector.

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

using namespace std;

vector<string>v;

void helper(string s,int l,int r)

{

    if(l==r)

    {

        v.push_back(s);

        return;

    }

    for(int i=l;i<=r;i++)

    {
```

```
        swap(s[i],s[l]);
        helper(s,l+1,r);
        swap(s[i],s[l]);
    }
}
int main()
 {
          int t;
          cin>>t;
          while(t--)
          {
              string s;
              cin>>s;
              helper(s,0,s.size()-1);
              sort(v.begin(),v.end());
              for(int i=0;i<v.size();i++)
              {
                  cout<<v[i]<<" ";
              }
              v.clear();
              cout<<endl;
          }
          return 0;
}
```

3) Longest palindrome in a string

Given a string S, find the longest palindromic substring in S. Substring of string S: S[i . . . . j] where 0 ≤ i ≤ j < len(S). Palindrome string: A string which reads the same

backwards. More formally, S is palindrome if reverse(S) = S. Incase of conflict, return the substring which occurs first (with the least starting index).

NOTE: Required Time Complexity O(n2).

Input:

The first line of input consists number of the testcases. The following T lines consist of a string each.

Output:

In each separate line print the longest palindrome of the string given in the respective test case.

Soln:- We use 2d dp to store true or false corresponding to substring from index i to j if it is a palindrome. Initially the single elements are palindromes and we check pairs of two also and store their results in dp. Now we use bottom up approach and for each pair of i and j, we check if str[i]==str[j] and value at dp[i+1][j-1] is true then we also make it true else false because if i+1 to j-1 is a palindrome then we need to check if characters at i and j are same. If same it is also a palindrome. We keep the account of max(j-i+1) to store maxlength.

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

using namespace std;

bool dp[10001][10001]={};

string helper(string str)

{

    int arr[2];

    arr[0]=0;

    arr[1]=0;

    int maxlength=1;

    for(int i=str.size()-1;i>=1;i--)

    {

        if(str[i]==str[i-1])

        {

            dp[i-1][i]=true;
```

```
        maxlength=2;
        arr[0]=i-1;
        arr[1]=i;
    }
    else
    {
        dp[i-1][i]=false;
    }
}
for(int i=str.size()-3;i>=0;i--)
{
    for(int j=i+2;j<str.size();j++)
    {
        if(dp[i+1][j-1] && (str[i]==str[j]))
        {
            if((j-i+1)>=maxlength)
            {
                arr[0]=i;
                arr[1]=j;
                maxlength=max(maxlength,j-i+1);
                dp[i][j]=true;
            }
        }
        else
        {
            dp[i][j]=false;
        }
```

```
        }
    }
    string out="";
    for(int i=arr[0];i<=arr[1];i++)
    {
        out+=str[i];
    }
    return out;
}
int main()
 {
          int t;
          cin>>t;
          while(t--)
          {
              string str;
              cin>>str;
              for(int i=0;i<str.size();i++)
              {
                  dp[i][i]=true;
              }
              cout<<helper(str)<<endl;
          }
          return 0;
}
```

4) Recursively remove all adjacent duplicates

Given a string s, recursively remove adjacent duplicate characters from the string s. The output string should not have any adjacent duplicates.

Soln:- We recursively remove first adjacent alphabets from s and check if the edited s doesn't have any successive same elements. In while loop we check if same elements are there and we remove them in recursive function until not same elements remains.

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

using namespace std;

string out="";

bool checker(string str)

{

    for(int i=1;i<str.size();i++)

    {

        if(str[i]==str[i-1])

        {

            return false;

        }

    }

    return true;

}

void helper(string str,int s,int e,bool flag)

{

    //cout<<s<<" "<<e<<" "<<" "<<flag<<out<<endl;

    if(e==str.size())

    {

        if(str[s-1]!=str[e-1])

        {

            out+=str[e-1];

        }
```

```
        return ;
    }
    if(str[s]==str[e])
    {
        helper(str,s+1,e+1,false);
    }
    else
    {
        if(!flag)
        {
            helper(str,s+1,e+1,true);
        }
        else
        {
            out+=str[s];
            helper(str,s+1,e+1,true);
        }
    }
}
int main()
 {
          int t;
          cin>>t;
          while(t--)
          {
              string str;
              cin>>str;
```

```
            out="";
            helper(str,0,1,true);
            while(!checker(out))
            {
                string temp=out;
                out="";
                helper(temp,0,1,true);
            }
            cout<<out<<endl;
        }
        return 0;
}
```

5) Check if string is rotated by two places

Given two strings a and b. The task is to find if the string 'b' can be obtained by rotating another string 'a' by exactly 2 places.

Soln:- Take a temp string and store rotated string str1 in it by 2 places in both directions and compare it with str2. If equal return true else false.

```
bool isRotated(string str1, string str2)
{
    if(str1.size()==1 || str2.size()==1)
    {
        return str1==str2;
    }
    string temp=str1;
    for(int i=0;i<str1.size()-2;i++)
    {
        temp[i+2]=str1[i];
```

```
    }
    temp[0]=str1[str1.size()-2];
    temp[1]=str1[str1.size()-1];
    //cout<<temp<<endl;
    if(temp==str2)
    {
        return true;
    }
    temp=str1;
    for(int i=2;i<str1.size();i++)
    {
        temp[i-2]=str1[i];
    }
    temp[str1.size()-2]=str1[0];
    temp[str1.size()-1]=str1[1];
    //cout<<temp<<endl;
    if(temp==str2)
    {
        return true;
    }
    return false;
}
```

6) Roman Number to Integer

Given a string in roman no format (s) your task is to convert it to an integer . Various symbols and their values are given below.

Soln:- If current alphabet's equivalent integer is less than next then subtract from total sum else add to total sum.

```
int helper(char ch)
```

```
{
    if(ch=='I')
    {
        return 1;
    }
    else if(ch=='V')
    {
        return 5;
    }
    else if(ch=='X')
    {
        return 10;
    }
    else if(ch=='L')
    {
        return 50;
    }
    else if(ch=='C')
    {
        return 100;
    }
    else if(ch=='D')
    {
        return 500;
    }
    else
    {
        return 1000;
    }
}
int romanToDecimal(string &str) {
    long long sum=0;

    for(int i=1;i<str.size();i++)
    {
        if(helper(str[i])>helper(str[i-1]))
        {
            sum-=helper(str[i-1]);
        }
        else
```

```
        {
            sum+=helper(str[i-1]);
        }
    }
    sum+=helper(str[str.size()-1]);
    return sum;
}
```

7) Anagram

Given two strings a and b consisting of lowercase characters. The task is to check whether two given strings are an anagram of each other or not. An anagram of a string is another string that contains the same characters, only the order of characters can be different. For example, “act” and “tac” are an anagram of each other.

Soln:- Store the frequency of each character in a and b in separate arrays and compare for each index of frequency arrays if they are different. If they are same return true else false.

```
bool isAnagram(string a, string b){
    int freq1[26]={0};
    int freq2[26]={0};
    for(int i=0;i<a.size();i++)
    {
        freq1[int(a[i])-97]++;
    }
    for(int i=0;i<b.size();i++)
    {
        freq2[int(b[i])-97]++;
    }
    for(int i=0;i<26;i++)
    {
        if(freq1[i]!=freq2[i])
        {
            return false;
        }
    }
    return true;
}
```

8) Remove Duplicates

Given a string without spaces, the task is to remove duplicates from it.
Soln:- We keep a hashmap to record if the current character has occurred before. If true we skip it else we add it and update its value in map to true.

```
string removeDups(string s)
        {
            string out="";
            unordered_map<int,bool> m;
            for(int i=0;i<26;i++)
            {
                m[i+97]=false;
                m[i+65]=false;
            }
            for(int i=0;i<s.size();i++)
            {
                if(!m[int(s[i])])
                {
                    out+=s[i];
                    m[int(s[i])]=true;
                    if(int(s[i])<92)
                    {
                        m[int(s[i])+32]=true;
                    }
                    else
                    {
                        m[int(s[i])-32]=true;
                    }
                }
            }
            return out;
        }
```

9) Form a palindrome
Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
Soln:- We create a 2D-DP table to store the no of additions required for each substring and fill the table in diagonal approach. We check if present s and e indexed characters are equal. If they are then no of insertions are equal to one (s+1) to (e-1) else we check the minimum of no of insertions in (s+1)(e) and (s)(e-1) and add 1 to it and store it in the present dp table. Finally we return the dp[0][str.size()-1].

```
#include<bits/stdc++.h>
using namespace std;
int dp[51][51];
int minAdded(string str)
{
    int s,e;
    for(int i=1;i<str.size();i++)
    {
        for(s=0,e=i;e<str.size();s++,e++)
        {
            if(str[s]==str[e])
            {
                dp[s][e]=dp[s+1][e-1];
            }
            else
            {
                dp[s][e]=1+min(dp[s+1][e],dp[s][e-1]);
            }
        }
    }
    return dp[0][str.size()-1];
}
int main()
 {
          int t;
          cin>>t;
          while(t--)
          {
              string str;
              cin>>str;
              dp[51][51]={0};
              cout<<minAdded(str)<<endl;
          }
          return 0;
}
```

10) Longest Distinct characters in string

Given a string S, find length of the longest substring with all distinct characters. For example, for input "abca", the output is 3 as "abc" is the longest substring with all distinct characters.

Soln:- We take start as -1 and initial maxLen as 0 . We use a hashmap to keep the latest occurrence of each character and if it is a new character then we don't update the start , we just add it to the map otherwise we update the occurrence of the character as well as start which will be at the old position of this character as this character already exists . Each time we just update the maxLen as max of current and i-start.

```
#include<bits/stdc++.h>
using namespace std;
int main()
 {
        int t;
        cin>>t;
        while(t--)
        {
            string str;
            cin>>str;
            int maxLen=0;
            int start=-1;
            unordered_map<char,int> m;
            for(int i=0;i<str.size();i++){
                if(m.count(str[i])>0)
                {
                    start=max(start,m[str[i]]);
                }
                m[str[i]]=i;
                maxLen=max(maxLen,i-start);
            }
            cout<<maxLen<<endl;
        }
        return 0;
}
```

11) Implement Atoi

Your task is to implement the function atoi. The function takes a string(str) as argument and converts it to an integer and returns it.

Soln:- We take a num as initialise it with 0, next we add 10 to the power strLen-i-1 x Number. If any character is other than 0 to 9 we return -1.

```
int atoi(string str)
{
    long long num=0;
    for(int i=1;i<str.size();i++)
```

```
    {
        if(str[i]>='0' && str[i]<='9')
        {
            num+=(pow(10,str.size()-i-1))*(int(str[i])-int('0'));
        }
        else
        {
            return -1;
        }
    }
    if(str[0]=='-')
    {
        num*=-1;
    }
    else
    {
        num+=(pow(10,str.size()-1))*(int(str[0])-int('0'));
    }
    return num;
}
```

12) Implement strstr

Your task is to implement the function strstr. The function takes two strings as arguments (s,x) and locates the occurrence of the string x in the string s. The function returns and integer denoting the first occurrence of the string x in s (0 based indexing).

Soln:- We make a out string of length x's from s and compare it with x. If equal we return 0 else we keep iterating by adding next and removing previous character from out and compare it with x. When found return starting index else -1.

```
int strstr(string s, string x)
{
     int len=x.size();
     string out="";
     for(int i=0;i<len;i++)
     {
         out+=s[i];
     }
     if(out==x)
     {
         return 0;
     }
```

```
    int start=x.size()-1;
    int f=0;
    while(start<s.size() && out!=x)
    {
        //cout<<out<<endl;
        out.erase(out.begin());
        start++;
        out+=s[start];
        if(out==x)
        {
            f=1;
            break;
        }
    }
    if(f)
    {
        return start-x.size()+1;
    }
    return -1;
}
```

13) Longest Common Prefix in an Array

Given a array of N strings, find the longest common prefix among all strings present in the array.

Soln:- We start from first index and check for each string if all characters are same. If they are, then we add the corresponding character to the string else breakout.

```
string longestCommonPrefix(string arr[], int N)
   {
       string out="";
       int maxlength=0;
       int ind=0;
       for(int i=0;i<N;i++)
       {
           if(arr[i].size()>maxlength)
           {
               maxlength=arr[i].size();
           }
           if(ind<maxlength && isSame(arr,ind,N))
           {
               out+=arr[i][ind];
```

```
        }
        else
        {
            break;
        }
        ind++;
        //cout<<arr[i].size()<<endl;
    }
    if(out.size()==0)
    {
        out+="-";
        out+="1";
    }
    return out;
}
```

LINKED LIST

Note:- Left 9

1) Finding middle element in a linked list

Given a singly linked list of N nodes. The task is to find the middle of the linked list. For example, if given linked list is 1->2->3->4->5 then the output should be 3.

If there are even nodes, then there would be two middle nodes, we need to print the second middle element. For example, if given linked list is 1->2->3->4->5->6 then the output should be 4.

Soln:- First we count the number of nodes then depending upon odd or even increment and divide it by 2. Next iterate the temp pointer by new count times.

```
int getMiddle(Node *head)
{
  Node *temp1=head;
  int count=0;
  while(temp1->next!=NULL && temp1!=NULL)
  {
     count++;
     temp1=temp1->next;
  }
  if(count%2==1)
  {
```

```
        count++;
    }
    count/=2;
    temp1=head;
    while(count--)
    {
        temp1=temp1->next;
    }
    return temp1->data;
}
```

2) Reverse a linked list
Given a linked list of N nodes. The task is to reverse this list.
Soln:- Take three pointers prev, curr and next. Point prev to null, curr to head and next to head's next. Now update all three pointers in each step in while loop with condition that next is not NULL. Put curr's next as prev, prev as curr, next node as curr and next's next as new next. Finally update curr's next as prev and put curr as head.

```
class Solution
{
    public:
    struct Node* reverseList(struct Node *head)
    {
        if(head==NULL || head->next==NULL)
        {
            return head;
        }
        Node *prev=NULL;
        Node *curr=head;
        Node *nextP=head->next;
        while(nextP!=NULL)
        {
            curr->next=prev;
            prev=curr;
            curr=nextP;
            nextP=curr->next;
        }
        curr->next=prev;
        head=curr;
    }
};
```

3) Rotate a Linked List

Given a singly linked list of size N. The task is to rotate the linked list counter-clockwise by k nodes, where k is a given positive integer smaller than or equal to length of the linked list.

Soln:- We take a curr pointer to head and move it ahead by k-1 to reach the point of reversal.If we already reached the end of list then return head. Next iterate the kth node pointer till the end of list. Point the kth pointer at head, head at curr->next and curr as NULL.

```
Node* rotate(Node* head, int k)
{
    Node *temp1=head;
    k--;
    while(k-- && temp1!=NULL)
    {
        temp1=temp1->next;
    }
    if(temp1==NULL)
    {
        return head;
    }
    Node* kth=temp1;
    while(kth->next!=NULL)
    {
        kth=kth->next;
    }
    kth->next=head;
    head=temp1->next;
    temp1->next=NULL;
    return head;
}
```

4) Reverse a Linked List in groups of given size.

Given a linked list of size N. The task is to reverse every k nodes (where k is an input to the function) in the linked list.

Soln:- First reverse first k nodes of the linked list and then call the function recursively for next nodes by passing head->next = remaining one. Reverse function is the same as general reverse function of linked list. In the while loop, we keep a check that curr->next!=NULL as else we need to break out and use the nodes available till now. So

before passing in recursion, we check if we have reached the end of the list. If we have we just return the pointer to last node else recursively bring the next reversed nodes.

```
struct node *reverse (struct node *head, int k)
{
   node *prev=NULL;
   node *curr=head;
   node *nextP=head->next;
   int count=k;
   count--;
   while(count-- && curr->next!=NULL)
   {
      curr->next=prev;
      prev=curr;
      curr=nextP;
      nextP=nextP->next;
   }
   curr->next=prev;
   if(nextP!=NULL)
   {
      head->next=reverse(nextP,k);
   }
   return curr;
}
```

5) Intersection Point in Y Shapped Linked Lists

Given two singly linked lists of size N and M, write a program to get the point where two linked lists intersect each other.

Soln:- We take a hashmap of node*, bool type to store what nodes we have visited. Initially all are non-visited so they are false by default and we traverse the first linked list and marks all its nodes as visited or true and while traversing second list we check if current node has been already visited. If it is then we return its value else we mark it true and move ahead. We return -1 at the end.

```
int intersectPoint(Node* head1, Node* head2)
{
   unordered_map<Node*,bool>m;
   Node* temp=head1;
   temp = head1;
   while(temp!=NULL)
   {
      m[temp]=true;
```

```
        temp=temp->next;
    }
    temp = head2;
    while(temp!=NULL)
    {
        if(m[temp])
        {
            return temp->data;
        }
        m[temp]=true;
        temp=temp->next;
    }
    return -1;
}
```

6) Detect Loop in linked list
Given a linked list of N nodes. The task is to check if the the linked list has a loop. Linked list can contain self loop.
Soln:- We take two pointers pointing the head and we increment them such that we make one jump by 1 and other by 2. If they meet they we return true else we return false.

```
bool detectLoop(Node* head)
    {
        Node* temp1=head;
        Node* temp2=head;
        while(temp1!=NULL && temp2!=NULL && temp2->next!=NULL)
        {
            temp1=temp1->next;
            temp2=temp2->next->next;
            if(temp1==temp2)
            {
                return true;
            }
        }
        return false;
    }
```

7) Remove loop in Linked List
You are given a linked list of N nodes. Remove the loop from the linked list, if present.

Note: X is the position of the node to which the last node is connected to. If it is 0, then there is no loop.
Soln:- We make an unordered_map to store the occurrence of a node in the linked list. If there is a loop then we reach at a node again and since its true we make it's previous node's next as null else we return without doing anything.

```
void removeLoop(Node* head)
{
    unordered_map<Node*,bool>m;
    Node* temp=head;
    m[temp]=true;
    temp=temp->next;
    Node* prev=head;
    while(temp!=NULL)
    {
        if(m[temp])
        {
            break;
        }
        m[temp]=true;
        temp=temp->next;
        prev=prev->next;
    }
    if(temp==NULL)
    {
        return;
    }
    prev->next=NULL;
}
```

8) Nth node from end of linked list
Given a linked list consisting of L nodes and given a number N. The task is to find the Nth node from the end of the linked list.
Soln:- Count no of nodes and subtract n from count and from starting, decrement count at each node and when it becomes 1 return its data.
If anywhere it becomes less than equal to 0 return -1.

```
int getNthFromLast(Node *head, int n)
{
     Node* temp=head;
     int count=1;
     while(temp!=NULL)
```

```
    {
        temp=temp->next;
        count++;
    }
    temp=head;
    count-=n;
    if(count<=0)
    {
        return -1;
    }
    while(1)
    {
        if(count==1)
        {
            return temp->data;
        }
        temp=temp->next;
        count--;
    }
}
```

17) Given a linked list of 0s, 1s and 2s, sort it.
Given a linked list of N nodes where nodes can contain values 0s, 1s, and 2s only. The task is to segregate 0s, 1s, and 2s linked list such that all zeros segregate to head side, 2s at the end of the linked list, and 1s in the mid of 0s and 2s.
Soln:- Count the number of 0 1 and 2 and attach the nodes using while loop until there individual count is positive.

```
Node* segregate(Node *head) {
        Node* temp=head;
        int count1=0;
        int count2=0;
        int count0=0;
        while(temp!=NULL)
        {
            if(temp->data==0)
            {
                count0++;
            }
            else if(temp->data==1)
            {
```

```
            count1++;
        }
        else
        {
            count2++;
        }
        temp=temp->next;
    }
    Node* main = new Node(-1);
    temp=main;
    while(count0--)
    {
        Node* temp1= new Node(0);
        temp->next = temp1;
        temp=temp->next;
    }
    while(count1--)
    {
        Node* temp1= new Node(1);
        temp->next = temp1;
        temp=temp->next;
    }
    while(count2--)
    {
        Node* temp1= new Node(2);
        temp->next = temp1;
        temp=temp->next;
    }
    return main->next;
}
```

18) Delete without head pointer

You are given a pointer/ reference to the node which is to be deleted from the linked list of N nodes. The task is to delete the node. Pointer/ reference to head node is not given.
Note: No head reference is given to you. It is guaranteed that the node to be deleted is not a tail node in the linked list.

Soln:- We take two pointers to head and increment one to next, then put next's data to previous and so on until next's next is NULL. At last we make previous's next as NULL so in this way last node is made NULL. We basically overwrite the values by putting next's to previous so that node to be deleted is replaced by some other value.

```
void deleteNode(Node *node)
{
   Node* temp=node;
   Node* temp2=node;
   temp=temp->next;
   temp2->data=temp->data;
   while(temp->next!=NULL)
   {
      temp=temp->next;
      temp2=temp2->next;
      temp2->data=temp->data;
   }
   temp2->next=NULL;
}
```

16) Implement Stack using Linked List

Let's give it a try! You have a linked list and you have to implement the functionalities push and pop of stack using this given linked list. Your task is to use the class as shown in the comments in the code editor and complete the functions push() and pop() to implement a stack.

Soln:- In the push function- If top is NULL(stack empty) we create a new node and make it top and return else we create a new node and point its next to top and make it top itself.

In the pop function- If the top is NULL(stack empty) return -1 else store the value at top and make top as top->next to lose the top most element.

```
void MyStack ::push(int x) {
    if(top==NULL)
    {
        StackNode* head = new StackNode(x);
        top=head;
        return;
    }
    StackNode* head = new StackNode(x);
    head->next=top;
    top=head;
    return;
}

/* The method pop which return the element
  poped out of the stack*/
```

```
int MyStack ::pop() {
    if(top==NULL)
    {
        return -1;
    }
    int val=top->data;
    top=top->next;
    return val;
}
```

15) Implement Queue using Linked List
Implement a Queue using Linked List.
A Query Q is of 2 Types
(i) 1 x (a query of this type means pushing 'x' into the queue)
(ii) 2 (a query of this type means to pop an element from the queue and print the poped element)
Soln:- In push function:- if the queue in empty, we add new node and rear and front both point on it else we add a new element , point it's next to rear and make it the new rear.
In pop function:- If the queue is empty(front is NULL) return -1, else if there is one element then we return front's data and make both front and rear as NULL else we traverse a temp variable till one less than front , return front's data and make temp as new front and temp's next as NULL.

