June 17 SDE Interview Questions - Solutions. Taken from GFG and Leetcode.

1. Find duplicate in an array of 1 to N numbers only.

a. Brute Force: For each element check whether another element exist or not.
b. Using extra space : Use map or set.
c. Without extra space: O(nlogn): Sort the array, check i and i+1.
d. O(N) solution: For each i, mark a[i] as -1, if the same element occurs again a[j] will be already -1.

```cpp
class Solution {
public:
    int findDuplicate(vector<int >& nums) {
        int i=0,n=nums.size();
        while(i<n)
        {
            if (nums[i] == -1)
               return i;
            int cur = i;

            i = nums[i] ;

            nums[cur] = -1;
        }
        return 0;
    }
};
```

Simply revisiting the similar index does the trick.

e. From above point, we can assume that there is a cycle in the linked list. We can then use hare and tortoise technique to detect the entry point of the cycle. Approach is still O(N) but doesn't modify the array.

1a. What if more than one element is duplicate ?
[4,3,2,7,8,2,3,1]

a. Instead of using -1, mark each element as its negative value. If you find that the number is being revisited then add it to the answer. To check it, the value is already -ve when you visit it.
Start with i = 0

0	1	2	3	4	5	6	7
4	3	2	7	8	2	3	1
4	3	2	**-7**	8	2	3	1
4	***3***	**-2**	-7	8	2	3	1
4	**-3**	***-2***	-7	8	2	3	1
4	-3	-2	***-7***	8	2	**-3**	1
4	-3	-2	-7	***8***	2	-3	**-1**
4	**-3**	-2	-7	8	***2***	-3	-1

So 3 is duplicate

4	-3	**-2**	-7	8	2	***-3***	-1

So 2 is suplicate

```
class Solution {
public:
    vector<int> findDuplicates(vector<int>& nums) {
        vector<int> ans;
        for(int i=0,n=nums.size();i<n;i++)
        {
            int num = abs(nums[i]) -1 ;
            if(nums[num]<0){
                ans.push_back(num+1);
            }
            nums[num]*=-1;
        }


     return ans;
    }
};
```

2. Sort an Array of 0’s , 1’s and 2’s

a. Using count sort will do the trick in O(N) but will cost extra space.
b. Count 0 and 1 to get exact positions of where 1 and 2 must start. Now if we encounter 0 in 1 & 2’s region, swap them until that position is not claimed by a 1 or 2 whatever. Do the same for 1 and 2 too. It is a two pass approach.

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int i=0,r,b,w,wc=0,bc=0,n = nums.size(),temp;

        for(i=0;i<n;i++)
        {
            if(nums[i] == 0)++wc;
            else if(nums[i] == 1) ++bc;
        }

        bc+=wc;
        r=0;
        w=wc;
        b = bc;

        for(i=0;i<n;)
        {
            if(nums[i] == 0 && i>=wc)
            {
                temp = nums[i];
                nums[i] = nums[r];
                nums[r] = temp;
                ++r;
            }
            else if(nums[i] ==1 && (i<wc || i>=bc))
            {
                temp = nums[i];
                nums[i] = nums[w];
                nums[w] = temp;
                ++w;
            }
            else if(nums[i] == 2 && i<bc)
            {
                temp = nums[i];
                nums[i] = nums[b];
                nums[b] = temp;
                ++b;
            }
            else{
                i++;
            }
        }

    }
};
```

c. Let zero = 0 and two = N-1.
Now swap each 0 with zero++ if i>zero
and swap each 2 with two-- if i<two. This is a single pass approach.

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
       int n = nums.size(),second=n-1, zero=0;
            for (int i=0; i<=second; i++) {
                while (nums[i]==2 && i<second) swap(nums[i],
nums[second--]);
                while (nums[i]==0 && i>zero) swap(nums[i],
nums[zero++]);
            }
    }
};
```

3. Find the missing Number.
Given an array with numbers 0 to N, find the missing number from the array.

a. Expected sum = n*(n+1)/2 , lets say sum = S
Missing number is n*(n+1)/2 -S

```
// Find missing number
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int n=nums.size(),sumshouldbe = n*(n+1)/2,sum=0;
        for(int i=0;i<n;i++)sum+=nums[i];
        return sumshouldbe - sum;

    }
};
```

b. Initialize X = 0,
for each i, X ^= (i^arr[i])   ; ^ = XOR
At last X^=N
each number will appear even number of time,so effect will be 0 except for the missing number. Hence X is the answer.

c. If array is sorted Use Binary search O(log N)

4. Merge two sorted arrays arr1 and arr2 given that arr1 has enough space left to store arr2.

a. Use merge step of merge sort. Costs Extra O(N) space.
b. To avoid using extra space, simply start from the last ie n+m-1th position and use merge function. It can be proved that arr1 will not be overwritten before that element is pushed in the array at some other position. Let's say N'th element is being overwritten, that means all M elements are already pushed. Hence it won't get touched. It may happen that all elements of arr1 are greater than arr2 . In that case the elements of arr2 will remain. Otherwise arr1 will be sorted in place in the other case.

```
// merge the sorted arrays
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int
n) {
          int i=m-1;
          int j=n-1;
          int k = m+n-1;
          while(i >=0 && j>=0)
          {
               if(nums1[i] > nums2[j])
                    nums1[k--] = nums1[i--];
               else
                    nums1[k--] = nums2[j--];
          }
          while(j>=0)
               nums1[k--] = nums2[j--];
    }


};
```

5. Max sum subarray with negative elements. Find subarray with atleast one element.

a. Use Kadane’s algorithm. Make sure to initialise maxsofar = INT_MIN. That way we will consider the negative elements too. Otherwise if all the elements are positive or 0, initialize it as 0.

```
// MAX subarray
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int i,n=nums.size(), sum=0,maxsum=INT_MIN;
        for(i=0;i<n;++i)
        {
            sum+=nums[i];
            if(sum>maxsum){
                maxsum=sum;
            }
            if(sum<0)sum=0;
        }
        return maxsum;
    }
};
```

6. Merge the overlapping intervals.
Given N intervals in the form (Li , Ri), merge those which are overlapping with each other.

a. Simply sort them using left index. And whatever keeps on connecting, skip that, and restart the same where a disconnected interval is found.

Start with {L1, _}
Keep skipping while Rk >= Lk+1.. Keep on updating R to the max available.
Add {L0, Rk } to the answer; make another {Lk+1,_} and repeat the same.
At last add {Lm, Rn}

```
// Merge overlapping intervals
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> res;
        int i,n=intervals.size();
        if(n==0)return res;
        sort(intervals.begin(),intervals.end());
        vector<int> overlap;
        overlap.push_back(intervals[0][0]);
        int r = intervals[0][1];
        for(i=1;i<n;i++)
        {
            if(intervals[i][0]<=r)
            {
                r = max(r,intervals[i][1]);
            }else{
                overlap.push_back(r);
                res.push_back(overlap);
                overlap.clear();
                overlap.push_back(intervals[i][0]);
                r = intervals[i][1];
            }

        }
        overlap.push_back(r);
        res.push_back(overlap);
        return res;
    }
};
```

7. Set matrix Zeroes
Given a matrix of all 0's and 1's. Set all the rows and colums =0 Where a 0 is present in any cell
eg

```
5 10 15      5  0  15
1  0  3      0  0  0
11 2  1      11 0  1
```

a. Use two sets to store the rows and cells seperately where there is a 0 in those cells. After that update them in place afterwards. It will take O(m+n) extra space.
b. Instead of using extra sets, mark the first row and column as 0 where a cell contains zero. Start from first row and second column, and do this process. We start from second column to avoid confusion of : who updated (0,0), the row or the column? Note that if there is a 0 in first row, matrix[0][0] is updated but not for first column. So check the column seperately.
Mark the internal (M-1)x(N-1) matrix first then, check the cell [0][0] to mark the first row. Atlast if first column contained 0 , mark it also.This approach does everything inplace with O(1) space.

```
// set matrix zero.
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int i=0,j=0,n=matrix.size(),m=matrix[0].size();
        bool c=false;
        for(i=0;i<n;i++)
        {
            if(matrix[i][0] == 0)
            {
                c=true;
            }
            for(j=1;j<m;j++)
            {
                if(!matrix[i][j])
                {
                    matrix[i][0] = matrix[0][j] = 0;
                }
            }
        }

        for(i=1;i<n;i++)
        {
            for(j=1;j<m;j++)
            {
                if(matrix[i][0] == 0 || matrix[0][j]==0){
                    matrix[i][j] = 0;
                }
            }
        }

        if(matrix[0][0]==0)
        {
            for(j=0;j<m;j++)
                matrix[0][j] = 0;
        }

        if(c){
            for(i=0;i<n;i++)
            {
                matrix[i][0]=0;
            }
        }

    }
};
```

8. Generate the pascal’s triangle.
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...

Simply create a triangle of only 1’s
The pattern can be observed easily.
For each row,starting from 3rd ie 2nd index, each element is sum of its upper and diagonally left element. At last a 1 is present which is not a matter of concern.

```
// Pascal Triangle
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        l = []
        for i in range(numRows):
            l.append([1]*(i+1))
        for i in range(2,numRows):
            for j in range(1,len(l[i-1])):
                l[i][j] = l[i-1][j-1] + l[i-1][j]
        return l
```

9. Next permutation.
Given an array of numbers or a string, return the next permutaion in sorted sequence order... if no such permutaion exist, return the sorted array.
ie 1 3 2 => 2 1 3
and “abcd” => “abdc”

a. Use c++ stl next_permutation(arr.begin(),arr.end()) (Not preferred)
b. Use proper algorithm,
    i. From right to left, Find the first element that isn’t in descending order.
    ii. Swap it with the closest larger element in its right.
    iii. Sort the rest array from that index (excluding itself.)

Eg
    aebdca
First element not in descending order is ‘b’
    ae**b**dca
Nearest larger is ‘c’
    ae**b**d**c**a
replace
    ae**c**dba
sort
    aec**abd**

```
// Next Permutation
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int i=0,n=nums.size(),k=-1;
        for(i=n-2;i>=0;i--)
        {
            if(nums[i]<nums[i+1]){
                k = i;
                break;
            }
        }
        if(k==-1)
        {
            sort(nums.begin(),nums.end());
        }
        else{
            int j,diff=INT_MAX;
            for(i=n-1;i>k;i--)
            {
                if(nums[i]>nums[k] and nums[i]-nums[k]<diff)
                {
                    diff = nums[i] - nums[k];
                    j = i;
                }
            }
            swap(nums[k],nums[j]);
            sort(nums.begin()+k+1,nums.end());
        }


    }
};
```

10. Inversion Count using merge sort.
Given an array, inversion count is the number of elements such that for i<j , A[i]>A[j].

a. In the merge step, there are two arrays, L and R. Both are sorted, For each element in L such that L[i] > R[j] , it means that all elements from L[i] to last index of L are greater that R[j] , so inversion count can be increased with mid – i + 1 where mid is the last index of L.

L = 2 **5** 6   R = 1 **3** 4 means 5 and 6 both are greater than 3.

```
// Inversion Count using merge sort

class Solution {
public:
    int globalinversion=0;
    void merge(vector<int>& A , int l , int mid , int r)
    {
        vector<int> temp(r-l+1);
        int i=l,j=mid+1,k=0;
        while(i<=mid and j<=r){
            if(A[i]<=A[j]){
                temp[k++] = A[i++];
            }else{
                temp[k++] = A[j++];
                globalinversion += mid-i+1;
            }
        }

        while(i<=mid)temp[k++]=A[i++];
        while(j<=r)temp[k++]=A[j++];
        for(k=0;k<r-l+1;k++){
            A[l+k] = temp[k];
        }
    }
    void mergeSort(vector<int>& A , int l , int r)
    {
        if (l==r)return;
        int mid = (l+r)/2;
        mergeSort(A,l,mid);
        mergeSort(A,mid+1,r);
        merge(A,l,mid,r);

    }
    bool isIdealPermutation(vector<int>& A) {
        int localinversion=0,i,n= A.size();
        for(i=0;i<n-1;i++)if(A[i]>A[i+1])localinversion++;
        mergeSort(A,0,n-1);
        return (localinversion == globalinversion);
    }
};
```

10a. Lets say array contains the elements between 0 to N-1 only.
 a. Merge sort : NlogN
b. BIT : NlogN , but won't work if A[i] is too large. Since A[i]<=N-1, we can use it here.

10b. For a permutation of {0,1,...N-1} lets define:
local inversion : A[i] > A[i+1] for each i
global inversion : A[i] > A[j] if i<j.
Note that local inversion is also a global inversion but reverse is not true.
Check whether local == global or not.
a. Use the previous approaches to count both and check if they are equal or not.
b. For each A[i]>A[i+1], simply swap them and check whether the array is now sorted or not. If it is sorted now, global = local.
c. Extending to last approach, if the elements A[i] at are at A[i]th A[i+1]th or A[i-1]th position, they can be sorted using previous apprach, so simply check for | A[i] – i|>1 for any element means that it can't be sorted and hence local ≠ global.

```
// Inversion Count for Array [0,1...N-1]
class Solution {
public:

    bool isIdealPermutation(vector<int>& A) {
        for(int i=0,n=A.size();i<n;i++)
            if(A[i]!=i && A[i]!=i+1 && A[i]!=i-1)return false; //
Can also use condition => [abs(A[i]-i)>1]
        return true;


    }
};
```

11. Stock Buy and sell:
Given a list of integers, ith element is the price of the stock on ith day. You can buy a stock once and sell it once, Find the maximum possible profit.

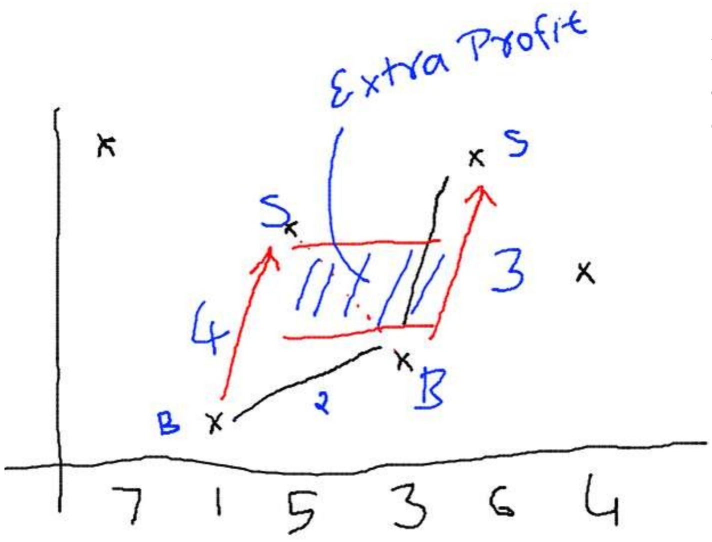
a. Idea is to find two numbers A[i] & A[j] ; i<j, with A[j] – A[i] having the maximum value.
From left to right keep an eye on the minimum element, because A[j] – A[i] is maximum if A[i] is minimum; A[j] is maximum. For each price, check the value of A[j] – minsofar, and whatever gives the max at last, print that.

```
// Buy and Sell stock
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int i,n=prices.size(),minsofar=INT_MAX,ans=0;
        for(i=0;i<n;i++)
        {
            if(prices[i]<minsofar){
                minsofar = prices[i];
            }
            ans = max(ans,prices[i] - minsofar);
        }
        return ans;

    }
};
```

11b. What if we are allowed to buy and sell multiple times?

a. We can use the same logic as before but consider the example below:
[7 1 5 3 6 4]
instead of taking 6 – 1 = 5, We will take (5 – 1) + (6 – 3) = 7.
Idea is simple, breaking the range like this will give us more profit because it is a straight slope,from 1 to 6. If we add a hinderence or a valley the difference will either remain the same or it will increase.
Eg [1 4 6] gives (4-1) + (6-4) = 5
But [1 4 3 6] gives (4-1) + (6 – 3) = 6
So find the contiguous increasing subarray , add the difference of endpoints in the answer.

Don’t forget to handle the last index seperately.



If we observe the graph carefully, we find that answer is nothing but ΣA[i] – A[i-1] (>0) ie only positive differences.

```
// Buy and Sell stock with multiple transactions
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int i,n=prices.size(),ans=0,start = 0;
        for(i=1;i<n;i++)
        {
            if(prices[i]<prices[i-1])
            {
                ans += prices[i-1] - prices[start];
                start = i;
            }
            if(i==n-1)
            {
                ans+=max(0,prices[i] - prices[start]);
            }
        }
        return ans;

    }
};
```

12. Rotate matrix.
Given an NxN matrix, rotate the matrix 90 degree.

a. take transpose, (swap upper and lower triangle)
b . To rotate 90 degree, Take mirror image about mid column. To rotate -90 degree take mirror image about mid row.

Why this works:
consider:

```
1 2 3      7 4 1
4 5 6 =>  8 5 2
7 8 9      9 6 3
```

We need to convert the row into columns and columns into rows for rotating, and that is the definition of transpose.

```
// Rotate NxN matrix by 90 degree.
class Solution {
public:
    // I already know the solution. We just need to
    //transpose the matrix and find the mirror image
    // about the mid Y axis
    void rotate(vector<vector<int>>& matrix) {
        int i=0,j=0,n=matrix.size();
        // Taking transpose of this matrix
        // ie swap the upper and lower triangle
        for(i=1;i<n;++i)
        {
            for(j=0;j<i;++j)
            {
                swap(matrix[i][j],matrix[j][i]);
            }
        }

        // Now simply take the mirror image
        // ie go upto mid, swap first,last & second,
        // second last.....upto mid

        for(i=0;i<n;i++){
            for(j=0;j<n/2;++j)
            {
                swap(matrix[i][j],matrix[i][n-j-1]);
            }
        }
    }
};
```

13. Excel sheet column number
Given a string s, find the serial number of this string in lexcographically orders dictionary of [A-Z] from ‘A’ to ‘ZZZZZZZZZ...Z’

a. Consider a number 2136. It is at 2136th position in number list. How? It is formed by skipping $2*10^3$ numbers, then $1*10^2$ numbers , $3*10^1$ numbers then 6 numbers.
Similarly in this case, A = 1, B = 2 , .. Z = 26. So
“AAB” will be formed by skipping $1*26^2 + 1*26^1 + 2$ strings.

```
// Excel Column number
class Solution {
public:
    // So basicaly we need to find the lexicographic index
    // of the given string in [A-Z] alphabet
    int titleToNumber(string s) {
        //Observe this:
        // 1 = 1, 2 = 2 , .. 100 = 100, 248 = 248
        // How do we get here?
        // 248 = 2*10^2 + 4*10^1 + 8*10^0
        // What is 10 here? Its the base.
        // For [A-Z], base is 26
        // Hence replace 10 by 26 in above logic
        int ans=0;
        for(int i=0,n=s.length();i<n;++i)
        {
            ans += pow(26,n-i-1)*(s[i]-64);
        }
        return ans;

    }
};
```

14. Find pow(X, N) in logN time. N can be negative or zero.

a. Brute force with O(N)
b. Use recursion, pow(X,N) = $[pow(X,N/2)]^2$; if N is even
= X * $[pow(X,N/2)]^2$; if N is odd

```
// pow x^n in log(n) time
class Solution {
public:
    double myPow(double x, long long n) {
        if(n==0)return 1.0;
        if(n<0)return 1.0/myPow(x,-n);
        if(n==1){
            return x;
        }
        if(n&1){
            double xx = myPow(x,n/2);
            return xx*xx*x;
        }else{
            double xx = myPow(x,n/2);
            return xx*xx;
        }
    }
};
```

15. Count trailing zeroes in N!.

a. Find N! And then count the zeroes.
b. Find the count of 2 and 5 for each number from 1 to N that divides each number. Doing it for each number will take O(N*sqrt(N)) time. So only calculate Σ N/pow(2,i) and Σ N/pow(5,i)

```
// Count trailing zeroes, a naive approach
class Solution {
public:
    int trailingZeroes(int n) {
        int count2=0,count5=0,i;
        int p = 1;
        while(true){
            double x = pow(2,p);
            if(x<=n){
            count2 += n/pow(2,p);
            p++;
            }else break;
        }
```

```
            p=1;
            while(true){
                double x = pow(5,p);
                if(x<=n){
                count5 += n/pow(5,p);
                p++;
                }else break;
            }
           return min(count2,count5);
        }
};
```

c. Or only count Σ N/pow(5,i) because count of 2 will be always smaller.

```
// Count Trailing zeroes
class Solution {
public:
    int trailingZeroes(int n) {
        if(n==0)return 0;
        return n/5 + trailingZeroes(n/5);
    }
};
```

16. GCD in logN time:
Use Euclid algorithm which solves a*x + b*y = g where g = GCD of a and b.
Extended Euclid also calculates the values of x and y.
It takes O(log(min(a,b))) time.

```
// Euclid and Extended Euclid Algo for reference;
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b,a%b);
}


int gcd(int a, int b, int& x, int& y) {    // extended
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int gcd = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return gcd;
}
```

17. Unique Paths in a grid : Given an MxN grid, Find the number of uniques paths from TopLeft to BottomRight where only Right and Down moves are allowed.

a. We need to take exactly N- 1 moves to Right and M-1 moves to down. Consider any permutation of RRR..DDD.., we will reach to the BottomRight. If we count the number of such permutaions, we get the answer.
The answer will ne (M+N-2)! / (M-1)! / (N-1)!
We can’t calculate this value for large matrix (even if M or N is 100)

```
// Grid Unique Paths
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        return factorial(m+n-2)//factorial(m-1)//factorial(n-1)
```

b. Another way is to use DP, count the number of ways to reach a cell[i][j] = number of ways to reach cell[i-1][j] + number of ways to reach cell [i][j-1]

0	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35

```
int countWays(int m , int n)
{
     int arr[m][n] , i , j;
     for(i=0;i<m;i++)
     {
          for(j=0;i<n;j++)
          {
               if(i==0 or j==0)arr[i][j] = 1;
               else arr[i][j] = arr[i-1][j] + arr[i][j-1];
          }
     }
     return arr[m-1][n-1];
}
```

18. Puzzles Link :https://www.geeksforgeeks.org/puzzles/

19. Two Sum
Given an array of integers, return indices of the two numbers such that they add up to a specific target.You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].

a. Idea is to use hashing. Use map or set to store each element. For arr[i] check whether target – arr[i] is already present in the map/set or not. Cost: O(N) time and O(N) space

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unorderd_map<int,int> mp;
        vector<int> ans;
        for(int i=0;i<nums.size();++i){

            if(mp.find(target-nums[i])!=mp.end() && mp[target-
nums[i]]!=i){
                ans.push_back(i);
                ans.push_back(mp[target-nums[i]]);
                return ans;
            }
            mp[nums[i]]=i;
        }

        return ans;

    }
};
```

19a. What if the array is sorted already.
a. It turns out that any number can be written as sum of two numbers i and j where i<=j. So simply traverse the array using two pointers i and j, i at the start , j at the end. We don't need extra space this way.

If arr[i] + arr[j] > target , --j;
else if arr[i] + arr[j] < target , ++i;
else return {i,j}

Consider these examples:

[1, 12, 12, 14, 14, 20]   26

[5, 10, 10, 11, 11, 14, 21, 23, 24, 25]   48

[5, 5, 6, 12, 15, 15, 19, 21]   27

```
pair<int,int> twosum(vector<int> arr , int target){
        int i=0,j=arr.size();
        while(i<j){
            if(arr[i]+arr[j]<target)++i;
            else if(arr[i]+arr[j]>target)--j;
            else return make_pair(i,j);
        }
    }
```

20. 4 Sum

Given an array nums of n integers and an integer target, are there elements a, b, c, and d in nums such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.
The solution set must not contain duplicate quadruplets.
Example:
Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.
A solution set is:

```
[
  [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]
]
```

20a. Before jumping to the solution, let's look at 3 sum problem.
The problem states that you have to print all the unique triplets that sum up to zero.

a. A naive solution is to fix any one number, and find a pair that sums up to the negative of that number.
Eg [1,2,-3,5,-7].
If we fix 1, we have 2,-3 hence [1,2,-3]
If we fix 2, we have 5,-7 hence [2,5,-7] and so on....

Now fixing the number is one thing but finding other two is simply 2 Sum problem. Here our target is negative of the fixed number. To rectify the duplicates we can use hashmap.
We can sort the array beforehand to have O(N) time to find the pairs. Overall the complexity is O(N^2)

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& a) {
        vector<vector<int>> ans;
        set<vector<int>> st;
        int i,j,k,n=a.size();
        sort(a.begin(),a.end());
        for(i=0;i<n;i++)
        {
            int target = -a[i];
            j= i+1;
            k = n-1;
            while(j<k)
            {
                if(a[j]+a[k]>target)k--;
                else if (a[j]+a[k]<target)j++;
                else{
                    st.insert({a[i],a[j],a[k]});
                    //break;
                    k--;
                    j++;
                }
            }

        }
        for(auto i:st)ans.push_back(i);
        return ans;
    }
};
```

b. We can modify the above code, we won't use set this time because it is taking extra space and time because afterall, we are calculating all the duplicates.
The idea is , **skip all the duplicate elements** after we find a valid triplet, because they will keep on adding.

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& a) {
        vector<vector<int>> ans;
        int i,j,k,n=a.size();
        sort(a.begin(),a.end());
        for(i=0;i<n;   )
        {
            int target = -a[i];
            j= i+1;
            k = n-1;

            while(j<k)
            {
                if(a[j]+a[k]>target)k--;
                else if (a[j]+a[k]<target)j++;
                else{
                    ans.push_back({a[i],a[j],a[k]});
                    while(j<k && a[j] == ans[ans.size()-1][1])j++;
                    while(j<k && a[k] == ans[ans.size()-1][2])k--;
                }
            }
            i++;
            while(i<n && a[i]==a[i-1])i++;

        }
        // for(auto i:st)ans.push_back(i);
        return ans;
    }
};
```

This solution doesn’t take extra space and works in O(N^2) .

Now coming back to our 4 sum problem, we can use the similar approach. Fix 2 numbers, localtarget = Target – fixed1 – fixed 2, and apply the approach used for 2 sum problem.

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& a, int tar) {
        vector<vector<int>> ans;
        int i,j,k,l,n=a.size();
        sort(a.begin(),a.end());
        for(l=0;l<n;){
            for(i=l+1;i<n;)
            {
                int target = tar - a[i] - a[l];
                j= i+1;
                k = n-1;

                while(j<k)
                {
                    if(a[j]+a[k]>target)k--;
                    else if (a[j]+a[k]<target)j++;
                    else{
                        ans.push_back({a[l],a[i],a[j],a[k]});
                        while(j<k && a[j] == ans[ans.size()-1]
[2])j++;

                        while(j<k && a[k] == ans[ans.size()-1]
[3])k--;
                    }
                }
                i++;
                while(i<n && a[i]==a[i-1])i++;

            }
            l++;
            while(l<n && a[l]==a[l-1])l++;
        }

        return ans;
    }
};
```

b. We can also use recursion ie if k == 2 use 2 sum otherwise fix the current number and use an call the function for k-1. But using this approach doesn’t reduce space or time because it can take upto O(N) space.

21. Longest Consecutive subsequence: Find the length of Longest Consecutive Sequence which is defined as a[i]+1 = a[i+1] for all valid i;
[1,2,3,5,6,7,8,9,11,12] = 5
[1,1,2,3,4,4,5,7] = 5

a. Sort the array and count the length. Note that if we find duplicates, skip them.

```
class Solution {
public:
    int longestConsecutive(vector<int>& a) {
        int i=0,n= a.size(),ans=0,mx=1;
        if(n==0)return 0;
        sort(a.begin(),a.end());

        for(int i=0;i<n-1;i++){
            if(a[i]+1==a[i+1])
            {
                mx++;
            }
            else if (a[i]!=a[i+1]){
                ans = max(ans,mx);
                mx=1;
            }
        }
        ans = max(ans,mx);
        return ans;
    }
};
```

b. Use a hash map to store the array. For each element x, count the size upto the point where x+1 is present in the array. After that if this chain is having greater length, make it the current answer.
In c++, set, map etc work in O(logn) so it won’t make any difference. But we can always make our own hashmap.

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        s = set(nums);
        ans=0
        for i in s:
            if i-1 not in s:
                j = i+1
                while j in s:
                    j+=1
                ans = max(ans , j-i)
        return ans
```

22. Longest subarray with sum 0.

Input: arr[] = {15, -2, 2, -8, 1, 7, 10, 23};
Output: 5
Explanation: The longest sub-array with

elements summing up-to 0 is {-2, 2, -8, 1, 7}

Input: arr[] = {1, 2, 3}
Output: 0
Explanation:There is no subarray with 0 sum

Input: arr[] = {1, 0, 3}
Output: 1
Explanation: The longest sub-array with
elements summing up-to 0 is {0}

a. Use brute force to find all ranges (i,j). This will take O(N^2) time.
b. A better approach is:
Observe the prefix sum of : [15 -2 2 -8 1 7 10 23]
[15 13 15 7 8 15 25 48]

here we get a repeating sum; Why? Because the subarray between these two points is having 0 sum.
Start calculating the prefix sum.
Lets store each sum in a map of sum->index. Whenever we encounter a sum that is already present, that will mean that there is a subarray with 0 sum between this index and the index of previous sum. Hence update our answer as max(ans, differnece b/w indecies).

```
def longestZeroSum(l):
    if len(l)==0:return 0
    n = len(l)
    d = dict([(l[0],0)])
    ans = 0
    for i in range(1,n):
        l[i]+=l[i-1]
        if l[i] in d:
            ans = max(ans , i - d[l[i]] )
        else:
            d[l[i]] = i

    return ans

print(longestZeroSum([1, 0, 3]))
```

22a. Subarray Sum Equals K
Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.
Example 1:
Input:nums = [1,1,1], k = 2
Output: 2   ie [1,1] and [1,1]
Input: nums = [0,0,0,0],k=0
Output: 10   ie all subarrays
a. Idea is as simple as above approach. Just like we were searching for sum = 0 ; we can also check whether there is any summation in the map that is equal to current_sum – k. That will mean that the subarray that lies between those two points has a sum = k.
We can use this logic to check whether there exist such subarray or not. Now we need to count that how many such subarrays are present in the entire array.

Answer is simple, whenever there is sum = current_sum – k, we can say that all those sums contribute in the answer. Hence We just need to keep an eye on the overall frequencies of all the sums, and whenever we encounter the above condition, add the frequency in our answer.

Lets take an example:

INPUT: [1,1,-1,4,2,-2]   k = 1

prefix sum = [1,2,1,5,7,5]
frequency_count = {0:1}
Note that there is key-value pair of 0->1 that is the empty subarray.
Now traverse the prefix_sum,
i=0; sum=1, check for 1 – 1 = 0, ans+=1; fc = {0:1,1:1}
i=1, sum = 2 check for 2 – 1 = 1, ans +=1; fc = {0:1,1:1,2:1}
i=2, sum = 1 , check for 1 – 1 = 0, ans+=1; fc = {0:1,1:2,2:1}
......
i=5, sum = 5,check for 5-1 = 4, ans+=0; fc = {.....}
Hence the answer is 3.

```python
class Solution:
    def subarraySum(self, l: List[int], k: int) -> int:
        if len(l)==0:return 0
        if len(l) == 1: return 1 if k == l[0] else 0
        n = len(l)
        d = Counter([0,l[0]])
        ans = 0
        if k == l[0]: ans+=1
        for i in range(1,n):
            l[i]+=l[i-1]
            if l[i] - k in d:
                ans += d[l[i]-k]
            d[l[i]]+=1

        return ans
```

23. Count number of subarrays with given XOR.
a. Take two pointers i and j, generate all possible combinations => all subarrays and count the ones having the given XOR. This solution is inefficient iff N^2 is too large. Complexity $O(N^2)$.

b. Just like the solution of 22a, we can take prefix XOR, and count the subarrays in the similar way.

Why it works?

Suppose the array is [a,b,c,d,e,f], and we are at 4th position for now,
If a^b^c^d^e == e^k, we will count this subarray in our answer,
Moreover, if any of the prefix XOR is having XOR = k let’s say a^b = e^k and b^c^d = e^k, then prefix XOR upto now is a^b^c^d^e, lets replace them by k, a^e^k^e, and this value is e^k
ie a^k = k^e ie from b upto 3rd postion, the subarray is having XOR = k.

A proper algorithm from GFG:
i. Initialize ans as 0.
ii. Compute xorArr, the prefix xor-sum array.

iii. Create a map mp in which we store count of
   all prefixes with XOR as a particular value.
iv. Traverse xorArr and for each element in xorArr
   (A) If m^xorArr[i] XOR exists in map, then
       there is another previous prefix with
       same XOR, i.e., there is a subarray ending
       at i with XOR equal to m. We add count of
       all such subarrays to result.
   (B) If xorArr[i] is equal to m, increment ans by 1.
   (C) Increment count of elements having XOR-sum
       xorArr[i] in map by 1.
v. Return ans.

```
class Solution:
    def subarrayXORCount(self, l: List[int], k: int) -> int:
        if len(l) == 0: return 0
        if len(l) == 1: return 1 if k == l[0] else 0
        n = len(l)
        d = Counter([0,l[0]])  # We already have XOR =0 and l[0]
        ans = 0
        if k == l[0]: ans+=1
        for i in range(1,n):
            l[i]^=l[i-1]
            if l[i] ^ k in d:
                ans += d[l[i]^k]
            d[l[i]]+=1
            if l[i] == k: ans+=1

        return ans
```

23a. XOR Queries of a Subarray
Given the array arr of positive integers and the array queries where queries[i] = [Li, Ri], for each query i compute the XOR of elements from Li to Ri (that is, arr[Li] xor arr[Li+1] xor ... xor arr[Ri]). Return an array containing the result for the given queries.

a. I simply used segment tree here...

```
class Solution {
public:
    int segtree[120000];
    void build(vector<int>& arr , int i,int l , int r){
        if(l==r)
        {
            segtree[i] = arr[l];
            return ;
        }
        int mid = (l+r)/2;
        build(arr,2*i+1,l,mid);
        build(arr,2*i+2,mid+1,r);
        segtree[i] = segtree[2*i+1]^segtree[2*i+2];
    }

    int query(int i,int l,int r,int left,int right)
    {
        if(l>right || r<left)return 0;

        if(l>=left && r<=right)return segtree[i];

        int mid = (l+r)/2;

        int ql = query(2*i+1,l,mid,left,right);
        int qr = query(2*i+2,mid+1,r,left,right);
        return ql^qr;
    }
    vector<int> xorQueries(vector<int>& arr, vector<vector<int>>&
queries) {
        int l,r,n=arr.size();
        vector<int> res;

        build(arr,0,0,n-1);
        // for(int i=0;i<15;i++)cout<<segtree[i]<<" ";cout<<endl;
        for(auto i:queries){
            // cout<<i[0]<<" "<<i[1]<<endl;
            res.push_back(query(0,0,n-1,i[0],i[1]));
        }
        return res;
    }
};
```

b. The above solution works well if we have to update the array. But there is nothing like that, so simply using a prefix array will also do the trick.

```
class Solution {
public:

    vector<int> xorQueries(vector<int>& arr, vector<vector<int>>&
queries) {
        int i,j=-1,n=arr.size();
        vector<int> res(queries.size());
        for(i=1;i<n;i++)arr[i]^=arr[i-1];
        for(auto &i: queries){
            res[++j] = i[0]>0?arr[i[0]-1]^arr[i[1]]:arr[i[1]];
        }
        return res;
    }
};
```

24. Longest Substring Without Repeating Characters
Given a string, find the length of the longest substring without repeating characters.
Example 1:
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
Example 2:

Input: "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
Example 3:
Input: "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.

a. This problem is easy.
Make a hashmap to keep an eye on frequency of characters. If it is guarenteed that characters are english alphabet, use a small array else use an array of size 256, initialised to 0.
Iterate from left to right, for each character increase its frequency.
If a character has already frequency>0, put the start pointer to the next to it and decrease the frequencies of all the in between characters.
Update the answer and maxsofar accordingly.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {

        int start = 0,end = 0, maxsofar=0, ans=0;
        int arr[256]={0,};
        for(int i=0,n=s.length();i<n;++i)
        {
            if(arr[s[i]]){
                ans = maxsofar>ans?maxsofar:ans;
                while(s[start]!=s[i])
               {
                    arr[s[start]]--;
                    ++start;
               }
               ++start;
                maxsofar = i-start+1;
            }
            else{
                ++arr[s[i]];
                maxsofar++;
            }
        }
        ans = maxsofar>ans?maxsofar:ans;
        return ans;
    }
};
```

25. Reverse a linked list.

We have done this problem many times. But beware of empty linked list ;)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *p,*q,*r;
        if (head == NULL)return NULL;
        q = NULL;
        r = NULL;
        p = head;
        while(p!=NULL)
        {
            r = q;
            q = p;
            p = p->next;
            q->next = r;
        }
        q->next = r;
        return q;
    }
};
```

26. Find the middle of a linked list.
Given a non-empty, singly linked list with head node head, return a middle node of linked list.
If there are two middle nodes, return the second middle node.

a. Again beware of empty linked list. Use slow and fast pointers and test it on some lists of 0,1,2,3,4,5 length.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode *slow, *fast;
        if (head==NULL)return NULL;
        slow = fast = head;
        while(fast!=NULL && fast->next!=NULL)
        {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
};
```

27. Merge two sorted linked lists.

a. Use the merge function of mergesort.
Again beware of empty lists.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *head,*tail,*p,*q;
        if(l1==NULL)return l2;
        if(l2==NULL)return l1;
        if(l1->val<=l2->val){
            head = l1;
            p = l1->next;
            q = l2;
        }else{
            head = l2;
            p = l2->next;
            q = l1;
        }
        tail = head;
        while (p!=NULL && q!= NULL)
        {
            if(p->val<=q->val)
            {
                tail->next = p;
                p = p->next;
            }else{
                tail->next = q;
                q = q->next;
            }
            tail = tail->next;
        }
        while(p!=NULL)
        {
            tail->next = p;
            p = p->next;
            tail = tail->next;
        }
        while(q!=NULL)
        {
            tail->next = q;
            q = q->next;
            tail = tail->next;
        }
        return head;
    }
};
```

28. Remove Nth Node From End of List
Given a linked list, remove the n-th node from the end of list and return its head.
Example:
Given linked list: 1->2->3->4->5, and n = 2.
After removing the second node from the end, the linked list becomes 1->2->3->5.

a. Nth node from end means LIST_SIZE – N + 1st node from start. Calculate the size of the list and using two pointers, delete the node (back->next = front->next)
Also see for the edge cases like :

- what if list is empty
- what if Nth node from last doesn't exist.
- What if we are removing the first node.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *p,*q;
        int size_of_list=0;
        p = head;
        while(p!=NULL)
        {
            p = p->next;
            size_of_list++;
        }
        n = size_of_list - n+1;
        if (size_of_list>n)return head;

        p = head;q = NULL;
        int iamat=0;
        while(p!=NULL)
        {
            iamat++;
            if(iamat==n)
            {
                if (q == NULL)head = head->next;
                else q->next = p->next;
                return head;
            }
            q = p;
            p = p->next;
        }
        return NULL;
    }
};
```

29. Given a direct pointer to a node, delete that node from the list.
Example 1:
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

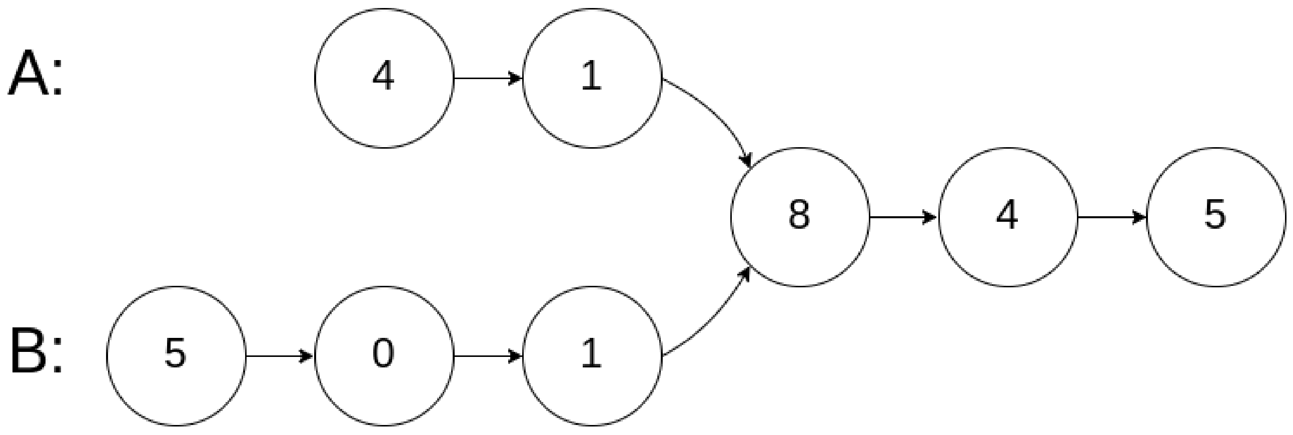
a. We can delete the node easily if we were given the head pointer too. But since it is not given, there is one more way. Copy each node->next->value to the current value. And delete the next node.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
NOTE: This code isn’t checking the edge cases so consider them.
Also it won’t work if we are dealing with the tail.
class Solution {
public:
    void deleteNode(ListNode* node) {
        if(node->next!=NULL)
        {
            node->val = node->next->val;
            node->next = node->next->next;

        }
    }
};
```

30. Adding Two numbers as a Linked List.
You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.
Example:
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *p,*q,*head,*tail;
        int rem = 0;
        if(l1==NULL)return l2;
        if(l2==NULL)return l1;
        head = new ListNode();
        tail = head;
        p = l1; q = l2;
        while(p!=NULL && q!=NULL)
        {
            int x = p->val + q->val + rem;
            rem = x/10;
            p = p->next;
            q = q->next;
            tail->next = new ListNode(x%10);
            tail = tail->next;
        }
        while(p!=NULL)
        {
            int x = p->val + rem;
            rem = x/10;
            p = p->next;
            tail->next = new ListNode(x%10);
            tail = tail->next;
        }
        while(q!=NULL)
        {
            int x = q->val + rem;
            rem = x/10;
            q = q->next;
            tail->next = new ListNode(x%10);
            tail = tail->next;
        }
        if(rem>0){

            ListNode *l3 = new ListNode(rem);
            tail->next = l3;
        }
        return head->next;

    }
};
```

31. Intersection of two Linked Lists
Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:
begin to intersect at node 8.



Example 1:
Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
Output: Reference of the node with value = 8

a. A pretty clean way is to visit the list using first list's head. Mark each node with a special value for example INT_MIN or something... Visit the list using second list head, and the first node having this special value is the intersection point. But this approach will change the linked list data so it is not very effective.

b. Other interesting way is, consider the non-common part,
there can be two cases, either they are equal or not equal
If they are equal, running two pointers from each part will meet somewhere;
Otherwise one of them will get NULL soon.
Now check which one is NULL, reset a pointer it to other list's head again, and run both until the other also gets NULL. Now this pointer is at a Node after which length of both lists is same. Again run two pointers and wherever they meet is the answer. If they don't meet then the list doesn't have an intersection point.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode
*headB) {
        ListNode *p,*q,*r,*s;
        p = r = headA;
        q = s = headB;

        // Make the pointers go to the last.
        while(p!=NULL && q!=NULL)
        {
            if(p==q){  // If they meet (equal length) then return
the pointer
                return p;
            }
            p = p->next;
            q = q->next;
        }

        // If the unmatched part is not equal in length
        // then start again a pointer to equalize the length

        if(p==NULL)
        {
            while(q!=NULL)
            {
                s = s->next;
                q = q->next;
            }
        }

        if(q==NULL)
        {
            while(p!=NULL)
            {
                r = r->next;
                p = p->next;
            }
        }
        // Now that the pointers have been equalized
        // redo the same again. if still you don't find
        // the intersection, return null
        while(r!=NULL && s!=NULL)
        {
            if(r==s)return r;
            r = r->next;
            s = s->next;
        }

        return NULL;
    }
};
```

32. Check if a given linked list is a palindrome or not.
Example 1: Input: 1->2   Output: false
Example 2: Input: 1->2->2->1   Output: true

a. If we can match the list from the mid point, the solution is done. Using slow and fast pointer reach to the mid point. From that point reverse the rest of the list using three pointer method. Now we can easliy match them.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode *slow, *fast;
        slow = fast = head;
        bool isodd = true;
        if(head == NULL || head->next == NULL)return true;
        while(true)
        {
            if(fast == NULL){
                isodd = false;
                break;
            }
            else if(fast->next == NULL)break;
            slow = slow->next;
            fast = fast->next->next;
        }
        if(isodd)slow = slow->next;

        ListNode *p,*q,*r;
        p = slow;
        q = NULL;
        r = NULL;
        while(p!=NULL)
        {
            r = q;
            q = p;
            p = p->next;
            q->next = r;
        }
        q->next = r;
        while(q!=NULL)
        {
            if(q->val!=head->val)return false;
            q = q->next;
            head = head->next;
        }
        return true;
    }
};
```

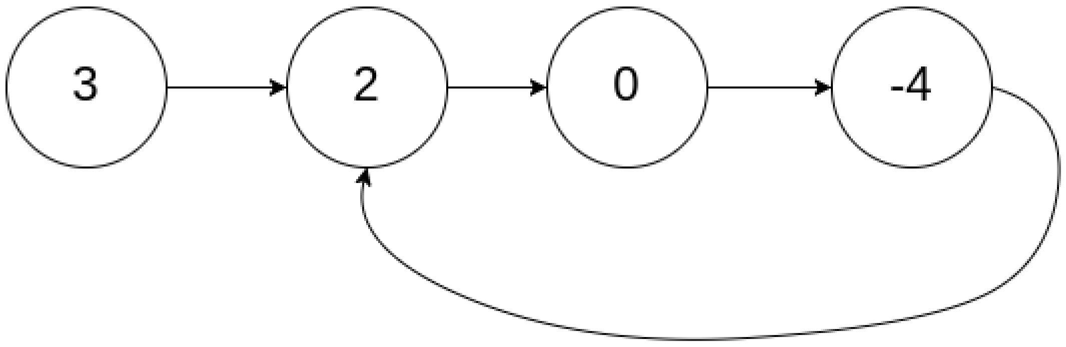
33. Reverse the given linked list in K-groups

a. The solution is straight forward, we really need to do what it says in the question.
Follow these step:

I. initialise some new pointers, p,q,r to reverse the list and head, tail to keep track of the list.
II. Initially head is a new pointer, let’s call it myhead to avoid conflict.
III. Set p = head. q and r will be initialised NULL for each group.
IV. Run a for loop of size K, reverse the entire list upto K nodes. Now two things may break the loop, we either hit the Kth node, or P becomes NULL.
V. If we hit the Kth node, simply put tail.next = q ie head of the reversed list and move tail to the end.
VI. Otherwise we have reversed the list non intentionally so we need to restore it. To do so, again reverse the list starting from where our current head ie q is. After reversing put tail.next = new head ie q.
VII. Return myhead -> next.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode *myhead,*tail,*p,*q,*r;
        if (head==NULL)return head;
        myhead = new ListNode();
        tail = myhead;
        int i;
         p = head;
        while(p!=NULL){
            q = NULL;
            r = NULL;
            for(i=0;i<k && p!=NULL;i++)
            {
                r = q;
                q = p;
                p = p->next;
                q->next = r;
            }
            if(i==k)
            {
                tail->next = q;
                while(tail->next!=0)
                {
                    tail = tail->next;
                }
            }
            else{
                p = q;
                q = NULL;
                r = NULL;
                while(p!=NULL)
                {
                    r = q;
                    q = p;
                    p = p->next;
                    q->next = r;
                }
                q->next = r;
                tail->next = q;
            }
        }
        myhead = myhead->next;
        return myhead;
    }
};
```

34. Detect a loop/cycle in linkedlist.



a. We can mark each node with a special value if it is known that what are the values present in the linked list But it won’t work otherwise.. If we encounter the node again, that means there is a cycle. This approach will modify the list so it’s not preferred.

```
bool hasCycle(ListNode *head) {
        ListNode *p;
        p = head;
        while(p!=NULL)
        {
            if(p->val==INT_MIN)return true;
            p->val = INT_MIN;
            p = p->next;
        }
        return false;
    }
```

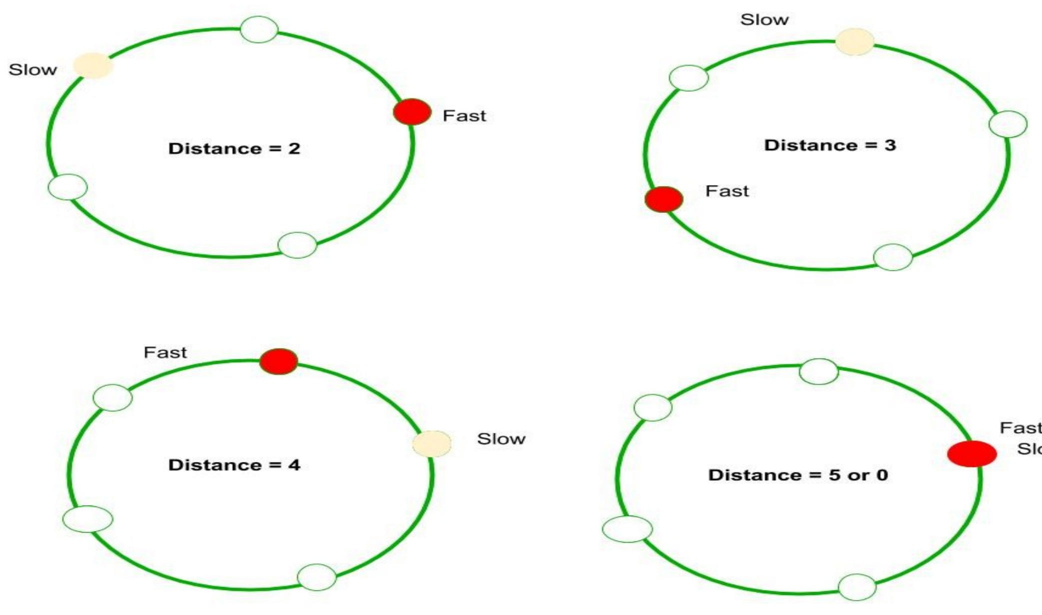
b. Extending the above idea, we can actually hash the node addresses instead because addresses are unique in a machine. If we revisit a node, there is a cycle. But it will cost extra space of O(N).

c. Use Floyd’s cycle detection algorithm, using two pointers, slow and fast.

```
    bool hasCycle(ListNode *head) {
        ListNode *slow,*fast;

        slow = fast = head;
        while(fast!=NULL &&  fast->next!=NULL )
        {
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast)return true;
        }
        return false;
    }
};
```

But why it works and how do we guarentee that the pointers will meet?
If a cycle is not present the pointers will never meet. Because faster pointer keeps on going and the slower one can’t catch upto it.
If they are in a cycle, one thing is interesting to observe:

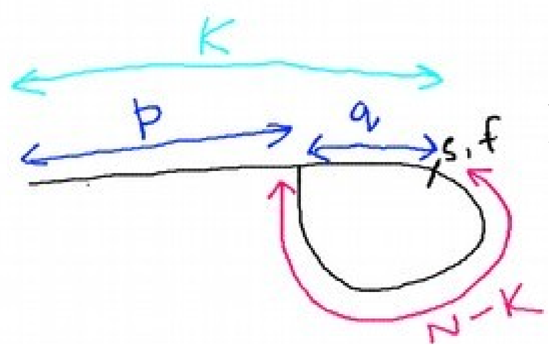
i. Initially both pointers are at 0 distance.
ii. Each iteration makes them one step apart, ie 0,1,2,...k
iii. If they are in a cycle of size N, they can get atmost N step apart (in either clockwise or anticlockwise direction). Here N step in any direction means they are now at same position.
iv. Since they are in same position, we can say the reverse that they are moving in a cycle, that’s all.



Refer https://www.geeksforgeeks.org/how-does-floyds-slow-and-fast-pointers-approach-work/

34a. Return the node from where the loop starts.
First of all check whether a cycle exists or not. If it exists, do the following:

i. put a new pointer say P on head.
ii. Move slow pointer and P at same pace until slow meets fast again. This will move P to K steps where K is the length of cycle.
iii. Suppose list is having N nodes and cycle is of length K. That means we need to move N-K more steps.
iv. We start another pointer Q from head and move P and Q at same speed. They will meet after N-K steps; that is the start of the loop.



Why step iv is true? Suppose slow pointer is k steps away and start of loop is p steps away. Start of loop and slow pointer are q steps away. Hence p+q = k. If we move poniter P ahead (cycle length = N-k+q) times, it will need x more steps to reach Nth node. Ie x = N – (N-k+q) = k-q = p steps. Q will meet P after p steps and P will reach to Nth node after p steps and that is the loop start point. Hence we can say the reverse, the point where P and Q meet, is the Nth node and that's where loop entry point starts.

```
ListNode *detectCycle(ListNode *head) {
        ListNode* s , *f, *p,*q;
        int k=0;
        if(head == NULL)return NULL;
        s = f = head;
        while(f!=NULL && f->next!=NULL)
        {
            s = s->next;
            f = f->next->next;
            if(s==f)break;
        }
        if (f==NULL || f->next==NULL)return NULL;
        p = head;
        do{
            p = p->next;
            f = f->next;
        }while(s!=f);
        q = head;
        while(p!=q){
            p=p->next;
            q= q->next;
        }
        return p;


    }
};
```

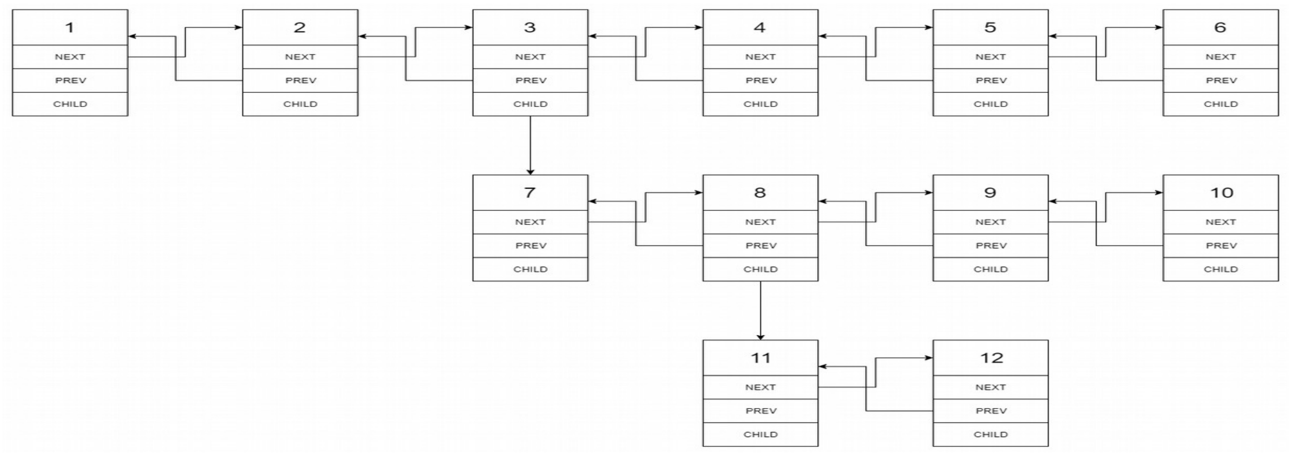
34b. Remove the loop in the list if it is present.
a. We can do that easily if we are following approach a or b.

b. Without extra space or without modifying the list, we have to use approach 34a and if we find a node X such that `X->next = startOfLoopNode`, put `X->next = NULL`.

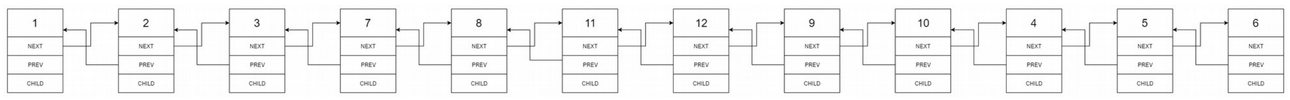
35. Flattening a linked list.

This question has multiple variants but all of them are similar in the sense that we need to append a new list (emerging out of a node) in between the current and next nodes. One variant on leetcode is about DLL where you need to maintain next and prev both pointers. The question is appended below: https://leetcode.com/problems/flatten-a-multilevel-doubly-linked-list/

35a.Flatten a Multilevel Doubly Linked List
Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
Output: [1,2,3,7,8,11,12,9,10,4,5,6]
INPUT:



OUTPUT:



a. Use a pointer P to traverse the list. Whenever a node having a child is encountered, move a pointer to child, and one to its next. Our local goal is to merge the entire list in between these two nodes.

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node* child;
};
*/

class Solution {
public:
    Node* flatten(Node* head) {
        Node *p,*q,*r;
        p = head;
        while(p!=0)
        {
            if(p->child!=0)
            {
                q = p->next;
                if(q==NULL){
                    p->next = p->child;
                    p->child->prev = p;
                }
                else{
                    r = p->child;
                    p->next = r;
                    r->prev = p;
                    while(r->next!=0)
                    {
                        r = r->next;
                    }
                    r->next = q;
                    q->prev = r;
                }
                p->child = 0;
            }
            p = p->next;
        }
        return head;
    }
};
```

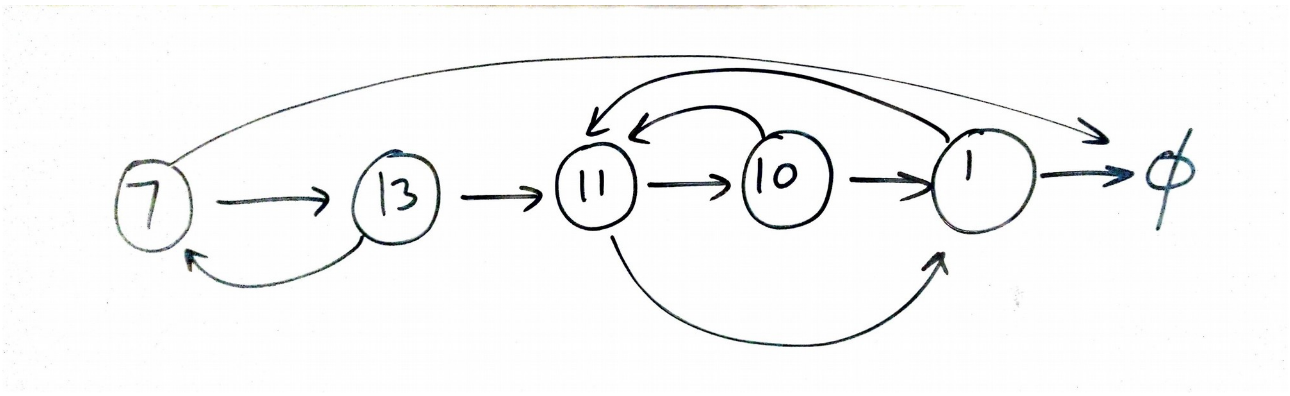
36. Rotate a linked list.
If you need to rotate it counter-clockwise, simply move to kth node (don't forget to make k = n%k because rotating it n times doesn't make any changes.) Move a pointer to kth node, one just behind it and other at the end and make proper changes.
In case you need to rotate clock wise, do the same except that move the pointer n-k times.

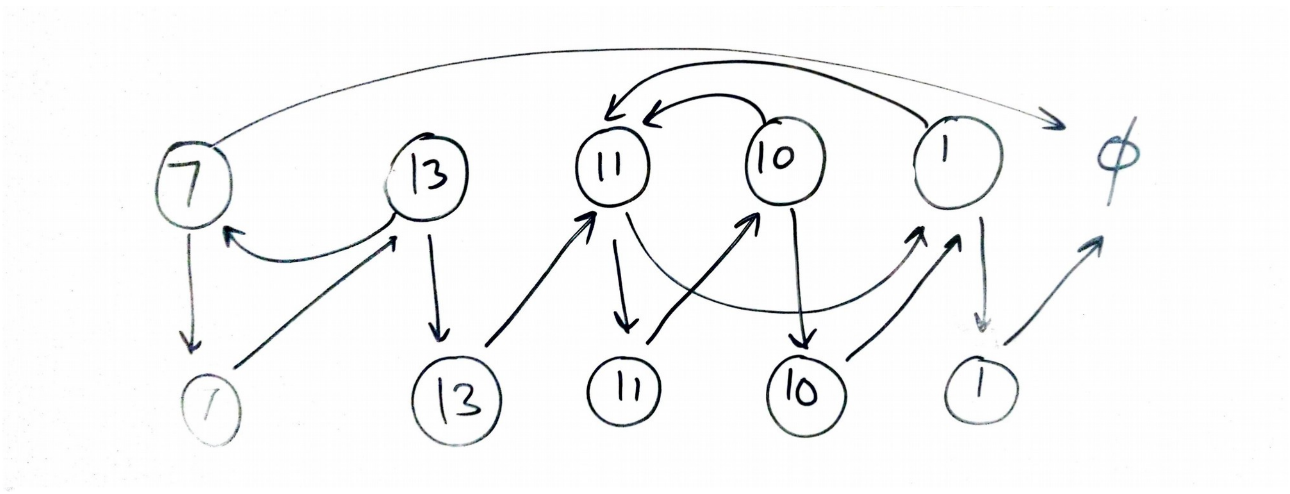
```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        ListNode *p,*q,*r;
        if(head==NULL || head->next==NULL)return head;
        int i,n=0;
        p = head;
        while(p!=NULL){
            n++;
            p = p->next;
        }
        k = k%n;
        if(k==0)return head;
        // This step can be done more easily so do the changes.
        for(i=0,p=head;i<k-1;i++)
        {
            p = p->next;
        }
        q = head;r = 0;
        while(p->next!=NULL)
        {
            p = p->next;
            r = q;
            q = q->next;
        }
        r->next = 0;
        p->next = head;
        head = q;
        return head;
    }
};
```

37. Clone a linked list with random and next pointer.

a. If we have extra space, we can use hashing to easily copy the node-node mapping.
b. The solution is as below but understanding the visualization is better.
   i. Make a new list with same data. We need to put each node in A->A'->B->B' ... pattern. Why? Because that will make it easy to store all the pointers as well as it will make it easy to access the random pointers in first list and corresponding random pointer in second list.
   ii. After that assign the random pointers using this interwoven list.
   iii. Separate this new list from the old list by deleting each node one by one and appending them to a new list.

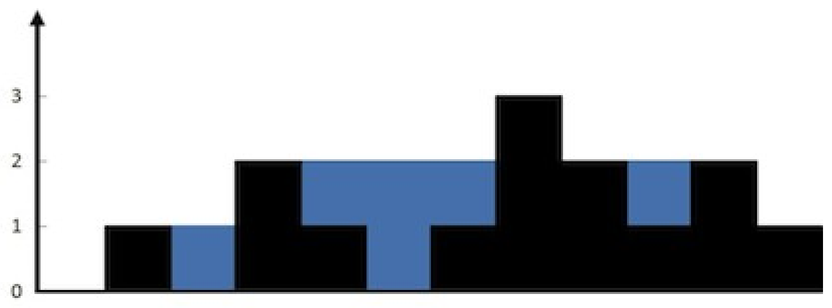
iv. Return the new list.





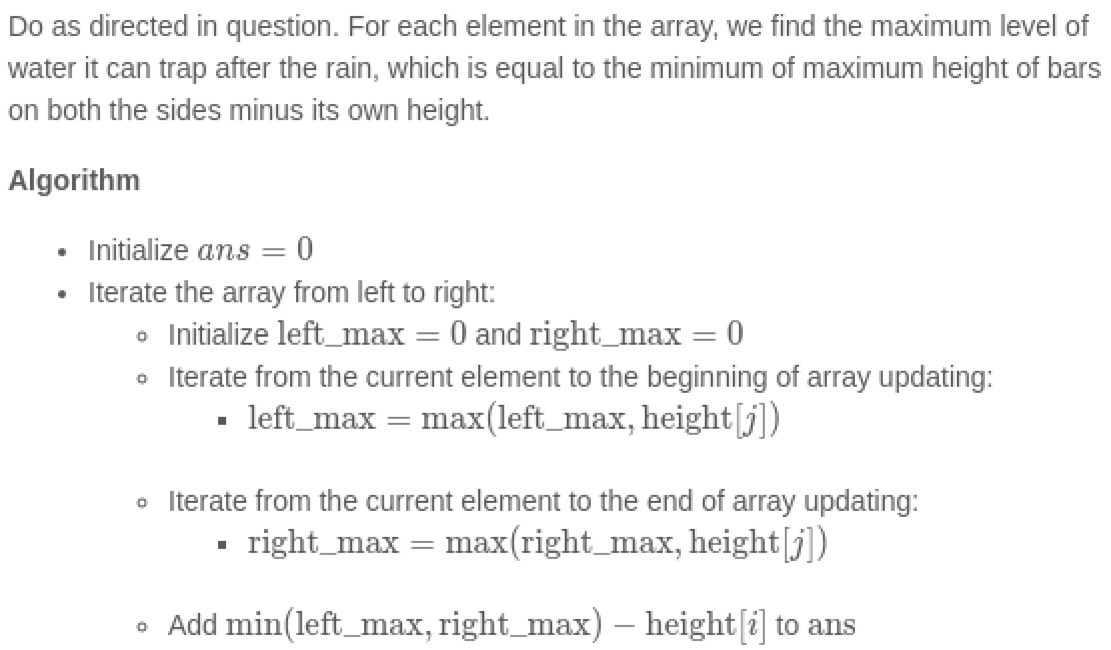
```
/*


// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/

class Solution {
public:
    // This question was asked in MICROSOFT TECH SET GO round 1
    // So I already know the solution....
    Node* copyRandomList(Node* head) {
        Node * p,*q,*myhead,*tail;

        myhead = new Node(-1); // Initialize a new head and tail.
        tail = myhead;
        p = head; // initialze P

        // First make a straight chain so that all nodes are
copied.
        while(p!=NULL){
            q = new Node(p->val); // Create a node with value at P
            q->next = p->next; // Save P->next reference;
            p->next = q; // append this node after p
            p = q->next; // move P

        }

        // Now nodes have been interwoven   A->A'->B->B'....

        // Now assign the pointers.. Visualization is the best way
        p = head;

        // Beware you are not trying to access NULL->next ;)
        while(p!=NULL)
        {
            p->next->random = (p->random==NULL)?NULL:p->random-
>next;
            p = p->next->next;
        }

        // Now seperate the interwoven list.

        p = head;
        while(p!=NULL)
        {
            tail->next = p->next;
            tail = tail->next;
            p->next = p->next->next;
            p = p->next;
        }
        return myhead->next;



    }
};
```
38. Merge two sorted linked lists.
39. Detect starting point of the loop.
40. 3 Sum problem.
// 38 and 39 are repeated. 40 is done with 4-Sum problem

41. Trapping Rain Water
Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.
Example:
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

a. This problem is a little hard if you solve it using brute force. We have to find two walls with equal or greater height than current wall. For that we need to run two loops, for maintaining both the pointers.
The logic is we need to find the left and right walls, as left<=right wall. Suppose their are K steps from left wall to right wall. Here total area = min(left,right)*K and if we delete the walls, whatever left is the water.



Do as directed in question. For each element in the array, we find the maximum level of water it can trap after the rain, which is equal to the minimum of maximum height of bars on both the sides minus its own height.

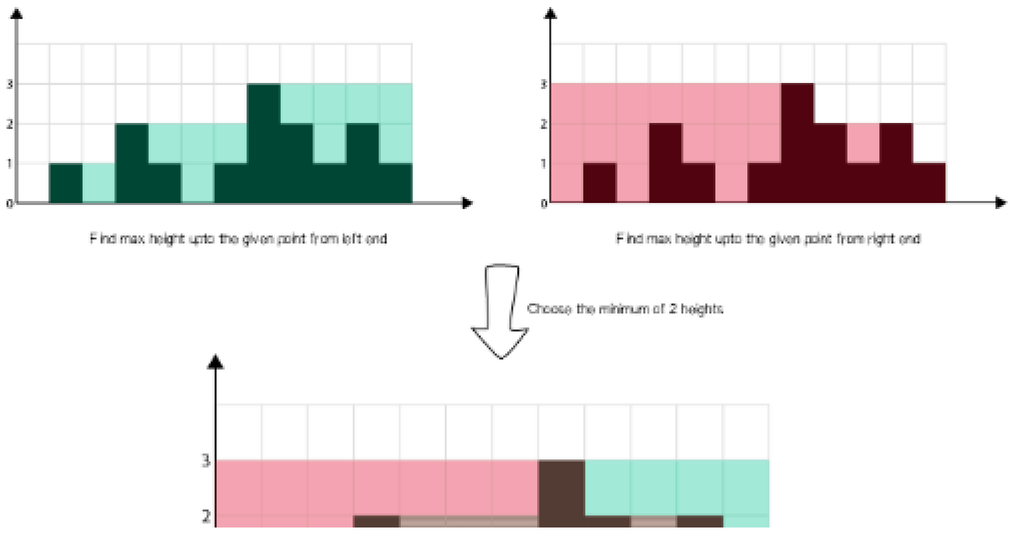
Algorithm

- Initialize $ans = 0$
- Iterate the array from left to right:
  - Initialize $\text{left\_max} = 0$ and $\text{right\_max} = 0$
  - Iterate from the current element to the beginning of array updating:
    - $\text{left\_max} = \max(\text{left\_max}, \text{height}[j])$
  - Iterate from the current element to the end of array updating:
    - $\text{right\_max} = \max(\text{right\_max}, \text{height}[j])$
  - Add $\min(\text{left\_max}, \text{right\_max}) - \text{height}[i]$ to ans

b. A better solution is :

In brute force, we iterate over the left and right parts again and again just to find the highest bar size upto that index. But, this could be stored. Voila, dynamic programming.

The concept is illustrated as shown:


Find max height upto the given point from left end
Find max height upto the given point from right end
Choose the minimum of 2 heights

Algorithm

- Find maximum height of bar from the left end upto an index i in the array left_max.
- Find maximum height of bar from the right end upto an index i in the array right_max.
- Iterate over the height array and update ans:
    - Add $\min(\text{left\_max}[i], \text{right\_max}[i]) - \text{height}[i]$ to ans

```
int trap(vector<int>& height) {
        int n = height.size(),i;
        if(n==0)return 0;
        vector<int> left(n),right(n);
        left[0] = height[0];
        right[n-1] = height[n-1];
        for(i=1;i<n;i++)
        {
            left[i] = max(left[i-1],height[i]);
            right[n-i-1] = max(right[n-i],height[n-i-1]);
        }
        int ans=0;
        for(i=0;i<n;i++){
            ans+= min(left[i],right[i]) - height[i] ;
        }
        return ans;
    }
```

42. Remove duplicate from sorted array.
Given a sorted array A, remove the duplicates inplace and return the new length of the array.

a. Solution is simple, fix two pointers, i and j. Move j while A[i] and A[j] are equal. Let K be a third pointer that we will use for maintaining the current length of our array. Now assign this element to a pointer at the start of the array. Repeat this process and return k.

```
 int removeDuplicates(vector<int>& a) {
        int i,j,k,n = a.size();
        if(n==0)return 0;
        k=-1;
        i=0;
        while(i<n)
        {
            j=i+1;
            while(j<n && a[j]==a[i])j++;
            a[++k] = a[i];
            i=j;
        }
        return ++k;
    }
```

b. One more approach using only two pointers:

```
int removeDuplicates(vector<int>& nums) {
    if (nums.size() == 0) return 0;
    int i = 0;
    for (int j = 1; j < nums.size(); j++) {
        if (nums[j] != nums[i]) {
            i++;
            nums[i] = nums[j];
        }
    }
    return i + 1;
}
```

43. Find maximum consecutive 1's.

a. This problem can also be solved using 2 pointers. Fix the current pointer, move the other while it is equal to 1. Update the answer with max consecutive series.

```
 int findMaxConsecutiveOnes(vector<int>& a) {
        int ans=0,i=0,j,n=a.size();
        while(i<n)
        {
            j=i;
            while(j<n && a[j]==1)j++;
            ans = max(ans,j-i);
            i=j+1;
        }
        return ans;
    }
```

44. N meetings in one room.
I got so many variations of this problem on internet.

The main statement is, *there are N meeting times scheduled in a office, each is given in the form Si->Fi where Si is start time and Fi is finish time. Some of these times may overlap.* The variations are like these:

i. *Find the minimum number of rooms to conduct all the meetings given that a room can be reused. Click Here to check out the problem*
ii. *There is only one meeting room in the office, what is the maximum number of meetings that can be conducted? Click here to check the problem*
iii. *At max how many meetings can a person attend given that he can attend only one meeting at a time.*

However the solution of all these follows similar concept.

a. If we sort the meetings according to the start time, a lot of smaller meetings can be skipped in case the first meeting is too long. Hence sorting the meetings according to finish time makes sense because the earlier a meeting finishes, the shorter it is.