```
void MyQueue:: push(int x)
{
    if(front==NULL)
    {
        QueueNode* head = new QueueNode(x);
        front = head;
        rear = head;
        return;
    }
    QueueNode* head = new QueueNode(x);
    head->next = rear;
    rear = head;
    return;
}

/*The method pop which return the element
  poped out of the queue*/
int MyQueue :: pop()
```

```
{
    if(front==NULL)
    {
        return -1;
    }
    else if(front==rear)
    {
        int val = front->data;
        front=NULL;
        rear=NULL;
        return val;
    }
    QueueNode* temp = rear;
    while(temp->next!=front)
    {
        temp=temp->next;
    }
    int val=temp->next->data;
    front = temp;
    temp->next=NULL;
    return val;
}
```

14) Check if Linked List is Palindrome

Given a singly linked list of size N of integers. The task is to check if the given linked list is palindrome or not.

Soln:- Make a stack and in first traversal add list's element to the stack. In next traversal of the list, check if current top and list element is same. If they are different, return false else in the end return true.

```
bool isPalindrome(Node *head)
{
    stack<int> s;
    Node* temp = head;
    while(temp!=NULL)
    {
        s.push(temp->data);
        temp=temp->next;
    }
    temp=head;
    while(temp!=NULL)
```

```
    {
        if(temp->data!=s.top())
        {
            return false;
        }
        temp=temp->next;
        s.pop();
    }
    return true;
}
```

13) Add two numbers represented by linked lists

Given two numbers represented by two linked lists of size N and M. The task is to return a sum list. The sum list is a linked list representation of the addition of two input numbers.

Soln:- First reverse both the linked list and add them while taking care of the carry and keep adding the node at the end of output linked list. At the end check if the value of carry is 1 , if it is 1 then append one more node with value 1 else nothing. Finally reverse the linked list and return it.

```
struct Node* reverseList(struct Node *head)
    {
      if(head==NULL || head->next==NULL)
      {
          return head;
      }
      Node *prev=NULL;
      Node *curr=head;
      Node *nextP=head->next;
      while(nextP!=NULL)
      {
          curr->next=prev;
          prev=curr;
          curr=nextP;
          nextP=curr->next;
      }
      curr->next=prev;
      head=curr;
      return head;
    }
    struct Node* addTwoLists(struct Node* first, struct Node* second)
```

```
{
    first = reverseList(first);
    second = reverseList(second);
    Node* temp1 = first;
    Node* temp2 = second;
    Node* head = new Node(0);
    Node* temp=head;
    int carry=0;
    while(temp1!=NULL || temp2!=NULL)
    {
        int sum=0;
        if(temp1!=NULL)
        {
            sum+=temp1->data;
            temp1 = temp1->next;
        }
        if(temp2!=NULL)
        {
            sum+=temp2->data;
            temp2 = temp2->next;
        }
        sum+=carry;
        carry=sum/10;
        sum%=10;
        Node* x = new Node(sum);
        temp->next = x;
        temp = temp->next;
    }
    if(carry)
    {
        Node* x = new Node(1);
        temp->next = x;
        temp = temp->next;
    }
    head = reverseList(head->next);
    return head;
}
```

12) Pairwise swap elements of a linked list

Given a singly linked list of size N. The task is to swap elements in the linked list pairwise.
For example, if the input list is 1 2 3 4, the resulting list after swaps will be 2 1 4 3.
Note: You need to swap the nodes, not only the data. If only data is swapped then driver will print -1.
Soln:- First swap first two nodes and then call recursively to merge remaining based on swapping only. We store first two nodes and call second's next as recursive call to remaining list. We also put condition that third node should not be null otherwise we just make previous' next as null and at the end return the list.

```
Node* pairWiseSwap(struct Node* head)
  {
    if(head->next==NULL || head==NULL)
    {
       return head;
    }
    Node* ans = head;
    Node* temp1 = head->next;
    Node* temp2 = head->next->next;
    ans->next=NULL;
    temp1->next = ans;
    if(temp2!=NULL)
    {
       ans->next = pairWiseSwap(temp2);
    }
    return temp1;
  }
```

11) Intersection Point in Y Shapped Linked Lists
Given two singly linked lists of size N and M, write a program to get the point where two linked lists intersect each other.
Soln:- We count no of nodes in both the linked list and take their difference. Now we traverse the longer list till their difference so that at that particular node, both the lists have equal length. Now we traverse both the lists in parallel and if we counter same node then we return its value else we return -1 at the end.

```
int intersectPoint(Node* head1, Node* head2)
{
  int count1=0;
  int count2=0;
  Node* temp = head1;
  while(temp!=NULL)
```

```
{
    count1++;
    temp=temp->next;
}
temp = head2;
while(temp!=NULL)
{
    count2++;
    temp=temp->next;
}
Node* temp1;
if(count1>count2)
{
    temp = head1;
    int x = count1-count2;
    while(x--)
    {
        temp = temp->next;
    }
    temp1 = head2;
}
else
{
    temp = head2;
    int x = count2-count1;
    while(x--)
    {
        temp = temp->next;
    }
    temp1 = head1;
}
while(temp!=NULL)
{
    if(temp==temp1)
    {
        return temp->data;
    }
    temp = temp->next;
    temp1 = temp1->next;
}
```

```
    return -1;
}
```

10) Merge two sorted linked lists

Given two sorted linked lists consisting of N and M nodes respectively. The task is to merge both of the list (in-place) and return head of the merged list.

Soln:- Take two temp variables pointing each one to heads of input lists. We check, if data of first node is less than we put it as answer and call recursively the rest of the lists else we put ans as other node's data and call similarly recursively. In the end we return the ans node comprising of sorted list.

```
Node* sortedMerge(Node* head_A, Node* head_B)
{
    if(head_A == NULL)
    {
        return head_B;
    }
    if(head_B == NULL)
    {
        return head_A;
    }
    Node* temp1 = head_A;
    Node* temp2 = head_B;
    Node* ans;
    if(temp1->data < temp2->data)
    {
        ans = temp1;
        ans->next = sortedMerge(temp1->next,temp2);
    }
    else
    {
        ans = temp2;
        ans->next = sortedMerge(temp1,temp2->next);
    }
    return ans;
}
```

STACK AND QUEUE

Left:- 6,9,10

1) Parenthesis Checker

Given an expression string x. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.

For example, the function should return 'true' for exp = “[()]{}{[()()]()}” and 'false' for exp = “[(])”.

Soln:- We use a stack and push the character if it is an opening bracket else we check if the current top is opposite of current closing bracket then we pop eles we return false from that only. In the end we check if the stack is empty, if it is then we return true else false.

```
bool ispar(string x)
   {
      stack<char> s;
      for(int i=0;i<x.size();i++)
      {
         if(x[i]=='('|| x[i]=='{' || x[i]=='[')
         {
            s.push(x[i]);
         }
         else
         {
            if(x[i]==')')
            {
               if(s.empty() || s.top()!='(')
               {
                  return false;
               }
               s.pop();
            }
            else if(x[i]==']')
            {
               if(s.empty() || s.top()!='[')
               {
                  return false;
               }
               s.pop();
            }
            else
            {
               if(s.empty() || s.top()!='{')
```

```
                {
                    return false;
                }
                s.pop();
            }
        }
    }
    if(s.empty())
    {
        return true;
    }
    return false;
}
```

2) Next Greater Element

Given an array arr[ ] of size N having distinct elements, the task is to find the next greater element for each element of the array in order of their appearance in the array. Next greater element of an element in the array is the nearest element on the right which is greater than the current element.

If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

Soln:- We first store -1 in vector corresponding to last element. Next we traverse the array from second last element and check if it's value is greater than top. If it is then we pop the stock till it is just smaller and then push the top's value to vector and current array value in the stack. Finally we just reverse the vector and return it.

```
vector<long long> nextLargerElement(vector<long long> arr, int n){
    vector<long long> v;
    v.push_back(-1);
    stack<long long> s;
    s.push(arr[n-1]);
    for(int i=n-2;i>=0;i--)
    {
        if(arr[i]<s.top())
        {
            v.push_back(s.top());
        }
        else
        {
            while(!s.empty() && arr[i]>=s.top())
            {
```

```
                s.pop();
            }
            if(s.empty())
            {
                v.push_back(-1);
            }
            else
            {
                v.push_back(s.top());
            }
        }
        s.push(arr[i]);
    }
    reverse(v.begin(), v.end());
    return v;
}
```

3) Queue using two Stacks
Implement a Queue using 2 stacks s1 and s2 .
A Query Q is of 2 Types
(i) 1 x (a query of this type means pushing 'x' into the queue)
(ii) 2   (a query of this type means to pop element from queue and print the poped element)
Soln:- When we want to push then we push to stack s1 and if we want to pop then we shift element of s1 to s2 as we have to pop bottom most element in s1 so when we transfer s1 to s2, s1's bottom becomes s2's top so we pop it and put remaining back to s1. If s1 is initially empty we return -1 else we return the top ele;

```
void StackQueue :: push(int x)
 {
   s1.push(x);
 }

/*The method pop which return the element poped out of the queue*/
int StackQueue :: pop()
{
   if(s1.empty())
   {
      return -1;
   }
   while(!s1.empty())
```

```
    {
        s2.push(s1.op());
        s1.pop();
    }
    int ele = s2.top();
    s2.pop();
    while(!s2.empty())
    {
        s1.push(s2.top());
        s2.pop();
    }
    return ele;
}
```

4) Stack using two queues.
Implement a stack usign two queues q1 and q2.
Soln:- In the push operation, just push the value to q1 and in pop operation, push the queue's value to another queue except the last element as it is to be popped. After the transfer, just pop the single remaining element and transfer the new formed queue (q2) back to q1.

```
void QueueStack :: push(int x)
{
    q1.push(x);
}

/*The method pop which return the element popped out of the stack*/
int QueueStack :: pop()
{
    if(q1.empty())
    {
        return -1;
    }
    while(q1.size()>1)
    {
        q2.push(q1.front());
        q1.pop();
    }
    int ele = q1.front();
    q1.pop();
    while(!q2.empty())
```

```
    {
        q1.push(q2.front());
        q2.pop();
    }
    return ele;
}
```

5) Get minimum element from stack
You are given **N** elements and your task is to Implement a Stack in which you can get minimum element in O(1) time.
Soln:- We make two stacks and during pushing, we push the element to s1 and check if s2 is empty or element is <= minEle, then we push it to s2. When we pop, we check if s1.top() is same as minimum element. If it is then we pop s2 also as it contains minimum element in order of input. While fetching minEle, we check if s2 is empty else we return it's top.

```
stack<int> s1;
stack<int> s2;

/*returns min element from stack*/
int _stack :: getMin()
{
    if(s2.empty())
    {
        return -1;
    }
    return s2.top();
}

/*returns poped element from stack*/
int _stack ::pop()
{
    if(s1.empty())
    {
        return -1;
    }
    int ele = s1.top();
   if(ele==getMin())
   {
      s2.pop();
   }
```

```
  s1.pop();
  return ele;
}

/*push element x into the stack*/
void _stack::push(int x)
{
  s1.push(x);
  if(s2.empty() || x<=getMin())
  {
     s2.push(x);
  }
}
```

7) Circular tour

Suppose there is a circle. There are **N** petrol pumps on that circle. You will be given two sets of data.

1. The amount of petrol that every petrol pump has.

2. Distance from that petrol pump to the next petrol pump.

Find a starting point where the truck can start to get through the complete circle without exhausting its petrol in between.

Note : Assume for 1 litre petrol, the truck can go 1 unit of distance.

Soln:- We first compute petrol and distance sum and if petrol is less than distance then return -1. Then traverse the array and keep a variable sum and keep adding the difference of petrol and distance. If sum becomes 0 then make start as i+1 and reset sum as 0. Finally return the start .

```
int tour(petrolPump p[],int n)
   {
      int petrolSum = 0;
      int distanceSum = 0;
      for(int i=0;i<n;i++)
      {
         petrolSum+=p[i].petrol;
         distanceSum+=p[i].distance;
      }
      if(petrolSum<distanceSum)
      {
         return -1;
      }
      int sum = 0;
```

```
    int start = 0;
    for(int i=0;i<n;i++)
    {
        sum+=(p[i].petrol-p[i].distance);
        if(sum<0)
        {
            sum = 0;
            start = i+1;
        }
    }
    return start;
}
```

8) First non-repeating character in a stream

Given an input stream of **A** of **n** characters consisting only of lower case alphabets. The task is to find the first non repeating character, each time a character is inserted to the stream. If there is no such character then append '#' to the answer.

Soln:- We first make an array of size 26 to store the frequency of the characters till current index and we use queue to queue up the elements which has frequency 1. So in each iteration we clear the queue till we get the front element as the one with 1 frequency and if frequency of current element is 1 then we push it to queue. If currently the queue is empty then we append ‘#’ else we append q.front(). Finally we return the output string.

```
string FirstNonRepeating(string A){
                    queue<char> q;
                    int arr[26]={0};
                    string out = "";
                    for(int i=0;i<A.size();i++)
                    {
                        arr[int(A[i])-97]++;
                        while(!q.empty() && arr[int(q.front())-97]>1)
                    {
                        q.pop();
                    }
                        if(arr[int(A[i])-97]==1)
                    {
                        q.push(A[i]);
                    }
                        if(q.empty())
                        {
```

```
                out+='#';
            }
            else
            {
                out+=q.front();
            }
        }
        return out;
    }
```

TREES

Note:- Left 16

1) Left View of Binary Tree

Given a Binary Tree, print Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Soln:- We first push the value of root in vector and then keep a level to make sure that we enter only single element from each level. We first go to left and incase we cant find a element we go to right and keep a global variable to keep the record of level we are currently on.

```
vector<int> v;
int k;
void helper(Node *root,int level)
{
   if(root==NULL)
   {
      return;
   }
   if(level==k)
   {
      k++;
      v.push_back(root->data);
   }
   helper(root->left,level+1);
   helper(root->right,level+1);
}
vector<int> leftView(Node *root)
```

```
{
    v.clear();
    k=0;
    helper(root,k);
    return v;
}
```

2) Check for BST
Given a binary tree. Check whether it is a BST or not.
Note: We are considering that BSTs can not contain duplicate Nodes.
Soln:- We make two variables minVal and maxVal to make sure that the current node's value lie in between them. So initially they are INT_MIN and INT_MAX and as we move to left and right subtrees, we tighten them to make sure the values are in correct range. While going left we make sure the value is less than root and while going right, value is greater than root so we return false if either value is outer bound.

```
bool helper(Node* root,int minVal,int maxVal)
    {
        if(root==NULL)
        {
            return true;
        }
        if(root->data<minVal || root->data>maxVal)
        {
            return false;
        }
        return helper(root->left,minVal,root->data-1) &&
helper(root->right,root->data+1,maxVal);
    }
    bool isBST(Node* root)
    {
        return helper(root,INT_MIN,INT_MAX);
    }
```

3) Bottom View of Binary Tree
Given a binary tree, print the bottom view from left to right.
A node is included in bottom view if it can be seen when we look at the tree from bottom.
Soln:- We use two maps, one to store the distance from root and other to maintain the level to make sure we take the bottom one only. So we check if distance till present

node already exist in the map, if it does then we check if this distance's level is more than current level if it is then we update the dist value in map else we not and then we call it for left and right subtree.

```
vector<int> v;
map<int,int> m;
map<int,int> l;
void helper(Node* root,int dis,int level)
{
    if(root==NULL)
    {
        return;
    }
    if(l[dis]<=level)
    {
        m[dis]=root->data;
        l[dis]=level;
    }
    helper(root->left,dis-1,level+1);
    helper(root->right,dis+1,level+1);
}
vector <int> bottomView(Node *root)
{
   v.clear();
   m.clear();
   l.clear();
   helper(root,0,0);
//   for(auto ite:m)
//   {
//       cout<<ite.first<<" "<<ite.second<<endl;
//   }
   for(auto ite:m)
   {
       v.push_back(ite.second);
   }
   return v;
}
```

4) Vertical Traversal of Binary Tree

Given a Binary Tree, find the vertical traversal of it starting from the leftmost level to the rightmost level.

If there are multiple nodes passing through a vertical line, then they should be printed as they appear in **level order** traversal of the tree.
Soln:- We maintain a map of int and multimap to store for each dis a multimap which can store level and root->data. Dis here is the distance from the root as done in bottom view to store the elements of a particular dis which themselves would be in a vertical order so we store all elements having same dis as a list as they correspond to same vertical order. So we store the elements in a multimap which stores their level and their value. We store them in a multimap to make sure that upper elements appear first then bottom ones so map ensures this constraint and we use multimap instead of normal map as there may be multiple elements with same level so they are here stored in order of their appearance ie level order as we first store left and then right so this ensures the level order for same level elements with same dis. Finally we push the elements of the map in the vector by iterating the main map then further its multimap(the one the main map is itself mapped to).

```
 map<int,multimap<int,int> > m;
    vector<int> v;
    void helper(Node* root,int dis,int level)
    {
        if(root==NULL)
        {
            return;
        }
        m[dis].insert({level,root->data});
        helper(root->left,dis-1,level+1);
        helper(root->right,dis+1,level+1);
    }
    vector<int> verticalOrder(Node *root)
    {
        m.clear();
        v.clear();
        helper(root,0,0);
        for(auto ite:m)
        {
            for(auto x:ite.second)
            {
                v.push_back(x.second);
            }
        }
        return v;
    }
```

5) Level order traversal in spiral form
Complete the function to find spiral order traversal of a tree
Soln:- Do normal level order traversal by making a vector of vectors and pushing the elements to new level vector on reaching new level. We make variable k and check if k is equal to current vector size. If it is then we push a vector signifying that we reached a new level else we push the current element to v[k] and call for left and right subtree. Finally we reverse the vector alternatively and push al its elements to a new vector.

```
vector<vector<int> > v;
void helper(Node* root,int k)
{
    if(root==NULL)
    {
        return;
    }
    if(k==v.size())
    {
        v.push_back(vector<int>());
    }
    v[k].push_back(root->data);
    helper(root->left,k+1);
    helper(root->right,k+1);
}
//Function to return a list containing the level order traversal in spiral form.
vector<int> findSpiral(Node *root)
{
    v.clear();
    helper(root,0);
    for(int i=0;i<v.size();i+=2)
    {
        reverse(v[i].begin(),v[i].end());
    }
    vector<int> ans;
    for(int i=0;i<v.size();i++)
    {
        for(int j=0;j<v[i].size();j++)
        {
            ans.push_back(v[i][j]);
        }
    }
```

```
    return ans;
}
```

6) Connect Nodes at Same Level

Given a binary tree, connect the nodes that are at same level. You'll be given an addition **nextRight** pointer for the same.

Initially, all the **nextRight** pointers point to **garbage** values. **Your function** should set these pointers to point next right for each node.

Soln:- We store all nodes similar to level order traversal but this time not just value, we store whole node and finally while vector traversal, we make current's nextRight the next element while working in a particular row. We do it for each row only for size -1 and make nextRight of last element of each row as NULL.

```
vector<vector<Node* > > v;

    void helper(Node* root,int k)

    {

        if(root==NULL)

        {

            return;

        }

        if(k==v.size())

        {

            v.push_back(vector<Node*>());

        }

        v[k].push_back(root);

        helper(root->left,k+1);

        helper(root->right,k+1);

    }


    void connect(Node *root)

    {

        v.clear();
```

```
    helper(root,0);
    for(int i=0;i<v.size();i++)
    {
        for(int j=0;j<v[i].size()-1;j++)
        {
            v[i][j]->nextRight = v[i][j+1];
        }
        v[i][v[i].size()-1]->nextRight = NULL;
    }
}
```

7) Lowest Common Ancestor in a BST

Given a Binary Search Tree (with all values unique) and two node values. Find the Lowest Common Ancestors of the two nodes in the BST.

Soln:- If current root is NULL return NULL else check if current root is less than both n1 and n2 then call for right subtree else if current root is greater than both n1 and n2 call for left subtree else return current root as if both above case fails then this is the least common ancestor.

```
Node* LCA(Node *root, int n1, int n2)
{
    if(root==NULL)
    {
        return root;
    }
    if(root->data<n1 && root->data<n2 )
    {
        return LCA(root->right,n1,n2);
    }
```

```
    else if(root->data>n1 && root->data>n2)
    {
        return LCA(root->left,n1,n2);
    }
    return root;
}
```

8) Binary Tree to DLL

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (leftmost node in BT) must be the head node of the DLL.

Soln:- We basically store the rightmost and leftmost nodes from both left and right subtrees and temp as root node. We make temp's left as left trees' rightmost node and temp's right as right trees' leftmost node and we check if leftmost node of left subtree and rightmost node of right subtree are null. If they are then we make them as temp and we return their pair. We are storing the endpoints of till formed linked list and returning the leftmost and rightmost nodes of currently built doubly linked list. And finally we return it's leftmost node which is its starting. We are calling recursion for small subproblems to form small doubly linked lists coming from left and right which can then be merged with root so we obtain leftList-root-rightList which is forming a bigger linked list and giving us answer.

```
pair<Node*,Node*> helper(Node* root)
    {
        if(root==NULL)
        {
            return pair<Node*,Node*>(NULL, NULL);
        }
        Node* temp = root;
        pair<Node*,Node*> one = helper(root->left);
```

```
    Node* prevFirst = one.first;
    Node* prevSecond = one.second;
    pair<Node*,Node*> two = helper(root->right);
    Node* nextFirst = two.first;
    Node* nextSecond = two.second;
    temp->right = nextFirst;
    if(nextFirst!=NULL)
    {
        nextFirst->left = temp;
    }
    temp->left = prevSecond;
    if(prevSecond!=NULL)
    {
        prevSecond->right = temp;
    }
    if(prevFirst==NULL)
    {
        prevFirst = temp;
    }
    if(nextSecond==NULL)
    {
        nextSecond = temp;
    }
    return make_pair(prevFirst,nextSecond);
}
Node * bToDLL(Node *root)
{
```

```
    Node* ans = helper(root).first;
    return ans;
}
```

9) Determine if Two Trees are Identical

Given two binary trees, the task is to find if both of them are identical or not.

Soln:- We check if current value of both's root is same or not, is same then we call for right and left subtrees for both r1 and r2 else if different we return false. If all conditions are checked then we return true at the end.