After that for (ii) and (iii) we need to simply find the longest non overlapping interval.
For (i) we need to find the maximum number of meetings that we have skipped, that is the number of rooms we need to conduct parellel meetings. A good data structure for fast deletion like linked list can be used in this case.

```
// Code for 44(ii)
#include<bits/stdc++.h>
using namespace std;
struct meeting{
    int s,f,pos;
};
vector<meeting> a;
bool comp(const meeting& p1 , const meeting& p2)
{
    if(p1.f==p2.f)return p1.s<p2.s;
    return p1.f<p2.f;
}
vector<int > ans;
int main()
{
    int t,n,i,l,r;

    cin>>t;
    while(t--)
    {
        a.clear();
        cin>>n;
        for(i=0;i<n;i++) // input provides start times in one line
        {                // and finish times in other line.
            cin>>l;
            a.push_back({l,0,i+1});
        }
        for(i=0;i<n;i++)
        {
            cin>>r;
            a[i].f= r;
        }
        sort(a.begin(),a.end(),comp); // Sort the meetings.
        ans.clear();

        r = a[0].f;
        ans.push_back(a[0].pos); // We will attend the first
        for(i=1;i<a.size();++i)  // meeting
        {
            if(r<=a[i].s){ // After that each meeting that doesn't
                ans.push_back(a[i].pos);// Overlap must be pushed.
                r = a[i].f; // Update the new boundary
            }

        }
        for(i=0;i<ans.size();++i)cout<<ans[i]<<" ";
        cout<<endl;

    }
}
```

45. Activity selection problem.
The problem statement differs but the concept is what we used in problem 44(ii).
Problem Statement is: *You are given N activities with their start and finish times. 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 time.*

46. Greedy way to exchange minimum number of coins.
Given a set of available denominations, you need to find the minimum number of coins to exchange the given amount.
a. We can solve this problem by sorting the denominations in decreasing order and fulfilling the amount using largest denomination first.

```
int coinChange(vector<int>& coins, int amount) {
        int i,n=coins.size(),ans=0;
        if(n==0)return -1;
        sort(coins.begin(),coins.end());
        for(i=n-1;i>=0;i--)
        {   //cout<<amount<<" "<<coins[i]<<endl;
            if(amount>=coins[i]){
                ans+=amount/coins[i];
                amount -= coins[i] * (amount/coins[i]);
            }
            if(amount==0)return ans;
        }
        return -1;
    }
```

This solution can be proved wrong for many test cases because greedy algorithm is a poor choice for this problem.
Eg [1,8,9] 32 can be made using 8*4 = 4 coins. But greedy gives 9*3 + 1*5 = 8 coins.

47. Fractional Knapsack problem.
Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
The optimal solution is to take the maximum possible item having largest value/weight ratio. If this whole item is taken, move to the other one having the highest value.

```
struct item{
    int value,weight;
};
bool comp(const item& a , const item& b){return 
1.0*a.value/a.weight>1.0*b.value/b.weight;}

double fractionalKnapsack(vector<item>& a , int W)
{
    int i,n=a.size();
    double ans=0;
    sort(a.begin(),a.end(),comp);
    //for(i=0;i<n;i++)cout<<a[i].value<<" "<<a[i].weight<<"
";cout<<endl;
    for(i=0;i<n;i++) {
        if(a[i].weight<=W) {
            ans+=a[i].value;
            W-=a[i].weight;
        } else {
            ans += (double)W * ((double)a[i].value/
(double)a[i].weight);
            W = 0;
        }

        if(W==0)return ans;
    }
    return ans;
}
```

48. Minimum number of platforms required for a railway.

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 waits. We are given two arrays which represent arrival and departure times of trains that stop.

Examples:
Input: arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}
        dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}
Output: 3
There are at-most three trains at a time
(time between 11:00 to 11:20)

a. Suppose the times are given in string format. If they were in integer form, there is no problem otherwise we can convert them to an equivalent integer.
A. If array size is too big convert 9:00 to 900, 11:20 to 1120 ... etc
B. Otherwise sort the array and from the beginning use mapping like 9:00 =1 , 9:10 = 2 and so on..

We can use prefix sum for ranges to solve the problem, the time range at which most of the trains arrive, is our answer.

i. Compute the prefix array by putting MAP[arr[i]] += 1 and MAP[dep[i]+1] -= 1.
ii. Compute the prefix sum.
iii. Find the maximum and that's the answer.

```
int minimumPlatforms(vector<pair<int,int>>& schedule){
    map<int,int> mp;
    int n = schedule.size();
    for(register int i=0;i<n;i++) {
        mp[schedule[i].first]++;
        mp[schedule[i].second+1]--;
    }

    for(auto i = ++mp.begin();i!=mp.end();++i){
        i->second += prev(i,1)->second;
    }
    int ans = 0;
    for(auto i:mp){
        ans = max(ans, i.second);
    }
    return ans;

}
```

49. Job sequencing problem.
Given a set of N jobs where each job i has a deadline and profit associated to it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done.

a. Using greedy strategy, we sort the Jobs on the basis of their profit. After that we will try to place each of them at the extreme end(near to their deadline). If two jobs have same profit, we will put the job having nearer deadline first.

```
struct Job // from gfg
{
   char id;      // Job Id
   int dead;    // Deadline of job
   int profit;  // Profit if job is over before or on deadline
};

// This function is used for sorting all jobs according to profit
bool comparison(Job a, Job b)
{
     return (a.profit > b.profit);
}

// Returns minimum number of platforms reqquired
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of profit
    sort(arr, arr+n, comparison);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n];  // To keep track of free time slots

    // Initialize all slots to be free
    for (int i=0; i<n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
       // Find a free slot for this job (Note that we start
       // from the last possible slot)
       for (int j=min(n, arr[i].dead)-1; j>=0; j--)
       {
          // Free slot found
          if (slot[j]==false)
          {
             result[j] = i;  // Add this job to result
             slot[j] = true; // Make this slot occupied
             break;
          }
       }
    }

    // Print the result
    for (int i=0; i<n; i++)
       if (slot[i])
         cout << arr[result[i]].id << " ";
}
```

50. N Queens Problem.

You are given an NxN chessboard and N Queens. Your task is to place exactly N Queens on the chess board in a way that no Queen can kill any other Queen in one move.

a.

i. A Queen can attack in 8 directions, N,E,S,W,NE,NW,SE,SW.
ii. We must place each Queen one by one, checking for where it is violating the above conditions.
iii. If any of the conditions is violated, we must backtrack.

See the code below, it’s enough to understand the solution.

```
class Solution {
public:
    bool isSafe(vector<string>& board, int row, int col){
        // Check whether placing a Queen here is safe or not?

        int i,j;

        // Check in N direction
        for(i=row-1;i>=0;i--)
            if(board[i][col]=='Q')
                return false;

        // Check in NE direction
        for(i=row-1,j=col-1;i>=0&&j>=0;i--,j--)
            if(board[i][j]=='Q')
                return false;

        // Check in NW direction
        for(i=row-1,j=col+1;i>=0&&j<board.size();i--,j++)
            if(board[i][j]=='Q')
                return false;

        // If all clear, board is safe
        return true;

    }
    void solve(vector<string>& board, int i,int
n,vector<vector<string>>& ans){

        // If we filled n rows, push the current board
        // and backtrack for more solutions
        if(i>=n){
            ans.push_back(board);
            return;
        }

        // Now for the current row ie 'i', try to place
        // a Queen in a column if it is safe.
        // After that remove it because we can't place more
        // than one Queen in a row.
        for(int j=0;j<n;j++)
        {
            if(isSafe(board,i,j)){  // Is it safe?
                board[i][j] = 'Q'; // If yes, place a queen
                solve(board,i+1,n,ans); // Now check for next row
                board[i][j] = '.'; // remove the Queen
            }
        }

    }
    vector<vector<string>> solveNQueens(int n) {
        // We have to return every possible solution of this
problem
        // Answer is a 3-D grid having multiple 2-D grids
        // Each row is represented by a string
        // (.) means blank and Q means there is a queen
        vector<vector<string>> ans;

        // Prepare the board
        vector<string> board;
        string s ="";
        int i,j;
        for(i=0;i<n;i++)s+=".";
        for(i=0;i<n;i++)board.push_back(s);

        // Board prepared! Now solve the problem,
        // here I am passing the ans vector so that
        // I can push each board whenever I reach
        // to a solution.

        solve(board,0,n,ans);
        return ans;
    }
};
```

51. Solving Sudoku using backtracking

You are given a *valid* sudoku grid, with some prefilled numbers, you have to fill the empty spaces using digits 0-9 such that 3 sudoku rules: i. Column ii. Row iii. The 3x3 grid; must contain unique digits only.

a. For each move we check whether it is safe to play or not. Go through the following code to understand the process.

```
class Solution {
public:
    // Fantastic!! I didn't imagine that I would get this in
single
    // shot. Sudoku is easier than N Queen!!!


    // Function to check for safe move.
    bool isSafe(vector<vector<char>>& board , int r , int c , char
num)
    {
        int i,j;
        // Check for entire Row
        for(i=0;i<9;i++)if(board[i][c]==num)return false;

        // Check for entire column
        for(i=0;i<9;i++)if(board[r][i]==num)return false;
        // Shift to the start of current 3x3 grid
        r = (r/3)*3;
        c = (c/3)*3;

        // Check the entire 3x3 grid
        for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                if(board[r+i][c+j]==num)
                    return false;

        // Otherwise it is safe
        return true;

    }
    bool solve(vector<vector<char>>& board, int r, int c)
    {
        // If you have successfully reached to the end
        // return true
        if(r==9 || c==9){  // c=9 will never occur
            return true;
        }

        // If this row, col already has a number,
        // skip it and go for the next one
        if(board[r][c]!='.'){
            return c==8?solve(board,r+1,0):solve(board,r,c+1);
        }

        // Otherwise try each digit one by one and
        // check for next non empty row,col
        for(int d=1;d<10;d++) // iterate over 1 to 9
        {
                if(isSafe(board, r, c, '0'+d)){  // Is putting
this number here safe?
                    board[r][c] = '0'+d;   // if yes, put the
number
                    bool isright = c==8?
solve(board,r+1,0):solve(board,r,c+1);  // Now go for the next
                    if(isright)return true; // If you have solved
the puzzle, congrats
                    board[r][c] = '.'; // Otherwise erase the
number and go for other number

                }
        }
        // Since none of the above worked, backtrack.
        return false;
    }
    void solveSudoku(vector<vector<char>>& board) {
        solve(board,0,0);
    }
};
```

52. M Coloring Problem

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

The solution is more simple than the Sudoku Puzzle. But before starting the coding, let’s understand few points:

i. The problem asks whether it is possible to color the graph using ‘m’ colors or not? We can also get a variation, find the smallest value of ‘m’ to color a given graph.
ii. We can use adjacency list for representing the graph but it becomes complex(maintain separate visited array and pass it everywhere) for determining the last node of graph. So I will use adjacency matrix instead.
iii. Colors can range from 1 to N, the number of nodes, so we have to determine an upper limit. Note that every graph has a solution if ‘m’ is not restricted.

I have solved this problem in such a way that if you need to tell the smallest value of M, you can find it in assignedcolors[]. If you want to determine whether it is possible to color the graph or not, you can tell that also. The problem link is Here.

Below is the implementation:

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

// Assign it something like N
#define MAX_COLOR 50

// Check whether coloring currentNode with this color is safe or not?
bool isSafe(vector<vector<bool>>& graph, vector<int>& assignedcolors, int color,
int V, int n){
    for(int i=0;i<n;i++)  // Traverse the adjacent neighbours
        if(graph[V][i])
            if(assignedcolors[i]==color) // Any of them has this color then it's
not safe
                return false;

    return true;// Otherwise it is safe.
}

bool solve(vector<vector<bool>>& graph, vector<int>& assignedcolors, int
currentVertex, int n)
{
    if(currentVertex == n) // We have colored all nodes
    {
        return true;
    }

    for(int color = 0;color<MAX_COLOR; color++){ // Choose a color
        if(isSafe(graph, assignedcolors, color, currentVertex,n)){ // Check if
it safe to fill this color
            assignedcolors[currentVertex] = color; // Try to fill
            bool solved = solve(graph, assignedcolors ,currentVertex+1,n); //
Check is it solvalble now?
            if(solved)return true; // If solved, congrats
            assignedcolors[currentVertex] = -1; // Else remove this color
        }
    }

    return false; // I guess this statement won't get executed.
}
void clearGraphAndColors(vector<vector<bool>>& graph, vector<int>&
assignedcolors)
{
    for(int i=0;i<50;i++){
        for(int j=0;j<50;j++)
        {
            graph[i][j] = false;
        }
    }
    for(int i=0;i<50;i++){
        assignedcolors[i] = -1;
    }
}
int main()
{
      int t,n,m,i,u,v,e;
      // It is given in constraints that N <=50
      vector<vector<bool>> graph(50,vector<bool>(50,false));
      vector<int> assignedcolors(50);
      cin>>t;
      while(t--){
          clearGraphAndColors(graph,assignedcolors);
          cin>>n; // Number of nodes
          cin>>m; // Value of M
          cin>>e; // Number of edges
          for(i=0;i<e;i++){
              cin>>u>>v;
              --u;--v;
              graph[u][v] = graph[v][u] = true;
          }
          bool ans = solve(graph, assignedcolors, 0,n);
          if(ans==false){
              cout<<0<<endl;
          }else{
              int mx=-1;
              for(i=0;i<n;i++){
                  mx = max(mx,assignedcolors[i]);
              }
              if(mx<m){
                  cout<<1<<endl;
              }else{
                  cout<<0<<endl;
              }
          }



      }
      return 0;
}
```

53. Rat in the Maze Problem

Consider a rat placed at (0, 0) in a square matrix m[ ][ ] of order n and has to reach the destination at (n-1, n-1). The task is to find a sorted array of strings denoting all the possible directions which the rat can take to reach the destination at (n-1, n-1). The directions in which the rat can move are ‘U'(up), ‘D'(down), ‘L’ (left), ‘R’ (right).

In this problem, we need to check for the safe move as: a. Don't visit the already visited cells and b. this cell shouldn't be a wall.

// I will add the code later

54. Print all the permutations of a string/ array.

a. Recursively call for each index. Try each element at that index.

```
void solve(vector<int>& a, int index, vector<bool>& visited, vector<int>& res,
vector<vector<int>>& ans){
        // It may seem like I am using too much arguments but I am only using
references to the vectors
        // Plus I am using 0 global variables i.e. the code is reusable and
readable.
        // Here if I use global variables, I will need only index ie one
variable.

        if(index == a.size()) { // Congrats you reached at the end.
            ans.push_back(res); // Push the current permutation
            return;
        }

        // Start using each number of the array at current index

        for(int i=0;i<visited.size();i++){
            if(visited[i]==false){
                res[index] = a[i]; // Put this element
                visited[i] = true; // mark it visited
                solve(a, index+1, visited, res, ans); // Solve for next index
                res[index] = INT_MIN; // Erase the element
                visited[i] = false; // Mark it unvisited
            }
        }
    }
    vector<vector<int>> permute(vector<int>& a) {
        int i,n =a.size();
        vector<vector<int>> ans;
        vector<bool> visited(n,false);
        vector<int> res(n,INT_MIN);
        solve(a,0,visited,res,ans);
        return ans;
    }
```

55. Word Break.
This problem is interesting. You are given a non empty word S and a dictionary of other words D. Your task is to determine whether it is possible to form the string S using the words of Dictionary D or not? Or in other words, is it possible to break the string S in such a way that every substring is in the dictionary?

See some examples from leetcode:
Example 1:
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
Example 2:

Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
Example 3:
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false

a. I wrote a recursive solution with proper comments below, you should go through it:

```
class Solution {
public:
    // I couldn't come up with a backtracking solution so I searched it on
internet
    // I found a video here https://www.youtube.com/watch?v=WepWFGxiwRs
explaining
    // the dynamic programming approach. I understood the recursive idea of his
approach
    // Let's take first example as it is easy.
    // s = "leetcode", wordDict = ["leet", "code"]
    // We need to break the string at different points and check whether both
belong to a
    // dictionary or not
    //     for example break "leetcode" as {"l eetcode", "le etcode","lee
tcode"...,"leetcod e"}
    // We will recursively check whether both of them exist in the dictionary or
not,
    // Or if they can be made up of a existing word in our dictionary
    //      For example we broke it as "leet code" since leet and code both are
in the
    // dictionary we return true
    //     Otherwise we will break leet and code into parts as "l eet", "le
et",... and
    //     "c ode","co de",... and when we get a partition such that each part
exist in
    //     dictionary, we return true.


    // It takes the substring, and returns whether its parts are in dictionary
or not
    bool solve(string& s , int l , int r, set<string>& dict){
        string S = s.substr(l,r-l+1); // The substring s[l:r+1] in Python3

        if(dict.find(S)!=dict.end())return true; // If it exists return true

        else if(l==r) return false; // It can't be broken anymore so sorry...

        for(int i=l;i<r;i++){ // Make breaking points,

            bool x = solve(s, l, i, dict) && solve(s,i+1,r,dict); // Check
whether the breaking
                                    // point breaks in two valid strings or not

            if(x)
                return true;
        }

        return false;
    }
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> dict;
        for(auto i:wordDict){
            dict.insert(i);
        }
        return solve(s, 0,s.length()-1, dict);
    }
};
```

b. But the above solution won’t work for larger strings because it takes too much stack space and checks for all possible strings. But it contains overlapping sub-problems(l and r are getting repeated many times) so we can turn it into dynamic programming.

The memoization approach for the above approach is as following, so go through the changes:

```
  bool solve(string& s , int l , int r, set<string>& dict,vector<vector<int>>&
dp){
        if(dp[l][r]!=0)return dp[l][r]==1?true:false;

        string S = s.substr(l,r-l+1); // The substring s[l:r+1] in Python3

        if(dict.find(S)!=dict.end()){
            dp[l][r] = 1;
            return true;} // If it exists return true

        else if(l==r) {
            dp[l][r] = 2;
            return false;} // It can't be broken anymore so sorry...

        for(int i=l;i<r;i++){ // Make breaking points,

            bool x = solve(s, l, i, dict,dp) && solve(s,i+1,r,dict,dp); // Check
whether the breaking
                                        // point breaks in two valid strings or not

            if(x) {
                dp[l][r] = 1;
                return true;
            }
        }
        dp[l][r] = 2;
        return false;
    }
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> dict;
        vector<vector<int>> dp(s.length(), vector<int>(s.length(), 0));
        for(auto i:wordDict){
            dict.insert(i);
        }
        return solve(s, 0,s.length()-1, dict,dp);
    }
```

One interesting thing that this solution got accepted on leetcode with 10% and 20% (time& space) which is greater than the dynamic programming solution.

c. Now convert it to pure dynamic programming:

```
bool wordBreak(string s, vector<string>& wordDict) {
        set<string> dict;
        vector<vector<bool>> dp(s.length(), vector<bool>(s.length(), false));
        for(auto i:wordDict){
            dict.insert(i);
        }
        int i,j,n=s.length();
        for(i=0;i<n;i++){
            if(dict.find(s.substr(i,1))!=dict.end()){
                dp[i][i] = true;
            }
        }

        for(i=1;i<n;i++) {
            for(j=0;j<n-i;j++) {
                if(dict.find(s.substr(j,i+1))!=dict.end()) {
                    dp[j][j+i] = true;
                }else{
                    for(int k=0;k<i;k++){
                        if(dp[j][j+k]&dp[j+k+1][j+i]){
                            dp[j][j+i]= true;
                            break;
                        }
                    }

                }
            }
        }
        return dp[0][n-1];
    }
```
55a. Modify the above solution and print all words that build the string. Also print all possible ways.
In the recursive(memoized) solution, the moment r == word.size(), we have reached to the right end. That means if the word is breakable, the current break points will be one of the answers.

56. Combination Sum-I
Given a set of candidate numbers (candidates) (without duplicates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.
The same repeated number may be chosen from candidates unlimited number of times
Note:
All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.
Example 1:
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
```
[
  [7],
  [2,2,3]
]
```
Example 2:
Input: candidates = [2,3,5], target = 8,
A solution set is:
```
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

a. Below is the implementation:

```
class Solution {
public:
    // Backtracking is so easy... I just had to do some practice.

    // last argument 'i' is taken to avoid using duplicate sets.
    // This way we will only take new elements in single direction.
    void solve(int target, int current_sum, vector<int>& temp,
vector<vector<int>>& ans,vector<int>& candidates, int i){
        if(current_sum == target){ // Target achieved
            ans.push_back(temp);
            return;
        }

        for(;i<candidates.size();i++) // Start with current element, take each
candidate into consideration
        {
            if(current_sum+candidates[i]<=target){  // Safe move condition
                temp.push_back(candidates[i]);  // push this element
                solve(target, current_sum+candidates[i], temp,ans,candidates,i);
                temp.pop_back(); // erase it
            }
        }
    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> ans;
        vector<int> temp;
        solve(target,0,temp,ans,candidates,0);
        return ans;
    }
};
```

57. Combination Sum II
Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.
Each number in candidates may only be used once in the combination.
Note:
All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.
Example 1:
Input: candidates = [10,1,2,7,6,1,5], target = 8,
A solution set is:

```
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

Example 2:

Input: candidates = [2,5,2,1,2], target = 5,
A solution set is:

```
[
  [1,2,2],
  [5]
]
```

a. This question differs from the previous one because we can't use an element more than once. To avoid that, in solve() function, I have passed the next index of the candidate array instead of the current element.

Now one more thing that there can be duplicates now. In the previous question, we were allowed to use the cuurent element as many time as we wanted and so the candidates were duplicate. But now we can't use the same element. Eg candidates are [1,7,7,1] and target is 8, with previous approach, we will get [1,7][1,7][7,1][7,1] and all four are duplicate. Using set is a bad idea for vectors. Hence we must use something different. Do you remember how we removed duplicates in 3-sum and 4-sum problem? Yes we will use that approach here too. First sort the candidates and then use each candidate only once at a position.
Again take the candidates = [1,1,7,7] and target = 8

push 1, push 1, push 7 is invalid [1,1], backtrack
push 1, push 7 is valid [1,7], backtrack
now remove 7   [1]
now skip the elements till they are 7
since no more elements are there, we can not push, so backtrack
remove 1 []
skip all 1's...
push 7   [7]
push 7 is invalid move so backtrack...
We can't generate any more combinations, so answer is [1,7]

```
Below is the implementation:

class Solution {
public:
    // Backtracking is so easy... I just had to do some practice.
    void solve(int target, int current_sum, vector<int>& temp,
vector<vector<int>>& ans,vector<int>& candidates, int i){
        if(current_sum == target){
            ans.push_back(temp);
            return;
        }

        for(;i<candidates.size();i++)
        {
            if(current_sum+candidates[i]<=target){
                temp.push_back(candidates[i]);
                solve(target, current_sum+candidates[i],
temp,ans,candidates,i+1);
                temp.pop_back();
            }
            while(i+1<candidates.size() && candidates[i]==candidates[i+1])++i;
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(),candidates.end());
        vector<vector<int>> ans;
        vector<int> temp;
        solve(target,0,temp,ans,candidates,0);
        return ans;
    }
};
```

58. Palindrome Partitioning
Given a string s, partition s such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of s.
Example:
Input: "aab"
Output:
```
[
  ["aa","b"],
  ["a","a","b"]
]
```

a. Since we need to print all possibilities, backtracking is again a good choice. However, one more variation of this question is about finding the minimum number of partitions to get all palindrome substrings. That can be done using a recursive strategy similar to chain matrix multiplication.

In this question, the concept is, we will try to use the substrings of length 1, 2, 3 ... |S|. If these strings are palindrome, we will recursively check for the next portion of the string. Whenever we see that a partition perfectly fits, we need to print that solution and backtrack. Also if a partition overfits, we need to terminate and backtrack.

```
class Solution {
public:
    bool isPalindrome(string& s, int l, int r){
        while(l<=r){
            if(s[l++]!=s[r--])
                return false;
        }
        return true;
    }
    void solve(string& s, int l,vector<string>& temp, vector<vector<string>>&
ans){
        if(l==s.length()){
            ans.push_back(temp);
            return;
        }else if(l>s.length()){
            return;
        }

        for(int i=1;i<=s.length();i++){
            if(l+i-1<s.length() && isPalindrome(s,l,l+i-1)){
                temp.push_back(s.substr(l,i));
                solve(s,l+i,temp,ans);
                temp.pop_back();
            }
        }
    }
    vector<vector<string>> partition(string s) {
        vector<vector<string>> ans;
        vector<string> temp;
        solve(s,0,temp,ans);
        return ans;
    }
};
```

59. Subset Sum
Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

a. Use backtracking as follows:

```
class Solution {
public:
       // here i is the index of next element
       // we are taking the element and erasing it while backtracking
    bool solve(int& target, int sum, vector<int>& a, int i){
        if(target==sum){
            return true;
        }
        else if(sum>target) return false;

        for(;i<a.size();i++){
            if(sum+a[i]<=target){
                bool x = solve(target,sum+a[i],a,i+1);

                if(x)
                    return true;
            }
        }
        return false;
    }
    bool canPartition(vector<int>& a) {
       int target=0;
        for(int i=0;i<a.size();i++){
            target+=a[i];

        }
        if (target&1)return false;
        target = target/2;
        return solve(target, 0, a, 0);

    }
};
```

b. Another backtracking solution which is more efficient is as follows:

```
class Solution {
public:
    bool solve(vector<int>& a, int sum, int i) {
        if(i==a.size())return false;
        if(sum == a[i]) return true;
        if(sum < a[i]) return false;
        return solve(a,sum-a[i],i+1) || solve(a,sum,i+1);
    }

    bool canPartition(vector<int>& a) {
       int target=0;
        for(int i=0;i<a.size();i++){
            target+=a[i];

        }
        if (target&1)return false;
        sort(a.rbegin(),a.rend());
        target = target/2;
        return solve(a,target,0);
    }
};
```

c. If we can find a subset having sum == setsum/2, we can solve the problem. To solve that, we need to solve the subset sum problem which states that "Given a set S and target N, is there a subset with sum N in this set?"

To solve that we can use 0-1 knapsack. We take the current element or we don't take. Below is the DP solution:

```
bool canPartition(vector<int>& a) {
        int target=0;
        for(int i=0;i<a.size();i++){
            target+=a[i];
        }

        if (target&1)return false;
        sort(a.rbegin(),a.rend());
        target = target/2;
        vector<bool> dp(target+1,false);
        dp[0] = true;
        for(auto i:a) {
            for(int j=target;j>0;j--) {
                if(i <= j)
                    dp[j] = dp[j] || dp[j-i];
            }
        }
        return dp[target];

    }
```

60. K-th permutation sequence
The set [1,2,3,...,n] contains a total of n! unique permutations.
By listing and labeling all of the permutations in order, we get the following sequence for n = 3:
"123"
"132"
"213"
"231"
"312"
"321"
Given n and k, return the kth permutation sequence.
Note:
Given n will be between 1 and 9 inclusive.
Given k will be between 1 and n! Inclusive.

a. We can generate all possible permutations and return the kth.

```
class Solution:
    def getPermutation(self, n: int, k: int) -> str:
        return "".join(map(str,list(permutations(range(1,n+1)))[k-1]))
```

b. Remember “Excel sheet column number” problem? Our solution will be similar to that.

Observe the following permutations for n=4

```
0  (1, 2, 3, 4)
1  (1, 2, 4, 3)
2  (1, 3, 2, 4)
3  (1, 3, 4, 2)
4  (1, 4, 2, 3)
5  (1, 4, 3, 2)
6  (2, 1, 3, 4)
7  (2, 1, 4, 3)
8  (2, 3, 1, 4)
9  (2, 3, 4, 1)
10 (2, 4, 1, 3)
11 (2, 4, 3, 1)
12 (3, 1, 2, 4)
13 (3, 1, 4, 2)
```

```
14 (3, 2, 1, 4)
15 (3, 2, 4, 1)
16 (3, 4, 1, 2)
17 (3, 4, 2, 1)
18 (4, 1, 2, 3)
19 (4, 1, 3, 2)
20 (4, 2, 1, 3)
21 (4, 2, 3, 1)
22 (4, 3, 1, 2)
23 (4, 3, 2, 1)
```

First position takes each integer 6 times, where 6 is 3!. If we fix the first position, the first position of rest array takes each integer 2 times that is 2!, and similarly 1! for 3rd position.
Suppose we want to know the 17th permutation.
So initially A = [1,2,3,4]
element at first position(0th index) = 17//(3!) = 2nd element ie 3.
So we put 3 at front. A = [3,1,2,4]
Again, we have rest of the array as [1,2,4] and we have generated the 12th permutation .
Now about [1,2,4] we need to generate 17-12 (or 17%(3!))= 5th permutation.
So 5/(2!) = 2, ie element at 2 index = 4
So array becomes [3,4,1,2].
Now for rest [1,2] we need to generate 5%(2!) = 1st permutation
1/(1!) = 1. Hence third postion goes to element at 1st index ie 2 (in [1,2] 1 is at 0th index, 2 is at 1st index)
So array becomes [3,4,2,1], that’s the answer.

Below is the implementation in python3:

```
  def getPermutation(self, n: int, k: int) -> str:
        k-=1
        l = [str(i) for i in range(1,n+1)]
        f = factorial(n-1)
        ans = ""
        while k>0:
            ans += str(l[k//f])
            l.remove(l[k//f])
            k = k%f
            f//=len(l)


        for i in l:
            ans+=str(i)

        return ans
```

b. A recursive solution :

```
class Solution:
    def solve(self,l,k,f):
        if len(l) == 1:
            return str(l[0])
        ans = str(l[k//f])
        l.remove(l[k//f])
        return ans + self.solve(l,k%f,f//len(l))
    def getPermutation(self, n: int, k: int) -> str:
        k-=1
        l = [i for i in range(1,n+1)]
        f = math.factorial(n-1)
        return self.solve(l,k,f)
```

61. Nth root of a number.
This problem has many variations, like:

i. Find nth root of x if root(x,n) is a whole number. Else return -1
ii. Find the closest integer of root(x,n). ie root(4,2) = 2 and root(8,2) = 2.
iii. Given an error measure E, find the root (x,n) given that atmost ±E error is tolerable.

The idea is to use Binary search over an interval [0,x/2].
Code for First:

```
int root(int x, int n){
    if(x==0 || x==1) return x;
    int start=0,end = x/2,mid;
    double m;
    while(start<=end){
        mid = (start+end)/2;
        m = pow(mid , n);

        if(m==x){
            return mid;
        }else if(m>x){
            end = mid-1;
        }else{
            start = mid+1;
        }
    }
    return -1;
}
```

But if we need to return the closest integer, we return the value of ‘end’ instead of -1.

```
int root(int x, int n){
    if(x==0 || x==1) return x;
    int start=0,end = x/2,mid;
    double m;
    while(start<=end){
        mid = (start+end)/2;
        m = pow(mid , n);

        if(m==x){
            return mid;
        }else if(m>x){
            end = mid-1;
        }else{
            start = mid+1;
        }
    }
    return end;
}
```

For the third variation, we need to use double/float datatype.
We have to make some changes in the binary search function too:

```
def root(x,n,e):
    if(x==0 or x==1 or n==1):return x

    start = 0
    end = x/2
    while start<end:
        mid = (start+end)/2
        m = mid**n
        if x-e <= m<= x+e:
            return mid
        elif m>x:
            end = mid   # Not mid-1 because we don’t know what fraction to
reduce
        else:
            start = mid
    return start  # loop breaks when start <= end, that is the most accurate
answer
```

62. Find median in Matrix

Given a matrix with N rows and M columns, with each row sorted in non decreasing order. Your task is to find the median of all elements in this matrix. It is given that N*M is odd(ie median is an integer, not average of two middle elements.)
Example:
Input : 1 3 5
            2 6 9
            3 6 9
Output : Median is 5
If we put all the values in a sorted
array A[] = 1 2 3 3 5 6 6 9 9, middle element is 5

Input: 1 3 4
           2 5 6
           7 8 9
Output: Median is 5
a. The idea is that for a number to be median there should be exactly (n/2) numbers which are less than this number.
So we need to find the max and min of this matrix and apply binary search for a number such that there are exactly r*c/2 numbers smaller than it.

(a) We find max and min of entire matrix by comparing first and last columns (each row is sorted so min max must be one of first and last column).
(b) We use binary search over (min,max).
(c) For mid = (min+max)/2, we count the number of smaller elements ineach row. If we use binary search in each row, we will get the position of mid and that is the count of smaller elements.
(d) If this count is less than r*c/2, we set min = mid+1. Else max = mid-1
(e) The moment count becomes exactly r*c/2, we return it.

Below is the code:

```
# Function to find median in the matrix
def binaryMedian(m, r, d):
    mi = m[0][0]
    mx = 0
    for i in range(r):
        if m[i][0] < mi:
            mi = m[i][0]
        if m[i][d-1] > mx :
            mx =  m[i][d-1]

    desired = (r * d + 1) // 2

    while (mi < mx):
        mid = mi + (mx - mi) // 2
        place = [0];

        # Find count of elements smaller than mid
        for i in range(r):
             j = bisect_right(m[i], mid)  // j = position of m[i] means there
are j smaller elements
```

```
                place[0] = place[0] + j
            if place[0] == desired: return mid
            if place[0] < desired:
                mi = mid + 1
            else:
                mx = mid
        return   mi
```

62a. Given a matrix of size n x m , let’s say you convert the 2-D matrix in linear array. What is the element at index i of this linear array.
a. The matrix is n rows of m elements. So i-th index will point at i/m th row and i%m th column. If we want to search an element in this matrix using binary search,we can do that using the code below:

```
bool searchMatrix(vector<vector<int>>& a, int target) {
        if(a.size()==0 || a[0].size()==0)return false;
        int n = a.size(), m = a[0].size(),i,j,start=0,end = m*n-1,mid;
        while(start<=end){
            mid = (start+end)/2;
            if(a[mid/m][mid%m] == target)return true;
            else if(a[mid/m][mid%m] > target){
                end = mid-1;
            }
            else{
                start = mid+1;
            }
        }
        return false;
    }
```

63. Single Element in a Sorted Array
You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.
Follow up: Your solution should run in O(log n) time and O(1) space.
Example 1:
Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2
Example 2:
Input: nums = [3,3,7,7,10,11,11]
Output: 10

a.Since array is sorted, that means the duplicate values will be together. We can linearly search for the first element that has no neighbours equal to itself. This will take O(N) time.
b. Since exactly one element has no duplicate, there will be odd number of elements in the array. We can use binary search in this problem. In each half if we find the element at mid, congrats, otherwise we will go to the next half where there are odd number of elements(excluding the duplicate element).
Eg [1,1,2,3,3,4,4,8,8], mid = 4th index ie 3, having a dupllicate on left.
We see that left subarray has 3 elements, so our target must be there.
Now [1,1,2,....], mid = 1st index ie 1, having a duplicate on right.
We see that in left 0 elements are there. But in right subarray, 1 element is present and 1 is odd. So we move to right this time.
[..1,2,3,...], mid = 2nd index ie 2 and it has no duplicate neighbours, so it is the answer.

```
int singleNonDuplicate(vector<int>& a) {
        int n = a.size(), start = 0, end = n,mid;
        while(start<=end){
            mid = start + (end - start)/2;
            if(mid>0 && a[mid]==a[mid-1]){
                if((mid-start-1)&1){
                    end = mid-2;
                }else{
                    start = mid+1;
                }
            }else if (mid<n-1 && a[mid] == a[mid+1]){
                if((mid-start)&1){
                    end = mid-1;
                }else{
                    start = mid+2;
                }
            }else{
                return a[mid];
            }
        }
        return 0;
    }
```

64. Search Element in sorted and rotated array.
Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.
Your algorithm's runtime complexity must be in the order of O(log n).
Example 1:
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
Example 2:
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1

a. If the given array is rotated, the first element must be greater than last element and this applies to all subarrays.
To do this in O(logN) time, we can check at middle points whether in the subarray [start:mid], the first element if greater than mid or not. If it is, the array is rotated,but it implies that the rest of the array is sorted. So we can use binary search on the rest of the array. Lets understand with an example:
Initially A = [4,5,6,7,0,1,2] and target = 0
since first > last, array is rotated. So we break it in two parts:
[4,5,6,7] and [0,1,2]. Since first element is smaller than last in [4,5,6,7] , we can apply binary search. Also [0,1,2] is sorted, so applying binary search will return the answer.

Take one more example:
[14,18,2,3,5,7,8,11] target = 8
first>last, so break in two parts.
[14,18,2,3] and [5,7,8,11]. Here first array is rotated, that means second is sorted. So we will apply binary search in that array.
For first array, break it. [14,18] and [2,3] Both are sorted so use binary search.

```
class Solution {
```

```
public:
    int bsearch(vector<int>& a, int l, int r, int target){
        while(l<=r){
            int mid = (l+r)/2;
            if(a[mid]==target)return mid;
            else if(a[mid]>target) r = mid-1;
            else l = mid+1;
        }
        return -1;
    }
    int find(vector<int>&a, int l, int r,int target){
        if(l>r)return -1;
        if(a[l]>a[r]){
            int mid = (l+r)/2;
            int x = find(a,l,mid, target);
            if(x!=-1)return x;
            x = find(a,mid+1,r, target);
            return x;
        }else{
            return bsearch(a,l,r,target);
        }
    }
    int search(vector<int>& a, int target) {
        return find(a,0,a.size()-1,target);
    }
};
```

b. If we can find the minimum element in array in logN time, we can Use two binary searches to solve this problem more easily, without any recursion.
We can actually find the smallest element using binary search.

Let’s observe the array [14,18,2,3,5,7,8,11].
The smallest one will be on right of the rotated part. So lets start the binary search:

mid = 3rd index ie 3. Since 3<11 (the right end), that means(array is strictly increasing) the smallest element is in left subarray(including mid), so we will search in [14,18,2,3].
Ths time middle one is 18.
Since 18>3 that means(the rotated part is also there) smallest one is on the right side, so we search in [2,3]
mid is 2 now and 2<3 so go for left subarray.
[2] Now only one element is left so this is our smallest element.