```
bool isIdentical(Node *r1, Node *r2)
{
    if(r1==NULL && r2==NULL)
    {
        return true;
    }
    if(r1!=NULL && r2==NULL)
    {
        return false;
    }
    if(r2!=NULL && r1==NULL)
    {
        return false;
    }
    if(r1->data!=r2->data)
    {
        return false;
    }
    if(!isIdentical(r1->left,r2->left) || !isIdentical(r1->right,r2->right))
```

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

10) Symmetric Tree

Given a Binary Tree. Check whether it is Symmetric or not, i.e. whether the binary tree is a **Mirror image of itself** or not.

Soln:- We use two roots here as while checking mirror we have to make sure that tree's left is equal to it's right so we use a helper function which take two roots basically and check if both's data is different. If both are NULL return true, if one is NULL and other is not then return false. Else check if both have different data. If different data then return false. Now call for right and left subtrees while passing root1->left, root->right and root1->right,root2->left separately in the helper function to make sure that left and right parts are same for smaller sub problems since right and left are mirror of each other.

The essence of taking two different roots is to compare two different trees which are essentially same ie to ease the process of checking the mirror otherwise it is difficult to traverse and do two comparisons in the same tree.

```
bool helper(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;
    }
    if(root1->data!=root2->data)
    {
        return false;
    }
    if(!helper(root1->left,root2->right) || !helper(root1->right,root2->left))
    {
        //cout<<"HI"<<endl;
        return false;
    }
    return true;
}
bool isSymmetric(struct Node* root)
{
          return helper(root,root);
}
```

11) Height of Binary Tree

Given a binary tree, find its height.

Soln:- If root is NULL, return 0 else return 1 + maximum of left and right subtree's height ie call for left and right subtree to give their respective heights.

```
 int height(struct Node* node){
        if(node==NULL)
        {
            return 0;
```

```
        }
        return 1+max(height(node->left),height(node->right));
    }
```

14) Count Leaves in Binary Tree

Given a Binary Tree of size **N** , You have to count leaves in it. For example, there are two leaves in following tree

Soln:- Take a global variable count and increment it each time you encounter a node whose right and left values are NULL.

```
int countLeaf = 0;
void helper(Node* root)
{
    if(root==NULL)
    {
        return;
    }
    if(root->left==NULL && root->right==NULL)
    {
        countLeaf++;
        return;
    }
    helper(root->left);
    helper(root->right);
}
int countLeaves(Node* root)
{
    countLeaf=0;
    helper(root);
```

```
    return countLeaf;
}
```

12) Maximum Path Sum between 2 Leaf Nodes

Given a binary tree in which each node element contains a number. Find the maximum possible sum from one leaf node to another.

Soln:- We do it similar to find maximum path sum but we just do not use max in l and r as here we surely require left and right to include leaf nodes so we return 0 if root is NULL then we bring left and right sums and while bringing them we check if either of left and right nodes are NULL. If they are we return root->data + any if available and we do not update ans here as max(ans,root->data+l+r) because we dont have other leaf so we just return the value else we if both are there then we update the ans as max of current ans and (leftSum+rightSum+root->data) and return max(l,r) + root->sum to include maximum path ie to contribute maximum possible sum from present root node so we choose max of left and right and join it in root->node and return the value to be joined in its parent call. Finally we obtain maximum sum connecting leaf nodes. We are basically contributing maximum possible path or segment of root and its child to global sum path that's why we are returning (root's data + max child) and if root has only one child then we return it only and if root has no child then we return root only.

```
int helper(Node* root,int &ans)
{
    if(root==NULL)
    {
        return 0;
    }
    int l = helper(root->left,ans);
    int r = helper(root->right,ans);
    if(root->left==NULL && root->right!=NULL)
    {
        return root->data+r;
    }
```

```
    if(root->right==NULL && root->left!=NULL)
    {
        return root->data+l;
    }
    if(root->left==NULL && root->right==NULL)
    {
        return root->data;
    }
    ans = max(ans,l+r+root->data);
    return max(l,r)+root->data;
}
int maxPathSum(Node* root)
{
    int ans = INT_MIN;
    helper(root,ans);
    return ans;
}
```

13) Diameter of Binary Tree

Given a Binary Tree, **find diameter of it**.

The diameter of a tree is the number of nodes on the longest path between two end nodes in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).

Soln:- Diameter of the tree is maximum of left diameter, right diameter and (1+leftHeight+rightHeight) so we find left and right heights and add 1 and find maximum of these 3 quantities as diameter may or may not include the root so we call for left and right subtrees as well to find the diameter as they individually may contain long chain of nodes. Hence one case is root + left + right heights and individual left and right diameters.

```
int height(Node* root)
   {
      if(root==NULL)
      {
         return 0;
      }
      return 1+max(height(root->left),height(root->right));
   }
   int diameter(Node* root)
   {
      if(root==NULL)
      {
         return 0;
      }
      int leftHeight = height(root->left);
      int rightHeight = height(root->right);
      int maxHeight = 1+leftHeight+rightHeight;
      return max(maxHeight,max(diameter(root->left),diameter(root->right)));
   }
```

15) Check for Balanced Tree

Given a binary tree, find if it is height balanced or not.

A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

Soln:- If root is NULL return true . If absolute difference of heights of left and right subtrees is greater than one then return false. Now if left and right subtrees are themselves not balanced then return false ie both of them should be balanced so if either of them is unbalanced we return false. If all these cases are fine then finally return true.

```
int height(Node* root)
{
    if(root==NULL)
    {
        return 0;
    }
    return 1+max(height(root->left),height(root->right));
}
bool isBalanced(Node *root)
{
    if(root==NULL)
    {
        return true;
    }
    if(abs(height(root->left)-height(root->right))>1)
    {
        return false;
    }
    if(!isBalanced(root->left) || !isBalanced(root->right))
    {
        return false;
    }
    return true;
}
```

HEAP

Note:- Left 3

1) Find median in a stream

Given an input stream of **N** integers. The task is to insert these numbers into a new stream and find the median of the stream formed by each insertion of **X** to the new stream.

Soln:- We create two heaps s and g. ‘S’ has elements in decreasing order and g in increasing order so basically the stream of numbers is arranged in a way such that s and g’s top always hold middle elements of the stream. If we observe at any instant then we see that if we combine elements from right to left in s with left to right in g then we form a stream of increasing numbers where s and g’s top have median elements. So when we insert the elements:- if s is empty means first element of stream then simply push it to s and update median as this element else - if size of s is more than size of g then we will receive the element and push it and make sizes equal so we check if incoming element is more than current median. If is it less than we transfer s.top to g and pop is from s and push this element to s because being less than median element, this element had to be in s to make sure that size of s and g become equal, we first push first element of s to g which then becomes it’s first element else if x is more than median than the incoming element is to be moved to g directly without any intermediate transfer as it was meant to be in g. After this step, the median is updated to be average of s and g’s top element. Now if s and g have same size then we check if incoming element is less than median, if less it is moved to s else to g to maintain consistency. Now since this addition would make size of a and g unequal, we would add to either s or g and take the median as s’ top if added to s else g’s top if added to g. Finally last case is if size of s is less than g- here it is similar to first case and just reverse of it. We check if incoming element is less than median. If less we push it directly to s because s is already less and deserves the element else(x is moe than median) we first push g’s top to s which becomes s’ top and now pop it from g and add incoming element to g. This makes both s and g equal in size and now median is average of s and g’s top. The median is global variable which is being updated after each addition so it is directly returned each time getMedian() is called.

```
priority_queue<double> s;
    priority_queue<double,vector<double>,greater<double> > g;
    double median = 0;
    void insertHeap(int &x)
    {
        if(s.size()==0)
```

```
{
    s.push(x);
    median = x;
    return;
}
if(s.size()>g.size())
{
    if(x<median)
    {
        g.push(s.top());
        s.pop();
        s.push(x);
    }
    else
    {
        g.push(x);
    }
    median = (s.top()+g.top())/2.0;
    return;
}
else if(s.size()==g.size())
{
    if(x<median)
    {
        s.push(x);
        median = s.top();
    }
```

```
        else
        {
            g.push(x);
            median = g.top();
        }
        return;
    }
    else
    {
        if(x<median)
        {
            s.push(x);
        }
        else
        {
            s.push(g.top());
            g.pop();
            g.push(x);
        }
        median = (s.top()+g.top())/2.0;
        return;
    }
}

//Function to balance heaps.
void balanceHeaps()
{
```

```
}

//Function to return Median.
double getMedian()
{
    return median;
}
```

2) Heap Sort

Given an array of size N. The task is to sort the array elements by completing functions **heapify**() and **buildHeap**() which are used to implement Heap Sort.

Soln:- We call buildHeap function inside heapSort function which first traverse array from mid to 0 and heapify it. Heapify function makes a max-heap basically by bringing the present element to its right place. It compares the current element with its both children which are referenced by 2i+1 and 2i+2 indices if current element is in position 'i'. So we call heapify recursively till we can find a bigger children for current element. We basically update the index largest which stores the index of maximum element amongst element and its children. If element itself is largest then we stop else we pass this largest index which is actually new index of swapped element with which we began the function. We call this function recursively until present element is at its correct position and we call heapify for mid to 0 position in the loop. After completing the loop we can say that we now have a max-heap ie each parent is bigger than their children. Next we use another loop to sort the array ie since first element is greatest, we swap it to last and call heapify for rest of the heap ie excluding last element. After heapify is done we now have biggest element at root from the remaining. We again swap it with 0th element as again heapify. This would start giving us array from behind in decreasing order so actually sorting the array.

```
void heapify(int arr[], int n, int i)
{
    int l = 2*i+1;
    int r = 2*i+2;
```

```
    int largest = i;
    if(l<n && arr[l]>arr[largest])
    {
        largest = l;
    }
    if(r<n && arr[r]>arr[largest])
    {
        largest = r;
    }
    if(largest!=i)
    {
        swap(arr[i],arr[largest]);
        heapify(arr,n,largest);
    }
}

public:
//Function to build a Heap from array.
void buildHeap(int arr[], int n)
{
    for(int i=n/2-1;i>=0;i--)
    {
        heapify(arr,n,i);
    }
    for(int i=n-1;i>0;i--)
    {
        swap(arr[0],arr[i]);
```

```
            heapify(arr,i,0);
        }
    }



    public:
    //Function to sort an array using Heap Sort.
    void heapSort(int arr[], int n)
    {
        buildHeap(arr,n);
    }
```

5) Kth largest element in a stream

Given an input stream **arr[]** of **n** integers, find the **k**th largest element for each element in the stream.

Soln:- We use a priority queue and make sure that top element in it is kth largest element. So we iterate the array and if size of queue is less than k then we simply push the element to queue else we check if its top is greater than current element. If it is then we already have our kth largest element so we do nothing else we pop first element from queue and push current element to the queue. Now we check if size of queue is less than k. If less than k then push -1 in output vector else push top element of queue.

```
vector<int> v;
    vector<int> kthLargest(int k, int arr[], int n) {
        priority_queue<int, vector<int>, greater<int> > pq;
        for(int i=0;i<n;i++)
        {
            if(pq.size()<k)
            {
                pq.push(arr[i]);
```

```
        }
        else
        {
            if(arr[i]>pq.top())
            {
                pq.pop();
                pq.push(arr[i]);
            }
        }
        if(pq.size()<k)
        {
            v.push_back(-1);
        }
        else
        {
            v.push_back(pq.top());
        }
    }
    return v;
}
```

6) Merge K sorted linked lists

Given **K** sorted linked lists of different sizes. The task is to merge them in such a way that after merging they will be a single sorted linked list.

Soln:- Write a function to merge two linked lists. Now traverse the lists in the array and keep merging the lists in a global Node* temp which is holding the lists merged till now. Finally return the temp which now holds the sorted linked list which is a combination of all given linked lists.

```
Node* mergeTwoLists(Node* first,Node* second)
   {
      Node* temp1 = first;
      Node* temp2 = second;
      Node* ans;
      if(temp1==NULL)
      {
         return temp2;
      }
      if(temp2==NULL)
      {
         return temp1;
      }
      if(temp1->data<temp2->data)
      {
         ans = temp1;
         ans->next = mergeTwoLists(temp1->next,temp2);
      }
      else
      {
         ans = temp2;
         ans->next = mergeTwoLists(temp1,temp2->next);
      }
      return ans;
   }
   Node * mergeKLists(Node *arr[], int K)
   {
```

```
        Node* temp  = arr[0];
        for(int i=1;i<K;i++)
        {
            temp = mergeTwoLists(temp,arr[i]);
        }
        return temp;
    }
```

RECURSION

Note:- Left 4

1) Flood fill Algorithm

You don't need to read or print anyhting. Your task is to complete the function **floodFill()** which takes image, sr, sc and newColor as input paramater and returns the image after flood filling.

Soln:- Create an array to mark if we have visited a particular cell so that we do not keep moving in an infinite loop. Then we check if current row and column are out of bound. If they are, we return. If current cell's value is not same as the one specified at sr,sc then we return. Else we change current cell as newColor and call for left,right,up,down.

```
bool v[101][101]={};
    void colour(vector<vector<int>>& image,int r,int c,int newColor,int replaced)
    {
        if(v[r][c])
        {
            return;
        }
        if(r==image.size() || r==-1)
        {
            return;
        }
```

```
        if(c==image[0].size() || c==-1)
        {
            return;
        }
        if(image[r][c]!=replaced)
        {
            return;
        }
        image[r][c] = newColor;
        v[r][c] = true;
        colour(image,r-1,c,newColor,replaced);
        colour(image,r,c-1,newColor,replaced);
        colour(image,r+1,c,newColor,replaced);
        colour(image,r,c+1,newColor,replaced);
    }
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)
{
        int replaced = image[sr][sc];
        colour(image,sr,sc,newColor,replaced);
        return image;
    }
```

2) Number of paths

You don't need to read input or print anything. Your task is to complete the function **numberOfPaths()** which takes the integer **M** and integer **N** as input parameters and returns the number of paths..

Soln:- Number of paths are (n+m-2)C(n-1) or (n+m-2)C(m-1) so we calculate this value and returns it.

```
long long printNcR(int n, int r)
{
    long long p = 1, k = 1;
    if (n - r < r)
        r = n - r;

    if (r != 0) {
        while (r) {
            p *= n;
            k *= r;
            long long m = __gcd(p, k);
            p /= m;
            k /= m;

            n--;
            r--;
        }
    }

    else
        p = 1;
    return p;
}
long long numberOfPaths(int m, int n)
{
    return printNcR(m+n-2,n-1);
}
```

5) Josephus problem

Given the total number of persons **n** and a number **k** which indicates that **k-1** persons are skipped and **kth** person is killed in circle in a fixed direction.

The task is to choose the **safe place in the circle** so that when you perform these operations starting from **1st place** in the circle, you are the last one remaining and survive.

Soln:- We initialise a vector and push elements from 1 to n in it. We initialise initial position as 0 and inside the loop we check while v's size is greater than 1. We take an iterator as v.begin to be used to delete an element later. We make k jumps from present position take its modulo with current vector's size and erase the element at this position. This all processing is being done in a loop and after breaking out from it we return v[0] which is the first and only remaining element in the vector.

```
int josephus(int n, int k)
  {
    vector<int> v;
    for(int i=1;i<=n;i++)
    {
       v.push_back(i);
    }
    int ini = 0;
    while(v.size()>1)
    {
       auto ite = v.begin();
       ini--;
       ini+=k;
       ini%=(v.size());
       v.erase(ite+ini);
    }
    return v[0];
```

```
}
```

HASHING

Note: Left 13, 17

1) Sort an array according to the other
Given two integer arrays **A1[ ]** and **A2[ ]** of size **N** and **M** respectively. Sort the first array **A1[ ]** such that all the relative positions of the elements in the first array are the same as the elements in the second array **A2[ ]**.
See example for better understanding.
Note: If elements are repeated in the second array, consider their first occurrence only.
Soln:- We make a hashmap and store the count of each element in array 1, then we traverse array 2 and for each of it's element we push the element from the map to vector while decrementing the count of the element. At the end we traverse the original array 1 and for each element such that its occurrence is atleast one we push it in another vector and sort it and push it to output vector and return it.

```
vector<int> sortA1ByA2(vector<int> A1, int N, vector<int> A2, int M)
{
    vector<int> v;
    //sort(A1.begin(),A1.end());
    unordered_map<int,int> m;
    for(int i=0;i<A1.size();i++)
    {
        m[A1[i]]++;
    }
    for(int i=0;i<M;i++)
    {
        int ele = A2[i];
        while(m[ele]--)
        {
            v.push_back(ele);
        }
    }
    vector<int> temp;
    for(int i=0;i<N;i++)
    {
        if(m[A1[i]]>0)
        {
```

```
            temp.push_back(A1[i]);
        }
    }
    sort(temp.begin(),temp.end());
    for(int i=0;i<temp.size();i++)
    {
        v.push_back(temp[i]);
    }
    return v;
}
```

2) Sorting Elements of an Array by Frequency
Given an array **A[]** of integers, **sort** the array according to **frequency** of elements. That is elements that have higher frequency come first. If frequencies of two elements are same, then smaller number comes first.
Soln:- We store occurrence count of each element and then sort it according to count and print them while taking the help of vector. We sort the vector to according to count and print the numbers.

```
#include<bits/stdc++.h>
using namespace std;
bool cmp(pair<int,int> &a, pair<int,int> &b)
{
    if(a.second == b.second)
    {
        return a.first < b.first;
    }
    return a.second > b.second;
}
int main()
 {
    int t;
    cin>>t;
    while(t--)
    {
        unordered_map<int,int> m;
        int n;
        cin>>n;
        int a[n];
        for(int i=0;i<n;i++)
        {
```

```
            cin>>a[i];
            m[a[i]]++;
        }
        vector<pair <int,int> > v;
        for(auto ite:m)
        {
            v.push_back(make_pair(ite.first,ite.second));
        }
        sort(v.begin(),v.end(),cmp);
        for(int i=0;i<v.size();i++)
        {
            int ele = v[i].second;
            while(ele--)
            {
                cout<<v[i].first<<" ";
            }
        }
        cout<<endl;
    }
          return 0;
}
```

3) Largest subarray with 0 sum

Given an array having both positive and negative integers. The task is to compute the length of the largest subarray with sum 0.

Soln:- We traverse the array and make a hashmap to store the sum till each index. While traversing we also keep checking if the sum till current is 0. If it is 0 then maxlength is ‘i+1’ because this is the max subarray length which has sum 0. We check at index if sum is repeated because if it repeated then sum between those was 0 so if we encounter same sum again then we take maxlength as max of maxlength and current index - previous recorded index. Finally we return the maxlength.

```
int maxLen(int A[], int n)
{
    unordered_map<int,int> m;
    int sum = 0;
    int maxlen = 0;
    for(int i=0;i<n;i++)
    {
        sum+=A[i];
        if(sum==0)
```

```
        {
            maxlen = i+1;
        }
        if(m.count(sum)>0)
        {
            maxlen = max(maxlen,i-m[sum]);
        }
        else
        {
            m[sum] = i;
        }
    }
    return maxlen;
}
```

4) Common Elements

Given two **lists V1 and V2 of sizes n and m** respectively. Return the list of elements common to both the lists and return the list in sorted order. Duplicates may be there in the output list.

Soln:- We map the elements in first vector to its count and while traversing the next vector, we check if its count in map is positive. If it is then we push it to multiset else we dont. Finally we put all elements of multiset in vector and return it.

```
vector<int> common_element(vector<int>v1,vector<int>v2)
{
    unordered_map<int,int> m;
    for(int i=0;i<v1.size();i++)
    {
        m[v1[i]]++;
    }
    multiset<int> s;
    for(int i=0;i<v2.size();i++)
    {
        if(m[v2[i]]>0)
        {
            s.insert(v2[i]);
            m[v2[i]]--;
        }
    }
    vector<int> v;
    for(auto ite:s)
```

```
    {
        v.push_back(ite);
    }
    return v;
}
```

5) Find All Four Sum Numbers
Given an array of integers and another number. Find all the **unique** quadruple from the given array that sums up to the given number.

Soln:- First we sort the array and then we traverse the array in two nested loop to pick two of the four elements and to pick next two elements, we traverse the remaining array to evaluate ‘k - sum_till_now’ in single traversal by using beg<end ie. if sum of beg and end is greater than remaining then we decrement end else if it lesser than we increment beg else we store the vector of these 4 numbers in a set to arrange them lexicographically and at the end we move elements of set to output vector and return it.

```
vector<vector<int> > fourSum(vector<int> &arr, int k) {
    sort(arr.begin(),arr.end());
    set<vector<int> > v;
    for(int i=0;i<arr.size();i++)
    {
        for(int j=i+1;j<arr.size();j++)
        {
            int sum = 0;
            sum+=arr[i];
            sum+=arr[j];
            int rem = k-sum;
            int beg = j+1;
            int end = arr.size()-1;
            while(beg<end)
            {
                if((arr[beg]+arr[end])==rem)
```

```
            {
                vector<int> temp;
                temp.push_back(arr[i]);
                temp.push_back(arr[j]);
                temp.push_back(arr[beg]);
                temp.push_back(arr[end]);
                sort(temp.begin(),temp.end());
                v.insert(temp);
                temp.clear();
                beg++;
                end--;
            }
            else if((arr[beg]+arr[end])>rem)
            {
                end--;
            }
            else
            {
                beg++;
            }
        }
    }
}
vector<vector<int> > x;
for (auto it = v.begin();
     it != v.end();
     it++) {
```

```
        x.push_back(*it);
    }
    return x;
}
```

6) Swapping pairs make sum equal
Given two arrays of integers **A[]** and **B[]** of size **N** and **M**, the task is to check if a pair of values (one value from each array) exists such that swapping the elements of the pair will make the sum of two arrays equal.

Soln:- We first sort both the arrays and then put indexes at start of each array and then check the temp sums by adding second array's element to original sum and subtracting first array's element. If they are same , we return 1 else if first sum is greater than we increase first array's index else we increase second array's index. Outside the loop, we return -1.