```
class Solution {
public:
    int bsearch(vector<int>& a, int l, int r, int target){
        while(l<=r){
            int mid = (l+r)/2;
            if(a[mid]==target)return mid;
            else if(a[mid]>target) r = mid-1;
            else l = mid+1;
        }
        return -1;
    }

    int search(vector<int>& a, int target) {
        int lo,hi,mid;
        lo = 0;
        hi = a.size()-1;
        while(lo<hi){   // Find the smallest element
            mid = (lo+hi)/2;
            if(a[mid]>a[hi])lo = mid+1;
            else hi = mid;
        }
        int x = bsearch(a,lo,a.size()-1,target);
        if(x<0){
            return bsearch(a,0,lo-1,target);
        }
        return x;


    }
};
```

65. Kth largest element in two sorted arrays.
Given two sorted arrays nums1 and nums2 of size m and n respectively and an int k. Find the k-th largest element of these arrays.
Note that it is the k-th largest element in the sorted order, not the k-th distinct element.
Example 1:
Input: nums1 = [-2, -1, 3, 5, 6, 8], nums2 = [0, 1, 2, 5, 9], k = 4
Output: 5
Explanation: Union of above arrays will be [-2, -1, 0, 1, 2, 3, 5, 5, 6, 8, 9] and the 4th largest element is 5.
Example 2:
Input: nums1 = [2, 4], nums2 = [6], k = 1
Output: 6
Explanation: union of above arrays will be [2, 4, 6] and the 1st largest element is 6.
You may assume k is always valid, $1 \le k \le m + n$.

a. One straight forward solution is to merge the arrays and return the kth largest element directly.
b. Before solving this one, lets see a similar question 65a.

65a. Find median of two sorted arrays. The size of the arrays may differ.

a. Merge the arrays and return the median. We don’t need to merge the arrays physically but keep a count of how many elements we have merged (Or skipped) while traversing using 2 pointers. When we have skipped exactly (m+n+1)/2 elements, we are at the mid.

b. We can use binary search for the solution. https://www.youtube.com/watch?v=LPFhl65R7ww
Suppose we have two arrays A and B, having size m and n. Here m<=n, if not, we swap the arrays. We will search for the answer in smaller array, considering m available partitions. Suppose we are at midA in any iteration. Since we want exactly (n+m)/2 elements in the left of our partition, we must choose midB = (m+n+1)/2 – midA as the index in array B so that we always balance the arrays.
Why (m+n+1)/2 and not (m+n)/2? Actually it doesn’t matter, using (m+n+1)/2 will keep 1 extra element in left array in case (m+n) is odd, that’s all. Even if we use (m+n)/2 – midA, we will need to check at midB +1st index that will be a little inconvenient.

Lets say A = a b c d
and   B = e f g h i j k
If we are at ‘b’ in array A, that means we partitioned it as [a] [b,c,d].

Hence we must choose (m+n+1)/2 – midA = (4+7+1)/2 – 1 = 5 ie ‘j’ as the partition point in B.
That means B is partitioned as [e,f,g,h,i] [j,k].
That is [a,e,f,g,h,i] in left part and [b,c,d,j,k] in right. We can see that left part is having one extra element since (m+n) is odd.
Lets rewrite it:
[a] [b,c,d]
[e,f,g,h,i] [j,k]
This partition is valid if a<=j and i<=b ie the merged result is sorted.
In case (m+n) is odd, [**a**,e,f,g,h,**i**,b,c,d,j,k] has two mid candidates, a and i, so answer is max(a,i) ie max of what left partitions have at their ends.
Lets make a quick example for even (m+n).
[a]   [b c d]
[g h] [i j]
If (m+n) is even, merging these will result in [**a**,g,**h , *b*** , c , d , ***i***, j]. Here a,h are candidates for left partition’s rightmost element and b,i can be right partition’s leftmost elements. So the answer is avg(max(a,h),min(b,i)).

One edge case we must keep in mind: What if midA/midB is at 0? That means there is nothing in left side, so we must assume that midA-1 points to -∞. Similarly, if midA/midB are at right of array(ie m,m+1,.. and n,n+1,...) that means nothing is present at right so assume that value to be ∞.

Below is the code:

```
class Solution {
public:

    double findMedianSortedArrays(vector<int>& a, vector<int>& b) {
        int i,j,m = a.size(),n = b.size();
        if(m>n){
            return findMedianSortedArrays(b,a);
        }

        bool odd = (m+n)&1;
        if(m==0){
            if(odd)return b[n/2];
            return (double)(b[n/2]+b[n/2-1])/2.0;
        }

        int start = 0, end = m, midx,midy; // end = m because there can be right 
placed partition
        while(start<=end){
            midx = (start+end)/2;   // You land at the first point of the second 
half
            midy = (m+n+1)/2 - midx; // Here too. So consider left partition 
upto the left index ie mid-1 .
            int a_left = midx<=0?INT_MIN:a[midx-1],b_left = midy<=0?
INT_MIN:b[midy-1], a_right = midx>=m?INT_MAX:a[midx], b_right = midy>=n?
INT_MAX:b[midy] ;
            if(a_left<=b_right && b_left<=a_right){
                if(odd){
                    return (double)max(a_left,b_left);
                }else{
                    return (double)(max(a_left,b_left)+min(a_right,
b_right))/2.0;
                }
            }
            else if(a_left>b_right){   // If midx-1 is larger, shift to left
                end = midx-1;
            }else{
                start = midx+1;
            }



        }
        return 0;

    }
};
```

Using the logic discussed above we can find kth largest and kth smallest elements in logN time.

Talking about medians, it's very irritating to write different median formulas for even and odd size arrays. So here is a trick for Python lists(or any other language that support -ve index), *Use (A[N/2] + A[~(N/2)])/2* for both even and odd arrays.

Below is the demo how it works:
[0,1,2,3,4] is odd sized, N = 5. So N/2 = 2 and ~2 = -3 both point to same element.
[0,1,2,3,4,5] is even sized, N = 6. N/2 = 3 and ~3 = -4 , both point to desired elements.

To do the same in a container with positive indices only, Use :
(A[N/2] + A[N/2 - 1 + N%2]) / 2   (Typecast accordingly!)
For even size   ^^^^^ this term will use 0 else it will add 1 to keep it same for odd size.

66. Sliding window median.

Given an array of integers A and window of size K. Your task is to find the median of numbers in the window when we move the window from left to right.

a. One way is to use binary insertion sort to push and pop elements from the window:

```
def medianSlidingWindow(self, nums, k):
        window = sorted(nums[:k])  # O(klogk)
        medians = []
        for a, b in zip(nums, nums[k:] + [0]): # O(N)
            medians.append((window[k//2] + window[~(k//2)]) / 2.0)
            window.remove(a)            # O(K)
            bisect.insort(window, b)    # O(K)
        return medians
```

Time complexity is O(klogk) + O(Nk) ~~ O(Nk).

b. We have to maintain the middle element of the self balanced tree which can insert and remove in O(logK) time. This way we can achieve a faster solution.

```
vector<double> medianSlidingWindow(vector<int>& a, int k) {
        vector<double> medians;

        // Create a sorted Window ~~klogk
        multiset<int> window(a.begin(),a.begin()+k);

        // Put an iterator on k/2-th position~~k
        auto mid = next(window.begin(),k/2);

        for(int i=k; ;i++) { // ~~n
            // One simple way to calculate median in set
            double median =  (
                        (double)(*mid) +
                        *prev(mid,1-k%2)
                        )/2;
            medians.push_back(median);

            // If all medians pushed, return
            if(i == a.size())return medians;

            // Push a[i]
            window.insert(a[i]); //logk

            // If left portion grew, move it to left.
            if(a[i] < *mid){
                mid--;
            }

            // Erase a[i-k]
            // If a[i-k] will be removed from left, restore mid
            // otherwise in other conditions, mid will be at right
place
            // because we will remove the leftmost element using
lowerbound
            if(a[i-k] <= *mid) {
                mid ++;
            }
            window.erase(window.lower_bound(a[i-k]));//logk
        }

    }
```

Time complexity is klogk + k + Nlogk ~~ O(NlogK).

67. Given a number N, check whether N is a power of 2 or not.

a. Check the binary representaion of N. If there is only 1 set bit, return True (O(logN))

b. Right shift N in a loop and check for its parity. The moment it is odd and not 1, return False. (O(logN)).

c. Count the bits in the number using c = log2(N). If we can achieve the same number again using 1<<c, return True. (O(1))

d. Or a better way, if log2(N) is a whole number, return True. If fraction, return False.

e. One more, we can remove the rightmost set bit using N&(N-1), so if doing this results in 0, return True. (O(1)).

For a number of type X = 1000 ie (1)(0)*, X-1 will be 111 ie 1* with one less bit. If we take their bitwise & , the result is 0.

Similarly, lets take a number 110100100.
We will apply N &= (N-1) again and again
110100100 & 110100011 = 110100000 (The rightmost set bit is gone)
110100000 & 110011111 = 110000000
...
Eventually it will become 0

68. Count number of set bits in the given number.
To make it more interesting, you have to count number of set bits for all numbers in range [0,N].
(Note: The time complexities are for single element in the given range)
a. Use the binary representation to count the bits. (O(logN))

b. Right shift the bits and count the rightmost bits (N&1) (O(logN))

c. Use the method above (N&(N-1)). (O(logN))

d. In C++, we can use __builtin_popcount(N) to directly get the answer.

e. Or a clever way to use method c (DP):

```
numberofbits(a) = 1 + numberofbits(a&(a-1))
```

```
class Solution {
public:
    vector<int> countBits(int n) {
        vector<int> ans(n+1,0);
        for(int i=1;i<=n;i++){
            ans[i] = ans[i&(i-1)]+1;
        }
        return ans;
    }
};
```

69. Divide two numbers A/B without *, /, or % operator. (return Quotient)

a. Keep subtracting the number and count the number of times you subtracted. (-.-)
Just for fun, suppose * operator was allowed, we can keep multiplying B with i{1to A} and the moment it becomes larger, return i;

b. We cannot use multiplication but we can simulate a similar operation using left shift operator.
Suppose A = 35, B = 6. A/B = 5

In binary, 35 = 100011 and 6 = 110
We will leftshift 6 using ‘i’, and look for the larget ‘i’ for which 6<<i is less than 35.
i=0 35>6
i=1 35>12
i=2 35>24
i=3 35<48   current = 1<<2 = 4

So that means 6*4 = 24 is the current checkpoint. Now we need to make 35-24 = 9 using 6.
i=0 9>6
i=1 9<12 current = 1<<0 = 1
So answer is 4 + 1 = 5.

```
def divide(a,b):
    sign = False  # ie negative
    sign = ((a<0 and b<0) or (a>0 and b>0))  # If true, makes it
positive
    ## Make them positive
    a = abs(a)
    b = abs(b)
    q = 0
    while a>=b:    # while a is dividable because we
                   # can't divide the smaller number
        for i in range(a):
            if (b<<i) > a:
                q += 1<<(i-1)  # means previous 'i' was smaller
                a -= b<<(i-1)
                break

    # Here for negative numbers I am adding an extra -1
    # because answer doesn't match in python3 but with other
    # languages (like c++), it should match.
    return (1 if sign else -1)*q + (0 if sign else -1)
```

c. (From GFG) As every number can be represented in base 2(0 or 1), represent the quotient in binary form by using shift operator as given below :
Determine the most significant bit in the quotient. This can easily be calculated by iterating on the bit position i from 31 to 1.
Find the first bit for which divisor << i is less than dividend and keep updating the ith bit position for which it is true.
Add the result in temp variable for checking the next position such that (temp + (divisor << i)) is less than dividend.
Return the final answer of quotient after updating with corresponding sign.

```
def divide(dividend, divisor):

    # Calculate sign of divisor
    # i.e., sign will be negative
    # either one of them is negative
    # only iff otherwise it will be
    # positive

    sign = (-1 if((dividend < 0) ^
                  (divisor < 0)) else 1);
    # remove sign of operands
    dividend = abs(dividend);
    divisor = abs(divisor);

    # Initialize
    # the quotient
    quotient = 0;
    temp = 0;

    # test down from the highest
    # bit and accumulate the
    # tentative value for valid bit
    for i in range(31, -1, -1):
        if (temp + (divisor << i) <= dividend):
            temp += divisor << i;
            quotient |= 1 << i;

    return sign * quotient;
```

70. Power set.
Given a set S, print all the sets in the power set of S.
Eg Input: S = {1,2,3}
Output: PS = {},{1},{2},{3},{1,2},{2,3},{1,3},{1,2,3}

a. We can use the backtracking solution of subset sum here. Print every possibility, either take the current element or leave it.

b. Since this question is from bit manipulation section, lets add the solution using bits. Observe the pattern of binary numbers from 0 to 7:
000
001
010
011
100
101
110
111

Each number represents a set of power set. If poistion is 1, take that element, else don’t take.

```
n = int(input())
a = [i+1 for i in range(n)]
ans = []
for i in range(1<<n):
      x = bin(i)[2:]
      x = (n-len(x))*’0’ + x
      l=[]
      for j in range(len(x)):
            if x[j]=='1':
                  l.append(a[j])
      ans.append(l)
print(ans)
```

71. Find MSB of a given number.

a. Use the binary representation. O(logN)

b. Use $\log_2 n$ to count total bits. Return the count or 1<<count whatever needed. O(1)

c. Set all of the unset bits, add 1 and return ans/2. O(logN) or O(1)
Eg the number is 1001011 in binary
To set all bits for an 8 byte integer:

```
    n|=n>>1
    n|=n>>2
    n|=n>>4
    n|=n>>8
    ...
    n|=n>>32.
```

Convert it to 1111111,
add 1, make it 10000000,
divide by 2 or right shift, make it 1000000. This is the answer.

72. Find the square of a number without using * or / operator.

a. Use '+' to simulate multiplication and multiply N and N by adding N to itself N times. O(N)

b.

If n is even, it can be written as
$n = 2*x$
$n^2 = (2*x)^2 = 4*x^2$
If n is odd, it can be written as
$n = 2*x + 1$
$n^2 = (2*x + 1)^2 = 4*x^2 + 4*x + 1$

We don't need to multiply with 4, or divide by 2 because we can do that using left and right shift operators.
We can recursively find these values easily.

```
def square(n):
   if n==0:return 0
   if n<0:n = -n

   x = n>>1

   if n&1:
      return (square(x)<<2) + (x<<2) + 1
   else:
      return (square(x)<<2)
```

73. Implement Stack and Queue.
a. Implementing stack is easy using array. So we will implement it using linked list, to rectify the overflow error.
Using array: Implementation is easy, fixed size of array is used. We must handle stackoverflow condition explicitly. There can be wastage of memory.

Using Dynamic memory allocation: Don’t need to explicitly handle overflow error. We use only the required amount of memory so there is no wastage.

Pseudo code:

```
initially:
     top = null
push( value):
    stack_node = new stack_node()
    stack_node.data = value
    stack_node.next = top
    top = stack_node

pop():
    if top == null:
        throw ‘Stack Empty’
    temp = top
    top = top.next
    return temp.data
```

Just like stack, we can implement Queue using array and DMA. Using Array is more complex in case of Queue.
We need two functions enqueue(value) to insert data from the end and deqeue() to pop data from the front.
Using array: With linear array, we can get into a condition where memory is available but we cannot use it.(Tail is at the end). We can overcome this by using Modulus operation.

Using Array
Pseudo Code:

```
initially:
    N = from_User_input()
    queue[N]
    rear = 0
    front = 0
enqueue(value):
    if (rear+1)%N == front:
        throw “Queue Full”
    queue[rear] = value
    rear = (rear + 1)%N
dequeue():
    if front == rear:
        throw “Queue Empty”
    ret = queue[front]
    front = (front + 1)%N
    return ret
```

Using DMA:
Pseudo Code

```
initially:
    front = null
    rear = null
enqueue(value):
    queue_node = new queue_node()
```

```
        queue_node.data = value
        queue_node.next = null
        rear.next = queue_node
        rear = rear.next
dequeue():
        if front == null:
                return "Queue is Empty"
        temp = front.data
        front = front.next
        return temp
```

74. BFS
Read here if needed...

75. Implement Stack Using Queue

a. Idea is simple, we can use two Queues to implement a stack.
Push operation is same for both, we have to append a new value at the rear end.
But for pop operation, the queue doesn't allow us to remove the rear element.
But we can temporarily push the entire queue, except the rear element into the second queue(Which is now our current queue) and return the last element that is left.
Below is the Python implementation:

```python
class MyStack:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.current_queue = deque()
        self.temp_queue = deque()

    def push(self, x: int) -> None:
        """
        Push element x onto stack.
        """
        self.current_queue.append(x)


    def pop(self) -> int:
        """
        Removes the element on top of the stack and returns that 
element.
        """
        n = len(self.current_queue)
        for i in range(n-1):
            self.temp_queue.append(self.current_queue.popleft())
        ret = self.current_queue.popleft()
        self.current_queue, self.temp_queue = self.temp_queue, 
self.current_queue
        return ret
    def top(self) -> int:
        """
        Get the top element.
        """
        return self.current_queue[-1]


    def empty(self) -> bool:
        """
        Returns whether the stack is empty.
        """
        if len(self.current_queue)==0:
            return True
        return False
```

And implementation in C++:

```
class MyStack {
public:
    queue<int> *current, *temp;
    int length;
    /** Initialize your data structure here. */
    MyStack() {
        length = 0;
        current = new queue<int>();
        temp = new queue<int>();
    }

    /** Push element x onto stack. */
    void push(int x) {
        current->push(x);
        ++length;
    }

    /** Removes the element on top of the stack and returns that element. */
    int pop() {
        for(int i=0;i<length-1;i++){
            temp->push(current->front());
            current->pop();
        }
        int ret = current->front();
        current->pop();
        length--;
        queue<int> *t = current;
        current = temp;
        temp = t;
        return ret;
    }

    /** Get the top element. */
    int top() {
        for(int i=0;i<length-1;i++){
            temp->push(current->front());
            current->pop();
        }
        int ret = current->front();
        current->pop();
        temp->push(ret);
        queue<int> *t = current;
        current = temp;
        temp = t;
        return ret;
    }

    /** Returns whether the stack is empty. */
    bool empty() {
        return length==0;
    }
};
```

76. Implement Queue Using Stack.

a. We need to implement enqueue, dequeue functions using stack as the primary container. Just like previous one, we can easily implement push/enqueue. For dequeue/pop operation, we can use two stacks.

Suppose currently our main stack and temp looks like this:
[1,2,3,4,5] where 5 is at the top.
For dequeue operation, we need to return 1. So in the other stack, we can push each value popped by this main stack:
[ ], [5,4,3,2,1].
Now return the top of stack and pop it from our temporary stack.
[ ], [5,4,3,2]
Now before terminating, we need to rebuild our main stack. Pop each value in main stack back.
[2,3,4,5]

Below is the python implementation:

```
class MyQueue:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.stack = [] # will use as stack
        self.temp = []  # temp stack

    def push(self, x: int) -> None:
        """
        Push element x to the back of queue.
        """
        self.stack.append(x)


    def pop(self) -> int:
        """
        Removes the element from in front of queue and returns
that element.
        """
        while len(self.stack)!=0:
            self.temp.append(self.stack.pop())
        ret = self.temp.pop() # temp.top()
        while len(self.temp)!=0:
            self.stack.append(self.temp.pop())
        return ret



    def peek(self) -> int:
        """
        Get the front element.
        """
        while len(self.stack)!=0:
            self.temp.append(self.stack.pop())
        ret = self.temp.pop() # temp.top()
        self.temp.append(ret)
        while len(self.temp)!=0:
            self.stack.append(self.temp.pop())
        return ret


    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        return len(self.stack)==0
```

77. Balanced Parantheses:
Given a string S consisting of "() {} []", check whether parantheses are balanced or not.
Eg ()[{(){}}] is balanced but (){[}] is not balanced.

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        for i in s:
            if len(stack)>0 and i in ")]}":
                if i == ')' and stack[-1] == '(':
                    stack.pop()
                elif i == '}' and stack[-1] == '{':
                    stack.pop()
                elif i == ']' and stack[-1] == '[':
                    stack.pop()
                else:
                    stack.append(i)
            else:
                stack.append(i)
        return len(stack)==0
```

78. Next greater Element.
Given an array of all unique elements, for each element you have to find the first element greater than the current element in right side of it. (-1 if there is no element greater that this element.)
Eg
Input: [1,6,5,4,2,3,7,0] Output:[6,7,7,7,3,7,-1,-1]
a. A simple solution is to traverse the entire array for each element. It will take O(N^2) time.

b. We can also use stack to solve this problem.

We have to push all the elements that are in decreasing order into the stack. When we get a larger element than stack top, we pop stack, setting this element the greater of stack top if it is greater. We repeat this process and if at the end there are elements in the stack, there is no greater element for them so assign -1 for them.

Push the first element to stack.

- Pick rest of the elements one by one and follow the following steps in loop.

  1.Mark the current element as next.

  2.If stack is not empty, compare top element of stack with next.

  3.If next is greater than the top element,Pop element from stack. next is the next greater element for the popped element.

  4.Keep popping from the stack while the popped element is smaller thannext.nextbecomes the next greater element for all such popped elements

- Finally, push the next in the stack.
- After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

Demonstration:

[1,6,5,4,2,3,7,0]

stack	ans = []	
[]	[]	
[1]	[]	push 1
[6]	[1:6]	current = 6, pop 1
[6,5,4,2]	[1:6]	push 6,5,4,2 , we cannot pop anything
[6,5,4,3]	[1:6,2:3]	current = 3, pop 2, push 3
[7]	[1:6,2:3,3:7,4:7,5:7,6:7]	pop all the elements, all are smaller. Push 7
[7,0]	[1:6,2:3,3:7,4:7,5:7,6:7]	push 0
[]	[1:6,2:3,3:7,4:7,5:7,6:7,7:-1,0:-1]	for rest (7 and 0), put -1 as their ans.

Code for the same:

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& a,
vector<int>& b) {
```

```
            unordered_map<int , int> mp;
            stack<int> stk;
            for(auto i:b){
                if(!stk.empty()){
                    while(!stk.empty() && stk.top()<i){
                        mp[stk.top()] = i;
                        stk.pop();
                    }
                }
                stk.push(i);
            }
            while(!stk.empty()){
                mp[stk.top()] = -1;
                stk.pop();
            }
            for(int i=0;i<a.size();++i){
                a[i] = mp[a[i]];
            }
            return a;
        }
    };
```

79. Next Smaller Element

a. We can solve this question using the same concept as 78, we just need to pop elements when we find a smaller element.

80. LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a positive capacity.

a. A simple solution is using Queue/List with a hashmap. Queue to maintain the order of recently used items and hasmap to quickly access the values :

```
class LRUCache:

    def __init__(self, capacity: int):
        self.ru = deque() #recently used
        self.cap = capacity # capacity
        self.data = dict()

    def get(self, key: int) -> int:
        if key in self.data:
            self.ru.remove(key)
            self.ru.append(key)
            return self.data[key]
        return -1

    def put(self, key: int, value: int) -> None:
        if len(self.data)<self.cap:
            if key in self.data:
                self.ru.remove(key)
            self.ru.append(key)
            self.data[key] = value

        else:
            if key in self.data:
                self.ru.remove(key)
                self.ru.append(key)
                self.data[key] = value
            else:
                x = self.ru.popleft()
                del self.data[x]
                self.ru.append(key)
                self.data[key] = value
```

//

b. The problem is removing the item from queue and putting it to the rear again. We have several options like using a hashmap to refer to the item to the queue.

Challenge: Write this implementation with O(1) time complexity. Maybe creating your own data structure will help.

//

I have implemented LRU once again using Doubly Linked List and unordered_map (Hashtable). Idea is to use the map to quickly find the node in the Doubly linked List (DLL) just like array indexing, and use DLL's fast insertion deletion. Overall everything takes O(1) time. But this solution is not space efficient.

Below is the code: Note that only `LRUCache(int capacity)`,`int get(int key)`, `void put(int key, int value)` should be implemented. Rest are for insertion deletion in DLL.

```
class DLL{
public:
    int key,value;
    DLL *left, *right;
    DLL(){
        key = value=-1;
        left = right = NULL;
    }
    DLL(int key,int value){
        this->key = key;
        this->value = value;
        left = right = NULL;
    }

};
class LRUCache {
public:
    int capacity;
    int size;
    DLL *head,*tail;
    unordered_map<int,DLL*> mp;
    LRUCache(int capacity) {
        this->capacity = capacity;
        this->size = 0;
        this->head = NULL;
        this->tail = head;
    }

    int get(int key) {
        if(mp.find(key)!=mp.end()){
            DLL* node = mp[key];
            if(node!=head){
                popnode(node);
                appendtotop(node);
            }
            // cout<<node->value<<endl;
            return node->value;
        }
        // cout<<-1<<endl;
        return -1;
    }

    void put(int key, int value) {
        // cout<<capacity<<" "<<size<<" "<<head<<endl;
        if(mp.find(key)!=mp.end()) {
            DLL* node = mp[key];
            node->value = value;
            if(node!=head){
            popnode(node);
            appendtotop(node);
            }
        } else{
            insertnew(key,value);
        }
    }

    void insertnew(int key, int value){
        if(size==capacity){
            mp.erase(tail->key);
            popnode(tail);

            size--;
        }
        DLL* node;
        node = new DLL(key, value);
        if(head!=NULL){
            head->left = node;
```

```
            node->right = head;
        }
        else{
            tail = node;
        }
        head = node;
        size++;
        mp[key] = node;
    }

    void appendtotop(DLL* node){
        head->left = node;
        node->right = head;
        node->left = NULL;
        head = node;
    }

    void popnode(DLL* node){
        if(node == tail){
            DLL* temp = tail;
            tail = tail->left;
            temp->right = NULL;
            temp->left = NULL;
        } else{
            node->left->right = node->right;
            node->right->left = node->left;
        }

    }


};
```

81. Largest Rectangle in Histogram.

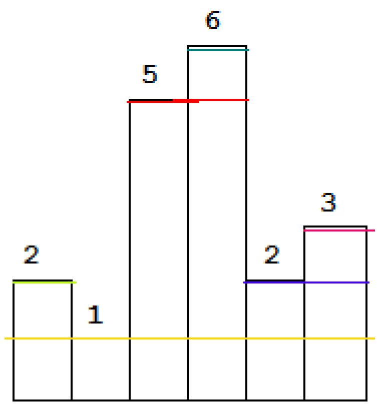
Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Example:

Input: [2,1,5,6,2,3]

Output: 10

a. We can solve this problem by considering each bar as minimum height and extending it as far as possible in left and right directions.



We can see that largest area is obtained by considering 5 as the shortest bar.
Take the current bar and start calculating area. For each i, area is max of area and i*minsofar.

This approach is of O(N^2).

b. An alternative approach is to use divide and conquer. Choose the smallest bar, and check three areas: 1. Area of A[min]*(r-l+1) 2. Max area in range (l,min-1) 3. Max area in range(min+1,r).

We have to return the answer of MaxRectangle(A , 0 , n-1).
To find minimum in range L,R, We need to use a data structure for RMQ, eg segment tree.{ O(N) to build and log(N) per query}

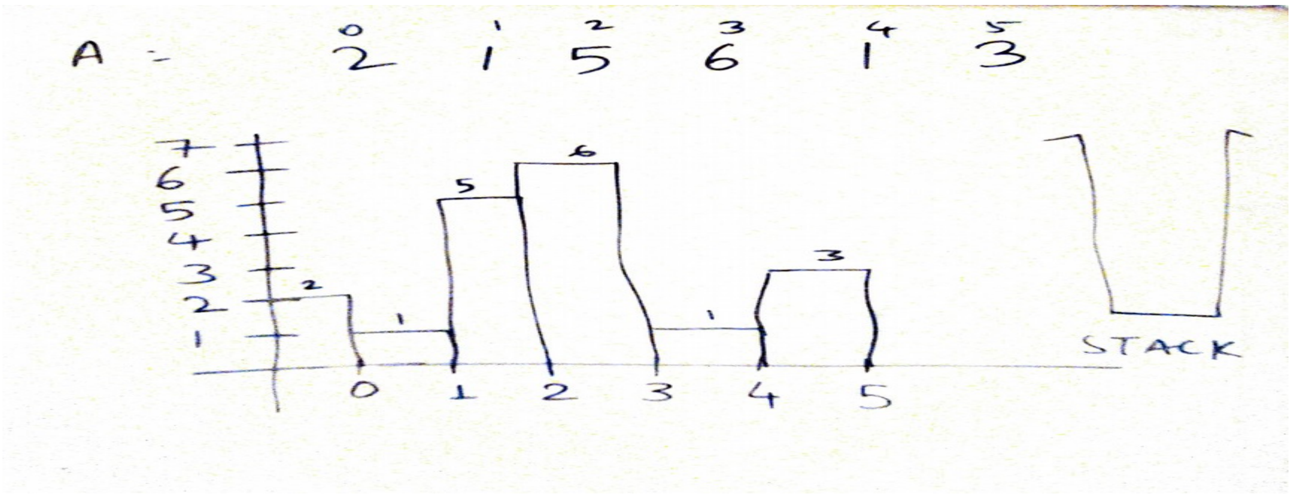
Then this algorithm will run just like Quick sort (depends on where we find the min). So in Best case, complexity is O(NlogN) and O(N^2) in worst case when we find min at left or right.

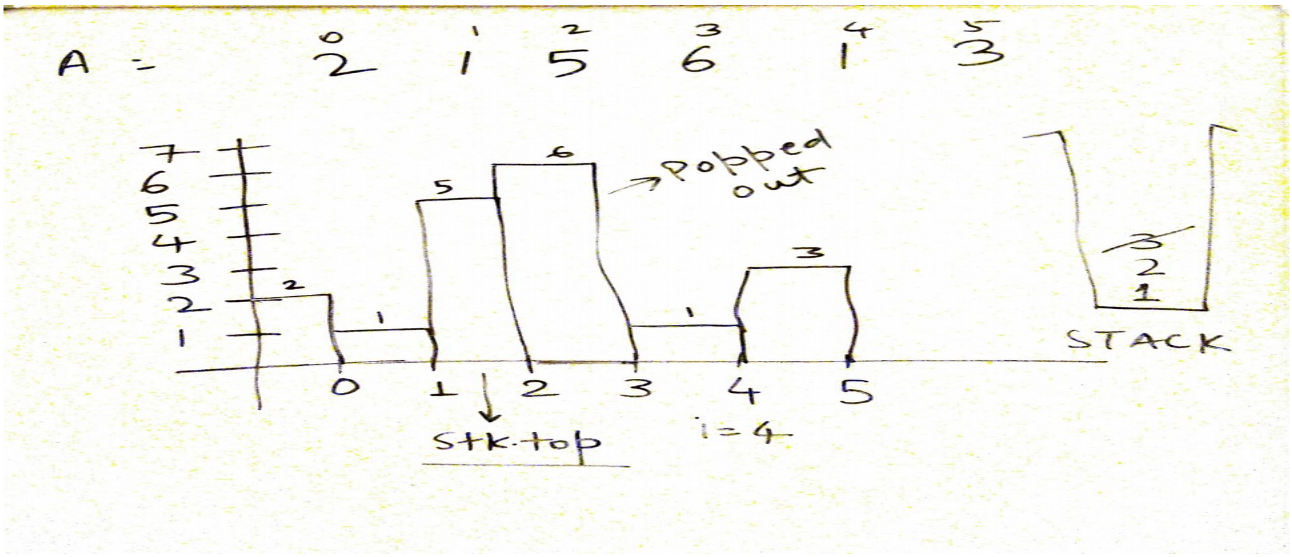
c. The best way to solve this problem is to use a stack to keep track of left bar smaller than or equal to current bar.
The reason is, suppose a bar is too large, it will not contribute in answer if we take the whole bar. But the bottom portion will always be there, even for the smaller bars.
Just like that, the smaller bars will not be available for larger bars so we can assume that larger bars are bars with equal size for the smaller bars.

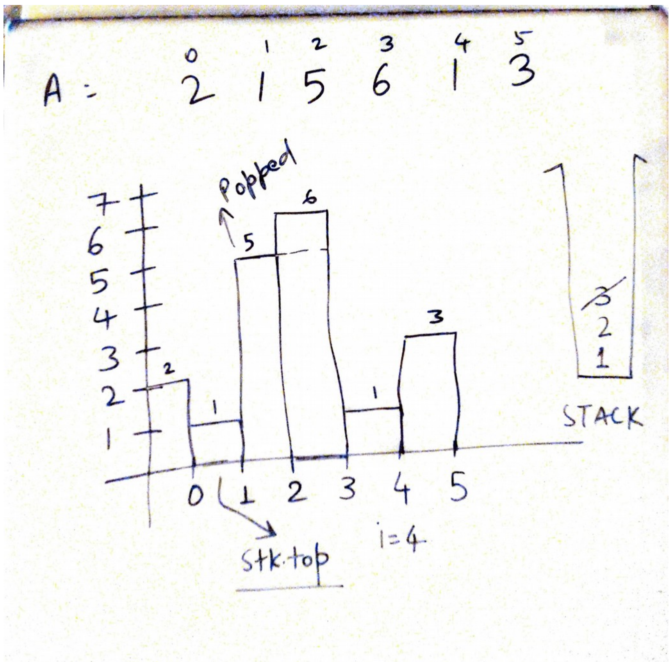
If stack is empty or a[i] > a[stack.top], push the index of current bar.
Other wise, keep popping the larger bars.

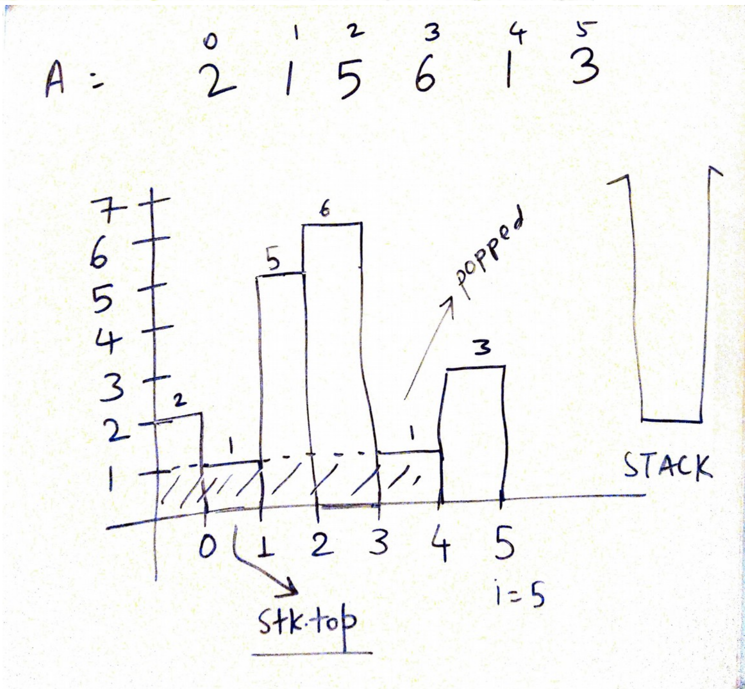
When popping a bar: For a bar in stack, we can say that it is greater than/ equal to all the bars below it and it is also greater than the incoming bar. It means when it was pushed on the stack, it may have removed the larger bars. Also the incoming bar a[i] may have removed the larger bars that

would have been in the stack before, above the current top. That means, considering the current top as smallest bar, we can say there are exaclty {index of top} – {index of next to top} bars that are greater than a[top]. Also there are exactly {i} – {index of top} bars in the right side. So the area formed by this is height of bar ie a[i] multiplied by all the bars in between the bar in stack and i-th bar(excluding both). We can check the number easily.
When we are done pushing, and reached at the end, if still there are bars in stack, that means there are bars in increasing order. So similarly we can calculate their areas, considering i = n(because all bars are greater in right side.)









Below is the code:

```
int largestRectangleArea(vector<int>& a) {
        stack<int> stk;
        int i,n=a.size(),area=0,tp,temp;
        for(i=0;i<n;i++){
            if(stk.empty() || a[i]>=a[stk.top()]){
                stk.push(i);

            }else{
                while(!stk.empty() && a[i]<a[stk.top()]){
                    tp = stk.top();
                    stk.pop();
                    if(stk.empty()){ // It's the smallest bar, so calculate area
                                    // from the start till this bar.
                        area = max(area , a[tp]*i);
                    }else{ // There is one or more smaller bars in stack, so take the
closest
                        // one ie current top. Calculate area from that bar upto ith
bar,
                        //  excluding both.
                        area = max(area , a[tp]*(i - stk.top() - 1));
                    }
                }
                stk.push(i);
            }
        }

        while(!stk.empty()){
            tp = stk.top();
                    stk.pop();
                    if(stk.empty()){ // It's the smallest bar, so calculate area
                                    // from the start till i-1st bar.
                        area = max(area , a[tp]*i);
                    }else{ // There is one (or more) smaller bar in stack, so take the
closest
                        // one ie current top. Calculate area from that bar upto ith
bar,
                        //  excluding both.
                        area = max(area , a[tp]*(i - stk.top() - 1));
                    }
        }
        return area;

    }
```

82. Sliding Window Maximum.

Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

a. Use brute force, O(N^2).

b. We need to keep the window in sorted order. To do the same, we can use a multiset(because we can have multiple values) . Insertion of a[i] and deletion of a[i-k] will take log(N) time. So overall time complexity is O(NlogN).