```
int findSwapValues(int A[], int n, int B[], int m)
        {
      sort(A,A+n);
      sort(B,B+m);
      int sum1 = 0;
      for(int i=0;i<n;i++)
      {
          sum1+=A[i];
      }
      int sum2=0;
      for(int i=0;i<m;i++)
      {
          sum2+=B[i];
      }
      int ind1=0;
      int ind2=0;
```

```
while(ind1<n && ind2<m)
{
    int temp1 = sum1-A[ind1]+B[ind2];
    int temp2 = sum2+A[ind1]-B[ind2];
    if(temp1==temp2)
    {
        return 1;
    }
    else if(temp1>temp2)
    {
        ind1++;
    }
    else
    {
        ind2++;
    }
}
return -1;
  }
```

7) Count distinct elements in every window

Given an array of integers and a number K. Find the count of distinct elements in every window of size K in the array.

Soln:- We store the occurrence of element in the map and push its size in vector next while removing and entering , we check if the element being removed makes its count 0

If it does then we decrement count by 1 and while entering the element, we check if it came from 0 to positive, if it did then we increment count by 1 as element may exist in map with count 0 which will give wrong answer. Finally return the output vector.

```
vector <int> countDistinct (int A[], int n, int k)
{
    vector<int> v;
    int count = 0;
    unordered_map<int,int> m;
    for(int i=0;i<k;i++)
    {
        m[A[i]]++;
    }
    count=m.size();
    v.push_back(count);
    for(int i=k;i<n;i++)
    {
        m[A[i-k]]--;
        if(m[A[i-k]]<=0)
        {
            count--;
        }
        if(m[A[i]]==0)
        {
            count++;
        }
        m[A[i]]++;
        v.push_back(count);
    }
    return v;
}
```

8) Array Pair Sum Divisibility Problem

Given an array of integers and a number k, write a function that returns true if given array can be divided into pairs such that sum of every pair is divisible by k.

Soln:- We store all the possible remainders from the array in the map ie we keep the occurrence of each remainder in the map and we traverse the map by checking that first-last second-secondLast are all equal to make sure to make pairs and if any of them cannot make a pair ie their occurrence is not equal then we return false. If the size of array is odd then also we return false as it is impossible to make a pair with odd number of elements.Because if (a+b) is divisible by k then (a%k) + (b%k) = k so this is the basic condition we are checking while checking for first-last, second-secondLast and so on to make sure suitable no of pairs are formed to ensure division by k.

```
bool canPair(vector<int> nums, int k) {
    if(nums.size()%2==1)
    {
        return false;
    }
    else
    {
         if(k==1)
        {
            return true;
        }
        map<int,int> m;
        for(int i=0;i<k;i++)
        {
            m[k]=0;
        }
        for(int i=0;i<nums.size();i++)
        {
```

```
    m[nums[i]%k]++;
}
if(m[0]%2==1)
{
    return false;
}
if(k%2==0)
{
    if(m[k/2]%2==1)
    {
        return false;
    }
    for(int i=1;i<=m.size()/2-1;i++)
    {
        if(m[i]!=m[k-i])
        {
            return false;
        }
    }
}
else
{
    for(int i=1;i<=m.size()/2;i++)
    {
        if(m[i]!=m[k-i])
        {
            return false;
```

```
            }
          }
        }
        return true;
    }
}
```

9) Longest consecutive subsequence

Given an array of positive integers. Find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the **consecutive numbers can be in any order.**

Soln:- We put the elements of the array in a hashmap and then traverse the array and check for each element if this element is first element of its consecutive subsequence so to check this, we check if element-1 exist, if it does then it is not the first element else we keep checking the map till we can't find the next element and in each iteration we increment the count to count the no of consecutive elements ahead this element corresponding to this element and while breaking out of the loop, simply update the maxCount as maximum of maxCount and count to have the max value of the length of subsequence.

```
int findLongestConseqSubseq(int arr[], int N)
{
    unordered_map<int,bool> m;
    for(int i=0;i<N;i++)
    {
        if(!m[arr[i]])
        {
            m[arr[i]]=true;
        }
    }
    int maxCount = 0;
```

```
    for(int i=0;i<N;i++)
    {
        int ele = arr[i];
        int count=0;
        if(!m[arr[i]-1])
        {
            int val = arr[i];
            while(m[val])
            {
                count++;
                val++;
            }
        }
        maxCount=max(count,maxCount);
    }
    return maxCount;
}
```

10) Array Subset of another array

Given two arrays: **a1[0..n-1]** of size **n** and **a2[0..m-1]** of size **m**. Task is to check whether a2[] is a subset of a1[] or not. Both the arrays can be sorted or unsorted. It may be assumed that elements in both array are distinct.

Soln:- We put all elements of first array in a map and for second array, we check for each element if it exist in the map. If it does then we return Yes else if even one is missing we return No.

```
string isSubset(int a1[], int a2[], int n, int m) {
    unordered_map<int,bool> mp;
    for(int i=0;i<n;i++)
```

```
    {
        mp[a1[i]]=true;
    }
    string str="Yes";
    for(int i=0;i<m;i++)
    {
        if(!mp[a2[i]])
        {
            str="No";
            break;
        }
    }
    return str;
}
```

11) Find all pairs with a given sum

Given two unsorted arrays **A** of size **N** and **B** of size **M** of distinct elements, the task is to find all pairs from both arrays whose sum is equal to **X**.

Soln:- We mapped elements of first array to true in a hashmap and for the second array we checked if m[X-B[i]] exists. If it exists, it means another sum pair of current element exists in the map and we push their pair to the map else we not. Finally we sort the vector and return it.

```
vector<pair<int,int>> allPairs(int A[], int B[], int N, int M, int X)
{
    vector<pair<int,int> > v;
    unordered_map<int,bool> m;
    for(int i=0;i<N;i++)
    {
```

```
        m[A[i]]=true;
    }
    for(int i=0;i<M;i++)
    {
        if(m[X-B[i]])
        {
            m[X-B[i]]=false;
            v.push_back(make_pair(X-B[i],B[i]));
        }
    }
    sort(v.begin(),v.end());
    return v;
}
```

14) Minimum indexed character

Given a string **str** and another string **patt**. Find the character in **patt** that is present at the minimum index in **str**. If no character of **patt** is present in **str** then print 'No character present'.

Soln:- We first traverse patt and make its characters true in map and then we traverse str, then if any character of str is there is patt we return current index of str else we return -1.

```
int minIndexChar(string str, string patt)
    {
        unordered_map<char,bool> m;
        for(int i=0;i<patt.size();i++)
        {
            m[patt[i]]=true;
        }
```

```
    for(int i=0;i<str.size();i++)
    {
        if(m[str[i]])
        {
            return i;
        }
    }
    return -1;
}
```

15) Check if two arrays are equal or not

Given two arrays **A** and **B** of equal size **N**, the task is to find if given arrays are equal or not. Two arrays are said to be equal if both of them contain same set of elements, arrangements (or permutation) of elements may be different though.

Note : If there are repetitions, then counts of repeated elements must also be same for two array to be equal.

Soln:- We store the occurrence of each element while traversing the first array ie we increment corresponding to a particular character while traversing the first array and decrement the same traversing the second array. After this we check if any element in map has count positive. If it is, then we return false else finally we return true.

```
bool check(vector<ll> arr, vector<ll> brr, int n) {
    unordered_map<int,int> m;
    for(int i=0;i<n;i++)
    {
        m[arr[i]]++;
    }
    for(int i=0;i<n;i++)
    {
        m[brr[i]]--;
```

```
    }
    for(auto ite:m)
    {
        if(ite.second>0)
        {
            return false;
        }
    }
    return true;
}
```

16) Uncommon characters

Given two strings A and B. Find the characters that are not common in the two strings.

Soln:- We map characters of first string in a map and characters of second string in another map using ie storing which characters are present in the map using bool. Then after storing, we iterate over a map and check if current character is also present in another map. If it is, then we mark false for the character in both maps. Finally we append the characters in string whose occurrence is true in both maps which means no same characters are there in both maps.

```
string UncommonChars(string A, string B)
        {
            unordered_map<char,bool> m;
            for(int i=0;i<A.size();i++)
            {
                m[A[i]]=true;
            }
            unordered_map<char,bool> m1;
            for(int i=0;i<B.size();i++)
            {
```

```
    m1[B[i]]=true;
}
string str="";
for(auto ite:m)
{
    int val = ite.first;
    if(m[val] && m1[val])
    {
        m[val]=false;
        m1[val]=false;
    }
}
for(auto ite:m)
{
    if(ite.second)
    {
        str+=ite.first;
    }
}
for(auto ite:m1)
{
    if(ite.second)
    {
        str+=ite.first;
    }
}
if(str.size()==0)
```

```
        {
            return "-1";
        }
        sort(str.begin(),str.end());
        return str;
    }
```

19) First element to occur k times

Given an array of **N** integers. Find the first element that occurs **K** number of times.

Soln:- We create a hash map and store the occurrence of each element, while incrementing, we keep checking if current element's count reached k. If it did we return it else we return -1 at the end.

```
int firstElementKTime(int a[], int n, int k)
  {
     unordered_map<int,int> m;
     for(int i=0;i<n;i++)
     {
        m[a[i]]++;
        if(m[a[i]]==k)
        {
            return a[i];
        }
     }
     return -1;
  }
```

19) Check if frequencies can be equal

Given a string **s** which contains only lower alphabetic characters, check if it is possible to remove at most one character from this string in such a way that frequency of each distinct character becomes same in the string.

Soln:- First we put frequencies of all characters in a map and then put value of distinct frequencies and store its corresponding no of characters in another map. Next we check if the second map has size equal to 1 then there is one frequency only ie all characters have equal frequency else if size is greater than 2 then we return false as we can only remove one character and finally we check for the case when the size of map is 2.

Now if the size of map is two we store the values of frequency and its corresponding number of characters in four variables and compare within them. First we compare no of elements for both frequencies and check which one is greater.Now if first ele is greater then second should be 1 as there can be only single different character. If it is not 1 we return false else we check furter. If count of second is 1. If it is then we return true as character with less frequency if 1 can be removed totally else we check if difference between first and second count is 1. If it is then we can remove it and we return true else false. Above is done similar to if first is less than second noEle. If both are equal then one of the count should be 1. If it is then we return true else false.

```
bool sameFreq(string str)
{
    unordered_map<char,int> m;
    for(int i=0;i<str.size();i++)
    {
        m[str[i]]++;
    }
   // for(auto ite:m)
   // {
   //     cout<<ite.first<<" "<<ite.second<<endl;
   // }
    unordered_map<int,int> m2;
    for(auto ite:m)
    {
        m2[ite.second]++;
```

```
 }
// for(auto ite:m2)
// {
//     cout<<ite.first<<" "<<ite.second<<endl;
// }
 if(m2.size()==1)
 {
    return true;
 }
 else if(m2.size()>2)
 {
    return false;
 }
 auto ite = m2.begin();
 int noEle1 = ite->second;
 int count1 = ite->first;
 ite++;
 int noEle2 = ite->second;
 int count2 = ite->first;
 if(noEle1>noEle2)
 {
    if(noEle2!=1)
    {
        return false;
    }
    if(count2==1)
    {
```

```
            return true;
        }
        if(count2-count1 == 1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else if(noEle1<noEle2)
    {
        if(noEle1!=1)
        {
            return false;
        }
        if(count1==1)
        {
            return true;
        }
        if(count1-count2 == 1)
        {
            return true;
        }
        else
        {
```

```
            return false;
        }
    }
    else
    {
        if(count1==1 || count2==1)
        {
            return true;
        }
        return false;
    }
}
```

GRAPHS

1) DFS of Graph

Given a connected undirected graph. Perform a Depth First Traversal of the graph.

Note: Use recursive approach to find the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph..

Soln:- We make a map and a global vector to store the answer. We first visit 0, push it into vector and mark it as true in map. Then we recursively call for all the adjacent nodes of 0 and so on through recursion.

```
unordered_map<int,bool> m;
    vector<int> ans;
    void helper(int V,vector<int> adj[])
    {
        m[V] = true;
```

```
    ans.push_back(V);
    for(auto ite:adj[V])
    {
        if(!m[ite])
        {
            helper(ite,adj);
        }
    }
}
    vector<int>dfsOfGraph(int V, vector<int> adj[]){
        ans.clear();
        m.clear();
        helper(0,adj);
        return ans;
    }
```

2) BFS of graph

Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.

Note: One can move from node u to node v only if there's an edge from u to v and find the BFS traversal of the graph starting from the 0th vertex, from left to right according to the graph. Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

Soln:- We first push 0 in the queue and use a map to check if the current node has been visited or not. We traverse for the top element in the queue and check if any element of the list is visited. If not then push it in vector, mark it true and push in queue. After each traversal, pop from the queue. Finally return the vector.

```
unordered_map<int,bool> m;
    vector<int>bfsOfGraph(int V, vector<int> adj[]){
```

```
        m.clear();
        vector<int> v;
        queue<int> q;
        v.push_back(0);
        q.push(0);
        while(!q.empty())
        {
            int val = q.front();
            for(auto ite:adj[val])
            {
                if(!m[ite])
                {
                    v.push_back(ite);
                    m[ite] = true;
                    q.push(ite);
                }
            }
            q.pop();
        }
        return v;
    }
```

3) Detect cycle in an undirected graph

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Soln:- We do a dfs search and pass the parent of current node also to make sure not to make it false everytime since during recursion checking, we check if the current node has been already visited. So we check further only if the current element is not the

parent of the node we are checking for ie the current node was not called by this element during recursion call. So we check for all vertices if they have been already visited. If they have not then we call isCyclic function to check for the non connected components. Now in the helper function, we return true only if isCyclic bool value is true because if is false then not necessarily the remaining node chain for that element would be false, it can be true for some other chain so we return true only if the cycle is present else we check for remaining nodes. If the loop did not run means cycle is not present so we simply return false.

```
unordered_map<int, bool> m;
    bool helper(int V,vector<int> adj[],int parent)
    {
        m[V] = true;
        for(auto ite:adj[V])
        {
            if(ite!=parent)
            {
                if(m[ite])
                {
                    return true;
                }
                else
                {
                    bool isCycle = helper(ite,adj,V);
                    if(isCycle)
                    {
                        return true;
                    }
                }
            }
```

```
    }
    return false;
}
    bool isCycle(int V, vector<int>adj[]){
        m.clear();
        for(int i=0;i<V;i++)
        {
            if(!m[i])
            {
                if(helper(i,adj,-1))
            {
                return true;
            }
            }
        }
        return false;
    }
```

4) Detect cycle in a directed graph

Given a Directed Graph with **V** vertices (Numbered from **0** to **V-1**) and **E** edges, check whether it contains any cycle or not.

Soln:- We use two maps, one to mark visited nodes and other to check if the node exists in the current path we are checking. We loop to check if a current node is visited or not in case of disconnected components or components which might not have arrived in the path till now. So at each recursive stack, we mark visited and path as true for the current node and run a loop to check for all the components connected with the current node through dfs and if the visited is false then call recursion for that node and at the end of the loop, just backtrack and mark path[V] as false to remove current src node from path and return false which shows that cycle was not detected with this node as the source and in the for loop we store the value and check if it is true. If yes then return

true. Here we do not return false directly because we might have not visited the node yet which might have cycle so we cannot directly conclude that cycle doesn't exist similar to done is undirected graph cycle.

```
unordered_map<int,bool> visited;

unordered_map<int,bool> path;

class Solution

{

    public:

        //Function to detect cycle in a directed graph.

        bool helper(int V,vector<int> adj[])

        {

            visited[V] = true;

            path[V] = true;

            for(auto ite:adj[V])

            {

                if(path[ite])

            {

                return true;

            }

            if(!visited[ite])

            {

                bool isCycle = helper(ite,adj);

                if(isCycle)

                {

                    return true;

                }

            }

            }
```

```
        path[V] = false;
        return false;
    }
    bool isCyclic(int V, vector<int> adj[])
    {
        visited.clear();
        path.clear();
            for(int i=0;i<V;i++)
            {
                if(!visited[i])
                {
                    if(helper(i,adj))
                    {
                        return true;
                    }
                }
            }
            return false;
    }
};
```

5) Topological sort

Given a Directed Graph with V vertices and E edges, Find any Topological Sorting of that Graph.

Soln:- We traverse from 0 to V-1 and check if the current node is visited. If not visited then we call the helper function in which we recursively go to deepest possible point in the graph from which we can’t go to any other node and push it to the stack because that node will be the one which does not have any dependency since if it had then it would not have been the last possible node. So we push it into the stack and in the

recursive call, we simply add that node to the stack which means that it is now the one without any dependency and so on. So we are pushing into stack when we can no longer find any dependency for the current node and after returning back to the previous recursion stack we push it since it has now been resolved with no dependency because for each node we traverse all of its directed nodes and push them into stack and when they are done we push the node itself because after pushing the deepest nodes they are assumed to be removed and now the calling node is the one with no dependency and so on till we return out of the last recursion stack. We run the loop till no node is unvisited.

```
stack<int> s;

unordered_map<int,bool> m;

class Solution

{

        public:

        //Function to return list containing vertices in Topological order.

        void helper(int V, vector<int> adj[])

        {

            m[V] = true;

            for(auto ite:adj[V])

            {

                if(!m[ite])

                {

                    helper(ite,adj);

                }

            }

            s.push(V);

        }

        vector<int> topoSort(int V, vector<int> adj[])

        {

            m.clear();
```

```
        for(int i=0;i<V;i++)
        {
            if(!m[i])
            {
                helper(i,adj);
            }
        }
        vector<int> v;
        while(!s.empty())
        {
            //cout<<s.top()<<" ";
            v.push_back(s.top());
            s.pop();
        }
        return v;
    }
};
```

6) Find the number of islands

Given a grid consisting of '0's(Water) and '1's(Land). Find the number of islands.

Note: An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i,e in all 8 directions.

Soln:- We check for each element for grid if it is visited or not. If not then we call the helper function to mark all possible adjacent nodes with value 1 as visited and the base case that if the node is already visited then return or if it is 0 then also return. We also check that i and j are valid indices ie not out of bound. Then we increment count each time we have to call helper function explicitly.

```
void helper(vector<vector<char>>& grid,int i,int j,int n,int m)
   {
```

```
    if(i<0 || i>=n || j<0 || j>=m)
    {
        return;
    }
    if(visited[i][j] || grid[i][j]=='0')
    {
        return;
    }
    visited[i][j] = true;
    helper(grid,i-1,j-1,n,m);
    helper(grid,i-1,j+1,n,m);
    helper(grid,i+1,j-1,n,m);
    helper(grid,i+1,j+1,n,m);
    helper(grid,i,j-1,n,m);
    helper(grid,i,j+1,n,m);
    helper(grid,i-1,j,n,m);
    helper(grid,i+1,j,n,m);
}
int numIslands(vector<vector<char>>& grid)
{
    int count = 0;
    int n = grid.size();
    int m = grid[0].size();
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
```

```
            visited[i][j] = false;
        }
    }
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            if(grid[i][j]=='1' && !visited[i][j])
            {
                helper(grid,i,j,n,m);
                count++;
            }
        }
    }
    return count;
}
```

7) Implementing Dijkstra Algorithm

Given a weighted, undirected and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S.

Note: The Graph doesn't contain any negative weight cycle.

Soln:- We create a vector of size V to store the distance of all vertices from the source node (we initialise dist[source] as 0) and an unordered set(we push source node here) to store those nodes whose minimum possible distance has already been calculated. So we run a while loop which runs till the size of set is less than V ie not all nodes have been put in the set and then we find the element which is not currently in the set with current minimum distance from the source using a helper function (in the first iteration the minimum distance element we get is one of the adjacent node of the source node because we updated their distance in the first step as their distance is fix equal to the weight of the edge between them and the source and then afterwards depends on the

value of the node) and then insert it in the set and iterate for its adjacency list to update their distance as (weight of edge + distance of parent node from source) if the current distance of the node is more than this value. Finally return the dist vector which stores the distance of all nodes from the source node.

```
int helper(unordered_set<int>& s,vector<int>& dist,int S)
  {
     int minDis = INT_MAX;
     int minNum = 0;
     for(int i=0;i<dist.size();i++)
     {
        if(minDis>dist[i] && s.find(i)==s.end())
        {
           minDis = dist[i];
           minNum = i;
        }
     }
     return minNum;
  }
  vector <int> dijkstra(int V, vector<vector<int>> adj[], int S) {
     vector<int> dist(V);
     for(int i=0;i<V;i++)
     {
        dist[i] = INT_MAX;
     }
     dist[S] = 0;
     unordered_set<int> s;
     s.insert(S);
     for(auto ite:adj[S])
```

```
        {
            dist[ite[0]] = ite[1];
        }
        while(s.size()!=V)
        {
            int num = helper(s,dist,S);
            s.insert(num);
            for(auto ite:adj[num])
            {
                if(s.find(ite[0])==s.end())
                {
                    if(dist[ite[0]]>ite[1]+dist[num])
                    {
                        dist[ite[0]] = ite[1] + dist[num];
                    }
                }
            }
        }
        return dist;
    }
```

8) Minimum Swaps to Sort

Given an array of n distinct elements. Find the minimum number of swaps required to sort the array in strictly increasing order.

Soln:- We store the original vector in another vector and sort it. Then we store the corresponding elements of both vectors in a map and we use another map to check if we have visited the element before. Now what we are doing is we make cyclic components in the graph which basically contains those elements which are in each others position so we assume them to be in a single cyclic component. Now the number

of nodes in a component - 1 is the number of swaps required to make them sorted. We are basically sorting the components because they all are wrongly arranged in each other's position and has nothing to do with other components. So we just have to count the no of nodes in each component and return the summation of (No of nodes - 1) for each component. So we iterate over the map to check which node is not yet visited. If not visited, we call the helper function which takes the key element of the map and check if it is visited. If not then visit it and make m[ele] = true and call recursively for the value of this key which itself is a key for some other value and so on. Recursion goes on and reaches a step where the cycle is completed and we encounter the current element to be already visited which was visited at the start of recursion stack. So we update temp which stores the number of nodes in current component and return from here. In main where this was called, we increment count as (temp - 1). We do it till all elements in m(visited map) are not visited or if any one is still false.

```
int temp = 0;

    void helper(unordered_map<int,bool>& m,unordered_map<int,int>& pairs,int ele,int count)

    {

        if(m[ele])

        {

            temp = count;

            return;

        }

        else

        {

            m[ele] = true;

            helper(m,pairs,pairs[ele],1+count);

        }

    }

          int minSwaps(vector<int>&nums){

              vector<int> x = nums;

              sort(x.begin(),x.end());
```

```
        unordered_map<int,bool> m;
        unordered_map<int,int> pairs;
        for(int i=0;i<nums.size();i++)
        {
            pairs.insert(make_pair(nums[i],x[i]));
        }
        int count = 0;
        for(auto ite:pairs)
        {
            if(!m[ite.first])
            {
                temp = 0;
                helper(m,pairs,ite.first,0);
                count += (temp-1);
            }
        }
        return count;
    }
```

9) Strongly Connected Components (Kosaraju's Algo)

Given a Directed Graph with **V** vertices **(**Numbered from **0 to V-1)** and **E** edges, Find the number of strongly connected components in the graph.

Soln:- We use **kosaraju** algorithm where we build the reverse graph and perform the topological sort algorithm to store the elements in the stack. Now to count no of connected components we are doing something similar to topological sort ie during topological sort, we store the least dependent element with no incoming at the the top and all outgoing at bottom so a component is actually a cycle so we have to treat it like a single node so that it can be analysed normally. Now a component will have a node outgoing externally as well as internally so we reverse all the nodes ie whole graph so that all outgoing nodes become incoming now we store elements of reverse graph in a

stack according to topological sort and compare it with original graph dfs traversal. So we traverse the stack from top and do the dfs traversal from the top element and mark all visited as true. Now pop from stack and if current top is already visited means it was part of component already visited so we just pop it out and during each dfs function call we increment count by 1 because it means it is a dfs call for a wholly new component which is not yet visited. Also it is for sure that we would not visit any other component from another component because all other nodes of other components would have visited already during a call to any node of that component.

Node:- All nodes of a component do not necessarily be together in stack because it depends on the arrow in the graph. Arrows may point out of that component and take us to other component so it is possible that nodes of a single component are not together or in a streak but this does not affect the traversal according to stack because we mark whole of a component's node true because if it is a component that means all of its nodes are reachable from each other or we can say they form a cycle. So on reaching any node of a component, that whole component is bound to be made true which does not affect traversal of any other component. Agar ek component se dusre ki taraf arrow hoga to problem nhi aegi kyuki wo reverse ho jaega in reverse tree and stack m delta time wo dusre component pe nhi jaega during topological sort so interference ni hogi stack m delta hue.

```
unordered_map<int,bool> m;
        stack<int> s;
   void helper(int V, vector<int> adj2[])
   {
      m[V] = true;
      for(auto ite:adj2[V])
      {
         if(!m[ite])
         {
            helper(ite,adj2);
         }
      }
      s.push(V);
   }
```

```
void dfs(int ele,vector<int> adj[])
{
    m[ele] = true;
    for(auto ite:adj[ele])
    {
        if(!m[ite])
        {
            dfs(ite,adj);
        }
    }
}
int kosaraju(int V, vector<int> adj[]) {
    vector<int> adj2[V];
    for(int i=0;i<V;i++)
    {
        for(auto ite:adj[i])
        {
            adj2[ite].push_back(i);
        }
    }
    for(int i=0;i<V;i++)
    {
        if(!m[i])
        {
            helper(i,adj2);
        }
    }
```

```
        int count = 0;
        m.clear();
        while(!s.empty())
        {
            if(!m[s.top()])
            {
                count++;
                dfs(s.top(),adj);
                s.pop();
            }
            else
            {
                s.pop();
            }
        }
        return count;
    }
```

10) Shortest Source to Destination Path

Given a 2D binary matrix A(0-based index) of dimensions NxM. Find the minimum number of steps required to reach from (0,0) to (X, Y).

Note: You can only move left, right, up and down, and only through cells that contain 1.

Soln:- We start from 0,0 and use BFS to traverse the matrix. We use a 2d array to mark all nodes as false for visited and start from 0,0. Now we use pair< pair<int,int>, int > datatype to store x,y and distance from 0,0 ie distance of x,y from 0,0. Now we check for each of left, right, down and up position wrt to queue's front node if it visitable. In the base case if 0,0 or destination has 0 then return -1. Else run a while loop till queue is empty and push the adjacent visitable node wrt to queue's front and mark them true

also pop front element. If at any stage queue's front node's x and y are destination's coordinate then return distance of that node. Finally return -1 means we could not find a path. Now BFS ensures that we reach the destination by shortest path because BFS moves single step ahead from all possible routes at each call so we would actually be spreading in all possible directions by 1 step at each iteration. Now if there is a longer route then it means it would make us reach later then or require more steps to spread till destination but since shorter path require less steps to spread till destination and less steps means less time because single step take some amount of fixed time so if it is 't' then n steps take 'nt' time (assuming minimum steps to reach destination) and if there are more steps then it takes more than 'nt' time.So it reach destination in 'nt' time which is minimum then we do not check for any other possible path because they would definitely be longer otherwise they would have reached earlier. So we reach the destination is minimum possible steps or we can say minimum possible time and when we reach there we return from the function and doesn't check further for any other ongoing path spread because we have already returned so program terminates.

```
bool isValid(vector<vector<int>>& A,int N,int M,int i,int j)
  {
     if(i<0 || i>=N || j<0 || j>=M)
     {
        return false;
     }
     if(A[i][j]==0)
     {
        return false;
     }
     return true;
  }
  int shortestDistance(int N, int M, vector<vector<int>> A, int X, int Y) {
    if(A[0][0]==0 || A[X][Y]==0)
    {
       return -1;
    }
```

```
queue< pair<pair<int,int>,int> > q;
bool visited[N][M];
for(int i=0;i<N;i++)
{
    for(int j=0;j<M;j++)
    {
        visited[i][j] = false;
    }
}
q.push({{0,0},0});
visited[0][0] = true;
while(!q.empty())
{
    int x = q.front().first.first;
    int y = q.front().first.second;
    int dist = q.front().second;
    if(x==X and y==Y)
    {
        return dist;
    }
    if(isValid(A,N,M,x-1,y) and !visited[x-1][y])
    {
        q.push({{x-1,y},1+dist});
        visited[x-1][y] = true;
    }
    if(isValid(A,N,M,x+1,y) and !visited[x+1][y])
    {
```

```
            q.push({{x+1,y},1+dist});
            visited[x+1][y] = true;
        }
        if(isValid(A,N,M,x,y-1) and !visited[x][y-1])
        {
            q.push({{x,y-1},1+dist});
            visited[x][y-1] = true;
        }
        if(isValid(A,N,M,x,y+1) and !visited[x][y+1])
        {
            q.push({{x,y+1},1+dist});
            visited[x][y+1] = true;
        }
        q.pop();
    }
    return -1;
}
```

11) Find whether path exist

Given a grid of size n*n filled with 0, 1, 2, 3. Check whether there is a path possible from the source to destination. You can traverse up, down, right and left.

The description of cells is as follows:

- A value of cell **1** means Source.
- A value of cell **2** means Destination.
- A value of cell **3** means Blank cell.
- A value of cell **0** means Wall.

Note: There are only a single source and a single destination.

Soln:- It is similar to shortest path to destination as here also we have to do bfs search to find if destination exists in the matrix. We first find the coordinates of source ie

grid[i][j] = 1 and mark all nodes as unvisited ie visited[i][j] = false for all nodes. Then we make a queue of type pair to push x and y as pairs of any node of matrix we are on and do bfs search. We check if the current front node of queue is the destination node and return 1 else we check if all adjacent nodes are visitable ie they do not contain 0 and ar not out of bound of matrix and are not visited yet then we push it into queue and mark it as visited. Then pop front from queue because it has already been used. Then return 0 outside the while loop.

```
bool isValid(vector<vector<int>>& grid,int n,int m,int i,int j)
    {
        if(i<0 || i>=n || j<0 || j>=m)
        {
            return false;
        }
        if(grid[i][j]==0)
        {
            return false;
        }
        return true;
    }
    bool is_Possible(vector<vector<int>>& grid)
    {
        int x , y;
        int n = grid.size();
        int m = grid[0].size();
        bool visited[n][m];
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
```

```
        if(grid[i][j]==1)
        {
            x = i;
            y = j;
        }
        visited[i][j] = false;
    }
}
queue< pair<int,int> > q;
q.push({x,y});
visited[x][y] = true;
while(!q.empty())
{
    int xcor = q.front().first;
    int ycor = q.front().second;
    if(grid[xcor][ycor]==2)
    {
        return 1;
    }
    if(isValid(grid,n,m,xcor-1,ycor) and !visited[xcor-1][ycor])
    {
        q.push({xcor-1,ycor});
        visited[xcor-1][ycor] = true;
    }
    if(isValid(grid,n,m,xcor+1,ycor) and !visited[xcor+1][ycor])
    {
        q.push({xcor+1,ycor});
```

```
            visited[xcor+1][ycor] = true;
        }
        if(isValid(grid,n,m,xcor,ycor-1) and !visited[xcor][ycor-1])
        {
            q.push({xcor,ycor-1});
            visited[xcor][ycor-1] = true;
        }
        if(isValid(grid,n,m,xcor,ycor+1) and !visited[xcor][ycor+1])
        {
            q.push({xcor,ycor+1});
            visited[xcor][ycor+1] = true;
        }
        q.pop();
    }
    return 0;
}
```

12) Minimum Cost Path

Given a square **grid** of size **N**, each cell of which contains integer cost which represents a cost to traverse through that cell, we need to find a path from top left cell to bottom right cell by which the total cost incurred is minimum.

From the cell (i,j) we can go (i,j-1), (i, j+1), (i-1, j), (i+1, j).

Note: It is assumed that negative cost cycles do not exist in the input matrix.

Soln:- We use djikstra's algorithm to find the shortest weight path between left top and bottom right. We create a 2d matrix of distance which stores the minimum distance from 0,0 cell and initialise all values as INT_MAX means they have not yet visited even once. Now we initialise distance of 0,0 as grid[0][0] because this weight is also included in the total distance. Now we make a set of pair of pairs to store distance of node x,y from 0,0. We run a while loop while set is not empty and we first store the first value of set ie s.begin() because set stores values according to ascending numbers so first iterator in

set has node with minimum distance till now so we change further values ahead according to this value. This is what we exactly do in Djikstra's algorithm where we do the operation for lowest weight edge or least distant node till now. Now the node removed from set has already reached its minimum possible value and values still in set somewhere down are to be further updated and nodes with distance as INT_MAX are not yet processed even once. So after storing the coordinates of first element of set in temp value and removing it form set, we check for its adjacent positions. First check if they are reachable ie not out of bound then check if their current distance is greater than distance of current node(parent node) + their grid value (edge weight). If greater then it means we have a better answer for that node so we update it else we move to some other plausible adjacent cell. If the distance of adjacent cell is INT_MAX means it is not in set and has not yet processed then we insert it directly into set with its updated distance ,x,y else if processed already but we now have a better answer for this cell so we first remove its previous valued cell which is already in the set (because it is still on the verge of updation or finding a better answer) and then insert new one (updated value) . Finally return the dist[n-1][m-1] ie distance of bottom right cell from left top (0,0).

```
bool isValid(int n,int m,int i,int j)
        {
            if(i<0 or i>=n or j<0 or j>=m)
            {
                return false;
            }
            return true;
        }
    int minimumCostPath(vector<vector<int>>& grid)
    {
        int n = grid.size();
        int m = grid[0].size();
        int dist[n][m];
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
```

```
    {
        dist[i][j] = INT_MAX;
    }
}
dist[0][0] = grid[0][0];
set< pair<int , pair<int,int> > > s;
s.insert({dist[0][0],{0,0}});
while(!s.empty())
{
    auto ite = s.begin();
    s.erase(s.begin());
    int x = (*ite).second.first;
    int y = (*ite).second.second;
    //cout<<x<<" "<<y<<endl;
    if(isValid(n,m,x-1,y))
    {
        if(dist[x-1][y] > dist[x][y] + grid[x-1][y])
        {
            if(dist[x-1][y]!=INT_MAX)
            {
                s.erase(s.find({dist[x-1][y],{x-1,y}}));
            }
            dist[x-1][y] = dist[x][y] + grid[x-1][y];
            s.insert({dist[x-1][y],{x-1,y}});
        }
    }
    if(isValid(n,m,x+1,y))
```

```
{
    if(dist[x+1][y] > dist[x][y] + grid[x+1][y])
    {
        if(dist[x+1][y]!=INT_MAX)
        {
            s.erase(s.find({dist[x+1][y],{x+1,y}}));
        }
        dist[x+1][y] = dist[x][y] + grid[x+1][y];
        s.insert({dist[x+1]+[y],{x+1,y}});
    }
}
if(isValid(n,m,x,y-1))
{
    if(dist[x][y-1] > dist[x][y] + grid[x][y-1])
    {
        if(dist[x][y-1]!=INT_MAX)
        {
            s.erase(s.find({dist[x][y-1],{x,y-1}}));
        }
        dist[x][y-1] = dist[x][y] + grid[x][y-1];
        s.insert({dist[x][y-1],{x,y-1}});
    }
}
if(isValid(n,m,x,y+1))
{
    if(dist[x][y+1] > dist[x][y] + grid[x][y+1])
    {
```

```
            if(dist[x][y+1]!=INT_MAX)
            {
                s.erase(s.find({dist[x][y+1],{x,y+1}}));
            }
            dist[x][y+1] = dist[x][y] + grid[x][y+1];
            s.insert({dist[x][y+1],{x,y+1}});
          }
        }
    }
    return dist[n-1][m-1];
}
```

13) Circle of strings

Given an array of lowercase strings **A[]** of size **N**, determine if the strings can be chained together to form a circle. A

string **X** can be chained together with another string **Y** if the last character of **X** is same as first

character of **Y.** If every string of the array can be chained, it will form a circle.

For eg for the array arr[] = {"for", "geek", "rig", "kaf"} the answer will be Yes as the given strings can be chained as "for", "rig", "geek" and "kaf"

Soln:- We first make the directed graph choosing first and last character of each string. This shows the plausible connection between first and last character which is to be analysed so after connecting the characters, we check the in and out degrees of every node. Now each node's in and out degree should be equal individually because if this is violated, it means it is now able to connect to properly as many nodes from start as it is connecting from end ie amount of connection for each node should be equal from both sides so it is necessary for in and out degrees to be equal. This can be explained by:-

Consider more than 1 words starting and ending with same character so to connect them, it must be ensured that equal number of words connect from either end of each such word ie a complete circle should be formed and no word should be dangling freely due to not being connected. So this condition can only be satisfied if each character has

same no of out connections as well as in connections to ensure uniformity and circle formation. Now this condition can also be satisfied in individual strongly connected components which means not single but more than one circles can be formed but to ensure formation of single circle, all the nodes should be part of same strongly connected components so we start a dfs search from any node. If all nodes are visited during a single dfs search then it means the strings can now form a circle without any problem so if any of these conditions is violated we return 0 else we return 1 at the end.

So we make a set to store all the extreme characters to store their in and out degrees properly in separate maps named indeg and outdeg. Now after storing them, we traverse the set and check for corresponding elements in the maps if their in and out degrees are same. If any of node does not follow this, we return 0. Now we also perform a dfs search on the graph and in the dfs search we mark the visited nodes as true. So if after dfs, any node is unvisited, it means it is part of some other strongly connected component so we return 0. Finally return 1 if none of above conditions becomes true.

```
void helper(char ele,unordered_map<char,bool>&
visited,unordered_map<char,vector<char> >& m)
  {
     visited[ele] = true;
     for(auto ite:m[ele])
     {
        if(!visited[ite])
        {
           helper(ite,visited,m);
        }
     }
  }
  int isCircle(int N, vector<string> A)
  {
     unordered_set<char> s;
     unordered_map<char,vector<char> > m;
     for(int i=0;i<N;i++)
```

```
{
    m[A[i][0]].push_back(A[i][A[i].size()-1]);
    s.insert(A[i][0]);
    s.insert(A[i][A[i].size()-1]);
}
unordered_map<char,int> indeg;
unordered_map<char,int> outdeg;
for(auto ite:s)
{
    indeg.insert({ite,0});
    outdeg.insert({ite,0});
}
for(auto ite:m)
{
    for(auto x:ite.second)
    {
        outdeg[ite.first]++;
        indeg[x]++;
    }
}
for(auto ite:s)
{
    if(indeg[ite]!=outdeg[ite])
    {
        return 0;
    }
}
```

```
    unordered_map<char,bool> visited;
    helper(*s.begin(),visited,m);
    for(auto ite:visited)
    {
        if(ite.second==false)
        {
            return 0;
        }
    }
    return 1;
}
```

14) Floyd Warshall

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. The Graph is represented as an adjacency matrix, and the matrix denotes the weight of the edges (if it exists) else -1. **Do it in-place.**

Soln:- We are basically finding the distance between any two nodes by going through a intermediate node ie let's say we have to find the minimum distance between two nodes i and j, then we can go through any node from 0 to n-1 ie i -> k -> j where i and j are all possible pairs from 0 to n-1 and k denotes the intermediate node between 0 to n-1 to we write 3 loops, outermost for k and inner two for the pairs (i,j) . Now we put a check that if dist[i][j] > dist[i][k] + dist[j][k] means if the current distance between i and j is more than i to k + k to j then update it provided that both the individual values are not -1 denoting that there is not an edge between those two values. Now this is a DP solution as when we are using values such as dist[i][k] and dist[j][k] then these intermediate values are also getting calculated maybe later or earlier so we are sure to get the minimum possible answer.

```
void shortest_distance(vector<vector<int>>&matrix){
        int n = matrix.size();
        for(int k=0;k<n;k++)
        {
```

```
            for(int i=0;i<n;i++)
            {
                for(int j=0;j<n;j++)
                {
                    if(matrix[i][j] > (matrix[i][k] + matrix[k][j]) and matrix[i][k]!=-1 and
matrix[k][j]!=-1)
                    {
                        matrix[i][j] = matrix[i][k] + matrix[k][j];
                    }
                }
            }
        }
```

15) Alien Dictionary

Given a sorted dictionary of an alien language having N words and k starting alphabets of standard dictionary. Find the order of characters in the alien language.

Note: Many orders may be possible for a particular test case, thus you may return any valid order and output will be 1 if the order of string returned by the function is correct else 0 denoting incorrect string returned.

Soln:- Now to know the precedence of a character, we traverse the corresponding strings and check if their corresponding character is same. If same then we cannot tell which character has priority so if found different, we know the character in later string has less precedence that's why that string is after the previous one. So we insert an edge from former to latter. We do it for all corresponding strings. For example:- `"baa","abcd","abca","cab","cad"`

Here, b has more priority than a because in first two strings baa comes before abcd and similarly 'd' has more priority than 'a' because in 2nd and 3rd strings only 4th character is different so d comes before a. We do it for all corresponding strings and add edge for each corresponding pair in the graph. Now we do the topological sort of the graph

because it gives us the characters according to their priority in correct order. If b is more prior than d then b->d will be broken as first d then b where d comes in bottom of stack and b on top so while forming final string, b comes before d. This is how we perform topological sorting and the order of characters in the stack forms the final string.

Note:- We were putting arrow towards less prior character so in topological sort, first one to come is the one with least priority because it does not contain any outgoing edge being least prior because it comes after all characters so it does not have outgoing edge. So that's why topological sort works here. Now after least prior is removed, then second least prior is put in stack because after removal of least prior, it is the one least prior in remaining characters.

```
stack<char> s;

class Solution{

    public:

    void topological(char ele,unordered_map<char,vector<char> >&
m,unordered_map<char,bool>& visited)

    {

        visited[ele] = true;

        for(auto ite:m[ele])

        {

            if(!visited[ite])

            {

                topological(ite,m,visited);

            }

        }

        s.push(ele);

    }

    string findOrder(string dict[], int N, int K) {

        unordered_map<char,vector<char> > m;

        for(int i=1;i<N;i++)

        {
```

```
            for(int j=0;j<min(dict[i].size(),dict[i-1].size());j++)
            {
                if(dict[i][j]!=dict[i-1][j])
                {
                    m[dict[i-1][j]].push_back(dict[i][j]);
                    break;
                }
            }
        }
        unordered_map<char,bool> visited;
        for(int i=0;i<K;i++)
        {
            char ch = char(i+97);
            if(!visited[ch])
            {
                topological(ch,m,visited);
            }
        }
        string str = "";
        while(!s.empty())
        {
            str += s.top();
            s.pop();
        }
        return str;
    }
};
```

16) Snake and Ladder Problem

Given a 5x6 snakes and ladders board, find the minimum number of dice throws required to reach the destination or last cell (30th cell) from the source (1st cell). You are given **N** ie - the total number of snakes and ladders and an array **arr** of **2*N** size where **2*i** and **(2*i+1)**th values denotes the starting and ending point respecitvely of ith snake or ladder. The board looks like the following.

Soln:- We create a graph by joining all the reachable numbers from all the numbers and if we reach start of a ladder then rather connecting that parent number to the start of ladder, we connect it with the end of ladder and similarly with end of snake if we can reach there. So for each number 1 to 30, we connect them with 6 numbers ahead because 6 is the maximum value of dice so from each value we can go 6 steps ahead. Now while connecting numbers with its the destination nodes, we check if the destination numbers for parent number exists in the snake ladder array in odd position because odd positioned values in the array are start of a ladder or snake so we directly connect the destination of the snake or ladder with the parent number we are adding edge if the start of snake or ladder is reachable from the the parent number (it is 1-6 steps away from the parent number) . Now after doing it for all the numbers from 1 to 30, we do a bfs search from 1 and check in how many minimum steps, we can reach 30. Bfs works here because we move to all possible single ahead steps in each move so since bfs return the value as soon as it finds it, now since less steps require less time so we find the answer for minimum steps for sure. After connecting the nodes in the graph, we can use any property or algorithm of the graph. In BFS, we make a queue of pair type to store distance or steps till now and destination number. This is with respect to 1 ie no of steps required to reach some number from 1.