```
vector<int> maxSlidingWindow(vector<int>& a, int k) {
        multiset<int> s;
        vector<int> ans;
        int i,n=a.size();
        for(i=0;i<k;i++){
            s.insert(a[i]);
        }
        ans.push_back(*(--s.end()));
        for(i=k;i<n;i++){
            s.erase(s.find(a[i-k]));
            s.insert(a[i]);
            ans.push_back(*(--s.end()));
        }
        return ans;
    }
```

c. The above solution is slow because of the slow insertion and deletion. We should observe a few points about the given problem before leading to a new solution.

i. Suppose we are at index i and it is the maximum of current window. We can say that upto next k elements, if there is no element greater than current element, it will remain the max.
ii. Also all the elements that are smaller than this value, in range i-k to i, have no meaning now. So it is better to remove them.
iii. So basically for the current ‘i’, we can remove all the elements smaller than a[i] that are in the left of i.
iv. Now if a new greater element comes afterwards, it will empty the container and will become the largest element.
v. When it goes outside the window, the next max will still be inside the container .

To do all that, we need to maintain every thing inside a linear data structure that keeps all the elements in sorted order. For that we can use a doubly ended queue which inserts and removes in O(1) at both ends. Since we are maintaining sorted order, we need to insert/remove at the end only.

Consider the list below:
[1,3,-1,-3,5,3,6,7] k = 3
We will use the index of these elements so that we can track which element is going outside the window. Note the front element in the de-que.

Elements	De-queue
Ai = 1	[1]
Ai = 3	[3] // remove all smaller from back
Ai = -1	[3,-1] // 1 will move out before 3, so it won’t matter – window starts here
Ai = -3	[3,-1,-3]
Ai = 5	[5]
Ai = 3	[5,3]
Ai = 6	[6]
Ai = 7	[7]

We may think that there can be too many deletions in one iteration, so complexity could be O(N*k) but overall N elements are inserted in the queue so there is no chance that we remove more that N elements. So complexity remains O(N) only.

```
vector<int> maxSlidingWindow(vector<int>& a, int k) {
        vector<int> ans;
        int i,n=a.size();
        /*
        dqueue https://www.geeksforgeeks.org/deque-cpp-stl/
        */

        deque<int> dq;
        for(i=0;i<k;i++){
            while(!dq.empty() && a[i] > a[dq.back()] )dq.pop_back();
            dq.push_back(i);
        }
        ans.push_back(a[dq.front()]);

        for(i=k;i<n;i++){
            if(!dq.empty() && i - k==dq.front())dq.pop_front();
            while(!dq.empty() && a[i] >a[dq.back()] )dq.pop_back();
            dq.push_back(i);
            ans.push_back(a[dq.front()]);
        }

        return ans;
    }
```

83. Implement Min Stack

You have to design a stack with performs push(), pop(), top() and getMin() operation all in constant time.

a. Use another stack in parallel which will get only minimum-so-far elements get pushed. When we pop the element from the main stack, we should remove it from other stack too.

```
class MinStack {
public:
    /** initialize your data structure here. */
    stack<int> a, b; // a is the main stack and b is the min stack
    MinStack() {

    }

    void push(int x) {
        a.push(x);
        if (b.empty() || x<=b.top()){
            b.push(x);
        }
    }

    void pop() {
        if (a.top() == b.top()){
            b.pop();
        }
        a.pop();
    }

    int top() {
        return a.top();
    }

    int getMin() {
        return b.top();
    }
};
```

b. One other way is to keep stack element and minsofar together as a pair. Method is easy but it will take more space even in best case.

```
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.l = list()

    def push(self, x: int) -> None:
        self.l.append([x , min(x,self.getMin())])

    def pop(self) -> None:
        self.l.pop()


    def top(self) -> int:
        return self.l[-1][0]

    def getMin(self) -> int:
        return self.l[-1][1] if self.l else float('+inf')
```

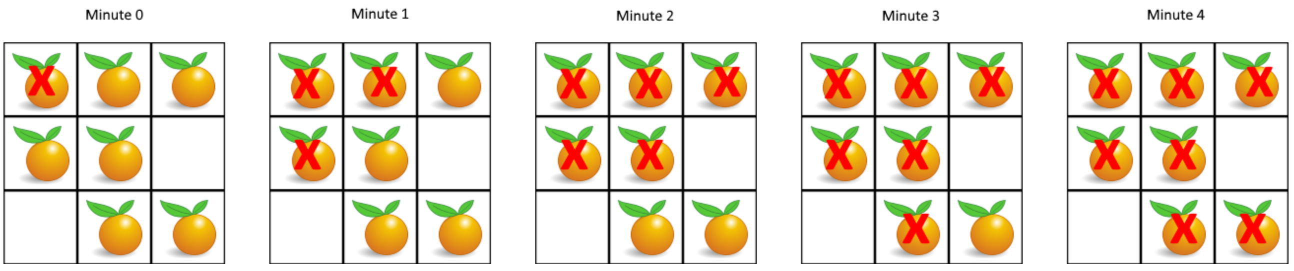
84. Rotten Oranges.

In a given grid, each cell can have one of three values:

- the value 0 representing an empty cell;
- the value 1 representing a fresh orange;
- the value 2 representing a rotten orange.

Every minute, any fresh orange that is adjacent (4-directionally) to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1 instead.



a. Before starting, lets consider few points:

- There can be one or more rotten oranges and it is possible that one rotten orange spreads more rapidly than the others. In such case we need to consider the fastest time.
- If we individually calculate the times for each rotten orange, that might give us a case where a fresh orange is left, so tracking it will be an obstacle.
- If we apply BFS for this problem, we must use all the rotten oranges simultaneously because if we don't do that, it will result in wrong answers for first point.

So the best way is to apply BFS. Create a queue for nodes containing row,column and time. Push each cell with rotten orange initially. Then use BFS until there is even a single fresh orange in any direction and mark them rotten or -1(anything).
After that the fresh oranges that are unreachable, will be left. If there are some, return -1.

```
class Solution {
public:
    struct node{
        int i,j,time;
    };
    int orangesRotting(vector<vector<int>>& a) {
        if(a.size()==0)return 0;
        int i,j,n = a.size(), m = a[0].size(),ans=0;
        bool oneishere = false;
        queue<node> q;
        for(i=0;i<n;i++){
            for(j=0;j<m;j++){
                if(a[i][j]==2){
                    q.push({i,j,0});
                }
            }
        }
        while(!q.empty()){
            node p = q.front();
            q.pop();
            ans = max(ans,p.time);
            if(p.i-1>=0 && a[p.i-1][p.j]==1){
                a[p.i-1][p.j]=-1;
                q.push({p.i-1,p.j,p.time+1});
            }
            if(p.i+1<n && a[p.i+1][p.j]==1){
                a[p.i+1][p.j]=-1;
                q.push({p.i+1,p.j,p.time+1});
            }
            if(p.j-1>=0 && a[p.i][p.j-1]==1){
                a[p.i][p.j-1]=-1;
                q.push({p.i,p.j-1,p.time+1});
            }
            if(p.j+1<m && a[p.i][p.j+1]==1){
                a[p.i][p.j+1]=-1;
                q.push({p.i,p.j+1,p.time+1});
            }

        }
        for(i=0;i<n;i++)
            for(j=0;j<m;j++)
                if(a[i][j]==1)
                    return -1;
        return ans;
    }
};
```

85. Reverse the string word wise.
Given a sentence in the form of a single string, reverse the string keeping the words as they are.
Eg input = “good morning folks”
   output = “folks morning good”

a. Using python we can do it easily:

```
class Solution:
    def reverseWords(self, s: str) -> str:
        return " ".join(reversed(s.split()))
```

b. Or a proper implementation in C++

```
string reverseWords(string s) {
        int i=0,j=0,n=s.length();
        string ans="",temp="";
        vector<string> v;
        for(i=0;i<n;i++){
            if(s[i]!=' '){
                temp += s[i];
            }
            else{
                if(temp!="")
                v.push_back(temp);
                temp = "";
            }
        }
        if(temp!="")
                v.push_back(temp);
        if(v.size()){
        ans = v[v.size()-1];
        v.pop_back();
        }
        for(auto i = v.rbegin();i!=v.rend();++i){
            ans+=" "+*i;
        }
        return ans;
```

```
    }
```

86. Longest Palindromic Substring

In a given String S, find the longest palindromic substring. If there are many, print any one.

a. Palindromic strings can be of even or odd length. We can fix centers for odd and even length and find : To what extent is this string a palindrome in both directions. We have to do this seperately for even and odd lengths.
This approach takes O(N^2) time and O(1) for space if we store only index and length, not the entire substring.

```
string longestPalindrome(string s) {
        int maxlen=0, i=0,j=0,n = s.length(),l;
        // For odd length
        for(i=0;i<n;i++){
            for(j=0;j<=n/2+1;j++){
                if(!(i-j>=0 && i+j<n && s[i-j]==s[i+j])){
                    if (2*(j-1)+1>maxlen){
                    maxlen = 2*(j-1)+1;
                    l = i-j+1;
                    }
                    break;
                }
            }
        }
        // For even length
        for(i=0;i<n;i++){
            for(j=1;j<=n/2+1;j++){

                if(!(i-j+1>=0 && i+j<n && s[i-j+1]==s[i+j])){
                    if(2*(j-1)>maxlen){
                    maxlen = 2*(j-1);
                    l=i-j+2;
                    }
                    break;
                }

            }
        }
        return s.substr(l,maxlen);
    }
```

b. A better way is to skip the duplicates. This way we don’t need to check for odd/even length separately, because when we skip duplicate, it will also consider the even length palindromes. See the code below.

```
string longestPalindrome(string s) {
        int maxlen=0, i=0,j=0,n = s.length(),l,k;

        // j = left, k = right.
        // maxlen is length of substring
        // l is the start index of our answer.

        for(i=0;i<n;){ // Iterate thourgh the string
            j=k=i; // initially set all same
            while(k<n-1 && s[k]==s[k+1])k++;  // Move the right pointer till duplicate
            i = k+1; // Set i to its next. If odd it will do i++ simply
            while(j>0 && k<n-1 && s[j-1]==s[k+1]){ // Now match left and right
                j--;k++;
            }
            if(k-j+1>maxlen){ /// If new length is greater, update
                maxlen = k-j+1;
                l = j;
            }
        }

        return s.substr(l,maxlen); // return the substring
    }
```

This approach is still O(N^2) but it is better and cleaner than previous approach.

c. Maybe using manacher's algorithm will speed up the solution.
   https://cp-algorithms.com/string/manacher.html

So let's first see the steps of manacher's algorithm then we will see how it works.

i. We want to find both even and odd length palindromes. So lets make a new string with a special symbol in between each character. So if s = "abc", our new string will be "#a#b#c#" where "#" is our special symbol for now.
ii. Note that we have doubled the string length so at the end we will map everything back.
iii. Initialise l = 0 and r = -1 . These two pointers will be used throughout the iteration to store our rightmost palindrome.
iv. Now we need one more variable K. K is the length of one side of our palindrome. Since there will be only odd length palindromes, we can say K is half length of the palindrome. We will start checking for palindrome starting from i-K and i+K
v. For each i (i is the new center), if i>r, K = 1, ie start from 0 length, otherwise K is the min of length left upto the r, and the K for mirror center.
vi. Calculate d[i], the length of this palindrome and update l and r if r>previous r.

We can map these lengths back to the original strings by taking their half. If it is guarenteed that palindromes are of odd length, we don't need to maipulate the main string.

Now why this works?

So for i>r, we are manually checking for palindrome, so it's clear.
For i<=r, there is the mirror palindrome about the middle character of current palindrome.
Lets observe it:
"abcdedcba"
About the character 'e', the strings are 'abcd' and 'dcba' which are exact mirror. What if we put a palindrome there?
Lets try:
'xabaeabax'
You can see that strings are still mirror images but now the palindrome around character 'b' is exact same as mirror image.
This way we can guarentee that if the length of mirror image doesn't cross 'r', the palindromic length of palindrome around center 'i' is same as that. Otherwise we are maunally checking it.

Now the question is why is it linear? Actually, we can notice that every iteration of this algorithm makes r increase by one. Also r cannot be decreased during the algorithm. So, this algorithm will make O(n) iterations in total.

Below is the detailed code:

```
string longestPalindrome(string s) {
            int n=s.length(),N = 2*n+1 , d[N];
```

```
        /*Let's prepare the string.. Interweave '#' symbol*/
        string S = "#";
        for(int i=0;i<n;i++){
            S += s.substr(i,1)+"#";
        }

        /*Let's calculate the length*/
        for(int i=0,l=0,r=-1;i<N;i++) {
            /* Computing k. k is the value upto which our current center can extend
            being a palindrome. If i>r, we start from k=1. If i<r, we have to start
            from the minimum of mirror value and r-i+1 (because we don't know if length
            at mirror position is valid outside r or not)
            */
            int k = (i>r)? 1: min(d[l+r-i]/2 , r-i+1);

            /* Now we will expand our current palindrome*/
            while(0<=i-k && i+k < N && S[i-k] == S[i+k]){
                ++k;
            }



            /* Note that k is 1 greater than what it should be.*/
            --k;

            /* Now assign the length*/
            d[i] = 2*k+1;



            /*Update l and r*/
            if(i+k > r){
                l = i-k;
                r = i+k;
            }

        }

        /* We have the length so we can also print the substring*/

        int maxlen=0,maxstart = 0;
        for(int i=0;i<N;i++){
            if(d[i]/2>maxlen){
                maxlen = d[i]/2;
                maxstart = i/2 - maxlen/2;
            }
        }

        return s.substr(maxstart, maxlen);
    }
```

87. Convert integers to roman numbers and vice versa.

87a.Given an integer, represent it in Roman numbers.
a. This question is simple, we just need to keep in mind that which mappings must we make ourselves. The mapping is given below but remember that we need to make combinations with nearest multiples of 10, ie we don't need to combine V, L, D (5,50,500). But we need to combine I,X,C (1,10,100) only. For reference the list is given below.

```
def intToRoman(self, num: int) -> str:
        values = [ 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 ]
        numerals = [ "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V",
"IV", "I" ]
        d = dict(zip(values,numerals))
        ans = ""
        for x in d.keys(): # In decreasing order.
            ans+=d[x]*(num//x)
            num-=x*(num//x)
        return ans
```

87b. Given the roman number, return the integer.

A roman number is always in decreasing order except the ones like CM, CD, ... IX, IV.
Keep adding the values of the main characters and when we encounter increasing order, reduce the double of previous character.

```
class Solution:
    def romanToInt(self, s: str) -> int:
        values = [ 1000, 500, 100, 50,10,5,1 ]
        numerals = [ "M", "D", "C",  "L",  "X", "V","I" ]

        d = dict(zip(numerals,values))
        ans = d[s[0]]

        for i in range(1,len(s)):
            if d[s[i]]>d[s[i-1]]:
                ans -= 2*d[s[i-1]]
            ans += d[s[i]]
        return ans
```

88. Implement atoi() and strstr().

a. atoi() is a c/c++ function that takes a string as an input and returns it's integer representation. The function first discards as many whitespace characters (as in isspace) as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many base-10 digits as possible, and interprets them as a numerical value.
The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.
If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed and zero is returned.

Below is the implementation:

```
int myAtoi(string s) {
        #define int long long
        bool neg = false;
        int i=0;
        int ans = 0;
        while(s[i++]==' ');
        i--;
        if(s[i]=='+'){
            neg = false;
            i++;
        } else if (s[i]=='-') {
            neg = true;
            i++;
        }

        while (s[i]>='0' && s[i]<='9') {
            ans*=10;
            ans+= s[i]-'0';
            i++;
            if((neg?-ans:ans)>=INT_MAX)return INT_MAX;
            else if((neg?-ans:ans)<=INT_MIN)return INT_MIN;
        }
        #undef int
        return neg?-ans:ans;
    }
```

b. strstr() : This function checks whether some string str1 is substring of str2 or not..
Returns a pointer to the first occurrence of str2 in str1, or a null pointer if str2 is not part of str1.
The matching process does not include the terminating null-characters, but it stops there.
// There can be more solutions like KMP algo, Z algo which are explained in upcoming tasks....

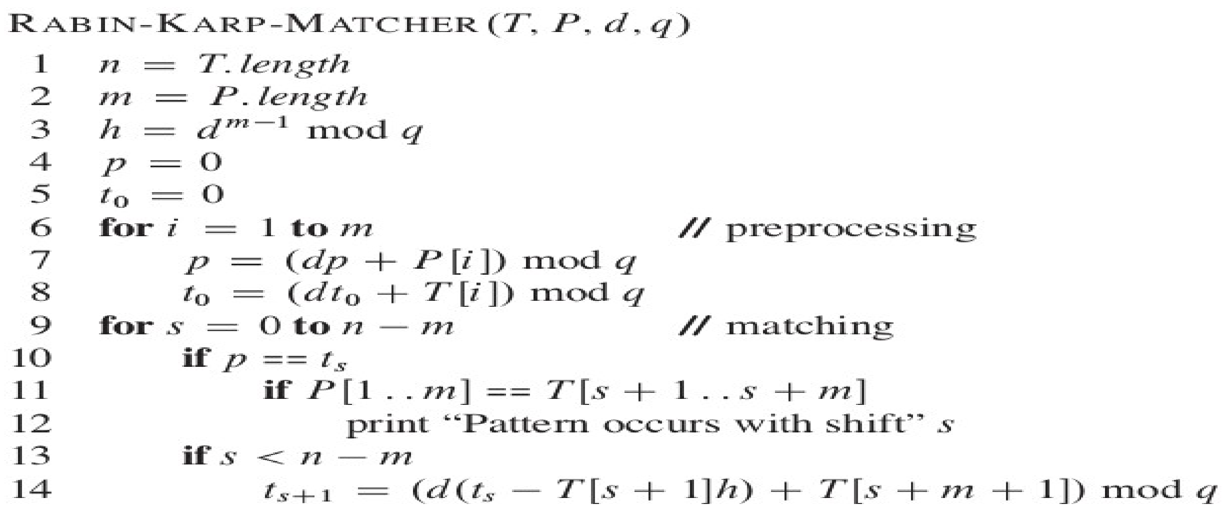
```
def strStr(self, haystack: str, needle: str) -> int:
        if needle in haystack:
            return haystack.index(needle)
        return -1
We have to implement the code below:::::
int strStr(string haystack, string needle) {
        char* i = strstr(&haystack[0], &needle[0]);
        if (i==NULL)return -1;
        return i - &haystack[0];
    }
```

89. Longest Common Prefix
Write a function to find the longest common prefix string amongst an array of strings.
If there is no common prefix, return an empty string "".
Example 1:
Input: ["flower","flow","flight"]
Output: "fl"

a. Simply compare each character.

```
string longestCommonPrefix(vector<string>& strs) {
        if (strs.size()==0) {
            return "";
        }
        if(strs.size()==1) {
            return strs[0];
        }

        string ans = "";
        int minlen = INT_MAX;

        for(string i:strs){
            minlen = min(minlen,(int)i.length());
        }

        for(int j=0;j<minlen;j++) {
            for (int i=1;i<strs.size();++i) {
                if(strs[i][j]!=strs[i-1][j]){
                    return ans;
                }
            }
            ans+=strs[0][j];
        }
        return ans;
    }
```

90. Rabin Karp Algorithm



```
RABIN-KARP-MATCHER(T, P, d, q)
1   $n = T.length$
2   $m = P.length$
3   $h = d^{m-1} \bmod q$
4   $p = 0$
5   $t_0 = 0$
6   for $i = 1$ to $m$                      // preprocessing
7       $p = (dp + P[i]) \bmod q$
8       $t_0 = (dt_0 + T[i]) \bmod q$
9   for $s = 0$ to $n - m$                  // matching
10      if $p == t_s$
11          if $P[1..m] == T[s+1..s+m]$
12              print "Pattern occurs with shift" $s$
13      if $s < n - m$
14          $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$
```

So we have a text T and a pattern P. Check whether P is in T or not ie pattern appears in the text or not?
This algo uses rolling hashing method. Details: https://cp-algorithms.com/string/string-hashing.html

So basically we take two prime numbers called ‘d’ and ‘q’. Here ‘d’ is the prime number nearer to alphabet size. We take it 31 for [a-z] and 53 for [A-Za-z] and so on.. Also ‘q’ is the largest prime number to make hashes well seperated.

For a substring S, we define it’s hash as: (let m = |S|)

$(S_0*d^{m-1} + S_1*d^{m-2} + ... S_{m-1}*d^0) \% q$

So what we are gonna do is:
For each substring of text T starting from index i, having length ‘m’, match it’s rolling hash value with pattern’s hash value.

Now consider this polynomial:

Pol = $10x^2 + 20x + 30$
We want to convert it to $20x^2 + 30x + 40$
What are the steps?
First of all remove the first element (ie $-10x^2$) : $20x + 30$
multiply it by x: $20x^2 + 30x$
Now add 40 : $20x^2 + 30x + 40$
here x^2 is a fixed value so we have precalculated it in variable ‘h’.

Below is the implementation in c++ with modifications for modular arithmetic.

```
int strStr(string T, string P) {
        // T -> text -> haystack
        // P -> pattern -> needle

        if(P == "") {
            return 0;
        }

        const int d = 31;
        const int q = (int)(1e9)+7;
        const int n = T.length();
        const int m = P.length();

        long long h = 1;
        for(int i=0;i<m-1;i++) {
            h*=d;
            h%=q;
        }
        long long p=0, t0=0;

        for(int i=0;i<m;i++) {
            p = (d*p + P[i] - 'a' + 1)%q;
            t0 = (d*t0 + T[i] - 'a' + 1)%q;
        }

        for(int s=0;s<n-m+1;s++) {
            if(p==t0){
                if(P == T.substr(s,m)) {
                    return s;
                }
            }

            if (s<n-m) {
                t0 = (d*( (q + t0 - ( (T[s] - 'a' + 1)*h )%q )%q ) + T[s+m] - 'a' + 1)%q;
            }
        }

        return -1;

    }
```

91. Z-function/ Prefix-function and Z-Algorithm.

Z-algorithm is also a linear time pattern matching algorithm. Before going into deep, lets study Z-function first.

Suppose the string is “aabcdaacaab”.
We initailise an empty array, say Z, of same length. For the first index the z-value is 0. Now for rest of the characters, *the value at* ***<u>Z[i] is the longest substring starting at i that is same as prefix of this string</u>***.
For example, S = “aabcdaacaab”
“**a a**bcdaacaab”
“**aa**bcd **aa**caab”
“**aab**cdaac **aab**”

We always compare the Z values, starting comparing from 0th index. We can compute this array easily in quadratic time.

Now coming to Z-Algorithm, we have two strings T(the text) and P(the pattern) and our goal is to check whether P occurs in T or not?

Lets say P = “abba” and T = “xcabbxabbaba”.
If we append these two strings, seperated with a special character (lets take ‘$’), the resulting string will be:
S = “a b b a $ x c a b b x a b b a b a”
Now if we compute the Z-array for this string, and we find a value equal to length of the pattern, we say that pattern exists.
“**a b b a** $ x c a b b x **a b b a** b a”

We appended $ so that we don’t compare more than required string, ie the Z-value shouldn’t accidently increase over pattern length.

Still the algorithm is O(N^2) in terms of time complexity. Our goal is to minimize it to linear time equivalent to O(N+M) where N is Pattern length and M is text length.

The idea is, suppose we found a long match in the middle of the string.
“**a b b a** $ x c **a b b** x a b b a b a”.
We have calculated the value for first ‘a’ for this match. Do we need to calculate it for other characters of this match? Answer is NO. Because they will be same as the values written in the prefix of our Z array. But if this value exceeds our current ‘r’ ie it is at the edge, we can’t guarentee that it has a larger match or not. So we simply skip that much characters and start comparing from that place instead of going for a full comparison.

So we will use something similar to what we did in Manacher’s algorithm.
First of all, lets take l = 0 and r = 0. ‘l’ and ‘r’ will maintain the latest prefix match with rightmost ‘r’.
For each i, if the Z value is inside boundary, we copy the corresponding value in previous match. Otherwise we start comparing after skipping those much characters.

Below is the implementation:

```
int strStr(string T, string P) {
        // T -> text -> haystack
        // P -> pattern -> needle
        if(P.length() == 0) return 0;

        int p = P.length(); // To regain the index.

        // Prepare the string
        P = P+"$"+T;

        int n = P.length();

        // Our Z-array
        vector<int> z(n,0);

        for(int i=1,l=0,r=0;i<n;++i) {

            // If we are inside the latest prefix match. So Z[i]
            // would be atleast length till boundary 'r' or
corresponding
            // value in latest prefix. To be in safe side: take
Min.
```

```
            if (i<=r) {
                z[i] = min(r-i+1 , z[i-l]);
            }

            // Now we need to compare starting from i + Z[i]
            // If the match is not expendible, this loop will
            // terminate immediately.
            while (i + z[i] < n && P[z[i]] == P[i+z[i]]) {
                ++z[i];
            }

            // This line is for first match. Replace it for all
            // of the occurences as needed.
            if (z[i] == p)return i - p - 1;

            // Now update the l and r if a larger 'r' is found.
            if (i + z[i] - 1 > r){
                l = i;
                r = i + z[i] - 1;
            }
        }


        return -1;

    }
```

92. KMP Algorithm.

KMP is also a linear time pattern matching algorithm. Just like Z-algorithm, it uses a similar kind of prefix function. Before jumping into the algorithm, lets understand the calculation of the prefix function.

Lets say the pattern we wanna search is “ababcabac”. We initialize an empty array prefix[] with same length as the pattern.
prefix[0] is 0 initially and we start from index 1.
<u>For each i, prefix[i] is the length of the suffix ending at ‘i’, which is also the prefix of the pattern.</u>
Calculation is easy. Logically, the suffix length is also the index of where the prefix ends.
So we keep matching the string[j] and string[i] and updating each index.

String	prefix
a **b** a b c a b a c	[0 **0** 0 0 0 0 0 0 0]
a b **a** b c a b a c	[0 0 **1** 0 0 0 0 0 0]
a **b** a **b** c a b a c	[0 0 1 **2** 0 0 0 0 0]
a b **a** b **c** a b a c	[0 0 1 2 0 0 0 0 0]

here a mismatch occured but its guarenteed that the prefix matched upto 2nd index (prefix[j-1]=2).
So we start matching from 2nd index.

a **b** a b **c** a b a c	[0 0 1 2 0 0 0 0 0]

Now still it isn’t matching so start from 0, because prefix[j-1] = 0

a b a b **c** a b a c	[0 0 1 2 0 0 0 0 0]

Now we are at start so put 0 if still not matching

a b a b c a b a c	[0 0 1 2 **0** 0 0 0 0]
a b a b c **a** b a c	[0 0 1 2 0 **1** 0 0 0]
a **b** a b c a **b** a c	[0 0 1 2 0 1 **2** 0 0]
a b **a** b c a b **a** c	[0 0 1 2 0 1 2 **3** 0]
a b a b c a b a c	[0 0 1 2 0 1 2 3 **0**]

Now in the similar fashion we can match the text using this precalculated information. This way whenever the suffix length becomes equal to pattern length, we say that there is a match.
For pseudo code, refer Cormen.
Below is the implementation:

```
int strStr(string T, string P) {
        // T -> text -> haystack
        // P -> pattern -> needle
        if(P.length() == 0) return 0;

        // Lets implement KMP algorithm this time.

        // Compute the prefix function
        int m = P.length();
        vector<int> prefix(m);
        prefix[0] = 0;

        // Suffix length is also equal to the index upto which
suffix = prefix
        int length_of_suffix = 0;

        for(int i=1;i<m;i++) {
            while(length_of_suffix > 0 && P[length_of_suffix] !=
P[i]){
                length_of_suffix = prefix[length_of_suffix-1];
            }

            // If now current index matches, increment length
            if(P[length_of_suffix] == P[i]) {
                ++length_of_suffix;
            }

            // Assign whatever the length is.
            prefix[i] = length_of_suffix;

        }

        // Now we have computed the prefix function.
        // Using this function for string matching is
        // pretty similar process. This time match it
        // with the other string.

        int n = T.length();
        for(int i=0,length_of_suffix=0; i<n ; i++) {
            while(length_of_suffix>0 && P[length_of_suffix] != 
T[i]) {
                // String is not matched upto current length so
                // Check for one less length
                length_of_suffix = prefix[length_of_suffix-1];
            }

            if(P[length_of_suffix] == T[i]) {
                ++length_of_suffix;
            }

            if(length_of_suffix == m) {
                return i - m + 1;
            }
        }

        return -1;
    }
```

93. Minimum number of characters to append in front to make the string palindromic.

According to the question, we are allowed to insert the characters only at the beginning of the string. Hence, we can find the largest segment from the beginning that is a palindrome, and we can then easily reverse the remaining segment and append to the beginning. This must be the required answer as no shorter palindrome could be found than this by just appending at the beginning.

For example: Take the string "abcbabcab". Here, the largest palindrome segment from beginning is "abcba", and the remaining segment is "bcab". Hence the required string is reverse of "bcab"(= "bacb") + original string(“abcbabcab") = "bacbabcbabcab".

This way the problem reduces to finding the longest palindromic prefix.

a. One way is to use brute force to check for each i (from n-1 down to 1) whether s[:i] is palindrome or not. This way it takes O(n^2) time.

b. Other way is to use some previous knowledge.

Lets use Z-function. It calculates the length of common prefix.
If we use our string as S + “#” + S’ (Where S’ = reversed string), it will calculate the longest prefix same as prefix of S.
abcbabcab#bacb**abcba**
A more clean way is to compute the prefix function of KMP instead of Z-function. It will also give the length of suffix that is also the prefix. So formally, the last index of this function for the string S + “#” + S’ will give the length of longest palindromic prefix of S.

Below is the code for the latter one:

```
string shortestPalindrome(string s) {
        int i,n;
        string t = string(s.rbegin(),s.rend());
        t = s + "$" + t;
        n = t.length();
        vector<int> prefix(n);
        int len=0;
        prefix[0] = 0;
        for(i=1;i<n;i++) {
            while(len>0 && t[len]!=t[i] ) {
                len = prefix[len-1];
            }
            if(t[len] == t[i]){
                len++;
            }
            prefix[i] = len;
        }
        return string(s.rbegin(),s.rbegin() + s.length() -
prefix[n-1]) + s;

    }
```

94. Check for Anagrams.

Two strings are anagram of each other if one can be shuffled to obtain the other.

a. Sort the strings and check whether they are same or not. O(nlogn)

```
def isAnagram(self, s: str, t: str) -> bool:
        return sorted(s) == sorted(t)
```

b. Count the frequency of each character and match it. O(n)

```
bool isAnagram(string s, string t) {
        vector<int> v(26,0);
        for (auto i:s){
            ++v[i-'a'];
        }
        for (auto i:t){
            --v[i-'a'];
        }
        for(auto i:v) {
            if(i!=0)return false;
        }
        return true;
    }
```

95. Count and say
The count-and-say sequence is the sequence of integers with the first five terms as following:

1. 1
2. 11
3. 21
4. 1211

5.   111221
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.

Given an integer n where 1 ≤ n ≤ 30, generate the nth term of the count-and-say sequence. You can do so recursively, in other words from the previous member read off the digits, counting the number of digits in groups of the same digit.

a. Idea is simple, do what it’s said in the question. We can use the simlar logic we used in {43. Find maximum consecutive 1’s.} This way we have to count each consecutive subarray and append the count before the digit. (Kinda similar to python’s groupby)

```
string countAndSay(int n) {
        if(n==1)return "1";
        string s = "1";
        for(int i=1;i<n;i++) {
            string t = "";
            int cnt=0;
            for(int l=0,r=0;l<s.length();) {
                cnt++;
                r++;
                while(r<s.length() && s[r] == s[r-1]){++cnt;++r;}
                t+=to_string(cnt) + s[l];
                l = r;

                cnt=0;
            }
            s = t;
        }
        return s;
    }
```

96. Compare Version Numbers.

Compare two version numbers version1 and version2.
If version1 > version2 return 1; if version1 < version2 return -1;otherwise return 0.

Few notable things are :
1.0 == 1 is True. Similarly 1.0.0 == 1 is also true.
1.2.4 < 3.0 is true. Also 1.0.1 > 1 is true.

a. So we can notice that we need to consider each number seperated by ‘.’ as a digit of a number and return the answer accordingly. For strings like 1.0 and 1, we can assume that there are some extra 0 with periods(‘.’) too.
Our goal is: Upto each ‘.’, get the numbers of both strings and compare them. If string length are different, consider the number to be 0 for the shorter string.
Observe the code below:

```
int compareVersion(string v1, string v2) {
        int i=0,j=0,n=v1.length(),m=v2.length(),x=0,y=0;
        while(i<n || j<m) {
            while(i<n && v1[i]!='.') {
                x = x*10+v1[i++]-'0';
            }
            while(j<m && v2[j]!='.') {
                y = y*10+v2[j++]-'0';
            }
            ++i;++j;
            if(x>y)return 1;
            else if(x<y) return -1;
            x=0;
            y=0;

        }
        return 0;
    }
```

97. Inorder Traversal of a binary tree.

a. Recursive approach: {LEFT::ROOT::RIGHT}

```
class Solution {
public:
    void inorderTraversal(TreeNode *root, vector<int>& ans) {
        if(root){
            inorderTraversal(root->left,ans);
            ans.push_back(root->val);
            inorderTraversal(root->right,ans);
        }
    }
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> ans;
        inorderTraversal(root, ans);
        return ans;
    }
};
```

b. Iterative approach:
We can mimic the recursive approach using a stack. Observe the recursive code, the root gets printed whenever it returns from it’s left and that happen only when left is null. So we keep pushing the root while it’s left is not null. When it is null, we print whatever is on top of the stack and pop it (because we just returned). From that node, we make root = root.right ; and continue the algorithm.

```
vector<int> inorderTraversal(TreeNode* root) {
        vector<int> ans;
        stack<TreeNode*> s;

        if(root==NULL)return ans;

        while(!s.empty() || root){
            if(root == NULL) {
                ans.push_back(s.top()->val);
                root = s.top()->right;
                s.pop();
            }else{
                s.push(root);
                root = root->left;
            }
        }

        return ans;
    }
```

98. Preorder Traversal.

Recursive Approach:

```
class Solution {
public:
    void preorder(TreeNode* root, vector<int>& ans) {
        if(root){
            ans.push_back(root->val);
            preorder(root->left, ans);
            preorder(root->right, ans);
        }
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        preorder(root, ans);
        return ans;
    }
};
```

Iterative Approach:

```
vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        stack<TreeNode*> s;
        while(!s.empty() || root) {
            if(root == NULL) {
                root = s.top()->right;
                s.pop();
            } else {
                ans.push_back(root->val);
                s.push(root);
                root = root->left;

            }
        }
        return ans;
    }
```

99. Postorder Traversal.

Recursive::

```
class Solution {
public:
    void postorder(TreeNode* root, vector<int>& ans) {
        if(root){
            postorder(root->left, ans);
            postorder(root->right, ans);
            ans.push_back(root->val);
        }
    }
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        postorder(root, ans);
        return ans;
    }
};
```

Iterative::
a. Iterative part of postorder is tricky because we have to maintain two callbacks, one for left and one for right. One approach is to use two stacks, Just like in recursive approach, we use root->left->right and print it in reverse.

1. Push root in first stack.
2. While first stack is not empty:
        Pop a node from first stack and push it into second stack
        Push left and right children of this node into first stack.
3. Print the second stack from top to bottom.

b. We can use single stack to solve this problem.
The idea is, we will push all the subroots in that stack. Suppose we reached the leaf node (NULL), that means in LEFT->RIGHT->ROOT traversal, we have reached LEFT. Now if a right sibling exist, we will continue this process for right subtree ie, if stack->top has right child, and we haven't traversed it yet, we have to push it in stack and continue. Otherwise we will push the stack->top in our answer.

See the steps below:

i. Initialize current = root and an empty stack.
ii. If current is not NULL, push left child in stack and set current = left
iii. Otherwise if stack top has right child, and it is not printed yet (It will be last printed element because we returned from there), set current = stack-top-right
iv. If it doesn't have a right child or it is already printed, pop it and print it.
v. Repeat ii to iv while either stack is not empty or current is not null.

See the code below :
```
vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        stack<TreeNode*> s;

        TreeNode* current = root;

        while(!s.empty() || current){
            if(current) {
                s.push(current);
                current = current->left;
            } else {
                TreeNode* temp = s.top();
                if(temp->right){
                     if(ans.size() && ans.back() == temp->right-
>val){
                         ans.push_back(temp->val);
                         s.pop();
                     } else{
                         current = temp->right;
                     }
                } else {
                    ans.push_back(temp->val);
                    s.pop();
                }
            }
        }

        return ans;
    }
```

100. Left / Right View of a binary tree.