```
int minThrow(int N, int arr[]){
    map<int,vector<int> > m;
    for(int i=1;i<2*N;i+=2)
    {
        m[arr[i-1]].push_back(arr[i]);
    }
    for(int i=1;i<=29;i++)
    {
        int f = 1;
```

```
    for(int j=0;j<2*N;j+=2)
    {
        if(i==arr[j])
        {
            f = 0;
            break;
        }
    }
    if(f)
    {
        for(int j=1+i;j<=i+6;j++)
        {
            if(m[j].size()>0)
            {
                m[i].push_back(m[j][0]);
            }
            else
            {
                m[i].push_back(j);
            }
        }
    }
}
queue<pair<int,int> > q;
q.push({1,0});
unordered_map<int,bool> visited;
visited[1] = true;
```

```
        while(!q.empty())
        {
            int ele = q.front().first;
            int dist = q.front().second;
            if(ele==30)
            {
                return dist;
            }
            for(auto ite:m[ele])
            {
                if(!visited[ite])
                {
                    q.push({ite,dist+1});
                    visited[ite] = true;
                }
            }
            q.pop();
        }
        return -1;
    }
```

GREEDY

Note:- Left 7,10

1) Activity Selection

Given **N** activities with their start and finish day given in array **start[ ]** and **end[ ]**. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a given day.

Note : Duration of the activity includes both starting and ending day.

Soln:- Make pair of vector to store start and end time together and then sort it according to end time. Then traverse the pair of vector and check if start time of current task is greater than finish time of previously finished task then increment ans by 1. We update the finishTime only if we did the previous task because simply checking from previous wont make sense.

```
bool sortbysec(const pair<int,int> &a,const pair<int,int> &b)
    {
        return (a.second < b.second);
    }
class Solution
{
    public:
    //Function to find the maximum number of activities that can
    //be performed by a single person.
    int activitySelection(vector<int> start, vector<int> end, int n)
    {
        vector<pair<int,int> > v;
        for(int i=0;i<n;i++)
        {
            v.push_back(make_pair(start[i],end[i]));
        }
        sort(v.begin(),v.end(),sortbysec);
        int ans = 1;
        int finishTime = v[0].second;
        for(int i=1;i<n;i++)
```

```
        {
            if(v[i].first>finishTime)
            {
                finishTime = v[i].second;
                ans++;
            }
        }
        return ans;
    }
};
```

2) N meetings in one room

There is **one** meeting room in a firm. There are **N** meetings in the form of **(S[i], F[i])** where **S[i]** is start time of meeting **i** and **F[i]** is finish time of meeting **i.**

What is the **maximum** number of meetings that can be accommodated in the meeting room when only one meeting can be held in the meeting room at a particular time? Also note start time of one chosen meeting can't be equal to the end time of the other chosen meeting.

Soln:- It is also same as previous ques of activity selection. Here also we combine start and end time in a pair and sort them according to finish time and traverse. We keep checking that if start time is greater than previous finish time. If true then we increment the ans by 1 exactly same as earlier.

```
bool sortbysec(const pair<int,int> &a,const pair<int,int> &b)
{
    return (a.second < b.second);
}

class Solution
{
```

```
    public:
    //Function to find the maximum number of meetings that can
    //be performed in a meeting room.
    int maxMeetings(int start[], int end[], int n)
    {
        vector<pair<int,int> > v;
        for(int i=0;i<n;i++)
        {
            v.push_back(make_pair(start[i],end[i]));
        }
        sort(v.begin(),v.end(),sortbysec);
        int ans = 1;
        int finishTime = v[0].second;
        for(int i=1;i<n;i++)
        {
            if(v[i].first>finishTime)
            {
                finishTime = v[i].second;
                ans++;
            }
        }
        return ans;
    }

};
```

3) Choose and Swap

You are given a string **s** of lower case english alphabets. You can choose any two characters in the string and replace all the occurences of the first character with the second character and replace all the occurences of the second character with the first character. Your aim is to find the lexicographically smallest string that can be obtained by doing this operation at most once.

Soln:- Now the best approach is to first store that which all characters are present in the string in an array of bool type. We start traversing the string from left and for current character check what smallest character is present in the string at later index obviously because if it is present at previous index then swapping will make string bigger rather smaller so when we pass an index we simply mark its character as false because it can no longer be used for swapping being already smaller. So we find the smallest possible character for the current character. If we find one then we swap all their occurrences in original string else we move to next character in string and mark this one as false to prevent wrong results production in future.

```
string chooseandswap(string s){
    bool a[26]={};
    for(int i=0;i<s.size();i++)
    {
        a[int(s[i])-97]=true;
    }
    int replace;
    int val;
    for(int i=0;i<s.size();i++)
    {
        val = int(s[i])-97;
        replace = val;
        for(int j=0;j<26;j++)
        {
            if(val>j && a[j])
            {
                replace = j;
```

```
                break;
            }
        }
        if(replace!=val)
        {
            break;
        }
        a[val]=false;
    }
    char toBeReplaced = char(val+97);
    char replacedWith = char(replace+97);
    for(int i=0;i<s.size();i++)
    {
        if(s[i]==toBeReplaced)
        {
            s[i]=replacedWith;
        }
        else if(s[i]==replacedWith)
        {
            s[i]=toBeReplaced;
        }
    }
    return s;
}
```

4) Maximize Toys

Given an array **arr[ ]** of length **N** consisting cost of **N** toys and an integer **K** depicting the amount with you. Your task is to find maximum number of toys you can buy with **K** amount.

Soln:- Sort the array and maintain a count variable to check if you can buy the current toy. We sort it to make sure to take maximum toys as their cost is in increasing order. So we check if deduction arr[i] cost from K would still keep it >=0. If this would then we decrement K by arr[i] and increment count by 1 else we break out. Finally return count.

```
int toyCount(int N, int K, vector<int> arr)
  {
     sort(arr.begin(),arr.end());
     int count = 0;
     for(int i=0;i<N;i++)
     {
        if((K-arr[i])>=0)
        {
           K-=arr[i];
           count++;
        }
        else
        {
           break;
        }
     }
     return count;
  }
```

5) Page Faults in LRU

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needs to be replaced when the new page

comes in. Whenever a new page is referred and is not present in memory, the page fault occurs and Operating System replaces one of the existing pages with a newly needed page. Given a sequence of pages and memory capacity, your task is to find the number of page faults using Least Recently Used (LRU) Algorithm.

Input:

The first line of input contains an integer **T** denoting the number of test cases. Each test case contains n number of pages and next line contains space seaprated sequence of pages. The following line consist of the capacity of the memory.

Note: Pages are referred in the order left to right from the array (i.e index 0 page is referred first then index 1 and so on). Memory is empty at the start.

Soln:- We maintain a queue to insert the elements and a map to check if current element is present in queue ie if value of element corresponds to true so what happens it that when we encounter current element then we check if it is true ie present in queue. If not present then we have to bring it from harddisk so in any case if element is not in queue ie false, we have to increment count which stores no of page faults. So if not present we check if queue's size is equal to capacity so if full then we pop once as queue's top stores least recently used element and push the current element which is stored in rear of queue. Now if queue is not full then simply push the element to queue. Now if element is already present in queue then we just have to update its position from somewhere in middle of queue to rear because it was most recently used so we use a temp queue and push current queue's element to temp and when pushing back we only push the element back to main queue only if temp's top is not equal to current element because it is to be moved to back. So after fully pushing the temp to main queue, we also push current element to main queue to make it most recent. Finally return the count which stores number of page faults.

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

using namespace std;

int main()

 {

        int t;

        cin>>t;

        while(t--)

        {

           int n;
```

```
cin>>n;
int a[n];
for(int i=0;i<n;i++)
{
    cin>>a[i];
}
int count = 0;
int capacity;
cin>>capacity;
queue<int> q;
unordered_map<int,bool> m;
for(int i=0;i<n;i++)
{
    if(!m[a[i]])
    {
        m[a[i]]=true;
        if(q.size()==capacity)
        {
            m[q.front()]=false;
            q.pop();
        }
        q.push(a[i]);
        count++;
    }
    else
    {
        queue<int> temp;
```

```
                while(!q.empty())
                {
                    temp.push(q.front());
                    q.pop();
                }
                while(!temp.empty())
                {
                    if(a[i]!=temp.front())
                    {
                        q.push(temp.front());
                    }
                    temp.pop();
                }
                q.push(a[i]);
            }
        }
        cout<<count<<endl;
    }
    return 0;
}
```

6) Largest number possible

Given two numbers '**N**' and '**S**' , find the **largest number** that can be formed with '**N**' digits and whose sum of digits should be equals to '**S**'.

Soln:- To make maximum number, we try to append as many 9's in the beginning and last numbers as remaining sum after all 9's are used up. We iterate till N and keep appending 9's till sum S is greater than equal to 9. When it drops to less than 9 we append that remaining value and make it 0 for rest iterations. Edge cases are if Nx9 is

less than S then number is not possible so return -1 and also if S is 0 and N is more than 1 return -1.

```
string findLargest(int N, int S){
    if(N*9<S)
    {
        return "-1";
    }
    if(S==0 && N>1)
    {
        return "-1";
    }
    string str = "";
    while(N--)
    {
        if(S>=9)
        {
            str+="9";
            S-=9;
        }
        else
        {
            string x = to_string(S);
            str+=x;
            S=0;
        }
    }
    return str;
```

```
}
```

8) Minimize the sum of product

You are given two arrays, **A** and **B**, of equal size **N**. The task is to find the minimum value of **A[0] * B[0] + A[1] * B[1] +…+ A[N-1] * B[N-1],** where shuffling of elements of arrays **A** and **B** is allowed.

Soln:- Sort array ‘A’ is ascending order and ‘B’ in descending order. Now add the products of corresponding elements in the variable prod. Or sort both in ascending order and add products of first element of A and last of B , second of A and second last of B and so on which is equivalent to above operations. Finally return the product.

```
long long int minValue(int a[], int b[], int n)
{
    sort(a,a+n);
    sort(b,b+n);
    long long int prod = 0;
    for(int i=0;i<n;i++)
    {
        prod+=(a[i]*b[n-i-1]);
    }
    return prod;
}
```

9) Huffman Decoding-1

Given a string **S**, implement Huffman Encoding and Decoding.

Soln:- We initialise curr node as root and traverse the string. If we find ‘0’ we move to left because that’s how huffman encoding encodes data else there is ‘1’ we move to right. We also check if currently left and right child of current node are NULL. If they are NULL then we append node’s data to output string and initialise curr and root because we have moved and found the character and now we re traverse the tree to find next character. Encountering both NULL tells us that we have reached the character. Finally return the output string.

```
string decode_file(struct MinHeapNode* root, string s)
{
    string ans = "";
    MinHeapNode* curr = root;
    for(int i=0;i<s.size();i++)
    {
        if(s[i]=='0')
        {
            curr=curr->left;
        }
        else
        {
            curr=curr->right;
        }
        if(curr->left==NULL && curr->right==NULL)
        {
            ans+=curr->data;
            curr=root;
        }
    }
    return ans;
}
```

11) Shop in Candy Store

In a candy store, there are **N** different types of candies available and the prices of all the N different types of candies are provided to you.

You are now provided with an attractive offer.

You can buy a single candy from the store and get at most **K** other candies (all are different types) for free.

Now you have to answer two questions. Firstly, you have to find what is the **minimum amount of money** you have to spend to buy all the **N** different candies. Secondly, you have to find what is the **maximum amount of money** you have to spend to buy all the N different candies.

In both the cases you must utilize the offer i.e. you buy one candy and get **K** other candies for free.

Soln:- We first sort the candies array and now to find the minimum cost what we do is to buy cheapest thing first and take costliest K along with it so we initialise i as 0 which is traverse index and end as N-1 and run while loop such that i<=end. Inside the loop, we add candies[i] ie current candy value to the minCost and decrement end by K because those K candies have been taken now. Similarly to find maximum cost, just reverse the process ie buy most costly candy and take K cheapest along with it so initialise end as 0 now and i as N-1 and run loop till i>=end and keep adding current cost to maxCost and increment end by K in each step. Finally push minCost and maxCost to vector and return v.

```
vector<int> candyStore(int candies[], int N, int k)
  {
     sort(candies,candies+N);
     int end = N-1;
     int minCost = 0;
     int i = 0;
     while(i<=end)
     {
        minCost+=candies[i];
        end-=k;
        i++;
     }
     end = 0;
     int maxCost = 0;
     i = N-1;
     while(i>=end)
```

```
        {
            maxCost+=candies[i];
            end+=k;
            i--;
        }
        vector<int> v;
        v.push_back(minCost);
        v.push_back(maxCost);
        return v;
    }
```

12) Geek collects the balls

There are two parallel roads, each containing **N** and **M** buckets, respectively. Each bucket may contain some balls. The balls in first road are given in an array **a** and balls in the second road in an array **b**. The buckets on both roads are kept in such a way that they are sorted according to the number of balls in them. Geek starts from the end of the road which has the bucket with a lower number of balls(i.e. if buckets are sorted in increasing order, then geek will start from the left side of the road).

Geek can change the road only at a point of intersection ie- a point where buckets have the same number of balls on two roads. Help Geek collect the maximum number of balls.

Soln:- We use the concept of merging the arrays with slight modification. We basically maintain two variables to store sum of values of both arrays just before any intersection. What we do is basically we store the sum of values and take their max before moving to the intersection to make sure that we have greatest sum till this particular intersection. So we keep incrementing the indexes and keep taking sums in two variables first and second. If a[i]<b[j] , it means a[i] is less so we keep its sum in first and increment i else if a[i]>b[j], it means we add b[j] to second and increment j. We do not increment both i and j at the same time so that we do not miss any intersection in between ie we only consider inequality cases eg consider 1 4 3 6 and 1 2 4 9 so here if we increment both at same time then we miss the intersection so we increment only one of i and j. When we reach the intersection, just store maximum of first and second in a global variable

and add a[i] to it which is intersection element. Now we store intersection in temp variable in case it is repeated in a streak in a or b array because if this is the case then we have to add all of them to global and because we can actually visit all of them through intersection so we check for both a and b till the index we can find this temp in them and add it to global variable. After coming out of loop we have maximum till now ie maximum just after crossing the intersection along with intersection values. Now we make both first and second as 0 because from here we can consider that we have to work in the remaining arrays exactly as we worked in beginning as we already have maximum till now. Finally if one of them could have become empty so add element of the remaining to first or second accordingly and then update res = res +max(first,second) and return final result(res).

```
int maxBalls(int N, int M, int a[], int b[]){
    int i = 0;
    int j = 0;
    int first = 0;
    int second = 0;
    int res = 0;
    while(i<N && j<M)
    {
       if(a[i]<b[j])
       {
          first+=a[i++];
       }
       else if(a[i]>b[j])
       {
          second+=b[j++];
       }
       else
       {
          res+=(max(first,second)+a[i]);
```

```
            int temp = a[i];
            i++;
            j++;
            while(i<N && a[i]==temp)
            {
                res+=a[i++];
            }
            while(j<M && b[j]==temp)
            {
                res+=b[j++];
            }
            first = 0;
            second = 0;
        }
    }
    while(i<N)
    {
        first+=a[i++];
    }
    while(j<M)
    {
        second+=b[j++];
    }
    res+=max(first,second);
    return res;
}
```

DYNAMIC PROGRAMMING

Note:- Left 9,13

1) Minimum Operations

Given a number N. Find the minimum number of operations required to reach **N** starting from **0**. You have 2 operations available:

- Double the number
- Add one to the number

Soln:-

Bottom-Up

We make a dp array of size n+1 and initialise dp[0] as 0 and dp[1] as 1 because they are independent entities. Now we iterate from 2 to n and check for each i if is odd or even. If odd then we make it as 1 + dp[i-1] as it would take one step to become so else if is even means it could have been doubled from some parent number so we make it as the minimum of dp[i-1] and dp[i/2] + 1 because it would take 1 + minimum of the possibilities.

```
int minOperation(int n)
   {
      int dp[n+1];
      dp[0]=0;
      dp[1]=1;
      for(int i=1;i<=n;i++)
      {
         if(i%2==0)
         {
            dp[i]=1+min(dp[i-1],dp[i/2]);
         }
         else
```

```
        {
            dp[i]=1+dp[i-1];
        }
    }
    return dp[n];
}
```

Top-Down

We make a global dp array and assign 0 to dp[0] and 1 to dp[1] and assign -1 to all remaining indexes and call helper function in which we pass n,n as num and n. Num is the number to be checked that how many steps are needed to reach the number num. If dp[num]!=-1 so we return its value else we find value of variable val1 as 1+helper(num-1) ie 1+for(num-1) and store it in ans then if num is even then we can reach num/2 so if even we find another val by passing num/2 and finally we update ans as min of val1 and val2. Put it in dp[num] and return it. This is recursion approach optimised by memoization in dp

```
int minOperation(int n)
{
    int dp[n+1];
    dp[0]=0;
    dp[1]=1;
    for(int i=1;i<=n;i++)
    {
        if(i%2==0)
        {
            dp[i]=1+min(dp[i-1],dp[i/2]);
        }
        else
```

```
        {
            dp[i]=1+dp[i-1];
        }
    }
    return dp[n];
}
```

2) Max length chain

You are given N pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed in this fashion. You have to find the longest chain which can be formed from the given set of pairs.

Soln:- We first sort the pairs according to first element to make sure that while taking the chain we can move in forward direction only and not in backward direction so we sort it and then we use helper function in which we traverse from ahead of the current index we have included and check if it's first value is greater than the maxNow which was previous' second value. So while traversing if any of the first value is greater than then we compute answer as max of current ans and (1+ helper of this index) and we find max for all such indexes and store it in ans and return ans finally. We check if dp[ind+1] is -1 if yes then compute and store the ans in dp and return it. We are basically taking the current element and hence adding 1 to the chain and computing for the remaining array so when ind==n just return 0 means we reached the end of array. We started with ind as -1 so that we can start checking from 0th index since loop starts from ind+1.

```
bool compare(val v1, val v2){
        return v1.first < v2.first;
}
int dp[101];
int helper(struct val p[],int n,int ind,int maxNow)
{
   if(ind==n)
   {
```

```
        return 0;
    }
    int ans = 0;
    if(dp[ind+1]!=-1)
    {
        return dp[ind+1];
    }
    for(int i=ind+1;i<n;i++)
    {
        if(p[i].first>maxNow)
        {
            ans = max(ans,1+helper(p,n,i,p[i].second));
        }
    }
    return dp[ind+1] = ans;
}
int maxChainLen(struct val p[],int n)
{
    for(int i=0;i<=n;i++)
    {
        dp[i]=-1;
    }
    sort(p,p+n,compare);
    return helper(p,n,-1,INT_MIN);
}
```

3) Minimum number of Coins

Given an infinite supply of each denomination of Indian currency { 1, 2, 5, 10, 20, 50, 100, 200, 500, 2000 } and a target value N.

Find the minimum number of coins and/or notes needed to make the change for Rs N.

Soln:- We try to include as big notes as we can and then if we can no longer sum them than we move to next smaller note. We keep doing it till the sum is same as the target N. This is a greedy approach actually.

```
vector<int> minPartition(int N)
   {
      vector<int> v;
      int arr[] = {2000,500,200,100,50,20,10,5,2,1};
      int sum = 0;
      int i = 0;
      while(sum<=N)
      {
         if(sum==N)
         {
            break;
         }
         if(sum+arr[i]<=N)
         {
            v.push_back(arr[i]);
            sum+=arr[i];
         }
         else
         {
            i++;
         }
```

```
    }
    return v;
}
```

4) Longest Common Substring

You don't need to read input or print anything. Your task is to complete the function **longestCommonSubstr()** which takes the string S1, string S2 and their length n and m as inputs and returns the length of the longest common substring in S1 and S2.

Soln:-

Bottom Up:-

We make a 2d DP array and store 0 in first row and first col. Then we check if s1[i-1] is equal to s2[j-1]. If they are, then make dp as dp of previous + 1 and store maximum of dp else dp[i][j] is 0. Finally return 0 as it contains the max answer.

```
nt longestCommonSubstr (string s1, string s2, int n, int m)
{
    int ans = 0;
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=m;j++)
        {
            if(i==0 || j==0)
            {
                dp[i][j]=0;
                continue;
            }
            if(s1[i-1]==s2[j-1])
            {
                dp[i][j] = dp[i-1][j-1]+1;
```

```
                ans = max(ans,dp[i][j]);
            }
            else
            {
                dp[i][j]=0;
            }
        }
    }
    return ans;
}
```

6) Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. Both the strings are of uppercase.

Soln:- We use dp to do this. First we initialize a 2d dp with all values -1 and then we call recursively. Now base case is when i=x or j=y so we return dp[i][j] as 0 from here else we check if dp[i][j] is -1 if it is then we compute by checking if s1[i]=s2[j] if same then ans = max of current ans and 1+remaining_strings ie i+1,j+1 else we take the max of i+1,j string and i,j+1 string and then return dp[i][j]=ans.

```
int helper(int x, int y, string s1, string s2,int i,int j)
{
    if(i==x || j==y)
    {
        return dp[i][j] = 0;
    }
    if(dp[i][j]!=-1)
    {
        return dp[i][j];
    }
    int ans = 0;
```

```
    if(s1[i]==s2[j])
    {
        ans = max(ans,1+helper(x,y,s1,s2,i+1,j+1));
    }
    else
    {
        ans = max({ans,helper(x,y,s1,s2,i+1,j),helper(x,y,s1,s2,i,j+1)});
    }
    return dp[i][j] = ans;
}
int lcs(int x, int y, string s1, string s2)
{
    for(int i=0;i<=x;i++)
    {
        for(int j=0;j<=y;j++)
        {
            dp[i][j]=-1;
        }
    }
    return helper(x,y,s1,s2,0,0);
}
```

7) 0 - 1 Knapsack Problem

You are given weights and values of **N** items, put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. Note that we have only **one quantity of each item**.

In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent values and weights associated with **N** items respectively. Also given an integer W which

represents knapsack capacity, find out the maximum value subset of **val[]** such that sum of the weights of this subset is smaller than or equal to **W.** You cannot break an item, **either pick the complete item, or don't pick it (0-1 property)**.

Soln:- We create a 2d DP as here we have two control variables which are i and weight so in the dp we store that which all weights give max value till index i so we use 2D dp here. Now we call a recursive function where we just check if ind==n if true then return 0 else we take max of taking the current value and not taking the current value and storing it in dp table. If dp[ind][tot] is -1 then compute the ans, store in the dp table and return it.

```
int helper(int W, int wt[], int val[], int n,int ind,int tot)
  {
     if(ind==n)
     {
        return 0;
     }
     if(dp[ind][tot]!=-1)
     {
        return dp[ind][tot];
     }
     int ans = helper(W,wt,val,n,ind+1,tot);
     if(W>=tot+wt[ind])
     {
        ans = max(ans,val[ind]+helper(W,wt,val,n,ind+1,tot+wt[ind]));
     }
     return dp[ind][tot] = ans;
  }
  int knapSack(int W, int wt[], int val[], int n)
  {
     for(int i=0;i<=1000;i++)
```

```
    {
        for(int j=0;j<=1000;j++)
        {
            dp[i][j]=-1;
        }
    }
    return helper(W,wt,val,n,0,0);
}
```

8) Maximum sum increasing subsequence

Given an array **arr** of **N** positive integers, the task is to find the **maximum sum increasing subsequence** of the given array.

Soln:- We traverse from left to right and update ans as max of current ans and value we got from i. We basically find the maximum sum increasing value till index i and get the maximum of all values. In helper function we just run a loop from 0 to i to being the maximum sum till now using recursive function and take its max. To speed up using DP, we check if dp[i]!=-1 then we return its value else we find the ans and store it in dp and return it.

```
int helper(int arr[],int n,int i)
    {
        int ans = arr[i];
        if(dp[i]!=-1)
        {
            return dp[i];
        }
        for(int j=0;j<i;j++)
        {
```

```
        if(arr[j]<arr[i])
        {
            ans = max(ans,arr[i]+helper(arr,n,j));
        }
    }
    return dp[i] = ans;
}
int maxSumIS(int arr[], int n)
{
    int ans = 0;
    for(int i=0;i<=n;i++)
    {
        dp[i]=-1;
    }
    for(int i=0;i<n;i++)
    {
        int val = helper(arr,n,i);
        //cout<<val<<" ";
        ans = max(ans,val);
    }
    return ans;
}
```

10) Edit Distance

Given two strings **s** and **t.** Find the minimum number of operations that need to be performed on str1 to convert it to str2. The possible operations are:

1. Insert
2. Remove

3. Replace

Soln:- We pass 0 0 as starting indices of both strings and then we check if corresponding characters of the string are same. If yes then simply call for the i+1,j+1 else there can be 3 cases ie to remove replace and add so if we are adding then i is same, j is increasing else if we are replacing then both are increasing else finally if we are removing a character, then i is increasing and j is same so we take min of all these cases. And base case is that if i reached end of s means we have to add the remaining length characters to make it equal to ‘t’. Else if j reached end of ‘t’ means we have to remove some characters from ‘s’ to make it same as ‘t’. Finally return minimum of all 3 cases.

```
int dp[101][101];
    int helper(string s,string t,int i,int j)
    {
        if(i==s.size())
        {
            return dp[i][j] = t.size() - j;
        }
        if(j==t.size())
        {
            return dp[i][j] = s.size() - i;
        }
        if(dp[i][j]!=-1)
        {
            return dp[i][j];
        }
        int ans = INT_MAX - 10;
        if(s[i]==t[j])
        {
            ans = min(ans,helper(s,t,i+1,j+1));
        }
```

```
    else
    {
        ans = min({ans,1+helper(s,t,i,j+1),1+helper(s,t,i+1,j),1+helper(s,t,i+1,j+1)});
    }
    return dp[i][j] = ans;
}
int editDistance(string s, string t) {
    for(int i=0;i<=s.size();i++)
    {
        for(int j=0;j<=t.size();j++)
        {
            dp[i][j]=-1;
        }
    }
    return helper(s,t,0,0);
}
```

11) Coin Change

Given a value N, find the number of ways to make change for N cents, if we have infinite supply of each of S = { S1, S2, .. , SM } valued coins.

Soln:- Here we send two subproblems in the recursive call. First is that we take the current index element and add it in tot which was 0 initially and second is to skip the current element and move ahead on remaining array. Base case is if tot=n then return 1 means we got a solution, or if tot>n means total sum exceeded so return 0 finally check if we reached end of the array and still sum is less than n then also return 0.

```
long long helper(int S[],int m,int n,int index,int tot)
{
    if(tot==n)
    {
```

```
        return 1;
    }
    if(tot>n)
    {
        return 0;
    }
    if(index>=m && tot<n)
    {
        return 0;
    }
    if(dp[index][tot]!=-1)
    {
        return dp[index][tot];
    }
    long long val1 = helper(S,m,n,index+1,tot);
    long long val2 = helper(S,m,n,index,tot+S[index]);
    //cout<<val1<<" "<<val2<<endl;
    return dp[index][tot] = (val1 + val2);
}
long long int count( int S[], int m, int n )
{
    for(int i=0;i<=m;i++)
    {
        for(int j=0;j<=n;j++)
        {
            dp[i][j]=-1;
        }
```

```
    }
    return helper(S,m,n,0,0);
}
```

12) Partition Equal Subset Sum

Given an array **arr[]** of size **N**, check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

Soln:- It is similar to subset sum to target where the target is half of the totalSum since we have to divide it into two subsets such that they have equal sum. So we calculate the total sum and then make it half then check if that sum can be formed using array elements by recursion. Two calls in recursion are one is to call with including current index element in total sum or not including current element in total sum. So we return that either of them is true by returning “||” of both the operations.

```
int helper(int N,int arr[],int ind,int currSum,int tot)
{
    if(ind==N && currSum<tot)
    {
        return 0;
    }
    if(currSum==tot)
    {
        return 1;
    }
    if(currSum>tot)
    {
        return 0;
    }
    return (helper(N,arr,ind+1,currSum+arr[ind],tot) || helper(N,arr,ind+1,currSum,tot));
}
```

```
int equalPartition(int N, int arr[])
{
    int totSum = 0;
    for(int i=0;i<N;i++)
    {
        totSum += arr[i];
    }
    if(totSum%2==1)
    {
        return 0;
    }
    return helper(N,arr,0,0,totSum/2);
}
```

14) Maximize The Cut Segments

Given an integer **N** denoting the Length of a line segment. You need to cut the line segment in such a way that the cut length of a line segment each time is either **x** , **y** or **z**. Here x, y, and z are integers.

After performing all the cut operations, your **total number of cut segments must be maximum**.

Soln:- We pass 3 recursive calls each for n-x, n-y, n-z and check in the base case if n is 0 then return 0 else if n is negative return INT_MIN. We take ans as maximum of these three cases + 1 and return it.

```
int helper(int n,int x,int y,int z)
{
    if(n==0)
    {
        return 0;
    }
```

```
    if(n<0)
    {
        return INT_MIN;
    }
    if(dp[n]!=-1)
    {
        return dp[n];
    }
    int ans = 1+max({helper(n-x,x,y,z),helper(n-y,x,y,z),helper(n-z,x,y,z)});
    return dp[n] = ans;
}
int maximizeTheCuts(int n, int x, int y, int z)
{
    for(int i=0;i<=n;i++)
    {
        dp[i]=-1;
    }
    int val = helper(n,x,y,z);
    if(val<0)
    {
        return 0;
    }
    return val;
}
```

15) Minimum sum partition

Given an integer array **arr** of size **N**, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum and find the minimum difference.

Soln:- We take the minimum of two recursive calls one is that we put the current element in first set or right set so in the recursive call we pass sum1,sum2+arr[ind] and sum1+arr[ind],sum2 and make it time efficient using dp. So we make a 2d DP of ind and sum where sum is the sum of all elements of the array. And ind is n so there are two control variables ind and sum hence 2d dp.

```
int helper(int arr[],int n,int ind,int sum1,int sum2)
        {
            if(ind==n)
            {
                return abs(sum1-sum2);
            }
            if(dp[ind][sum1]!=-1)
            {
                return dp[ind][sum1];
            }
            int ans =
min(helper(arr,n,ind+1,sum1+arr[ind],sum2),helper(arr,n,ind+1,sum1,sum2+arr[ind]));
            return dp[ind][sum1] = ans;
        }
        int minDiffernce(int arr[], int n)
        {
            dp.clear();
            int sum = 0;
            for(int i=0;i<n;i++)
            {
                sum += arr[i];
```

```
        }
        for(int i=0;i<=n;i++)
        {
            vector<int> v;
            for(int j=0;j<=sum+1;j++)
            {
                v.push_back(-1);
            }
            dp.push_back(v);
        }
        return helper(arr,n,0,0,0);
    }
```

16) Count number of hops

A frog jumps either 1, 2, or 3 steps to go to the top. In how many ways can it reach the top. As the answer will be large find the answer modulo 1000000007.

Soln:- We make a dp array and by observation the current and is sum of last three values of the array so we make the dp accordingly and return dp[n].

```
long long countWays(int n)
  {
    if(n==1)
    {
        return 1;
    }
    if(n==2)
    {
        return 2;
    }
    if(n==3)
```

```
        {
            return 4;
        }
        long long dp[n+1];
        dp[0]=0;
        dp[1]=1;
        dp[2]=2;
        dp[3]=4;
        for(int i=4;i<=n;i++)
        {
            dp[i] = (dp[i-1]%1000000007 + dp[i-2]%1000000007 +
dp[i-3]%1000000007)%1000000007;
        }
        return dp[n];
    }
```

18) Egg Dropping Puzzle

Suppose you have N eggs and you want to determine from which floor in a K-floor building you can drop an egg such that it doesn't break. You have to determine the minimum number of attempts you need in order find the critical floor in the worst case while using the best strategy.There are few rules given below.

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If the egg doesn't break at a certain floor, it will not break at any floor below.
- If the eggs breaks at a certain floor, it will break at any floor above.

For more description on this problem see wiki page

Soln:- We iterate for every floor from 1 to k and find the maximum value of that particular case and minimize it for each floor. For each case we are trying to find out that from which floor should be throw the eg first to ensure minimum possible answer hence we iterate from 1 to k to find the answer for each floor and store it if it is minimum

till now. Now after throwing the egg from any floow there can be two cases. One is that egg breaks, in that case we have one less egg and we have to check for below i-1 left floors else if doesn't break then we have to check for upper k-i floors with all eggs retained since egg didn’t break. So being the worst case we take maximum of both the cases when the egg break and doesn’t break for a particular floor and take the minimum for each floor to decide where to throw is first and so on and finally return the ans. We use a 2d DP of dp[n][k] since there are two control variables n and k to decide the minimum answer for a particular floor and no of eggs. So this is a top down approach.

```
int helper(int n,int k)
  {
     if(k==0 || k==1)
     {
        return k;
     }
     if(n==1)
     {
        return k;
     }
     if(dp[n][k]!=-1)
     {
        return dp[n][k];
     }
     int ans = INT_MAX;
     for(int i=1;i<=k;i++)
     {
        int res = 1+max(helper(n,k-i),helper(n-1,i-1));
        if(ans>res)
        {
           ans = res;
```

```
        }
    }
    return dp[n][k] = ans;
}
int eggDrop(int n, int k)
{
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=k;j++)
        {
            dp[i][j]=-1;
        }
    }
    return helper(n,k);
}
```

19) Optimal Strategy For A Game

You are given an array **A of size N**. The array contains integers and is of **even length**. The elements of the array represent N **coin** of **values V1, V2, ...Vn**. You play against an opponent in an **alternating** way.

In each **turn**, a player selects either the **first or last coin** from the **row**, removes it from the row permanently, and **receives the value** of the coin.

You need to determine the **maximum possible amount of money** you can win if you **go first**.

Note: Both the players are playing optimally.

Soln:- Now there can be 2 cases for each of two case. There can be two cases that we pick tha element from left or right so let's say we pick element from left side so now second player would pick element from remaining array such that we have minimum possible sum from remaining array so we first take minimum of both cases that other player picks left or right after we have picked left one. Now a similar case can be when

we pick from right side then also other player can also pick from remaining from either left or right so we take minimum of these and finally we add the element we picked to the recursive both cases and take the maximum of both the cases. Now here we are taking minimum because the other player is also playing optimally so by taking min we are ensuring that he picks element such that we would have least sum possible from the remaining array so we make him play optimally by taking min of whether he picks from left or right. And maximum comes into play because we are taking maximum of our choices ie from where can we get the maximum sum possible out of left and right index of the array.

```
int dp[1001][1001];

long long helper(int arr[],int n,int i,int j)

{

    if(i>=j)

    {

        return 0;

    }

    if(dp[i][j]!=-1)

    {

        return dp[i][j];

    }

    long long ans  =
max(arr[i]+min(helper(arr,n,i+2,j),helper(arr,n,i+1,j-1)),arr[j]+min(helper(arr,n,i,j-2),helper(
arr,n,i+1,j-1)));

    return dp[i][j] = ans;

}

long long maximumAmount(int arr[], int n)

{

    for(int i=0;i<=n;i++)

    {

        for(int j=0;j<=n;j++)
```

```
        {
            dp[i][j]=-1;
        }
    }
    return helper(arr,n,0,n-1);
}
```

20) Shortest Common Supersequence

Given two strings X and Y of lengths m and n respectively, find the length of the smallest string which has both, X and Y as its sub-sequences.

Note: X and Y can have both uppercase and lowercase letters.

Soln:- This is much similar to LCS, here find the length of lcs and return total sum length of both strings - length of lcs because we are removing the common characters from them which are already in the LCS. So we return (m + n - length_of(lcs)).

```
int lcs(char* X, char* Y, int m, int n,int i,int j)
    {
        if(i==m || j==n)
        {
            return 0;
        }
        if(dp[i][j]!=-1)
        {
            return dp[i][j];
        }
        int ans;
        if(X[i]==Y[j])
        {
            ans = 1+lcs(X,Y,m,n,i+1,j+1);
```

```
    }
    else
    {
        ans = max(lcs(X,Y,m,n,i+1,j),lcs(X,Y,m,n,i,j+1));
    }
    return dp[i][j] = ans;
}
int shortestCommonSupersequence(char* X, char* Y, int m, int n)
{
    for(int i=0;i<=m;i++)
    {
        for(int j=0;j<=n;j++)
        {
            dp[i][j]=-1;
        }
    }
    return m+n-lcs(X,Y,m,n,0,0);
}
```

DIVIDE AND CONQUER

Note:- Left 4,7

1) Find the element that appears once in sorted array

Given a sorted array arr[] of size N. Find the element that appears only once in the array. All other elements appear exactly twice.

Soln:- Since the elements are sorted so it means that if not interrupted in between by the single element, the current element should be equal to either its next or previous element if it appears twice. Now we use divide and conquer and we find mid index as mean of low and high which are initially 0 and n-1. Now there can be two possibilities that mid can be odd or even. If odd it means this element should be equal to its previous element considering indexes not interrupted in between by single element. If equal, we call recursively to check for mid+1 to high index else it means that the single element lies in left side so we call for low and mid. We do not use mid-1 since current element can also be single so we cannot take mid-1. Similarly if mid is even then its right element should be equal to current ie arr[mid]=arr[mid+1]. If true it means till now order was correct and we call for right subarray ie mid+2 and high else we call for low and mid as earlier in case of odd case. Now if low is equal to high, it means we have reached the element so we return it from here.

```
int ele;
    void search(int arr[],int low,int high)
    {
        if(low>high)
        {
            return;
        }
        if(low==high)
        {
            ele = arr[low];
            return;
        }
        int mid = (low+high)/2;
        if(mid%2==1)
        {
            if(arr[mid]==arr[mid-1])
            {
                search(arr,mid+1,high);
```

```
        }
        else
        {
            search(arr,low,mid);
        }
    }
    else
    {
        if(arr[mid]==arr[mid+1])
        {
            search(arr,mid+2,high);
        }
        else
        {
            search(arr,low,mid);
        }
    }

}
int findOnce(int arr[], int n)
{
    search(arr,0,n-1);
    return ele;
}
```

2) Search in a Rotated Array

Given a sorted and rotated array A of N distinct elements which is rotated at some point, and given an element key. The task is to find the index of the given element key in the array A.

Soln:- In the search function we check if low is greater than high then return -1 which means the element is not present. Next we find mid as (l+h)/2. Now check if key is equal to arr[mid] if yes return mid. Now since the array is rotated then either subarray from (l to mid) is sorted else (mid to h) is sorted. So we check if arr[l]<=arr[mid]. If true, it means left side is sorted so we further check if key is present in between them if yes then call search function recursively with low as low and high as mid-1 else if not present then call for right subarray with low as mid+1 and high as high only. Now if this subarray is not sorted then right one is sorted so we check if key exists between mid and high. If yes then call search recursively with low as mid+1 and high as high else if not present then search in left subarray with low as low and high as mid-1.

```
int search(int arr[], int l, int h, int key){
    if(l>h)
    {
        return -1;
    }
    int mid = (l+h)/2;
    if(arr[mid]==key)
    {
        return mid;
    }
    if(arr[l]<=arr[mid])
    {
        if(key>=arr[l] && key<=arr[mid])
        {
            return search(arr,l,mid-1,key);
        }
        else
        {
```

```
            return search(arr,mid+1,h,key);
        }
    }
    else
    {
        if(key>=arr[mid] && key<=arr[h])
        {
            return search(arr,mid+1,h,key);
        }
        else
        {
            return search(arr,l,mid-1,key);
        }
    }
}
```

3) Binary Search

Given a sorted array of size N and an integer K, find the position at which K is present in the array using binary search.

Soln:- Initialise low as 0 and high as n-1. Run while loop till low<=high and find mid at each step as (low+high)/2. Now check if key is equal to mid element ie arr[mid]. If yes return mid else check if key is less than arr[mid]. If less ,then element is present in left side else on right side so if less than make high as mid-1 else low as mid+1.

```
int binarysearch(int arr[], int n, int k){
    int l = 0;
    int h = n-1;
    while(l<=h)
    {
```

```
        int mid = (l+h)/2;
        if(k==arr[mid])
        {
            return mid;
        }
        else if(k<arr[mid])
        {
            h = mid-1;
        }
        else
        {
            l = mid+1;
        }
    }
    return -1;
}
```

5) Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

Given an array arr[], its starting position low and its ending position high.

Implement the partition() and quickSort() functions to sort the array.

Soln:- The basics concept of quicksort is to find a pivot element and place it at its correct position in final sorted array so here we find a pivot element by-

We call partition function where we mark our pivot element as last element of array and run a loop to swap the elements with jth index element if they are less than pivot. Now after coming out of the loop, we swap (j+1)th with pivot element ie (high) and return (j+1) which is the final position of pivot element ie j+1 and this now brings it at the correct position in sorted array ie all elements to its left are smaller than it and all

elements to its right are greater than it. So after getting pivot index we call recursively for left and right halves. Left half is from low to pivot-1 and right is from pivot+1 to high.

We swap the element from j+1 because initially j was low-1 so we swap by its one greater.

```
void quickSort(int arr[], int low, int high)
  {
     if(low<high)
     {
        int p = partition(arr,low,high);
        quickSort(arr,low,p-1);
        quickSort(arr,p+1,high);
     }
  }


  public:
  int partition (int arr[], int low, int high)
  {
     int pivot = arr[high];
     int j = low-1;
     for(int i=low;i<high;i++)
     {
        if(arr[i]<pivot)
        {
           j++;
           swap(arr[i],arr[j]);
        }
     }
     swap(arr[j+1],arr[high]);
```

```
    return (j+1);
}
```

6) Merge Sort

Given an array arr[], its starting position l and its ending position r. Sort the array using merge sort algorithm.

Soln:- We first find a mid element as means of l and r and then call mergeSort on left and right halves. For left we call on l and m <-> for right we call on m+1 and r. And then all merge function which takes arr and l , m and r. Now these elements are sorted in this array so we are basically merging sorted array so we put left half in one of the arrays and right half in another array. And merge them ie putting the merged in original array. This mergeSort is called recursively and its base case is hit when there is single element so we return from there.

```
void merge(int arr[], int l, int m, int r)
{
    int n1 = m-l+1;
    int n2 = r-m;
    int a[n1];
    int b[n1];
    for(int i=0;i<n1;i++)
    {
        a[i]=arr[l+i];
    }
    for(int i=0;i<n2;i++)
    {
        b[i]=arr[m+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++];
        }
        else
        {
            arr[k++]=b[j++];
        }
    }
    while(i<n1)
    {
        arr[k++]=a[i++];
    }
    while(j<n2)
    {
        arr[k++]=b[j++];
    }
}
public:
void mergeSort(int arr[], int l, int r)
{
    if(l>=r)
    {
        return;
```

```
    }
    int mid = (l+r)/2;
    mergeSort(arr,l,mid);
    mergeSort(arr,mid+1,r);
    merge(arr,l,mid,r);
  }
```

BACKTRACKING

1) N-Queen Problem

The n-queens puzzle is the problem of placing **n** queens on a (**n×n)** chessboard such that no two queens can attack each other.

Given an integer n, find all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens' placement, where the solutions are a permutation of [1,2,3..n] in increasing order, here the number in the *ith* place denotes that the *ith*-column queen is placed in the row with that number. For eg below figure represents a chessboard [3 1 4 2].