```
-->      1          <---
        /  \
-->   2     3        <---
       \     \
 -->    5     4      <---
```

For left view we need to print [1,2,5] and for right view we need to print [1,3,4]

a. Idea is simple, suppose we are traversing this tree in dfs order. Lets talk about left view first, we will traverse the left arm first. So we will print it. Now what if there are more leftmost nodes at a higher level? Again, we will traverse the leftmost first.
So the logic is to keep a variable called “maxsofar”, and whenever we jump to a higher level, print that node because it is guarenteed that it is the leftmost node of this level.
For right view, we must traverse right subtree first.

Below is the code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
```

```
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:

    void dfs(TreeNode* root , vector<int>& ans, int level, int&
maxlevel) {
        if(root){
            if(level>maxlevel){
                maxlevel = level;
                ans.push_back(root->val);
            }
            dfs(root->right, ans, level+1, maxlevel);
            dfs(root->left, ans, level+1, maxlevel);
        }
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> ans;
        int mx=-1;
        dfs(root, ans, 0, mx);
        return ans;
    }
};
```

b. Other way is to use BFS or level order traversal. We need to print the first node in each level. Now how do we determine when the new level start?

A pretty logical way is to push a NULL character at the end of each level. We need to monitor the front of the queue. When it is NULL, we have to push another NULL after pushing the children nodes.
Now at the end, we will push an extra NULL automatically in the end, so in our BFS, instead of checking whether queue is empty or not, we check whether q.front != NULL or not. Otherwise to keep it “q!=empty”, we have to remove all the null from the front and check for empty tree separately.

Below is the implementation:

```
vector<int> rightSideView(TreeNode* root) {
        vector<int> ans;
        queue<TreeNode*> q;
        q.push(root);
        q.push(NULL);
        while(q.front()){
            TreeNode* top = q.front();
            q.pop();

            if(top->left)q.push(top->left);
            if(top->right)q.push(top->right);

            if(q.front() == NULL){
                q.pop();
                q.push(NULL);
                ans.push_back(top->val);
            }
        }
        return ans;
    }
```

101. Bottom view of the tree.

First of all we need to be clear about bottom view. We assume that root is at 0th horizontal level and for each node, the left subtree is at lvl – 1 level and right one is at lvl + 1 level. Suppose we have the following tree:

```
           20
         /    \
       8        22
     /   \        \
   5       3        25
         /  \
       10    14
```

Lets rewrite it with its horizontal level order:

```
              20,0
             /    \
       8,-1         22,1
     /       \          \
   5,-2       3,0        25,2
             /   \
       10,-1      14,1
```
Now level wise, we can write the nodes as:

-2: 5
-1: 8->10
0 : 20 -> 3
1 : 22->14
2 : 25

So the bottom view will be:

[5,10,3,14,25] {last node of each level}.

If two nodes are at same horizontal and vertical level, we take the node that is from the right subtree. Or we can also say that it comes later in the level order traversal.
Another example:

```
              20
            /    \
          8        22
        /   \    /   \
      5       3 4      25
             / \
          10    14
```

Bottom view is 5,10,4,14,25 . We have 3 and 4 at same levels but we take 4 instead of 3 because it is from the right subtree.

a. Idea is simple, we can apply dfs. We need to create a hashmap to store the latest node at each horizontal level. Since we can travel back to an upper level in dfs, it is not guarenteed that each latest node at a certain level is also in bottom most levels. So we also maintain the current level of the node while traversing.

Below is the code;
```
void dfs(Node *root, int horizontal_level,int vertical_level, 
map<int,pair<int,int>>& info){
    if(!root)return;
    if(info.find(horizontal_level)!=info.end()){
        if(vertical_level>=info[horizontal_level].second){
            info[horizontal_level] = {root->data, vertical_level};
        }
    } else {
        info[horizontal_level] = {root->data, vertical_level};
    }
    dfs(root->left,horizontal_level-1,vertical_level+1,info);
    dfs(root->right,horizontal_level+1,vertical_level+1,info);
}
vector <int> bottomView(Node *root)
{
   map<int,pair<int,int>> info; // pair of {node.data,lvl}
   vector<int> ans;
   if(root == NULL) return ans;
   dfs(root,0,0,info);
   for(auto i:info)ans.push_back(i.second.first);
   return ans;
}
```

b. If we use BFS in this problem, we will need to put the horizontal level with the node too. That way whenever we find a new node with a level, we can directly update it because in BFS it’s guarenteed that it will be rightmost and bottom most.

102. Top view of the binary tree.

Just like the bottom view, we have to print the first node of vertical order traversal.

Since we coded the dfs solution for bottom view, we can reuse it but we should always update the hashmap when we find a node at lower level.

Lets code the BFS solution this time. We need to keep track of the horizontal level in queue, so we push pair of node and its level. And we will update the map only if a new level is found because others are already filled with top most nodes.

Lets observe the code and we will be able to grasp the logic about how to use it in bottom view too.

```
void topView(Node * root) {
        map<int,int> info;
        if(root==NULL)return;

        queue<pair<Node*,int>> q; //{ node and horizontal level}

        q.push({root, 0});
        while(!q.empty()) {
            pair<Node*,int> p = q.front();
            q.pop();
            if(info.find(p.second) == info.end()) {
                info[p.second] = p.first->data;
            }
            if(p.first->left)q.push({p.first->left, p.second - 1});
            if(p.first->right)q.push({p.first->right, p.second + 1});
        }

        for(auto i:info)cout<< i.second << " ";

    }
```

103. Level order traversal (Simple and spiral form)

We can do simple level order traversal using BFS straight forward. So let’s focus on zig zag traversal / spiral traversal only.

```
   3
  / \
 9  20
   /  \
  15   7
```

We must print 3 -> 20-> 9-> 15-> 7.

a. So the most simple method is to use BFS (With NULL markers) and store each level in an array. Later we can print even levels as they are and odd level in reverse manner. This approach is straight forward so it takes O(N) time. About space, BFS takes O(N) space but we are also using some extra space to store each level. We can avoid that by printing the even levels directly and storing the odd levels in a stack first and then printing them while popping them (Basically reversing them)

b. We can also use two stacks to solve this problem without using any queue.

Lets say we have two stacks namely Current and Next. We will traverse the Current stack and push children into Next stack and will swap them at the end of each level.

So initially we push root into Current stack. Now we are at even level so we push the left child and right child so that rightmost can come at the top. When we pop this node, stack will be empty so we will swap the stacks. For next level, we want that the leftmost child should come at top so we push right child and then left child for each node. This way we continue till the current stack is not empty (even after swapping the stacks) This will definetly terminate because we won't push null children into either of the stack, so last level won't contribute in pushing new children. So at the end both stacks are empty.

Below is the code:

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root)return ans;

        stack<TreeNode*> current, next;
        bool even = true;
        vector<int> temp;
        current.push(root);
        while(!current.empty()) {
            TreeNode* node = current.top();
            current.pop();
            temp.push_back(node->val);
            if(even){
                if(node->left)next.push(node->left);
                if(node->right)next.push(node->right);
            } else {
                if(node->right)next.push(node->right);
                if(node->left)next.push(node->left);
            }

            if(current.empty()){
                even^=true;
                ans.push_back(temp);
                temp.clear();
                swap(current,next);
            }
        }

        return ans;


    }
```

104. Height of a binary tree.

a. This one is simple if we use recursion.

```
int maxDepth(TreeNode* root) {
        if(!root)return 0;
        return 1+max(maxDepth(root->left) , maxDepth(root->right));
    }
```

105. Diameter of the binary tree.

By definition, diameter is the longest path between any two distinct nodes of the binary tree.

```
      1
     / \
    2   3
   / \
  4   5
```

a. If we observe the tree, we can see that if an internal node gets the largest diameter, it won't contribute further. But a diameter will always form while going through an ancestor so we can see that as a base case diameter is the height of left subtree + right subtree.

Hence diameter is the maximum of these three values:

i. Diameter of left subtree
ii. Diameter of right subtree
iii. Height of left subtree + Height of right subtree + 1

Below is the code:

```
class Solution {
public:
    int height(TreeNode* root){
        if(!root)return 0;
        return 1+max(height(root->left), height(root->right));
    }
    int diameterOfBinaryTree(TreeNode* root) {
        if(!root)return 0;
        int d1 = diameterOfBinaryTree(root->left);
        int d2 = diameterOfBinaryTree(root->right);
        return max(height(root->left) + height(root->right)  ,
max(d1,d2)) ;

    }
};
```

Above logic recalculates the height many times so it is of O(N^2).

b. Extending the above logic, we can do the same in linear time. As a base case, diameter is the sum of left and right subtree. So we can modify our height function a little bit to get the maximum left + right height we can get for any node.

Code:

```
class Solution {
public:
    int height(TreeNode* root, int& ans){
        if(!root)return 0;
        int left = height(root->left, ans);
        int right = height(root->right, ans);
        ans = max(ans, left+right);
        return 1+max(left,right);
    }
    int diameterOfBinaryTree(TreeNode* root) {
        int ans=0;
        height(root, ans);
        return ans;
    }
};
```

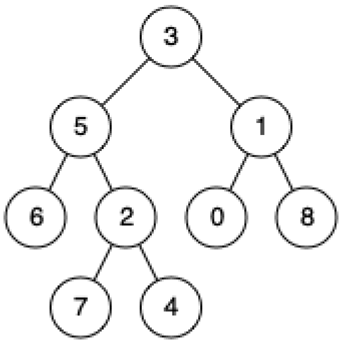
106. Check whether the given binary tree is height balanced or not.

A tree is height balanced iff for each node, the height of left and right subtree do not differ in more than 1.

a. We can simply check the height of every node. To make it simpler, observe the code below:

```
class Solution {
public:
    int height(TreeNode* root, bool& ans) {
        if(!root)return 0;
        int l = height(root->left, ans);
        int r = height(root->right, ans);
        ans &= abs(l-r)<=1;
        return max(l,r)+1;
    }
    bool isBalanced(TreeNode* root) {
        bool ans = true;
        height(root, ans);
        return ans;
    }
};
```

107. Find LCA of given binary tree.



If we observe the tree closely, we can see that pick any two nodes, their LCA always exist.
The LCA is always that node from where one node is in the left subtree and other is in right subtree.
If the nodes are in same chain, then the upper one is LCA. So we can use the following approaches to find the lowest common ancestor.

a. Find the path from root to first node and also the path from root to second node. The first node that mismatch in these paths is the LCA. See the code below:

```
class Solution {
public:
   bool dfs(TreeNode* root, TreeNode* target, vector<TreeNode*>& path) {
      path.push_back(root);
      if(!root) return false;

      if(root == target){
         return true;
      }

      if(dfs(root->left , target, path))return true;
      path.pop_back();
      if(dfs(root->right, target, path))return true;
      path.pop_back();
      return false;

   }
   TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
      vector<TreeNode*> path1,path2;
      dfs(root, p, path1);
      dfs(root, q, path2);
      int i;
      for(i=0;i<min(path1.size(), path2.size());i++) {
         if(path1[i]!=path2[i])return i==0?root:path1[i-1];
      }
      return path1[i-1];
   }
};
```

This approach is O(N) for time and space. We are using O(N) * 3 space for path1 path2 and stack space. And we also traverse O(N) three times.

More methods are listed here: https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-2-using-parent-pointer/

b. If the parent pointers are given, ie we can access the node->parent, the problem gets converted to finding the intersection of Y shaped linked list. That way we need O(logN) (Best case) to O(N) time (worst case) and O(1) space.

c. One interesting solution (learnt from Tushar Roy's Video) is given below. If a node is LCA, it will contain one node at left subtree and other at right subtree.
The algorithm is as follows:

I. We use dfs to traverse the tree. For a node, if it is NULL ie we are dealing with leaf nodes, return NULL.
II. If our current node is either of P and Q, return the current node.
III. Otherwise traverse the left and right node. If both traversals return NON-NULL nodes, return the current node.
IV. If any of them is NULL, we return the node which is not NULL.

We can dry run this algo on a tree.

If a tree has both P and Q in same branch, our algo will never traverse the lower node. Eg if P is above Q, we will return from P. Since everyone else will return NULL, we will eventually push P to the top and actually in these cases, P is the answer. So we don’t need to worry about this case.

Again, if a tree has the LCA in a lower level, we will be returning it always and other nodes will return NULL so it will be pushed to top. Hence this algo works.

```
class Solution {
public:
    TreeNode* dfs(TreeNode* root,TreeNode* p,TreeNode* q) {
        // Base case
        if(!root)return NULL;

        // If it is one of them, return
        if(root == p || root == q) return root;

        // Get the left and right candidates
        TreeNode * l = dfs(root->left,p,q);
        TreeNode * r = dfs(root->right,p,q);

        // If both are not null, return the current node
        if(l&&r) return root;

        // Otherwise return the NON-NULL node
        return l?l:r; //(returns the non-null only)

    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
        return dfs(root,p,q);
    }
};
```

108. Check whether two binary trees are same or not.

Apply dfs on both trees simultaneously and return false when a mismatch occur.

```
bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==NULL && q==NULL)return true;
        if((p==0 && q)||(q==0 && p))return false;
        if(p->val != q->val)return false;
        return isSameTree(p->left,q->left) && isSameTree(p->right , q->right);
}
```

109. Max Path Sum in a binary tree.

You are given a binary tree, your task is to return the maximum possible path sum between any two nodes.

Input:

```
      1
     / \
     2   3
```

Output : 6

```
       1
      / \
     2   3
```

Input:

```
   -10
    / \
   9  20
       /  \
   15    7
```

Output: 42

```
   -10
    / \
   9  20
       /  \
   15    7
```

Input:

```
        9
      /   \
     6    -3
            /  \
        -6   2
              /
            2
          /  \
       -6  -6
       /
     -6
```

Output: 16 (From 6 to 2)

```
        9
      /   \
     6    -3
            /  \
        -6   2
              /
            2
          /  \
       -6  -6
       /
     -6
```

a. We are allowed to take any path in this question. We can observe few things:

i. For any subtree, we can take the path formed in left subtree or the right subtree. But both must be linear paths, ie we can not take a path in ^ shape.

ii. We can take any longest path but the negative values become the hurdle. We can easily deal with them by taking 0 to ignore the negative chains.

iii. If there is path like in example 2, we can also use such path by computing it on the root of any subtree. So we will pass the reference of our ans variable so that we get the maximum possible ans.

So the process is:
Use dfs to traverse the entire tree. For any subtree, the path can consist of either left/right both or none. Instead of using multiple if-else, simply ignore those which are negative, and if both are poistive, simply take the maximum one. But if you don't want to take the root, return 0 because you can't take the entire subtree now(You can't skip any node).

May be the explanation is not too understandable.. But I will try to improve it.
See the code below: :::: Instead of using these much variables, use ? : and max function for one liner codes. See my first uncommented submission:: https://leetcode.com/problems/binary-tree-maximum-path-sum/

```
class Solution {
public:
    int dfs(TreeNode * root, int& ans){
        if(!root)return 0;  // Null node should return 0

        int l = dfs(root->left, ans);  // Compute the longest 
chain in left

        int r = dfs(root->right, ans); // Compute the longest 
chain in right

        // Now for our answer, we may take either the left chain,
        // Or the right chain, or both , or none.
        // But why will we ignore a chain? because it is 
negative!!!
        // So to avoid separate if else, use 0 to ignore them!!

        int maxYouCanGetFromLeft = max(l,0);  // ie take it or 
leave it
        int maxYouCanGetFromRight = max(r,0);


        // Hence all three cases are handled.
        ans = max(ans , maxYouCanGetFromLeft + 
maxYouCanGetFromRight + root->val );

        // So what do we return? We can't return a chain coming 
from
        // left, passing through the root and going to right. So 
we return
        // a single chain, that too with greatest sum!!!
        // But wait! What if we don't want this subtree at all?
        // Simply return 0 if it is giving negative value!!

        return max(0,max(l,r) + root->val);

    }
    int maxPathSum(TreeNode* root) {
        int ans = INT_MIN;
        dfs(root, ans);
        return ans;
    }
};
```

Time & Space Complexity: O(N)

110. Construct binary tree from preorder and inorder traversal.

We can use the traditional algorithm for this. Observe the code below:

```
class Solution {
public:
    TreeNode * dfs(vector<int>& preorder, vector<int>& inorder,
                   unordered_map<int , int>& mp,
                  int start, int end, int& preorder_index){

        // If nodes are not available....
        if(preorder_index >= preorder.size())
            return NULL;

        // Make the root of this subtree.
        TreeNode * root = new TreeNode(preorder[preorder_index]);

        // Now we don't need this node :P
        ++preorder_index;

        // In preorder, left subtree comes first, so recur
        // in left first.
        // If left node exist, assign it.
        if( start <= mp[root->val] - 1) {
            root->left = dfs(preorder, inorder, mp, start,
mp[root->val] - 1, preorder_index );
        }

        // If right node exits, assign it.
        if( end   >= mp[root->val] + 1) {
            root->right = dfs(preorder, inorder, mp, mp[root->val]
+ 1, end, preorder_index );
        }

        // Now return this subtree root.
         return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>&
inorder) {
        // For fast search of nodes we make a hashmap.
        unordered_map<int , int> mp; // taking this mapping
because duplicates do not exist.
        int preorder_index = 0;
        // this hashmap will map all the preorder nodes -> index
in inorder.
        for(int i=0;i<inorder.size();i++){
            mp[inorder[i]] = i;
        }

        // Call the dfs
        return dfs(preorder, inorder, mp, 0, inorder.size()-1,
preorder_index);
    }
};
```

111. Construct binary tree from postorder and inorder traversal.

Concept is similar but recur in right before left because postorder comes from right subtree.

```
class Solution {
public:
    TreeNode* dfs(vector<int>& inorder, vector<int>& postorder,
             unordered_map<int , int>& mp,
             int start, int end) {

        if(postorder.size() == 0) return NULL;

        TreeNode * root = new TreeNode(postorder.back());
        postorder.pop_back();

        if(end   >= mp[root->val]+1) {
            root->right = dfs(inorder, postorder, mp, mp[root-
>val]+1, end);
        }

        if(start <= mp[root->val]-1) {
            root->left = dfs(inorder, postorder, mp, start,
mp[root->val]-1);
        }

        return root;

    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>&
postorder) {
        unordered_map<int , int> mp; // taking this mapping
because duplicates do not exist.

        for(int i=0;i<inorder.size();i++){
            mp[inorder[i]] = i;
        }

        return dfs(inorder, postorder, mp, 0, inorder.size()-1);
    }
};
```

112. Symmetric Binary tree.

A symmetric tree looks like this:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following tree is not symmetric:

```
    1
   / \
  2   2
 /   /
3   3
```

a. We can use a recursive approach by using two pointers simultaneously. We will send one pointer to left and other to right and vice versa.
Observe the code below:

```
class Solution {
public:
    bool dfs(TreeNode* root1, TreeNode *root2) {
        if((root1==NULL && root2!=NULL) || (root1!=NULL &&
root2==NULL)) // If any of them is null,notnull or notnull,null
            return false;

        //if both are null return true
        if(root1 == NULL && root2 == NULL) return true;
        // if value isn’t same return false.
        if(root1->val != root2->val)
            return false;
        // recur for left and right
        return dfs(root1->left, root2->right) && dfs(root1->right,
root2->left);

    }
    bool isSymmetric(TreeNode* root) {
        return dfs(root,root);
    }
};
```

b. We can also solve this problem using iteration. We traverse the tree using bfs, but we start with two roots initially. In each step we pop two nodes from queue and they should be equal. Insert root1.left , root2.right ,root1.right, root2.left in the queue and repeat until queue is not empty.

```
bool isSymmetric(TreeNode* root) {
        queue<TreeNode*> q;
        q.push(root);
        q.push(root);
        while(!q.empty()) {
            TreeNode * a = q.front();
            q.pop();
            TreeNode * b = q.front();
            q.pop();
            // If both null, continue..
            if(a == NULL && b == NULL) continue;

            // If one of them is null, or it's a mismatch so
return false
            if(a == NULL || b == NULL || a->val!=b->val) return
false;

            // Now push leftmost and right most nodes, then middle
ones
            q.push(a->left);
            q.push(b->right);
            q.push(a->right);
            q.push(b->left);

        }
        return true;
    }
```

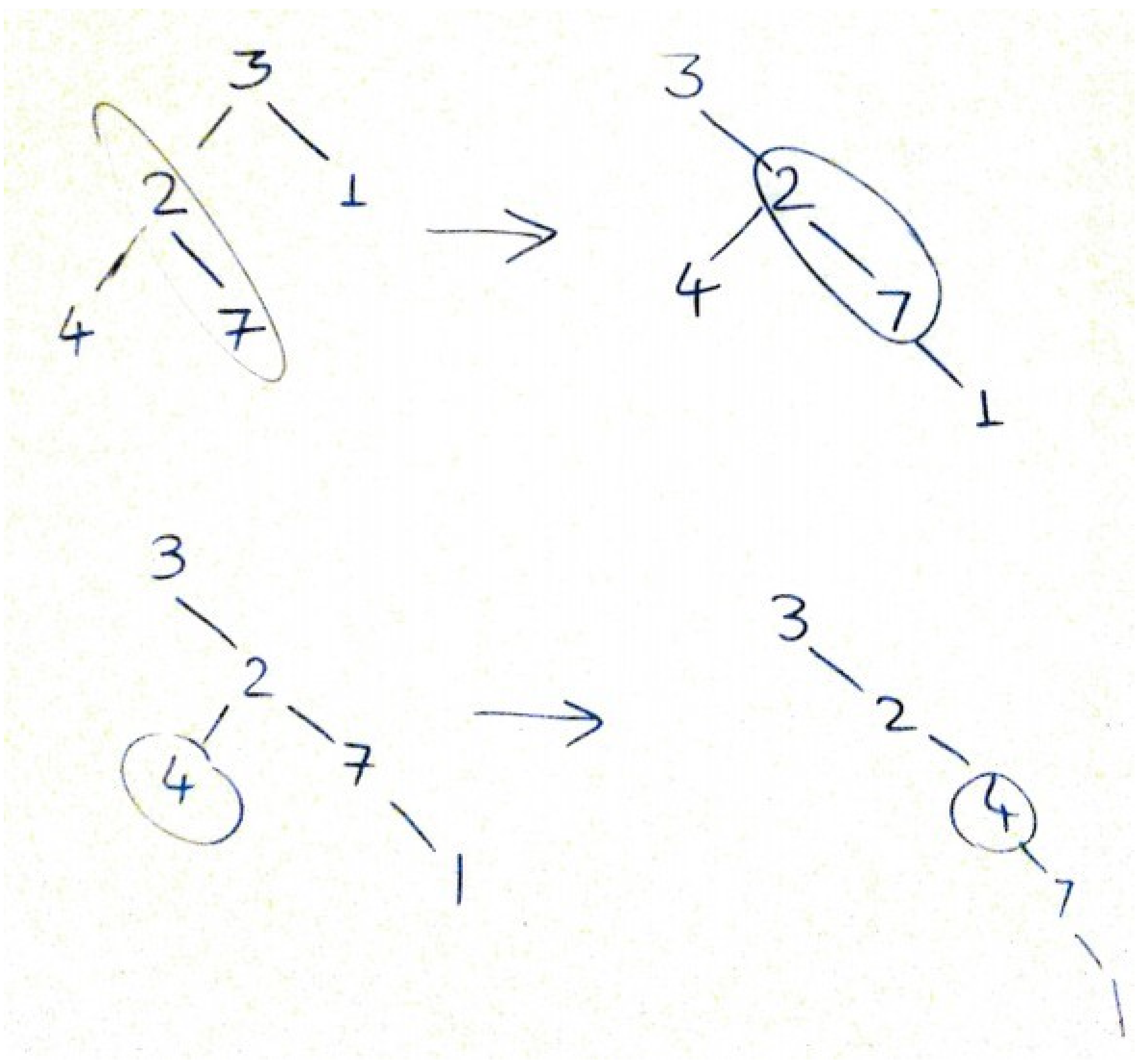
113. Flatten a binary tree.
Given a binary tree, flatten it to a linked list in-place.
For example, given the following tree:

```
 Tree                      Flattened Tree

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

a. We need to recursively flatten the left subtree, attach it to left and flatten the right subtree

```
class Solution {
public:
    /*
    So we need to convert
       a
      / \
     b   c

    into

    a
     \
      b
       \
        c

    So we first flatten the left subtree
    then push it in between root and right subtree
    And finally flatten the right subtree.
    */
    void flatten(TreeNode* root) {
        if(root == NULL) return;

        if(root->left == NULL && root->right == NULL) return;

        if(root->left != NULL) {
            flatten(root->left);
            TreeNode *right = root->right;
            root->right = root->left;
            root->left = NULL;
            while(root->right!=NULL)
                root = root->right;
            root->right = right;
        }
        flatten(root->right);
    }
};
```

b. We can also do it iteratively. We take the longest chain(skewed in right direction) in the left subtree and append it in between root and right subtree. We do it iteratively until we reach to a null node. See the figure below:



```
void flatten(TreeNode* root) {
        if(root == NULL) return;
        while(root) {
            if(root->left && root->right) {
                TreeNode * node = root->left;
                while(node->right) {
                    node = node->right;
                }
                node->right = root->right;
            }

            if(root->left)
                root->right = root->left;
            root->left = NULL;
            root = root->right;
        }
    }
```

114. Check if Binary Tree is mirror of itself or not

Same as 112.

115. Populate next right pointers of the given Binary Tree.

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

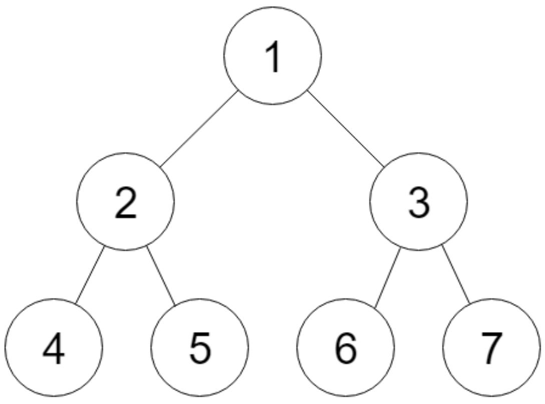


Figure A

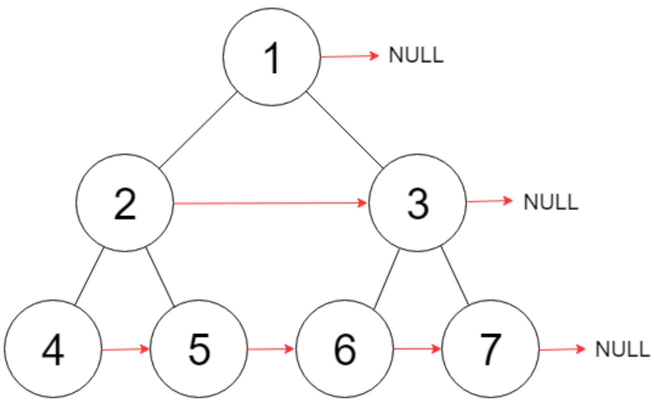


Figure B

a. We can solve this problem using BFS easily. We will initially push root and NULL in the queue and at each occurence of NULL, we push another NULL(Just like level order traversal where we were marking the end of each level). Meanwhile we set current_node ->next = queue.front().

b. We can also iteratively solve this problem since the given tree is Perfect Binary Tree. For each level, we set a node's left's next = node's right . Then we set node's right's next = node's next's left. Again apply restrictions so that we don't access NULL node's next/left/right pointers. Observe the code below:

```
Node* connect(Node* root) {
        if(!root)return root;
        Node* head = root;
        root->next = NULL;
        Node* p;
        while(root->left) {
            p = root;
            while(p->next) {
                p->left->next = p->right;
                p->right->next = p-> next->left;
                p = p->next;
            }
            p->left->next = p->right;
            p->right->next = NULL;
            root = root->left;
        }
        return head;
    }
```

116. Search given key in BST.

```
TreeNode* searchBST(TreeNode* root, int target) {
        if(!root)return root;

        while(root){
            if(root->val == target)
                return root;
            root = (target < root->val )?root->left:root->right;
            }
        return NULL;
    }
```

117. Construct Height Balanced Binary Search Tree using the given sorted array.
a. We will use recursion to assign proper values to tree nodes. If the given keys are not sorted, we can sort them first before applying the method below.

```
class Solution {
public:
    TreeNode* recur(vector<int>& a, int l, int r){
        if(l>r)return NULL;
        int mid = (l+r)/2;
        TreeNode *root = new TreeNode(a[mid]);
        root->left = recur(a, l, mid-1);
        root->right = recur(a,mid+1,r);
        return root;

    }
    TreeNode* sortedArrayToBST(vector<int>& a) {
        return recur(a,0,a.size()-1);
    }
};
```

Time complexity O(N) if sorted array is given, O(NlogN) otherwise.

118. Check if given binary tree is a BST or not.

a. If we write the inorder traversal of a BST, it is already sorted.

```
class Solution {
public:
    void inorder(TreeNode* root, vector<int>& in) {
        if(root){
            inorder(root->left, in);
            in.push_back(root->val);
            inorder(root->right, in);
        }
    }
    bool isValidBST(TreeNode* root) {
        if(!root) return true;
        vector<int> in;
        inorder(root, in);
        for(int i=1;i<in.size();++i){
            if(in[i]<=in[i-1])
                return false;
        }
        return true;
    }
};
```

b. Above approach takes an extra vector to store the inroder traversal. So we can do the same process without using extra space (except the stack used in recursion.)
We can write the inorder traversal while maintaining a variable that records the previously visited node. If the current node is having lesser or equal value, we return false.

Below is the code for better understanding:

```
class Solution {
public:
    bool inorder(TreeNode* root, long long& last) {
        if(root){
            // Visit left subtree. If it has an NON-BST subtree
            // immediately return false
            if(!inorder(root->left, last))return false;

            // If this node doesn't follow BST rules,
            // return false.
            if(root->val <= last){
                return false;
            }

            // Update the last node..
            last = root->val;

            // Now traverse in right subtree and return
            // the result whether it is balanced or not.
            return inorder(root->right, last);

        } else return true;
    }
    bool isValidBST(TreeNode* root) {
        if(!root) return true;

        long long last = LONG_MIN;
        return inorder(root, last);

    }
};
```

119. Find LCA of two nodes in BST.

We find the first node having one node in left subtree and other in right. Or if we reach either p or q, we return that node.

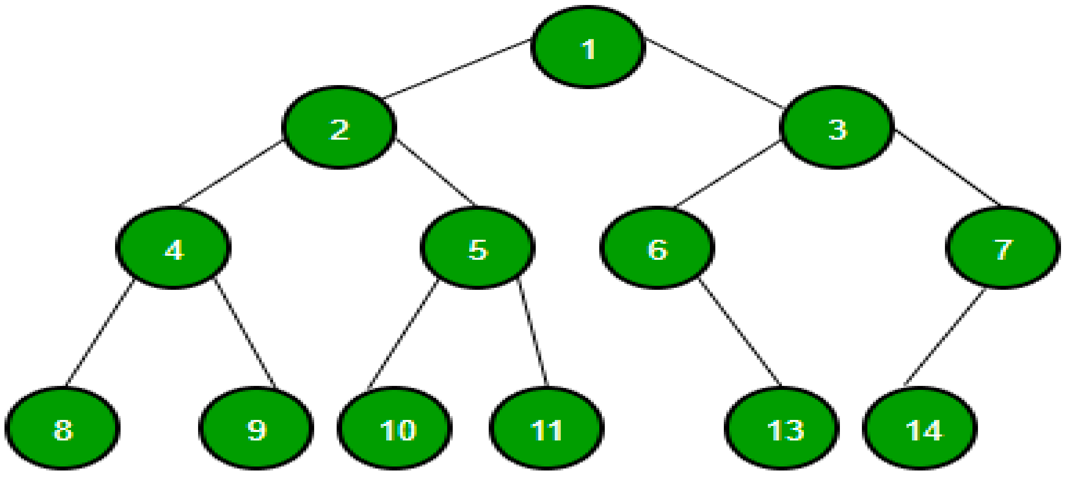
```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
        int l = p->val, r = q->val;
        if(l>r)
            swap(l,r);
        while(root){
            if(l < root->val && r > root->val)
                return root;
            if(l == root->val || r == root->val)
                return root;
            if(l< root->val)
                root = root->left;
            else
                root = root->right;

        }
        return NULL;
    }
```

Iterative approach takes O(1) space while recursive approach takes stack space. But runtime is same for both. Note that tree might not be balanced, so time complexity will be O(N).

120. Find the inorder predecessor/successor of a given Key in BST/ Binary Tree.

a. If parent pointer is given, we first search for the node. For BST, we can find the node in O(logN) time in best case(This is the only difference for BST and BT for this approach). We are talking about inorder traversal, so lets talk about successor first.
When we visit a node, we go to it’s right child. From there, the recursion gets us to the left most child of this node. But if right child doesn’t exist, we will go to an ancestor. If our current node is

the left child of any node, successor will be the parent node. If it is the right child, we will go to the first parent, that has called the left recursion. So in both cases, successor is the first node that is left child of its parent.



So we can define the algorithm as :

Predecessor:

i. Find the node.
ii. If the node has left child, move to left child. Goto its rightmost child, that is answer.
iii. Else starting from current node, move to parent nodes, return the parent of first node that is right child of its parent.

Successor:
We can simply exchange the words left and right :)

i. Find the node.
ii. If the node has right child, move to right child and return its leftmost child.
iii. Else moving to parents, return the parent of first node that is left child of its parent.

b. If the parent pointer is not given.
If the successor exist in subtree, we can use the same approach. But what about the successor that is an ancestor? Solution is simple, while traversing the tree, we will set a pointer to the node from where we are coming. We do this in when moving to right subtree. Similarly we can find the predecessor while moving to left subtree.
Again, the only difference between binary tree and BST is about the searching method.

```
// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{

    if(root == NULL) return;

    if(root->key == key){

        // Find the predecessor
        if(root->left) { // If root->left exist
            Node* t = root->left;
            while(t->right){
                t = t->right;
            }
            pre = t;
        }

        // Find the successor
        if(root->right) { // If root->right exist
            Node* t = root->right;
            while(t->left)
                t = t->left;
            suc = t;
        }
        return; // We don't need to recur anymore.
    }

    if(root->key > key) // In Binary tree we do this when we go 
left
    {
        suc = root; // because we will return here
        findPreSuc(root->left, pre, suc, key) ;
    }
    else { //In Binary tree we do this when we go right
        pre = root; // because we came from here
        findPreSuc(root->right, pre, suc, key) ;
    }
}
```

121. Floor and Ceil in a BST

a. One simple way is to traverse the entire tree and return the value less than equal to (floor) and value greater than eqaul to (ceil) the given key. This approach will take O(N) time and O(N) extra space.

b. We can also maintain a variable and find the latest value that is lesser than our key but greatest of all. That value is surely the floor value. We can achieve the same for the ceil.

The code below is from GFG:

```
int floor(Node* root, int key)
{
    if (!root)
        return INT_MAX;

    /* If root->data is equal to key */
    if (root->data == key)
        return root->data;
    /* If root->data is greater than the key */
    if (root->data > key)
        return floor(root->left, key);

    /* Else, the floor may lie in right subtree
      or may be equal to the root*/
    int floorValue = floor(root->right, key);
    return (floorValue <= key) ? floorValue : root->data;
}
```

122. Find Kth smallest / Kth largest Element in the BST.

a. The inorder traversal of the tree will give us the sorted array, we can easily find the kth largest and smallest element. But we will use extra space of O(N) this way. If we don't store the array and pass the reference of a counter and ans, it will still take O(N) space for the stack.

```
class Solution {
    void smallest(TreeNode* root, int k, int& i,int& ans){
        if(root){
            smallest(root->left,k,i,ans);
            i++;
            if(i==k)ans= root->val;
            smallest(root->right,k,i,ans);
        }
    }
public:
    int kthSmallest(TreeNode* root, int k) {
        int i=0,ans=-1;
        smallest(root,k,i,ans);
        return ans;
    }
};
```

b. If we don't want to store the array, we can traverse the array without recursion and as soon as we get the i'th element popped from the stack, we return the answer. This approach may still take O(N) extra space in worst case but it will be faster.

```
int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> stk;
        while(true){
            while(root){
                stk.push(root);
                root = root->left;
            }
            root = stk.top();
            stk.pop();
            if(--k==0)return root->val;
            root = root->right;
        }
  }
```

c. If we are allowed to change the implementation if the BST, we can maintain a count of insertion of the nodes and kth element. But it is not a too good idea because we don't get to reuse it for other values of k.

One more way is to create a threaded binary tree so that we can answer each query in O(k) time at max. Using the morris traversal will cost O(1) extra space and it will be reusable for all the valid values of k.