Soln:- We first initialize a matrix of size 20 as safe choice and then traverse for each row, each column if it is safe. Now we are denoting presence of queen by 1 in the matrix so we check if there is any 1 horizontally, vertically or diagonally corresponding to current cell. If there is then we return 0 meaning it is not safe ie we cannot put 1 there else we return 1. So now if the cell is safe we mark it 1 and find the value for next row using recursion. Now if value is true then we return 1 meaning the matrix marking is true till now else if false meaning we could not find a position for 1 in next row so we mark previously marked cell 1 as 0 and further check for remaining value. We basically start from the values we left on earlier ie when we marked previous value 1 it was wrong so we continue checking for next columns. If we could not find a suitable cell then we return 0 meaning we have to backtrack from previous marked cell. Now when we reach row=n , we traverse the array and put the indexes of cells with in a vector which is then pushed to global vector and we return 0 so that we can backtrack to previous result and check for any other possible combination as if we return 1 then we break out of the function but we want all possible queen positions so that's why we returned 0 to continue the traversal further.

```
vector<vector<int> > v;
    bool isSafe(int a[][20],int row,int col,int n)
    {
        for(int i=0;i<n;i++)
        {
            if(a[i][col]==1)
            {
                return 0;
            }
        }
        for(int i=0;i<n;i++)
        {
            if(a[row][i]==1)
            {
                return 0;
            }
        }
        int i=row;
        int j=col;
        while(i>=0 && j<n)
        {
            if(a[i][j]==1)
            {
                return 0;
            }
            i--;
            j++;
```

```
}
i = row;
j = col;
while(i<n && j<n)
{
    if(a[i][j]==1)
    {
        return 0;
    }
    i++;
    j++;
}
i = row;
j = col;
while(i<n && j>=0)
{
    if(a[i][j]==1)
    {
        return 0;
    }
    i++;
    j--;
}
i = row;
j = col;
while(i>=0 && j>=0)
{
```

```
        if(a[i][j]==1)
        {
            return 0;
        }
        i--;
        j--;
    }
    return 1;
}
int helper(int a[][20],int row,int n)
{
    if(row==n)
    {
        vector<int> temp;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(a[i][j]==1)
                {
                    temp.push_back(j+1);
                }
            }
        }
        v.push_back(temp);
        return 0;
    }
```

```
    for(int i=0;i<n;i++)
    {
        if(isSafe(a,row,i,n))
        {
            a[row][i]=1;
            int p = helper(a,row+1,n);
            if(p)
            {
                return 1;
            }
            a[row][i]=0;
        }
    }
    return 0;
}
vector<vector<int>> nQueen(int n) {
    v.clear();
    int a[20][20]={0};
    helper(a,0,n);
    return v;
}
```

2) Solve the Sudoku

Given an incomplete Sudoku configuration in terms of a 9 x 9 2-D square matrix (grid[][]), the task to find a solved Sudoku. For simplicity, you may assume that there will be only one unique solution.

Soln:- We do it much similar to n queen problem where we check for each box each number from 1 to 9 if it can come. In the helper function we check if row=N means we

reached end of the matrix so we return true. If col=N means we reached the end of row so we call helper recursively for next row and col 0. Now if current cell dont have 0 means it is already filled so we call helper for next cell ie row,col+1. Else if it is 0 then we iterate from 1 to 9 so check which number can come in the current cell. To check this we call the function isSafe which check if the number n can come in cell- row,col. Here we check if there is already the same number in grid vertically and horizontally and also in the same mini box. If the number is there we cant fill this number so we increment it and call the same isSafe and so on until we find a suitable number. If we found means if isSafe is true then put number n in grid[row][col] and then call for next column and store the value true or false. If true means next one is also true so we return true else check for next value. If no one is suitable then we return false finally to backtrack last call ie to reset last put element in the grid and change it so that later columns can be filled suitably. So here return false puts back to previous call and tries to change the then put element else if true it goes on traversing until we encounter false. So if p is false we reset the cell to 0 and continue traversing from the point we left while putting it then so that we can try for next elements and if no element is suitable then we move to previous cell and try to change its value. This goes on until the sudoku's number satisfy the value and finally we return true. When we get false so either we change the number or move to previous cell then to further previous cells to get desired combination. When the combination satisfies the value we return true finally.

```
bool isSafe(int grid[9][9],int row,int col,int num)
  {
     for(int i=0;i<9;i++)
     {
        if(grid[i][col]==num)
        {
           return false;
        }
     }
     for(int i=0;i<9;i++)
     {
        if(grid[row][i]==num)
        {
```

```
            return false;
        }
    }
    int temp = 3;
    int x = (row/temp)*temp;
    int y = (col/temp)*temp;
    for(int i=x;i<x+temp;i++)
    {
        for(int j=y;j<y+temp;j++)
        {
            if(grid[i][j]==num)
            {
                return false;
            }
        }
    }
    return true;
}
bool helper(int grid[9][9],int row,int col)
{
    if(row==9)
    {
        return true;
    }
    if(col==9)
    {
        return helper(grid,row+1,0);
```

```
    }
    if(grid[row][col]!=0)
    {
        return helper(grid,row,col+1);
    }
    for(int i=1;i<=9;i++)
    {
        if(isSafe(grid,row,col,i))
        {
            grid[row][col]=i;
            bool p = helper(grid,row,col+1);
            if(p)
            {
                return true;
            }
            grid[row][col]=0;
        }
    }
    return false;
}
bool SolveSudoku(int grid[9][9])
{
    return helper(grid,0,0);
}
void printGrid (int grid[9][9])
{
    for(int i=0;i<9;i++)
```

```
    {
        for(int j=0;j<9;j++)
        {
            cout<<grid[i][j]<<" ";
        }
    }
}
```

3) Rat in a Maze Problem - I

Consider a rat placed at **(0, 0)** in a square matrix of order **N * N**. It has to reach the destination at **(N - 1, N - 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **'U'(up)**, **'D'(down)**, **'L' (left)**, **'R' (right)**. Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.

Note: In a path, no cell can be visited more than one time.

Soln:- We make a global vector to store the strings and and a global strings to store the steps involved in reaching the maze. We use an array to store the path by marking the cells as 1. So we call helper function which checks for each cell if it is reachable or not so first check is that current cell's row and cell should be inside the range else we return false if row and cell are greater than equal to n or less than 0. So next check is that current cell is not 0 ie it is reachable so if 0 we return false also if output array is 1 means we have reached the cell already so we cannot reach it again so we return false. Else if we have reached row and cell as n-1,n-1 means we have reached the end of grid so we push_back the output string in vector and return false to backtrack and find other possible paths also. Now if all these do not work then means current cell is reachable so we mark output array as 1 and now we consider 4 possible directions, down left right up so before each of these calls we append corresponding D, L, R, U and then check if they are true then return true else backtrack and reset a[row][col] as 0 to backtrack. Then check for next values and finally if none is true then return false and reset the current cell as 0 only and pop_back string in each step after each of down, right, left and up to backtrack the previous calls.

Note by arhant sir:- Jab bhi kuch append kro ya output m add kro to backtrack krne ke liye utni bar reset kro ans ko jitni baar set kia h kyuki this is how backtracking works ki

jitna set kia h utna reset krte jao after each call to make sure consistency. So overall jitni baar set kro exactly utni baar reset kro to sahi kaam krega.

```
vector<string> v;
    string str="";
    int a[6][6];
    bool helper(vector<vector<int>> &m, int n,int row,int col)
    {
        if(row>=n || col>=n || row<0 || col<0)
        {
            return false;
        }
        if(m[row][col]==0 || a[row][col]==1)
        {
            return false;
        }
        if(row==n-1 && col==n-1)
        {
            v.push_back(str);
            return false;
        }
        a[row][col]=1;
        str+="D";
        int down = helper(m,n,row+1,col);
        if(down)
        {
            return true;
        }
```

```
    str.pop_back();
    str+="L";
    int left = helper(m,n,row,col-1);
    if(left)
    {
        return true;
    }
    str.pop_back();
    str+="R";
    int right = helper(m,n,row,col+1);
    if(right)
    {
        return true;
    }
    str.pop_back();
    str+="U";
    int up = helper(m,n,row-1,col);
    if(up)
    {
        return true;
    }
    str.pop_back();
    a[row][col]=0;
    return false;
}
vector<string> findPath(vector<vector<int>> &m, int n) {
    for(int i=0;i<n;i++)
```

```
    {
        for(int j=0;j<n;j++)
        {
            a[i][j]=0;
        }
    }
    v.clear();
    helper(m,n,0,0);
    return v;
}
```

`4) Word Boggle

Given a dictionary of distinct words and an M x N board where every cell has one character. Find all possible words from the dictionary that can be formed by a sequence of adjacent characters on the board. We can move to any of 8 adjacent characters, but a word should not have multiple instances of the same cell.

Soln:- We check for each word in the dictionary independently if it exists on the board. If the word exists, then we push it in the vector. We check for each possible starting index one the board because the single function works only for the connected indices. So for single set of connected indices we use the helper function to determine if the word exists on the board. We use base case which are that if the string formed till now is equal to the word we are checking so if it is then return true. If the cell's row and col go out of bound return false. If current cell's character is not equal to the character at current index of the word or if the current cell has already been visited return false. We are checking if we have visited the cell by maintaining an array a which stores 1 at the cell if we have found suitable character at the present cell. Now if everything went fine till this point then we append current index character of word to output string and call recursively for next all adjacent indices. If none of them could take the iteration forward then we pop the just appended character from the string and return false to backtrack to previous call. Now when we move to previous call, we continue checking from that point if we can find suitable character to be appended to the string. If this helper function's result is true then we add the word to v vector else not. It is also similar to previous backtracking solutions where we push to string and pop back same no of times to make sure we backtrack correctly.

```
vector<string> v;
    string str="";
    int a[60][60];
    bool helper(vector<vector<char> >& board,int row,int col,string word,int index)
    {
        if(str==word)
        {
            return true;
        }
        if(row==board.size() || row==-1 || col==board[0].size() || col==-1 ||
index==word.size())
        {
            return false;
        }
        if(board[row][col]!=word[index] || a[row][col]==1)
        {
            return false;
        }
        str+=word[index];
        a[row][col]=1;
        int leftTop = helper(board,row-1,col-1,word,index+1);
        if(leftTop)
        {
            return true;
        }
        int midTop = helper(board,row-1,col,word,index+1);
        if(midTop)
```

```
{
    return true;
}
int rightTop = helper(board,row-1,col+1,word,index+1);
if(rightTop)
{
    return true;
}
int left = helper(board,row,col-1,word,index+1);
if(left)
{
    return true;
}
int leftBottom = helper(board,row+1,col-1,word,index+1);
if(leftBottom)
{
    return true;
}
int midBottom = helper(board,row+1,col,word,index+1);
if(midBottom)
{
    return true;
}
int rightBottom = helper(board,row+1,col+1,word,index+1);
if(rightBottom)
{
    return true;
```

```
        }
        int right = helper(board,row,col+1,word,index+1);
        if(right)
        {
            return true;
        }
        str.pop_back();
        a[row][col]=0;
        return false;
    }
        vector<string> wordBoggle(vector<vector<char> >& board, vector<string>&
dictionary) {
            v.clear();
            for(int i=0;i<dictionary.size();i++)
            {
                for(int i=0;i<60;i++)
                {
                    for(int j=0;j<60;j++)
                    {
                        a[i][j]=0;
                    }
                }
                str="";
                int f=0;
                for(int j=0;j<board.size();j++)
                {
                    for(int k=0;k<board[0].size();k++)
```

```
            {
                if(helper(board,j,k,dictionary[i],0))
        {
            f=1;
            v.push_back(dictionary[i]);
            break;
        }
            }
            if(f)
            {
                break;
            }
        }
    }
    return v;
}
```

5) Generate IP Addresses

Given a string **S** containing only digits, Your task is to complete the function **genIp()** which returns a vector containing all possible combination of valid IPv4 ip address and takes only a string **S** as its only argument.

Soln:- We take a global string and a vector. Global string stores the possible string till now so we first append first character of input string because in any case it is bound to be there. Next we call the helper function and here we first check the base cases. We also maintain a variable noDots which count the no of dots in the output string. According to first base case, we check if no of dots exceeds 3 return false, next if output string's size is input string + 3 then it should be complete and if it does not have no of dots as 3 then return false finally if string is complete and has 3 dots in it so we check if it is valid. To check valid we check if each substring separated by the dot is between 0 and 255 and also a substring starting with 0 and having more than 1 character is invalid as eg 011 or 002 does not make sense. So if it is valid then push it to output vector.

After checking all these conditions, we now come to important part. Now we are at a particular index of input string so we have two choices here. One is two append current character to the output string or first add '.' and then the current character so for first case we append the current character to output string and call recursively for next index if it true then fine else we pop_back from the string to backtrack last call. Now we append a dot and current character so here we increment noDots by 1 and call recursively for next index if true then fine else we pop_back string two times since we appended two characters and decrement noDots by 1 and return false to backtrack meaning none of the possibilities worked well. So we backtrack the last call and check for them and so on until we find suitable string. When we find suitable string we still return false to find more valid strings as we have done in previous backtracking questions. Here also jitne changes kiye hamne states mei unhe wapas same to same revert krdia taki backtrack kr saken as Arhant sir said.

```
vector<string> v;
    string str="";
    int noDots = 0;
    bool ifStartZero(string temp)
    {
        if(temp.size()==1 && temp[0]=='0')
        {
            return false;
        }
        if(temp.size()>1 && temp[0]=='0')
        {
            return true;
        }
        return false;
    }
    bool ifValid(string x)
    {
        string temp="";
```

```
for(int i=0;i<x.size();i++)
{
    if(x[i]=='.')
    {
        if(temp.size()>3 || ifStartZero(temp))
        {
            return false;
        }
        int num = stoi(temp);
        if(num>255)
        {
            return false;
        }
        temp="";
    }
    else
    {
        temp+=x[i];
    }
}
if(temp.size()>3 || ifStartZero(temp))
{
    return false;
}
int num = stoi(temp);
if(num>255)
{
```

```
        return false;
    }
    return true;
}
bool helper(string s,int index)
{
    if(noDots>3)
    {
        return false;
    }
    if(str.size()==s.size()+3 && noDots!=3)
    {
        return false;
    }
    if(str.size()==s.size()+3 && noDots==3)
    {
        if(ifValid(str))
        {
            v.push_back(str);
        }
        //v.push_back(str);
        return false;
    }
    if(index==s.size())
    {
        return false;
    }
```

```
    str+=s[index];
    int val1 = helper(s,index+1);
    if(val1)
    {
        return true;
    }
    str.pop_back();
    str+=".";
    str+=s[index];
    noDots++;
    int val2 = helper(s,index+1);
    if(val2)
    {
        return true;
    }
    str.pop_back();
    str.pop_back();
    noDots--;
    return false;
}
vector<string> genIp(string &s) {
    v.clear();
    str+=s[0];
    helper(s,1);
    sort(v.begin(),v.end());
    return v;
}
```

BIT MANIPULATION

NOTE:- Left 8, 14

1) Find first set bit

Given an integer an **N.** The task is to return the position of **first set bit found from the right side** in the binary representation of the number.

Note: If there is no set bit in the integer N, then return 0 from the function.

Soln:- Initialise count variable as 1 and run loop till n is true. Each time check if n&1 is true. If it is return count else increment count by 1 and at the end return 0.

```
unsigned int getFirstSetBit(int n){
    int count=1;
    while(n)
    {
        if(n&1)
        {
            return count;
        }
        count++;
        n=n>>1;
    }
    return 0;
}
```

2) Rightmost different bit

Given two numbers **M** and **N**. The task is to find the position of the **rightmost different** bit in the binary representation of numbers.

Soln:- Just store rightmost bits of m and n both and check if both are same. If they are, then count++ else return count as they became different. And rightshift both m and n each time in the iteration. Finally return -1 if they are all same.

```
int posOfRightMostDiffBit(int m, int n)
   {
      int count = 1;
      while(m || n)
      {
         int mRight = m&1;
         int nRight = n&1;
         if(mRight!=nRight)
         {
            return count;
         }
         m=m>>1;
         n=n>>1;
         count++;
      }
      return -1;
   }
};
```

3) Check whether K-th bit is set or not

Given a number **N** and a bit number **K**, check if **Kth** bit of N is **set or not**. A bit is called set if it is 1. Position of set bit '1' should be **indexed starting with 0** from **LSB** side in binary representation of the number.

Soln:- Just right shift n k times and then check if right-most bit is 1 or not. If it is then return true else return false.

```
bool checkKthBit(int n, int k){
    while(k--)
    {
        n>>=1;
    }
    if(n&1)
    {
        return true;
    }
    return false;
}
```

4) Toggle bits given range

Given a non-negative number **N** and two values **L** and **R**. The problem is to toggle the bits in the range L to R in the binary representation of N, i.e, to toggle bits from the rightmost Lth bit to the rightmost Rth bit. A toggle operation flips a bit 0 to 1 and a bit 1 to 0. Print N after the bits are toggled.

Soln:- We put all bits of N is a vector and toggle them while iterating from L to R. Then we form the final sum using the altered bits in the vector.

```
int toggleBits(int N , int L , int R) {
    vector<int> v;
    while(N)
    {
        v.push_back(N&1);
        N>>=1;
    }
    //reverse(v.begin(),v.end());
```

```
    for(int i=L-1;i<R;i++)
    {
        v[i]^=1;
    }
    int count=0;
    for(int i=0;i<v.size();i++)
    {
        if(v[i])
        {
            count+=pow(2,i);
        }
    }
    return count;
}
```

5) Set kth bit

Given a number **N** and a value **K**. From the right, set the Kth bit in the binary representation of N. The position of Least Significant Bit(or last bit) is 0, the second last bit is 1 and so on.

Soln:- Form a new number by right shifting N by checking if current index we are on is equal to k. If it is not equal then simply do normal formation else if it is equal to k then we strictly add it irrespective of 0 or 1 as it has to be set.

```
int setKthBit(int N, int K)
{
    int i=0;
    int number=0;
    while(N)
    {
        if(i!=K)
```

```
        {
            if(N&1)
            {
                number+=pow(2,i);
            }
        }
        else
        {
            number+=pow(2,i);
        }
        N>>=1;
        i++;
    }
    return number;
}
```

6) Power of 2

Given a positive integer **N**. The task is to check if N is a power of **2**. More formally, check if **N** can be expressed as **2x** for some **x.**

Soln:- We check if any of intermediate bit in N before left most is 1. If it is then we return false else true. If number is 0 then also we return false.

```
bool isPowerofTwo(long long n){
    if(n==0)
    {
        return false;
    }
    while(n)
    {
```

```
        if(n!=1)
        {
            if(n&1)
            {
                return false;
            }
        }
        n>>=1;
    }
    return true;
}
```

7) Bit Difference

You are given two numbers **A** and **B**. The task is to **count the number of bits needed to be flipped** to **convert** A to B.

Soln:- We count the no of bits when xor of current bits of a and b are different. As if the xor is 1 it means bits are different else bits are same. Finally we return count of the different occurrences.

```
int countBitsFlip(int a, int b){
    int count=0;
    while(a || b)
    {
        if((a&1)^(b&1))
        {
            count++;
        }
        a>>=1;
        b>>=1;
```

```
    }
    return count;
}
```

9) Swap all odd and even bits

Given an unsigned integer **N**. The task is to swap all odd bits with even bits. For example, if the given number is 23 (**0**0**0**1**0**1**1**1), it should be converted to 43(0**0**1**0**1**0**1**1**). Here, every even position bit is swapped with adjacent bit on the right side(even position bits are highlighted in the binary representation of 23), and every odd position bit is swapped with an adjacent on the left side.

Soln:- We store last and second last bits in two variables and add pow(2,i)*second then pow(2,i)*first to make sure to take second bit first and first next. Then we right shift n by 2 and this loop runs till n is true.Finally we return the number.

```
unsigned int swapBits(unsigned int n)
{
    unsigned int number = 0;
    int i=0;
    while(n)
    {
        int first = n&1;
        int second = (n>>1)&1;
        number+=(pow(2,i++)*second);
        number+=(pow(2,i++)*first);
        n>>=2;
    }
    return number;
}
```

10) Count total set bits

You are given a number **N**. Find the **total count of set bits** for all numbers from 1 to N(both inclusive).

Soln:- We observe a pattern in the repeating bits to count the no of bits at a particular place. Like in left most place, the pattern is alternate 0 and 1, in second last left the pattern is two 0's and two 1's. Which repeats in this order of 4 8 16 and so on so we first increment n by 1 and calculate the ans for left by directly taking the half of n. Then we use powerOf2 to find no of bits for a number by using this variable and a variable totalPairs which is used to find if any bit is remaining to find being the number odd.

For more clarification visit:-

https://www.geeksforgeeks.org/count-total-set-bits-in-all-numbers-from-1-to-n-set-2/

```
int countSetBits(int n)
   {
      n++;
      int powerOf2 = 2;
      int cnt = n / 2;
      while (powerOf2 <= n) {
         int totalPairs = n / powerOf2;
         cnt += (totalPairs / 2) * powerOf2;
         cnt += (totalPairs & 1) ? (n % powerOf2) : 0;
         powerOf2 <<= 1;
      }
      return cnt;
   }
```

11) Longest Consecutive 1's

Given a number **N.** Find the length of the longest consecutive 1s in its binary representation.

Soln:- We repetitively perform right shift operation and count the number of set bits in continuation. If rightmost bit is not set then we take maxCount as max of maxCount and original count and finally return maxCount.

```
int maxConsecutiveOnes(int N)
  {
     int count=0;
     int maxCount = 0;
     while(N)
     {
        if(N&1)
        {
           count++;
        }
        else
        {
           maxCount = max(maxCount,count);
           count=0;
        }
        N>>=1;
     }
     maxCount = max(maxCount,count);
     return maxCount;
  }
```

12) Number is sparse or not

Given a number **N**. The task is to check whether it is **sparse or not**. A number is said to be a sparse number if **no two or more consecutive bits are set** in the binary representation.

Soln:- For each bit we check if there is a consecutive setbit by maintaining a count and right shifting the number by 1 so if count reaches 2 then we return false else we return true at the end.

```
bool isSparse(int n){
    int count = 0;
    while(n)
    {
        if(n&1)
        {
            count++;
        }
        else
        {
            count=0;
        }
        if(count==2)
        {
            return false;
        }
        n>>=1;
    }
    return true;
}
```

13) Party of Couples

In a party of N people, each person is denoted by an integer. Couples are represented by the same number. Find out the only singe person in the party of couples.

Soln:- We take xor of all numbers in the array and return it because it is the number that is single because others get cancelled out due to xor.

```
int findSingle(int N, int arr[]){
        int x = 0;
        for(int i=0;i<N;i++)
        {
            x^=arr[i];
        }
        return x;
    }
```