123. Find a pair with given sum in the BST.

In a BST, you need to find two distinct nodes whose sum is equal to a given number k. Your task is to check whether such nodes exist or not.

a. We can use the property of BST that its inorder traversal is a sorted array. We find the array and then we can use the two pointer method to check whether there are two nodes with sum = k or not.

```
class Solution {
public:
    void inorder(TreeNode* root, vector<int>& v){
        if(root){
            inorder(root->left,v);
            v.push_back(root->val);
            inorder(root->right,v);
        }
    }
    bool findTarget(TreeNode* root, int k) {
        vector<int> v;
        inorder(root,v);
        int i=0,j=v.size()-1;
        while(i<j){
            int sum = v[i]+v[j];
            if(sum == k) return true;
            if(sum>k){
                j--;
            }else i++;
        }
        v.clear();
        return false;
    }
};
```

b. Another way is to use a threaded binary tree. We will need to traverse the successor and predecessor both and then we need to find the answer using two pointers apprach. This way we don't need extra space.

124. BST iterator

a. We can store the inorder traversal in an array and imitate the iterator.

```
class BSTIterator {
public:
    int current = 0;
    vector<int> inorder ;
    BSTIterator(TreeNode* root) {
       BSTtraverse(root);
    }
    void BSTtraverse(TreeNode* root){
     if(root) {
            BSTtraverse(root->left);
            inorder.push_back(root->val);
            BSTtraverse(root->right);
        }
    }
    /** @return the next smallest number */
    int next() {
        if(hasNext())
            return inorder[current++];
        return -1;
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return current<inorder.size();
    }
};
```

b. Instead of traversing the array beforehand, we can use a stack to imitate the recursive procedure. This way we will need some extra space but it will be lesser than recursive approach.

```
class BSTIterator {
public:
    stack<TreeNode*> stk;
    BSTIterator(TreeNode* root) {
        // Move to the left most node.
        while(root){
            stk.push(root);
            root = root->left;
        }
    }

    /** @return the next smallest number */
    int next() {
        // Topmost node is the next node.
        TreeNode* topmost = stk.top();
        stk.pop();

        // We need to maintain the invariant. So if a right child
        // exist, goto its leftmost node. Otherwise we will pop
the next
        // node from stack in next call.

        if(topmost->right) {
            TreeNode* t = topmost->right;
            while(t)
```

```
                {
                    stk.push(t);
                    t = t->left;
                }
            }

            return topmost->val;
        }

        /** @return whether we have a next smallest number */
        bool hasNext() {
            return !stk.empty();
        }
};
```

125. Find the largest BST in the given binary tree.

You are given a binary tree, one or more subtree may be a binary search tree. If the size of the tree is defined as number of nodes in the tree, find the size of largest subtree such that it is also a BST.

a. One simplest way is to check all the subtrees of this tree whether they are BST or not. This approach will take O(N*N) (N for traversing all nodes and N to check for subtree).

b. Instead of checking from top to bottom, we can use a bottom up approach with recursive calls. A subtree is a BST if its left subtree and right subtree both are BST as well as the root is greater than max value in left subtree and less than greatest value in right subtree.

```
# Return the size of the largest sub-tree which is also a BST
ans = 0
def minmaxsize(root):
    global ans
    if(root == None):
        return float('+inf'), float('-inf'), 0, True
    leftmin,leftmax,leftsize,isleftbst = minmaxsize(root.left)
    rightmin,rightmax,rightsize,isrightbst =
minmaxsize(root.right)

    cur = root.data
    isthisbst = False
    if isleftbst and isrightbst and cur > leftmax and cur <
rightmin:
        ans = max(ans, leftsize + rightsize + 1)
        isthisbst = True

    return min(cur,leftmin,rightmin), max(cur,leftmax,rightmax),
leftsize + rightsize + 1, isthisbst

def largestBst(root):
    global ans
    ans = 0
    minmaxsize(root)
    return ans
```

126. Serialize and Deserialize Binary Tree.

Your task is to write two methods:

string Serialize(TreeNode * root) : Return a string representing the tree.
TreeNode * deserialize(string data): Return the root of the Tree.

Before moving to the solution, lets see some easier versions of this problem:

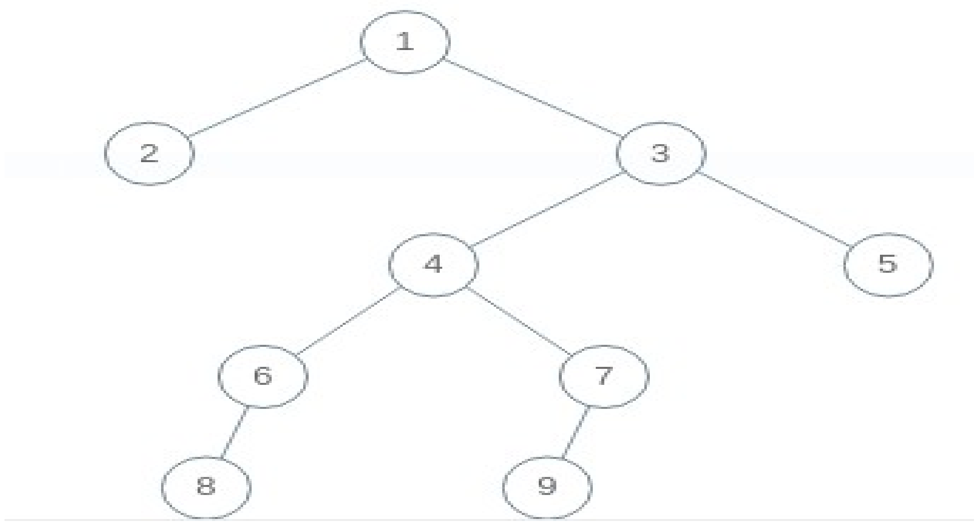
If given Tree is a Binary Search Tree: We can store the preorder/postorder traversal of the tree. The sorted version is inorder traversal so we can rebuild the tree using them.

If the given tree is Complete Tree: A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible. Using this definition, we can use the binary heap representation. Left and Right child of each node is at 2*i +1 and 2*i + 2 position.

If given Binary Tree is Full Tree?
A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

a. For a general binary tree. One way is to store preorder/postorder, inorder traversal of the tree. This way we can restore the binary tree.

b. One way is to store the BFS traversal. For each node, we push the left and right node. This is similar to what leetcode does. Consider the follwing tree:



[1,2,3,null,null,4,5,6,7,null,null,8,null,9]

We can take the non null nodes from front and set next 2 ndoe as left right nodes. We can recursively build the tree that way.
Observe the code below:

```
class Codec:

    def serialize(self, root):
        def doit(node):
            if node:
                vals.append(str(node.val))
                doit(node.left)
                doit(node.right)
            else:
                vals.append('#')
        vals = []
        doit(root)
        # print(' '.join(vals))
        return ' '.join(vals)

    def deserialize(self, data):
        def doit():
            val = next(vals)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = doit()
            node.right = doit()
            return node
        vals = iter(data.split())
        return doit()
```

:::: I will update more methods that are cleaner to understand as soon as I discover them ::::

127. Given a binary tree, convert the tree into a doubly linked list.

The doubly linked list must contain inorder traversal of the tree. We can use the tree node structure as dll.

a. We can recursively convert all the left subtree and right subtree into linked list and after that merge the root in between them. https://www.geeksforgeeks.org/in-place-convert-a-given-binary-tree-to-doubly-linked-list/

b. One more simpler way is to perform the inorder traversal and for every node, append that node to the tail of our DLL. After that we will simply return the list.

```
// This function should return head to the DLL
Node * headofdll, *tailofdll;
void myinorder(Node *root) {
    if(root){
        myinorder(root->left);
        Node* t = newNode(root->data);
        t->left = tailofdll;
        tailofdll->right = t;
        tailofdll = t;
        myinorder(root->right);
    }
}
Node * bToDLL(Node *root)
{
    headofdll = newNode(-1);
    tailofdll = headofdll;
    myinorder(root);
    headofdll->right->left = NULL;
    return headofdll->right;

}
```

c. If we need to convert the tree inplace, we can use the stack based approach to traverse the tree while converting each node to DLL node. For each node, find the successor and predecessor of that node and assign them as the left and right of the current node.

128. Find the running median of data stream.

a. Simplest way is to sort the array each time we add a new element, and then return the median. We can also use binary insertion sort to keep the array sorted. Both ways the complexity will be $O(N^2logN)$ to $O(N^3)$.

b. An efficient way is to use two heaps, one max heap that will represent first half of the array and one min heap that represent second half. For median we need only max of first half and min of second half.
For each element, we need to maintain in which heap we should push the element and if necessary we will pop the top element and push into other heap. Observe the code below:

```
class MedianFinder {
public:
    /** initialize your data structure here. */
    priority_queue<int> maxheap;
    priority_queue <int, vector<int>, greater<int>> minheap;
    int i;
    MedianFinder() {
        i=0;
    }

    void addNum(int num) {
        if(i==0){
            maxheap.push(num);
            ++i;
            return;
        }
        else if(i==1){
            if(maxheap.top()>num){
                minheap.push(maxheap.top());
                maxheap.pop();
                maxheap.push(num);
            } else{
                minheap.push(num);
            }
            ++i;
            return;
        }
        if(i%2==0){ // means both heap have same size
```

```
            if(num<=maxheap.top() || num<=minheap.top()){
                maxheap.push(num);
            } else {
                maxheap.push(minheap.top());
                minheap.pop();
                minheap.push(num);
            }
        } else { // Means maxheap has one extra element.
            if(num>=maxheap.top() || num>= minheap.top()){
                minheap.push(num);
            } else {
                minheap.push(maxheap.top());
                maxheap.pop();
                maxheap.push(num);
            }
        }
        ++i;
    }

    double findMedian() {
        if(i&1){
            return maxheap.top();
        } else {
            return 1.0*(maxheap.top() + minheap.top())/2;
        }
    }
};
```

Time Complexity: O(logN) to insert element, O(1) to find median.

c. Since maintaining two heaps is a complicated process, we need a single data structure for keeping the sorted sequence. A linear data structure like array (with insertion sort) and linkedlist (with merge node) will take linear time, so we can use a self balancing tree. A red-black tree will take atmost O(logN) time to insert a new element. Since it is balanced, the median is either the root itself, or root and one of the child node. We have to maintain two pointers and after insertion wherever side becomes lengthier, we increase or decrease the pointers.
See the code below:

```
class MedianFinder {
public:
    /** initialize your data structure here. */
    int size;
    multiset<int> rbtree;
    multiset<int>::iterator l,r;

    MedianFinder() {
        size=0;
        l = rbtree.end();
        r = rbtree.end();
    }

    void addNum(int num) {
        if(size == 0){vector<int> Solution::dNums(vector<int> &A,
int B) {
    vector<int> ans;
    unordered_map<int , int> count;
    for(int i=0;i<B;i++){
        count[A[i]]++;
    }
    if(A.size() >= B-1){
        ans.push_back(count.size());
    }
    for (int i=B,n=A.size();i<n;i++){
        ++count[A[i]];
        --count[A[i-B]];
        if(count[A[i-B]] == 0){
            count.erase(A[i-B]);
        }
        ans.push_back(count.size());
    }
    return ans;
}
            rbtree.insert(num);
            l = r = rbtree.begin();
            ++size;
            return;
        }
        rbtree.insert(num);
        if(size&1){ // Tree size is odd and l = r

            if(num >= *l){ // Because multiset inserts new element
at last
                ++r;
            } else {
                --l;
            }
        } else{ // If tree size is even, ie l!=r
             // consider 1 2 4 7 8 9 and try to insert 3,4,5,6,7,8
independently
            if(num<*l){
                --r;
            } else if(num>=*r) {
                ++l;
            } else if(num>=*l && num<*r){
                ++l;
                --r;
            }
        }
        ++size;

    }

    double findMedian() {
        if(size&1){
            return *l;
        } else
            return 1.0*(*l+*r)/2;
    }
};
```

129. Kth largest element in a data stream.

a. If we maintain the stream in sorted order, the Kth largest element will not change until we add an element larger than current kth larger. If we insert a large element it will be the next element. We can use multiset as an RBTree. Time Complexity: logN to add new element, O(1) to return kth largest, O(NlogN) for preprocessing.

```
class KthLargest {
public:
    multiset<int> rbtree;
    multiset<int> :: iterator kth;
    int K;
    KthLargest(int k, vector<int>& nums) {
        for(auto i: nums){
            rbtree.insert(i);
        }
        K=k;
        if(rbtree.size()>=k){
            kth = rbtree.begin();
            for(int i=0;i<rbtree.size() - k;i++){
                ++kth;
            }
        }
    }

    int add(int val) {
        rbtree.insert(val);
        if(rbtree.size() == K){
            kth = rbtree.begin();
            return *kth;
        }
        if(val>=*kth)++kth;
        return *kth;
    }
};
```

b. In the above solution, we can see that we are only interested in K elements only. So if we only store the largest K elements while preprocessing, we need to return only minimum element of the container having K element(that will be kth largest ofcourse).
 Here I have used a minheap so that I can easily insert new element and access min element in O(1).

```
class KthLargest {
public:
    // min heap
    priority_queue< int, vector<int>, greater<int>> heap;
    int K;
    KthLargest(int k, vector<int>& nums) {
        K = k;
        for(int i=0,n=min((int)nums.size(),k);i<n;i++){
            heap.push(nums[i]);
        }
        for(int i=k,n=nums.size();i<n;i++){
            if(nums[i]>heap.top()){
                heap.pop();
                heap.push(nums[i]);
            }
        }
    }

    int add(int val) {
        if(heap.size() == K){
            if(val > heap.top()){
                heap.pop();
                heap.push(val);
            }
            return heap.top();
        } else{
            heap.push(val);
            return heap.top();
        }
    }
};
```

130. Distinct Elements in a Window.

You are given a stream of integers A and a window size B. Your task is to count the number of distinct elements in each window of size B. It is possible that B is greater than size of A.

a. We can use a hashmap to store the count and whenever we leave a number behind while sliding the window, we will decrease its count and delete it if it is 0.

```
vector<int> distinctInWindow(vector<int> &A, int B) {
    vector<int> ans;
    unordered_map<int , int> count;
    for(int i=0;i<B;i++){
        count[A[i]]++;
    }
    if(A.size() >= B-1){
        ans.push_back(count.size());
    }
    for (int i=B,n=A.size();i<n;i++){
        ++count[A[i]];
        --count[A[i-B]];
        if(count[A[i-B]] == 0){
            count.erase(A[i-B]);
        }
        ans.push_back(count.size());
    }
    return ans;
}
```

131. Find Kth largest/Smallest element in unsorted array.

Here we will see the solution for Kth largest only, Kth smallest can be found using similar method.

a. Sort the array and return kth element from back.O(NlogN)

```
class Solution {
public:
    int findKthLargest(vector<int>& a, int k) {
        sort(a.begin(),a.end());
        return a[a.size() - k];
    }
};
```

b. We can use the minheap method we used in Kth largest element from data stream because concept is similar. O(KlogK + NlogK)

```
class Solution {
public:
    int findKthLargest(vector<int>& a, int k) {
        priority_queue<int , vector<int>, greater<int>> minheap;

        for(int i=0;i<k;i++)minheap.push(a[i]);
        for(int i=k;i<a.size();i++){
            if(a[i]>minheap.top()){
                minheap.pop();
                minheap.push(a[i]);
            }
        }
        return minheap.top();
    }
};
```

c. If we closely look at Quick Sort , this algorithm takes each element to its correct place after each partition. So we need to check for each partition and the moment we see that partition has swapped a pivot to kth element , we return. In best case, complexity is O(N) when we find the pivot in one partition, otherwise it can go upto $O(N^2)$.

132. Flood Fill Algorithm.

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

Example 1:
Input:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation:
From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected
by a path of the same color as the starting pixel are colored with the new color.
Note the bottom corner is not colored 2, because it is not 4-directionally connected
to the starting pixel.

a. We can use recursion to solve this problem.
If the given pixel is of same color as new color, we don’t need to proceed further so return.
Otherwise color this pixel and call the floodFill function for all 4 directions.

Below is the implementation:
```
class Solution {
public:
    vector<vector<int>>& floodFill(vector<vector<int>>& image, int sr, int sc,
int& newColor) {
        if(image.size()){
            if(image[sr][sc] == newColor)return image;
            int prevcolor = image[sr][sc];
            image[sr][sc] = newColor;
            if(sr-1 >= 0 && image[sr-1][sc] == prevcolor){
                floodFill(image, sr-1, sc, newColor);
            }
            if(sr+1 < image.size() && image[sr+1][sc] == prevcolor){
                floodFill(image, sr+1, sc, newColor);
            }
            if(sc-1 >= 0 && image[sr][sc-1] == prevcolor){
                floodFill(image, sr, sc-1, newColor);
            }
            if(sc+1 < image[0].size() && image[sr][sc+1] == prevcolor){
                floodFill(image, sr, sc+1, newColor);
            }
        }
        return image;
    }

};
```

We can also speed up this solution by using a custom stack and converting the recursion to iteration.

133. Clone an undirected connected graph.

This problem is a little bit tricky. See this article.

We have to clone the nodes while traversing them using BFS. To clone a node, we will use a hashmap to store their references as OriginalNode -> DuplicateNode. To clone the neighbours, we will create new nodes iff they are not available in hashmap(So that we don't create two or more references of same node.) We will create a copy, store it in a map (If node already created) and finally we will push that reference into that original node's clone (Using hashmap).

Read the code below for better understanding.

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> neighbors;

    Node() {
        val = 0;
        neighbors = vector<Node*>();
    }

    Node(int _val) {
        val = _val;
        neighbors = vector<Node*>();
    }

    Node(int _val, vector<Node*> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
};
*/

class Solution {
public:
    Node* cloneGraph(Node* node) {
        // If graph is empty return
        if(!node) return node;

        // Initialise a map and queue
        unordered_map<Node*, Node*> mp; // Store orginal -> clone
reference
        queue<Node*> q; // For BFS traversal

        // Make copy of first node (or root node)
        Node* nd = new Node();
        nd->val = node->val;
        mp[node] = nd; // Store it's reference


        // Start BFS
        q.push(node);
        Node * qnode;
        while(!q.empty()){
            qnode = q.front();
            q.pop();

            // TO avoid typing too much, I used a macro here
            // so its a reference to vector
            #define neigh qnode->neighbors

            // We will check each neighbor of original node
            for(int i=0;i<neigh.size();i++){

                // If its reference hasn't been created yet,
create it
                if(mp.find(neigh[i])==mp.end()){

                    nd = new Node();
                    nd->val = neigh[i]->val;

                    // Store it in map
                    mp[neigh[i]] = nd;

                    // and since we are visiting it first time
                    // push it into queue
                    q.push(neigh[i]);
                }

                // Finally old node or freshly created, push it
into
                // clone node's neighbors.
                mp[qnode]->neighbors.push_back(mp[neigh[i]]);
            }
        }
        #undef neigh

        // Return clone node.
        return mp[node];
    }
};
```

134. BFS in graph.
We have used BFS many times, just keep in mind that in case of undirected graphs, we only push unvisited nodes. So we may need a hashmap to store visited array.

135. DFS in graph.
DFS is also quite simple, for undirected graphs, we also pass a parent’s pointer so that we do not visit it again. We can also use a hashmap to keep track of unvisited node instead.

136. Detect cycle in directed and undirected graph.

Undirected Graph

a. We can use DFS to visit each node. For a node, if there is an **already visited neighbour,** that will mean there would have been another node from where we can come here, ie a cycle.

```
bool dfs(int node, int parent, vector<bool>& visited, vector<int>
g[]){
    // Visit this node
    visited[node] = true;

    // Visit each neighbour
    for(auto i: g[node]) {
        // Ignore the node from where we came
        if(i!=parent){

            // If it is visited, there is a cycle
            if(visited[i])
                return true;

            // If there is a cycle in neighbour, return true
            if(dfs(i,node,visited,g))
                return true;
        }
    }

    // Otherwise there is no cycle so return false.
    return false;
}
bool isCyclic(vector<int> g[], int V)
{
    // Initialise a mark array
   vector<bool> visited(V,false);

   // Apply dfs from each unvisited node. If a cycle exist
   // return true
   for(int i=0;i<V;i++){
       if(!visited[i] && dfs(i, -1, visited, g))
        return true;
   }
   // Else there is no cycle
   return false;

}
```

Directed Graph

a. This is a little trickier in case of indirected graph. If we use a similar approach as undirected graph, we can not answer correctly for a case like:

(2) -> (11) <- (9)

Here if we apply dfs from 2 and 9, it will detect a cycle because 11 is already visited by either of them. Hence the answer will be incorrect because there is not any cycle.

The correct approach is to keep track of all the nodes visited in current recursion. If we find that we are revisiting a node that was visited in current recursion, we return true. To do that we can mark each node in a seperate array when we visit them, and unmark them when we exit the recursion. This way we will know what elements are there in this chain.

Below is the code:
```
bool dfs(int node, vector<bool>& visited, vector<int>
adj[],vector<bool>& instack) {
    visited[node] = true;
    instack[node] = true;
    for(auto i:adj[node]) {
        if(visited[i]){
            if(instack[i])
                return true;

        } else{
            if(dfs(i,visited,adj,instack))
                return true;
        }

    }
    instack[node] = false;
    return false;
}
bool isCyclic(int V, vector<int> adj[])
{
    vector<bool> visited(V,false), instack(V,false);
    for(int i=0;i<V;i++) { // Call for all nodes if graph is not
                          //  connected
        if(!visited[i] && dfs(i,visited,adj,instack)){
            return true;
        }
    }
    return false;
}
```

137. Topological sort.

Topological sorting of a graph is a special order of nodes such that for every edge u -> v, u comes before v in the topological sorted sequence.
Graph must be directed and acyclic, ie DAG. There can be more than one source nodes in a DAG ie a node where we can not reach from any other node.

a. The idea is to apply DFS for each node(because there can be more than one source nodes). For each node, whenever there is no more unvisited neighbor left, we will push it into a stack. After we apply dfs for all the nodes, we will print the stack from top to bottom.

Since we push the nodes that don’t have unvisited neighbors at last, the source nodes will always be on top. And even if a node that is not a source node, is above a source node, that will have no way to reach the source, so topological order will be maintained.

Topological sorting is used in build systems. Suppose a system needs to install too many modules but the modules depend on each other. The task is to determine which module to install first so that there is no dependency issue. Here we can use topological sorting and install everything in order.

Below is the code:

```
void dfs(int node, vector<int> adj[], vector<bool>& visited,
stack<int>& s ) {
    // If this node has been visited by another source, leave it.
    // Otherwise apply dfs on it.
    if(!visited[node]) {

        // Visited the node.
        visited[node] = true;

        // Now visit all the unvisited neighbors
        for (auto i: adj[node]){
            dfs(i,adj, visited,s);
        }

        s.push(node);
    }
}
vector<int> topoSort(int V, vector<int> adj[]) {
    // Initialise a mark array and stack.
    stack<int> s;
    vector<bool> visited(V,false);

    // Apply dfs for each node.
    for(int i=0;i<V;i++){
        // Note that our dfs doesn't require parent
        // because directed acyclic graph won't get back to parent
        dfs(i, adj, visited, s);
    }

    // Now our ans is the nodes in stack from top to bottom.
    vector<int> ans;
    while(!s.empty()){
        ans.push_back(s.top());
        s.pop();
    }
    return ans;
}
```

138. Number of islands.
In a 2D grid of 0 and 1 only, 1 represent land and 0 represent water. Your task is to find number of island in the grid where an island is surrounded by water(0’s) and formed by connecting adjacent lands (NWSE, not diagonals).

a. We can use the same approach as flood fill algo. We will mark each land, and count the unvisited ones. See the code below:

```
void dfs(int r, int c, vector<vector<char>>& grid,
vector<vector<bool>>& mark) {
        if(r<0 || c<0 || r>=grid.size() || c>=grid[0].size())
            return;
        if(grid[r][c] == '0' || mark[r][c])return;
        mark[r][c] = true;
        dfs(r-1,c,grid,mark);
        dfs(r+1,c,grid,mark);
        dfs(r,c-1,grid,mark);
        dfs(r,c+1,grid,mark);
    }
    int numIslands(vector<vector<char>>& grid) {
        if(grid.size() == 0) return 0;

        vector<vector<bool>>
mark(grid.size(),vector<bool>(grid[0].size(),false));

        int count=0;

        for(int r=0; r<grid.size();r++) {
            for(int c=0;c<grid[0].size();c++) {
                if(grid[r][c] == '1' && !mark[r][c]){
                    count++;
                    dfs(r,c,grid,mark);
                }
            }
        }
        return count;
    }
```

139. Check whether the given graph is Bipartite.

A graph is bipartite, if we can partition the nodes into two different sets such that for every edge u->v, both u and v are in different sets.

a. We can use the graph coloring property for this problem. For a bipartite graph, the chromatic number is 2. For each node, we will color the node with a color (set A) and for each adjacent node, if it is not already colored, we will color it with a different color(set B). If the adjacent node is already colored, but with the same color as current node, we return false because these nodes are in same set.

We can do this using BFS and DFS both. Below implementation uses BFS. Colors are stored in a vector. ‘0’ means uncolored, ‘1’ means set A and ‘2’ means set B. So for each node popped from queue, we will try to color their adjacent nodes.

```
bool isBipartite(vector<vector<int>>& graph) {
        // Array for colors
        vector<int> color(graph.size(),0);
        // Queue for BFS
        queue<int> q;

        // node to store q.front
        int node;
        // Do it for each uncolored node if graph can be 
disconnected
        for(int i=0;i<graph.size();i++){
            // If node is not colored, apply bfs
            if(!color[i]) {
                q.push(i);
                color[i] = 1;

                while(!q.empty()) {
                    node = q.front();
                    q.pop();
                    for(auto child:graph[node]){
                        if(!color[child]){ // Fill the opposite 
color in children nodes
                            color[child] = (color[node]==1?2:1);
                            q.push(child);
                        } else {
                            // If a child is already colored with 
same color return false
                            if(color[child] == color[node] )
                                return false;
                        }
                    }
                }
            }
        }

        // If chromaticity is maintained as 2, return true.
        return true;
    }
```

140. Kosaraju's algorithm for Strongly Connected Components(SCC).

SCC: Strongly Connected comoponents are the subgraphs of a directed graph such that for every pair (u,v) in that subgraph, there is a path from u to v and from v to u.

Kosaraju's Algorithm: This is a 3 step algorithm that uses DFS.
Initialise a stack and an array to mark the visited nodes.
1. For each node, if it is not visited yet, apply DFS and push each node into stack after all of its children are visited(See the code of DFS).
2. Reverse all the edges of the graph.
3. Pop nodes from the stack one by one, and apply dfs on unvisited nodes again. Once an unvisited node is found, all the nodes on stack that are visited after that node, are the members of that component.
See detailed video and algorithm here:
https://www.youtube.com/watch?v=RpgcYiky7uw
https://www.geeksforgeeks.org/strongly-connected-components/
This algorithm work because we store all the connected chains in the stack just like we did in Topological sorting. After that when we reverse the edges and use dfs again, only those nodes that are strongly connected, can visit the same nodes again. Other nodes are either non reachable or they

are already visited because of the sorted ordering (by Finish time) in stack. We can also dry run the concept on a graph. When we reverse the edges, only SCC will have same DFS/BFS traversal as before.

We can notice that SCC is about reachability, so it isn't a concept of Undirected Graph because all nodes are reachable in an undirected connected graph.

Code:
```
void dfs(int node, vector<bool>& visited, stack<int>& s,
vector<int> adj[]) {
    // Do nothing if this node is already visited
    if(visited[node])
        return;

    // Mark it visited
    visited[node] = true;

    // Call DFS for all the neighbours
    for(auto child: adj[node]) {
        dfs(child, visited, s, adj);
    }

    // Push this node on the stack
    s.push(node);
}
int kosaraju(int V, vector<int> adj[])
{
    /*
    Step 1 : Apply DFS on all unvisited nodes. Push them in stack
             after the DFS ends;
    */

    vector<bool> visited(V,false);
    stack<int> s;

    for(int i=0;i<V;i++) {
        if(!visited[i]) {
            dfs(i,visited, s, adj);
        }
    }

    /*
        Now the stack is filled with those nodes.
    Step 2 : Reverse the edges.
    */

    vector<int> transposedGraph[V];
    for(int i=0;i<V;i++) {
        for (auto node: adj[i]) {
            transposedGraph[node].push_back(i);
        }
    }
    /*
        Step 3 : Now pop nodes from stack and use dfs again.
        For each unvisited node popped from stack, create a new
SCC.

        For this task, we only need count. Otherwise we can store
all the
        nodes popping from stack that are visited after a dfs on
unvisited node.
    */
    fill(visited.begin(), visited.end(), false);
    int count = 0;


    while(!s.empty()) {

        // Pop each node one by one
        int i = s.top();
        s.pop();

        // Use dfs if it is unvisited.
        if(!visited[i]) {
            // Append the nodes in a list if you also want the SCC
nodes
            ++count;

            // Apply dfs
            dfs(i,visited, s, transposedGraph);
        }
    }

    return count;

}
```

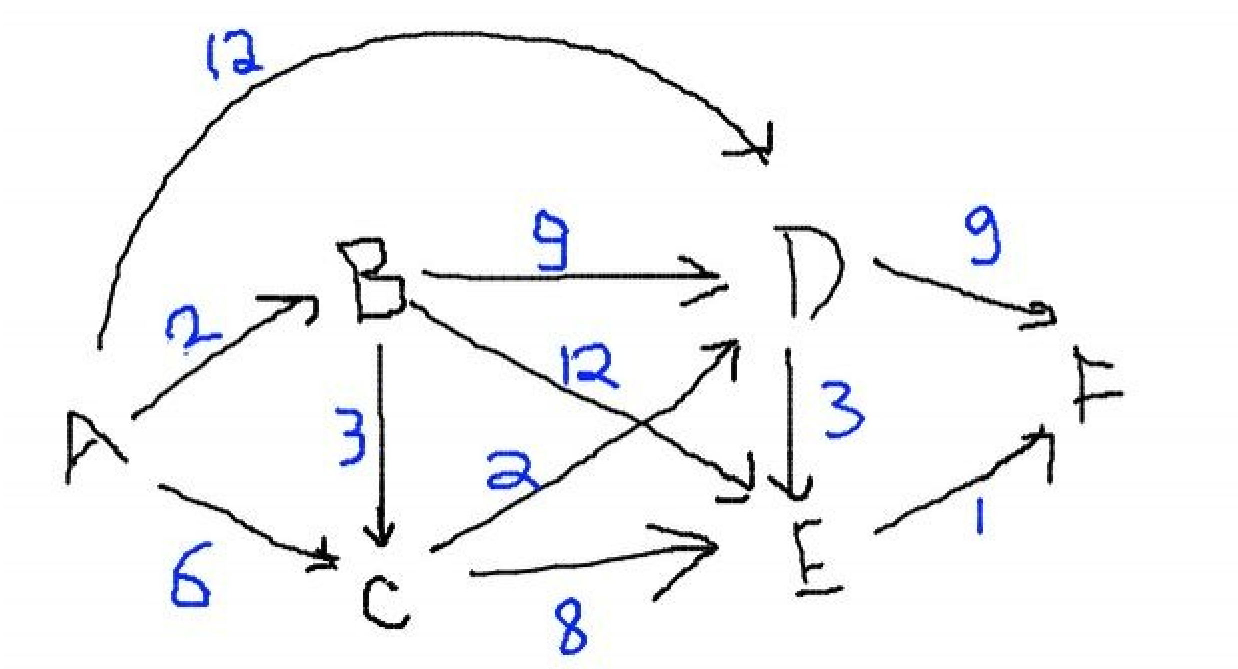
// For the graph algorithms below, I will add codes later. I am looking for online judges to test my functions.

141. Dijkstra Algorithm for Single Source Shortest Path.

Dijkstra algorithm is used to find the shortest path from a source node to all other nodes in the weighted graph. But this algorithm **can not detect negative cycles** in the graph. It is a greedy algorithm.

1. For the source node, the distance is 0. For other nodes it is INF initiallty.
2. We extract the unvisited node having minimum distance from source.
3. From this node, we visit all the unvisited neighbors and update their distance using triangle inequality. We also update the parent node if needed.
4. Repeat point 2 until there is an unvisited node left.
5. The distance array represent minimum distance and parent array represent parents to determine path.

Example:



Distance						Parent					
A	B	C	D	E	F	A	B	C	D	E	F
0	∞	∞	∞	∞	∞	-	-	-	-	-	-
/	**2**	6	12	∞	∞		A	A	A	-	-
	/	**5**	11	14	∞			B	B	-	-
		/	**7**	13	∞				C	C	-
			/	**10**	16					D	D
				/	**11**						E

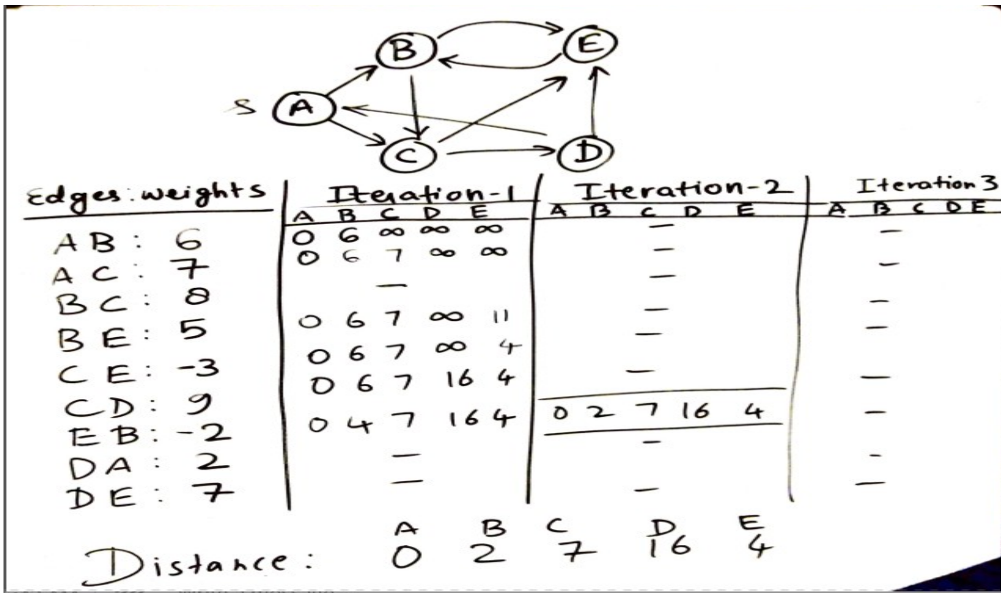
In this algorithm, we visit nodes in BFS style. So it takes O(E) to visit entire graph. Along with that, finding min at each step can take O(V) for each step. But if we make a minheap, this operation will take O(logV) time for each node. So overall complexity will be:

If we use min heap: O(V + E) * O(log V) = O(ElogV) (E >= V – 1 in connected graph)
Otherwise we will visit E edges with O(V) time to find min distance, that will be $O(V^2 + VE) = O(VE)$.

142. Bellman Ford Algorithm
Bellman Ford algorithm is also a single source shortest path algorithm. But **it can also detect negative weight cycles**. It is based on Dynamic Programming.

1. Initialise a distance array. Source distance is 0 and INF for all other nodes.
2. Let V = number of nodes in the graph.
3. Repeat step 4 V-1 times.
4. For each unvisited edge (u,v) , (It can be picked in BFS style.)
   Update distance[v] = min(distance[v] , distance[u] + w(u,v))
5. distance array represents the minimum distance from source. But it can be invalid if graph contains negative weight cycle.

6. For each edge (u,v), if distance[v] > distance[u] + w(u,v): There is a cycle.
Example:



Edges : weights	Iteration-1 A B C D E	Iteration-2 A B C D E	Iteration 3 A B C D E
AB : 6	0 6 ∞ ∞ ∞	—	—
AC : 7	0 6 7 ∞ ∞	—	—
BC : 8	—	—	—
BE : 5	0 6 7 ∞ 11	—	—
CE : -3	0 6 7 ∞ 4	—	—
CD : 9	0 6 7 16 4	—	—
EB : -2	0 4 7 16 4	0 2 7 16 4	—
DA : 2	—	—	—
DE : 7	—	—	—

	A	B	C	D	E
Distance :	0	2	7	16	4

This algorithm uses V iterations and visits E edges. So time complexity is O(VE). But dijkstra algorithm can be implemented in O(ElogV) so Bellman Ford is slower if negative weight cycles are not present.

143. Floyd Warshall Algorithm.

This algorithm calculates the shortest distance from each node U to each node V.
This algorithm uses adjacency matrix only. The reason is that it makes the edges during computation. That is quite complicated in adjacency list. If the graph is too sparse, using dijkstra is a better option. It is based on Dynamic Programming.

```
1. Initialise two matrix, one for distance, one for parent(If needed).
        Initially distance[u][u] = 0, par[u][u] = NULL
        distance[u][v] = W(u,v) if E(u,v) exist else INF
        par[u][v] = u if E(u,v) exist else NULL
2. For each vertex X:
     For each vertex U:
        For each vertex V:
            if(distance[U][V] > distance[U][X] + distance[X][V])):
                distance[U][V] = distance[U][X] + distance[X][V]
                 parent[U][V]   = parent[X][V]
```
Example:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

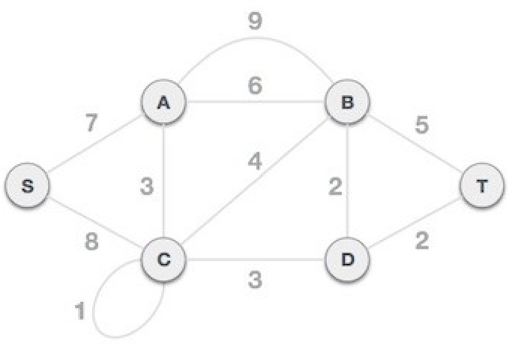
$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Time complexity of this algorithm is $O(V^3)$. For a too sparse matrix, we can also use dijkstra for better speed.

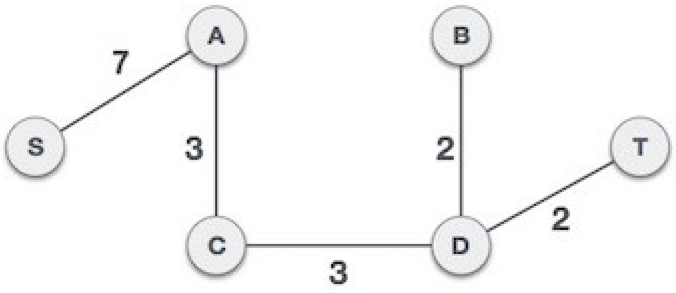
144. MST using Prim’s algoritm.

Prim’s algorithm is a greedy algorithm to find the minimum spanning tree. The idea behind this algorithm is to connect each disconnected node via minimum weighted edge possible. We traverse the graph in BFS style and for an edge (u,v) we will use this edge if both nodes are disconnected or they are connected with a longer/ more costly path.

1. Initialise two arrays for cost and parent (if needed). Cost of source = 0 and parent = NULL. Other nodes will have cost = INF. Source node will be our root node.
2. Take the node with minimum cost and **mark it visited**. Note that we should mark only this node.
3. Visit all the neighbours of this node (and do not mark them yet). For each edge U->V, update it as cost[V] = min(cost[V], W(U,V)). Parent of V will be U now. Parent[V] = U.
4. Repeat step 2 until all nodes are not visited.
5. Finally, sum of cost array is cost of MST. We can build the MST using parent array.
Example:



Cost						Parent					
S	A	B	C	D	T	S	A	B	C	D	T
0	∞	∞	∞	∞	∞	-	-	-	-	-	-
/	**7**	∞	8	∞	∞	-	S	-	S	-	-
	/	6	**3**	∞	∞	-	S	A	A	-	-
		4	/	**3**	∞	-	S	C	A	C	-
		2		/	**2**	-	S	D	A	C	D
		2		**/**	**/**	-	S	D	A	C	D



Time complexity of this algorithm is same as dijkstra algoritm ie O(V^2) for adjacency matrix and O(ElogV) for adjacency list with minheap.

145. Kruskal’s MST.
Kruskal’s algorithm is also a greedy algorithm but it is more costly than prim’s algorithm.

1. Sort the edges in increasing order of their weight.
2. Start from cheapest edge and connect vertices iff the vertices have no path between them yet.
3. Do this until all the nodes are not connected. We will need only V-1 such edges.

For this algorithm, step 1 is easy as we need to sort the weights. The time taking step is step 2 because before inserting an edge, we need to check the connectivity. One option is to use DFS but that will take O(V + E) time for all E edges, resulting in O(E^2) complexity. A better option is to use a disjoint set which can check the connectivity in O(logV) time.

Hence overall complexity is O(ElogE) for sorting and O(V+E)*O(logV) = ElogE + VlogV + ElogV (but E >> V - 1) = 3ElogE = O(ElogE) or O(ElogV)

146. Find the max product subarray.

Given an array, find a contiguous subsequence (subarray) having minimum 1 element, with largest possible product.

Example:
Input: [2,-3,1,0,5,-2,-3]
Output: - 2 * - 3 = 6

Input : [-2 , -3]
Output : -2

a. We can observe that array has only integers. So absolute value of the product will always increase as we take more and more elements. But the problematic elements are 0 and -ve numbers.

Let’s assume that there is no zero present in the array. Negative numbers will only affect us when they are present odd times. Eg [-1 -2 -3 -4] will result in positive product, so we can take the product as answer directly. Otherwise we have to remove atleast 1 negative number from the product.

Now the question is which one to remove? Take an array [a,-b,c,-d,e,-f,g]. Suppose we remove -d from the array. We are breaking the contiguous nature of this subarray. Hence that will result in two subarrays with samller products. If we remove a smaller number (in magnitude), it won’t affect the answer too much. Suppose we remove the leftmost/rightmost negative number, even if it is large, it is making the answer too negative. While any middle element will break the array resulting in small subarray as well as small product. So **the best choice is to remove the leftmost or righmost negative number** from the array.

Now if there are 0 in the array, it is simply working as boundary because at that point, the array product will be 0 no matter what.

We will simply scan the array from left to right and from right to left. We will use kadane like simulation to update the answer as soon as we get a larger product. It will automatically discard rightmost negative element in first scan and leftmost in other. This way we can easily find the answer in linear time in two scans.

```
int maxProduct(vector<int>& a) {
        if(a.size() == 0) return 0;

        int n = a.size(),ans=INT_MIN,product=1;

        // Scan from left to right.
        for(int i=0;i<n;i++){
            product*=a[i];
            ans = max(product, ans);
            product = product == 0 ? 1 : product;
        }

        // scan from right to left.
        for(int i=n-1,product=1;i>=0;i--){
            product*=a[i];
            ans = max(product, ans);
            product = product == 0 ? 1 : product;
        }

        return ans;
    }
```

::Dry run it on arrays with 1, 2 .. 10 elements, it will get more clear::

b. We can also solve it in one pass. The process is simple, we maintain two more variables, maxsofar and minsofar. Maxsofar will hold the maximum positive product and minsofar will hold the minimum negative product.
Initially, ans, maxsofar and minsofar will be first element of array.

From second element, update maxsofar as max of current element, product of maxsofar and current element and product of minsofar and current element. If the minsofar is too small and current element is negative, it will become a large positive value.
Similarly update minsofar as minimum of these three. (use separate variables to avoid comparing updated variables.)
If current element is 0, reset both minsofar and maxsofar.
Answer is max possible maxsofar. See the code below:

```
int maxProduct(vector<int>& a) {
        if(a.size() == 0) return 0;
        int n = a.size(),ans,minsofar,maxsofar;
        ans=minsofar=maxsofar=a[0];
        for(int i=1;i<n;i++)
        {
            if(a[i]==0) {
                maxsofar = minsofar = 0;
                ans = max(ans,0);
                continue;
            }

            int temp1 =
max(a[i],max(minsofar*a[i],maxsofar*a[i]));
            int temp2 =
min(a[i],min(minsofar*a[i],maxsofar*a[i]));
            maxsofar = temp1;
            minsofar = temp2;
            ans = max(ans, maxsofar);
        }
        return ans;
    }
```

147. Longest increasing subsequence.

a. If we fix the starting index, we can use a recursive approach to find the longest subsequence if we choose the current index, and if we don’t:

```
int solve(vector<int>& a, int cur, int prev){
        if(cur == a.size()) return 0;
        int if_cur_taken = 0;
        if(a[cur]>prev){
            if_cur_taken = 1 + solve(a,cur+1,a[cur]);
        }
        int if_not_taken = solve(a, cur+1, prev);
        return max(if_cur_taken, if_not_taken);
    }
    int lengthOfLIS(vector<int>& a) {
        if(a.size()==0) return 0;
        return solve(a,0,INT_MIN);
    }
```

This solution takes O(2^n) time and O(N) stack space. But we can use memoization that will take only O(N^2) space to store an array of dp[cur][prev]. Instead of using prev as value, we will use it’ s index. Also the time complexity will reduce to O(N^2).

b. There is one more way to solve it in O(N) extra space. Observe the follwing example:

Initially, dp[i] =1 because answer is 1 for single element.
For each i, if A[i]>A[j], that means LIS upto ith index is 1 + LIS upto jth index. We will take the max one.

A	10	9	2	5	3	7	101	18	
dp	1								
i=1	1	**1**							
i=2	1	1	**1**						
i=3	1	1	**1**	**2**					
i=4	1	1	**1**	2	**2**				
i=5	1	1	1	**2**	2	**3**			
i=6	1	1	1	2	2	**3**	**4**		
i=7	1	1	1	2	2	**3**	4	**4**	

See the code below:
```
int lengthOfLIS(vector<int>& a) {
        if(a.size()==0) return 0;
        int ans = 1;
        vector<int> dp(a.size(),1);
        for(int i=0,n=a.size();i<n;i++) {

            for(int j=0;j<i;j++) {
                if(a[i]>a[j]){
                    dp[i] = max(dp[i], dp[j]+1);
                    ans = max(ans, dp[i]);
                }
            }
        }
        return ans;
    }
```
Time complexity is O(N^2) but extra space is only O(N).

c. We can do this in O(NlogN) time. Before moving to solution, let’s observe few things.
Suppose array is [2,3,5,7,4]
We can see that if we take only starting 4 elements, it will make our current LIS [2,3,5,7]. If we want to take the LIS ending at 5th element(ie 4), we must skip all the smaller elements. That makes it [2,3,4] But it is smaller. Since there are no more elements, taking this element was not a good idea. But we can assume that we took this element.
How? We will replace it with it’s correct position in the LISsofar. In the above case our LISsofar is [2,3,5,7]. After adding 4, it will become [2,3,4,7]. Although it is not the correct LIS but it will tell us the longest LIS so far. If the array were [2,3,5,7,4,12], we can easily make LISsofar as [2,3,4,7,12] because our last state was not lost, we can still assume that 5 is there instead of 4.

To find the position, we can use binary search. Hence overall time complexity will be O(NlogN).

Reference: https://www.youtube.com/watch?v=TocJOW6vx_I

See the code below:
```
int lengthOfLIS(vector<int>& a) {
        if(a.size()==0) return 0;
        vector<int> seq;
        seq.push_back(a[0]);
        for(int i=1;i<a.size();++i) {
            if(a[i] > seq.back()) {
                seq.push_back(a[i]);
            } else {
                int index =
lower_bound(seq.begin(),seq.end(),a[i]) - seq.begin();
                seq[index] = a[i];
            }
        }

        return seq.size();
    }
```

148. Longest Common Subsequence
Given two arrays or strings, your task is to find the longest subsequence that is present in both of them.

a. The recursive approach is, if the last element of both strings match, we return 1 + LCS(i-1,j-1) otherwise we return max of LCS(i,j-1) and LCS(i-1,j). where i and j are the position where first and second string end.
For **1-based indexing** :

LCS(i,j) = 0 ; if i=0 or j=0
1 + LCS(i-1,j-1) ; if A[i] == B[j]
max(LCS(i-1,j) , LCS(i,j-1)); if A[i] ≠ B[j].

```
class Solution {
public:
    int solve(string& s1, string& s2, int i, int j) {
        if(i==-1 || j == -1) return 0;

        if(s1[i] == s2[j]){
            return 1 + solve(s1,s2,i-1,j-1);
        }

        else {
            return max(solve(s1,s2,i-1,j) , solve(s1,s2,i,j-1));
        }

    }
    int longestCommonSubsequence(string text1, string text2) {
        return solve(text1,text2,text1.size()-1,text2.size()-1);
    }
};
```

This approach will take exponential time but using memoization we can convert this code into O(N^2).

```
class Solution {
public:
    int dp[1000][1000];
    int solve(string& s1, string& s2, int i, int j) {

        if(i==-1 || j == -1) return 0;

        if(dp[i][j])return dp[i][j];

        if(s1[i] == s2[j]){
            dp[i][j] = 1 + solve(s1,s2,i-1,j-1);
            return dp[i][j];
        }

        else {
            dp[i][j] = max(solve(s1,s2,i-1,j) , solve(s1,s2,i,j-
1));
            return dp[i][j];
        }

    }
    int longestCommonSubsequence(string text1, string text2) {
        memset(dp,0,sizeof(dp));
        return solve(text1,text2,text1.size()-1,text2.size()-1);
    }
};
```

b. The dynamic programming approach is :

```
int longestCommonSubsequence(string a, string b) {
        int n = a.size()+1,m=b.size()+1,i,j;
        vector<vector<int>> dp(n,vector<int>(m,0));

        for(i=1;i<n;i++) {
            for(j=1;j<m;j++) {
                if(a[i-1] == b[j-1]) {
                    dp[i][j] = 1 + dp[i-1][j-1];
                } else {
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }
        return dp[n-1][m-1];
    }
```

149. 0-1 Knapsack

In this problem, we are given N items with a weight and value. And a knapsack of capacity W (by weight). Fill the knapsack to maximize the value of the items collected. But we can either take the whole item, or leave it.

a. A recursive approach is, for each item, either take it(if it fits in knapsack) or don’t take it. Below is the memoized approach of recursion:

```
#include<bits/stdc++.h>
using namespace std;
int dp[1001][1001];
int knapsack(int W, vector<int>& w, vector<int>& v, int i) {

    // If no item left or Remaining Capacity not left
    if (i == w.size() || W == 0) return 0;
    if(dp[W][i]) return dp[W][i];

    // If current item has more weight, we are bound to
    // skip it.
    if (w[i] > W) {
        dp[W][i] = knapsack(W, w, v, i+1);
        return dp[W][i];
    }
     // Else we take it or not take it.
    dp[W][i] = max(
            v[i] + knapsack(W-w[i], w, v, i+1),
            knapsack(W, w, v, i+1)
        );
    return dp[W][i];
}
int main()
 {
     int t,i,n,W,temp;
     vector<int> w, v;
     // Number of test cases
     cin>>t;

     while(t--) {
         cin>>n;
         w.clear();
         v.clear();
         memset(dp,0,sizeof(dp));
         // Capacity of knapsack
         cin>>W;

         // Values
         for(i=0;i<n;i++)
        {
            cin>>temp;
            v.push_back(temp);
        }
        // weights
         for(i=0;i<n;i++)
         {
            cin>>temp;
            w.push_back(temp);
        }
         cout<<knapsack(W,w,v,0)<<endl;

     }

     return 0;
}
```

b. Or we can use dynamic programming iteratively over parameters W and i.

```
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(
                    val[i - 1] + K[i - 1][w - wt[i - 1]],
                    K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}
```

150. Edit Distance.
You are given two words A and B. Your task is to convert A into B using three types of operations: Insert a character, Delete a character, Replace a character. Find the Minimum number of operations to perform the task above.

a. Suppose the given strings are :
EDIT
DISTANCE

If we denote each character of first string by i and other by j, lets see how these three operation will work-
Insert:

We can assume that we have inserted an equal character. Then moved both. But we will only move ‘j’ and that will simulate the same process because we don’t care what we have inserted.

```
  i                i                 i
EDIT             ENDIT            ENDIT
      j                j                 j
DISTANCE         DISTANCE         DISTANCE
```

Delete:
Similarly for delete, we assume we have deleted a character. But we will only move ‘i’.

```
  i               i           i
EDIT             EIT        EIT
      j                j                j
DISTANCE         DISTANCE         DISTANCE
```

Replace:
We will assume that we have replaced the character, so move i and j both.

We will recursively solve the subproblem assuming that we have applied all three operations. When either of the string ends, we return the max remaining length, assuming that we will insert replace or delete all the characters there.

Below is the recursive solution:

```
class Solution {
public:
    int solve(string& word1, string& word2, int i, int j) {

        // If string ends, return the max remaining length
        // We assume we will insert characters if word1 ended
        // and delete characters if word2 ended
        if(i == word1.size() || j == word2.size()) {
            return max(word1.size() - i, word2.size()-j);
        }

        // If both are equal, move ahead
        if(word1[i] == word2[j]) {
            return solve(word1,word2, i+1,j+1);
        }

        // Calculate the number of moves if we
        // Insert , delete and replace the current character
        int insert = solve(word1,word2,i,j+1);
        int del = solve(word1,word2,i+1,j);
        int replace = solve(word1, word2, i+1,j+1);

        // return the one having minimum number of moves
        return 1 + min(insert,min(del, replace));
    }
    int minDistance(string word1, string word2) {
        return solve(word1, word2, 0, 0);
    }
};
```

b. Use memoization in above solution.

```
class Solution {
public:

    int solve(string& word1, string& word2, int i, int
j,vector<vector<int>>& dp) {
        if(i == word1.size() || j == word2.size()) {
            return max(word1.size() - i, word2.size()-j);
        }
        if(dp[i][j]){
            return dp[i][j];
        }
        if(word1[i] == word2[j]) {
            dp[i][j] = solve(word1,word2, i+1,j+1,dp);
            return dp[i][j];
        }

        int insert = solve(word1,word2,i,j+1,dp);
        int del = solve(word1,word2,i+1,j,dp);
        int replace = solve(word1, word2, i+1,j+1,dp);

        dp[i][j] = 1 + min(insert,min(del, replace));
        return dp[i][j];
    }
    int minDistance(string word1, string word2) {
        vector<vector<int>>
dp(word1.size(),vector<int>(word2.size(),0));
        return solve(word1, word2, 0, 0,dp);
    }
};
```

c. Using iterative approach of dynamic programming:

```
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size()+1,vector<int>(word2.size()+1,0));
        for(int i=word1.size();i>=0;i--) {
            for(int j=word2.size();j>=0;j--) {
                if(i==word1.size() || j == word2.size()){
                    dp[i][j] = max(word1.size() - i, word2.size()-j);
                    continue;
                }

                if(word1[i] == word2[j]){
                    dp[i][j] = dp[i+1][j+1];
                }
                else {
                    dp[i][j] = 1 + min(dp[i+1][j+1],min(dp[i+1][j],dp[i][j+1]));
                }
            }
        }
        return dp[0][0];

    }
```

151. Maximum Sum increasing subsequence.

This problem is similar to LIS but the difference is that instead of length, we want the maximum sum. Find a subsequence such that it is strictly increasing as well as its sum is maximum.

a. One method is to use recursion by taking the current element and leaving it. We can use memoization but that depends on the constraints.

```
int a[1000];
int solve(int n, int cur, int prev){
    if(cur == n) {
        return 0;
    }

    int taken = 0;
    if(a[cur] > a[prev]){
        taken = a[cur] + solve(n, cur+1, cur);
    }

    int nottaken = solve(n, cur+1, prev);
    return max(taken, nottaken);
}
```

b. Other way is to use the same solution with O(N) space(LIS solution [b]).

```
#include<bits/stdc++.h>
using namespace std;
int a[1000010];

int dp[1000010];
int main()
 {
     int t,i,j,n;
     cin>>t;
     while(t--) {
         cin>>n;
         memset(dp,0,sizeof(dp));
         for(i=0;i<n;i++){
             cin>>a[i];
         }
         for(i=0;i<n;i++){
             dp[i] = a[i];
             for(j=0;j<i;j++){
                 if(a[i]>a[j]){

                     dp[i] = max(dp[i], dp[j]+a[i]);
                 }
             }
         }
         int ans = INT_MIN;
         for(int i=0;i<n;i++)ans = max(ans, dp[i]);
         cout<<ans<<endl;
```

```
        }
}
```

152. Matrix Chain Multiplication

Given N matrices A,B,C,D,...; your task is to find the minimum number of multiplications to multiply them all. Note that, to multiply two matrices of order nxm and mxp, we need nxmxp operations.

a. We have to try all possible combinations. Lets say Matrix are A B C D, we will try :
(A) (BCD) , (AB)(CD), (ABC)(D).
We will recursively solve all the subproblems.
Suppose we are given the matrix orders in an array :
[2 4 3 5] that represents 2x4, 4x3, 3x5 matrices.

```
int solve(int l, int r, vector<int>& a){
        if(l==r){
            return 0;
        }
        int ans = INT_MAX;
        for(int i=l;i<r;i++){
            ans = min(
                ans,
                solve(l,i,a) + a[l-1]*a[i]*a[r] + solve(i+1,r,a)
            );
        }

        return ans;
    }
Call this function as solve(1,a.size()-1,a)
l = 1 because first matrix takes first two numbers.
```

b. We can use memoization to make it faster.
Full code:

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

int solve(int l, int r, vector<int>& a,vector<vector<int>>& dp){
        if(l==r){
            return 0;
        }
        if(dp[l][r])return dp[l][r];
        int ans = INT_MAX;
        for(int i=l;i<r;i++){
            ans = min(
                ans,
                solve(l,i,a,dp) + a[l-1]*a[i]*a[r] +
solve(i+1,r,a,dp)
            );
        }
        dp[l][r] = ans;
        return ans;
    }
int main()
 {
     int t,n,i;

     cin>>t;
     while(t--){
         cin>>n;
         vector<int> a(n);
         vector<vector<int>> dp(n,vector<int>(n,0));
         for(i=0;i<n;i++)cin>>a[i];
         cout<<solve(1,n-1,a,dp)<<endl;
     }
     return 0;
}
```

153. Max sum Path.
Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.
Example:
Input:
```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```
Output: 12
Explanation: Because the path 1→3→5→2→1 maximizes the sum.

a. We can use the same method we used to count the number of paths. But this time we will store the sum instead. We can reach to a cell from either left, or top; So we can say that reaching to a cell will cost maximum of reaching to left cell or top cell, in addition with current cell.
So dp[i][j] = max(dp[i-1][j],dp[i][j-1]) + path[i][j].

```
1 3 1                1 4 5
1 5 1   ===>         2 9 10
4 2 1                6 11 12
```

```
int maxPathSum(vector<vector<int>>& grid) {
        for(int i=1;i<grid.size();++i) {
            grid[i][0] += grid[i-1][0];
        }
        for(int i=1;i<grid[0].size();++i) {
            grid[0][i] += grid[0][i-1];
        }
        for(int i=1;i<grid.size();i++){
            for(int j=1;j<grid[0].size();j++){
                grid[i][j] += max(grid[i-1][j],grid[i][j-1]);
            }
        }
        return grid[grid.size()-1][grid[0].size()-1];
    }
```

b. We can also use backtracking for this problem. We will try to visit both right and down. The previous solution is DP version of this solution.

```
class Solution {
public:
    int solve(vector<vector<int>>& a, int r, int c) {
        // If we are at any of the border, we return -INF
        // so that max() function takes the other path
        if(r==a.size() || c == a[0].size())
            return INT_MIN;

        // Visit right and down
        int right = solve(a,r,c+1);
        int down = solve(a,r+1,c);

        // If both right and down are borders,
        // we have reached the bottom right.
        if(right == INT_MIN && down == INT_MIN){
            return a[r][c];
        }

        // else return the max cost path
        return a[r][c] + max(right,down);
    }
    int maxPathSum(vector<vector<int>>& grid) {
        return solve(grid,0,0);
    }
};
```

154. Min Coin Change

::There is a variation of this problem which asks the total number of ways to make the given amount using the given denominations. We have solved that problem in Backtracking portion.::

Given different denominations of coins (with infinite supply), your task is to find the minimum number of coins to sum upto given amount. Return -1 if not possible.

a. We can use recursion over the given amount. Suppose the denominations are [1,2,5] and target is 11. We have to call recursion for (11-1), (11-2),(11-5) and return the one which gives minimum answer. If amount < 0, we return +INF since we are using min() of them.

```
class Solution {
public:

    int solve(vector<int>& coins, int amount){
        if(amount == 0) return 0;
        if(amount<0) return INT_MAX;
        int ans = INT_MAX;

        for(auto coin:coins) {
            ans = min(ans, solve(coins, amount-coin));
        }

        return (ans == INT_MAX)? (INT_MAX) : (ans + 1);
    }
    int coinChange(vector<int>& coins, int amount) {
        int ans = solve(coins,amount);
        return (ans>amount)?-1:ans;

    }

};
```

b. We can convert it into dynamic programming.
Again lets assume denominations = [1,2,5] and target = 11

DP	0	1	2	3	4	5	6	7	8	9	10	11
i=0	0											
i=1	0	1										
i=2	0	1	1									
i=3	0	1	1	1 +Min(3-1,3-2)								
i=4	0	1	1	2	1+Min(4-1,4-2)							
i=5	0	1	1	2	2	1						
i=6	..											
i=7												
i=8				....								
i=11	0	1	1	2	2	1	2	2	3	3	2	3

```
int coinChange(vector<int>& coins, int amount) {
        int dp[amount+1];
        dp[0] = 0;
        for(int i=1;i<=amount;i++){
            int cost = INT_MAX;
            for(auto coin:coins) {
                if(coin <= i)
                {
                    if(cost > dp[i-coin]){
                        cost = dp[i-coin];
                    }
                }
            }

            if(cost == INT_MAX){
                dp[i] = INT_MAX;
            } else {
                dp[i] = cost + 1;
            }
        }
        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }
```

155. Subset Sum
// Solved using DP already ==> 59[c]

156. Rod Cutting

You are given a rod of length N and an array denoting prices of rod pieces of length 1,2,..N. Your task is to find the maximum profit you can get after cutting the rod and selling it.
Eg :
N = 4
prices = [2,3,4,5]
Selling the whole rod = 5
But we can sell 4 pieces of length 1 = 2*4 = 8

a. We will try to cut the rod at all the places and recursively find the maximum profit of the subrods. Eg rod is of length 4 . We will recursively find maximum prices of (1,3), (2,2). The configuration with maximum profit is our answer.

```
 int cutRod(vector<int> price, int n)
{
   // Rod of 0 length has no value.
   if (n <= 0)
     return 0;

   int max_val = INT_MIN;

   // Recursively cut the rod in different pieces and compare
different
   // configurations.
   for (int i = 0; i<n; i++)
         max_val = max(max_val, price[i] + cutRod(price, n-i-1));

   return max_val;
}
```

b. We can see that this code will calculate various configurations many times. So we can store them and convert our solution into dynamic programming.

The price of length 0 is 0 initially. Max price of length N is equal to max price of all pairs that sum upto N, eg if N = 5, find max price of price[5] + dp[0], price[4] + dp[1], price[3] + dp[2] ..... price[1]+dp[4].

```
int cutRod(int price[], int n)
{
   int val[n+1];
   val[0] = 0;
   int i, j;

   // Build the table val[] in bottom up manner and return the
last entry
   // from the table
   for (i = 1; i<=n; i++)
   {
       int max_val = INT_MIN;
       for (j = 0; j < i; j++)
         max_val = max(max_val, price[j] + val[i-j-1]);
       val[i] = max_val;
   }

   return val[n];
}
```
Above solution uses O(N) extra space and O(N^2) time.

157. Egg Dropping Problem.
You are given E eggs, and you have access to a building with N floors from 1 to N.
Each egg is identical in function, and if an egg breaks, you cannot drop it again.
You know that there exists a floor F with 0 <= F <= N such that any egg dropped at a floor higher than F will break, and any egg dropped at or below floor F will not break.
Each move, you may take an egg (if you have an unbroken one) and drop it from any floor X (with 1 <= X <= N).
Your goal is to know with certainty what the value of F is.
What is the minimum number of moves that you need to know with certainty what F is, regardless of the initial value of F?

a. In this problem, we need to find the minimum number of tries to find the threshold floor F. A pretty good explanation is given in this video https://www.youtube.com/watch?v=o_AJ3VWQMzA by Gaurav sen.

We can use N moves starting from bottom, and the first floor where the egg breaks, is our answer. But this is not optimal. Other way is to use sqrt decomposition that will surely give us answer in 2√N moves. (See the video for better explanation).

The idea here, is to use all possible moves.
Lets say there are N floors and E eggs
Suppose we are at a floor numbered ‘x’. Here two things can happen, either the egg dropped from here breaks, or it doesn’t.
If it breaks, that means our threshold floor ‘F’ is below this floor. So we have x-1 floors to check and E-1 remaining eggs.
If it doesn’t break then it means that there are still N-x floors to check with E eggs. Note that if an egg doesn’t break, we can use it again.
Since we are not sure where to look, we are bound to consider the worst scenerio. Dropping the egg from xth floor is counted in the answer. So recursively the answer at floor X is :
F(X, E) = 1 + max(F(X-1,E-1), F(N-X,E))

So if we check the answer for all possible floors, the minimum one will be our answer.

Hence,
F(N,E) = min(F(X,E)) for X = 1,2,...N-1
If we expand the function, the recursive function will be::

$F(N,E) = \min_{x=0 \text{ to } N-1} (1 + \max(F(x-1,E-1), F(N-x, E)))$

But we need to define the base cases where this recursion will terminate. We can see that the parameters of F are only decreasing.
What if we have only 1 egg left? We are bound to start from the bottom to top and try each floor because we don't have any other option.
Hence,
F(N,1) = N.

What if there is only 1 floor(ie N =1)? How many moves we need to determine F? The answer is only 1 egg. Wether it breaks or not, the answer is 1.
Hence,
F(1,E) = 1

In this question, we just need to compute the above value.

```
int superEggDrop(int E, int N) {
        if(N == 1) return 1;
        if(E == 1) return N;

        int ans = N;
        for(int x = 1;x<N;x++) {
            ans = min(ans,
                     1 + max(
                         superEggDrop(E-1,x-1),
                         superEggDrop(E,N-x)
                        )

                     );
        }
        return ans;
    }
```

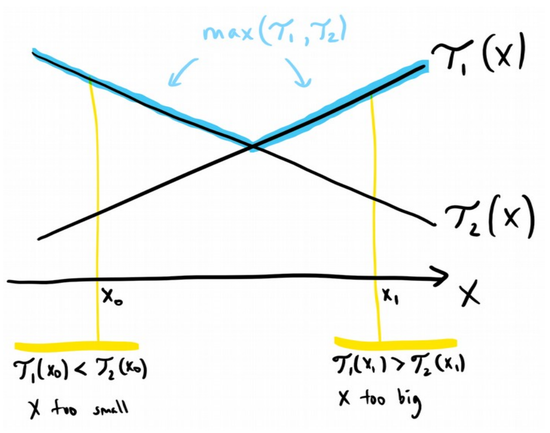
b. Since the recursive solution is too slow, we can use DP. It will take O(N*e*N) time and O(N*e) space.

```
int superEggDrop(int E, int N) {
        vector<vector<int>> dp(E+1,vector<int>(N+1,-1));
        for(int i=1;i<=E;i++){
            for(int j=1;j<=N;j++) {
                if(j == 1){
                    dp[i][j] = 1;
                } else if (i == 1) {
                    dp[i][j] = j;
                }
                else {
                    dp[i][j] = j;
                    for(int x = 1;x < j;x++) {
```

```
                                  dp[i][j] = min(dp[i][j],
                                                 1 + max(dp[i-1][x-1],dp[i]
[j-x])
                                                 );
                              }

                          }
                      }
                  }
              return dp[E][N];
          }
```

c. ::::::::::Can you do it O(eNlogN) or O(elogN) ::::?
If we observe the two calls
T1 = F(E-1,x-1)
T2= F(E,N-x)
We can see that T1 will increase as x increases because we have still x-1 ~ x floors to check.
Similarly T2 will always decrease because number of remaining floors to check are N-x that decrease as x increase.

Hence the graph between T1 and T2 is:



max(T1,T2) has a minimum at point P. We can see that whenever T1 < T2 (see x_0), P is ahead. Also whenever T1>T2 (see x_1), P is behind. This way we can use binary search to find the minimum point.

Below is the implementation:

```
int superEggDrop(int E, int N) {
        vector<vector<int>> dp(E+1,vector<int>(N+1,-1));
        for(int i=1;i<=E;i++){
            for(int j=1;j<=N;j++) {
                if(j == 1){
                    dp[i][j] = 1;
                } else if (i == 1) {
                    dp[i][j] = j;
                }
                else {
                    dp[i][j] = j;
                    int start = 1, end = j;
                    while(start+1<end) {
                        int x = (start+end)/2;
                        if(dp[i-1][j-1] == dp[i][j-x]){
                            start = end = x;
                        }
                        else if(dp[i-1][j-1] < dp[i][j-x]){
                            start = x;
                        } else {
                            end = x;
                        }
                    }
                        for(int x = 1;x<j;x++)
                            dp[i][j] = min(dp[i][j],
                                           1 + max(dp[i-1][x-1],
dp[i][j-x])
                                          );

                }
            }
        }
        return dp[E][N];
    }
```

For finding the min of all x, we were taking O(N) time before, which is now only logN. Hence time complexity is O(ENlogN) which is quite better and almost quadratic. We still use space O(E*N).

158. Word Break.
Done with backtracking. 58[c] is the DP solution.

159. Min Cut Palindrome Partitioning

Given a string S, find the minimum number of cuts to partition the string into all palindromes.
Eg “abacc” = “aba   cc” needs 1 cut
“abc” = “a b c” needs 2 cuts.

a. We can recursively find the answer by cutting at each point just like we did in matrix Chain multiplication. Cut the string at point i, and check the min for (l,i) and (i+1,r). A palindromic substring needs 0 cuts.
Below is the code:

```
class Solution {
public:

    bool isPalindrome(string& s, int l, int r){
        while(l<=r){
            if(s[l++]!=s[r--])
                return false;
        }
        return true;
    }
    int solve(string& s, int l, int r,vector<vector<int>>& dp) {
        if(dp[l][r] != -1)return dp[l][r];
        if(isPalindrome(s,l,r)) {
            dp[l][r] = 0;
            return 0;
        }

        int ans = INT_MAX;
        for(int i=l;i<r;i++){
            ans = min(ans, solve(s, l, i,dp) + solve(s,i+1,r,dp));
        }
        dp[l][r] = ans+1;
        return ans + 1;
    }
    int minCut(string s) {
        int n = s.length();
        vector<vector<int>> dp(n, vector<int>(n,-1));
        return solve(s,0,n-1,dp);
    }
};
```

If we use iterative DP, this solution will take $O(N^2)$ to build the table over l and r and O(N) to find minimum. Overall time complexity is $O(N^3)$.

b. We can also use recursion over a single variable. (See palindromic partition from backtracking.) In this problem, we have to partition the string linearly. Eg “abacc” can be partitioned as “a|b|a|cc” or “aba|cc”. We can see that each partition is having palindrome of a length > = 1.
The idea is simple, we have to take the longest palindromic prefix and cut the string. From there solve for the remaining suffix.

From the ith index, we will try different lengths, l = 0 ,1 , 2,....N-1 such that they fit in the answer and the length such that s[i:i+l] is palindromic, we cut the string.

Below is the recursive implementation for this approach.

```
class Solution {
public:

    bool isPalindrome(string& s, int l, int r){
        while(l<=r){
            if(s[l++]!=s[r--])
                return false;
        }
        return true;
    }
    int solve(string& s, int index){//, int r,vector<vector<int>>&
dp) {
        if(index == s.size()) return 0;
        int ans = INT_MAX;
        for(int length=0;length<s.size();length++){
            if(index + length < s.size() &&
isPalindrome(s,index,index+length)){
```

```
                ans = min(ans,1+solve(s,index+length+1));
            }
        }
        return ans;
    }
    int minCut(string s) {
        return solve(s,0)-1;
    }
};
```

We can memoize it to make it faster, or use iterative DP. But the problem is we are repeatedly checking for palindromes using linear time. This approach is faster but if we memoize the palindromes in form of isPalindrome[l][r], this approach will be perfectly O(N^2).

For palindromes in for [l,r],
if l == r, return True
if l +1== r, return true if s[l] == s[r] else false
else return isPalindrome(l+1,r-1) if s[l] == s[r] else False

This is the recursive version of this problem and we can memoize it to fasten the solution. Below is the recursive soution with memoization: (This code may look dirty but it is memoized and uses 0 global variables)

```
class Solution {
public:

    bool isPalindrome(string& s, int l, int r,vector<vector<int>>&
pal){
        if(l == r) return true;
        if(pal[l][r]) return pal[l][r] == 1? true:false;
        if(l+1 == r) {
            if(s[l] == s[r]) {
                pal[l][r] = 1;
                return true;
            }
            else {
                pal[l][r] = 2;
                return false;
            }
        }
        if(s[l] == s[r]){
            bool x = isPalindrome(s,l+1,r-1,pal);
            if(x)pal[l][r] = 1;
            else pal[l][r] = 2;
        } else {
            pal[l][r] = 2;
        }
        return pal[l][r] == 1? true:false;
    }
    int solve(string& s, int index, vector<int>&
dp,vector<vector<int>>& pal) {
        if(index == s.size()) return 0;
        if(dp[index] != -1) return dp[index];
        int ans = INT_MAX;
        for(int length=0;length<s.size();length++) {
            if(index + length < s.size() &&
isPalindrome(s,index,index+length,pal)) {
                ans = min(ans,1+solve(s,index+length+1,dp,pal));
            }
        }
        dp[index] = ans;
        return ans;
    }
    int minCut(string s) {
        vector<int> dp(s.size(),-1);
        vector<vector<int>> pal(s.size(),vector<int>(s.size(),0));
        return solve(s,0,dp,pal)-1;
    }
};
```

// Next target...Cheat Sheet on C++ STL

// Next Target C/C++/Python interview Questions. Then Jaba