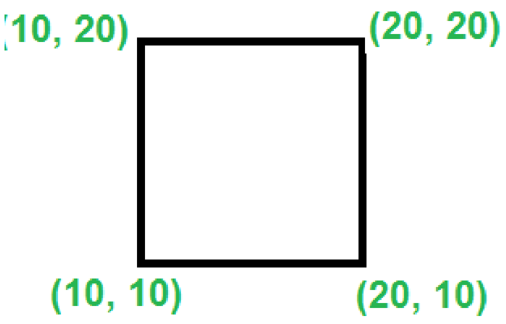
Amazon Question:

# How to check if given four points form a square

Given coordinates of four points in a plane, find if the four points form a square or not.

To check for square, we need to check for following.
a) All fours sides formed by points are same.
b) The angle between any two sides is 90 degree. (This condition is required as Quadrilateral also has same sides.
c) Check both the diagonals have same distance



```
//How to check if given four points form a square

class Point{
	int x;
	int y;
	Point(int x,int y){
		this.x=x;
		this.y=y;
	}
}
class PointDistanceSquare{
	public static void main(String[] args) {
		Point p1=new Point(0,0);
		Point p2=new Point(4,0);
		Point p3=new Point(0,5);
		Point p4=new Point(5,5);
		boolean flag=validate_square(p1,p2,p3,p4);
		System.out.println(flag);
	}
	public  static boolean validate_square(Point p1,Point p2,Point p3,Point p4){
		double d1=Distance(p1,p2);
		double d2=Distance(p1,p3);
		double d3=Distance(p1,p4);
		System.out.println(d1+" "+d2+" "+d3);
		if(d1==d2 && d3==Math.sqrt(d1*d1+d2*d2)){
			return true;
		}

		else if(d1==d3 && d2==Math.sqrt(d1*d1+d3*d3)){
			return true;
		}

		else if(d2==d3 && d1==Math.sqrt(d3*d3+d2*d2)){
			return true;
		}
		else{
			return false;
		}
	}
	public static double Distance(Point p1,Point p2){
		int x=Math.abs(p1.x-p2.x);
		int y=Math.abs(p1.y-p2.y);

		double d=Math.sqrt(x*x+y*y);
		System.out.println(d+" distance ");
		return d;
	}

}
```

## Check if a string can be obtained by rotating another string 2 places

Given two strings, the task is to find if a string can be obtained by rotating another string two places.

Examples:

```
Input : string1 = "amazon"

           string2 = "azonam"  // rotated anti-clockwise

Output : Yes



Input : string1 = "amazon"

           string2 = "onamaz"  // rotated clockwise

Output : Yes
```

```
//Check if a string can be obtained by rotating another string 2 places
class RotateString{
	public static void main(String[] args) {
		String s1="amazon";
		String s2="onamaz";
		boolean ans=check_substring(s1,s2);
		System.out.print(ans);


	}
	public static boolean check_substring(String s1,String s2){
		int l=s2.length();
		String clock_string="";
		clock_string+=s2.substring(l-2,l)+s2.substring(0,l-2);
		if(s1.equals(clock_string)){
			return true;
		}
		String anti_clock_string="";
		anti_clock_string+=s2.substring(2,l)+s2.substring(0,2);
		if(s1.equals(anti_clock_string)){
			return true;
		}
		return false;


	}


}
```

## Find the nearest smaller numbers on left side in an array

Given an array of integers, find the nearest smaller number for every element such that the smaller element is on left side.

Examples:

```
Input:  arr[] = {1, 6, 4, 10, 2, 5}

Output:         {_, 1, 1,  4, 1, 2}

First element ('1') has no element on left side. For 6,

there is only one smaller element on left side '1'.

For 10, there are three smaller elements on left side (1,

6 and 4), nearest among the three elements is 4.



Input: arr[] = {1, 3, 0, 2, 5}

Output:        {_, 1, _, 0, 2}
```

Expected time complexity is O(n).

```
//Check if a string can be obtained by rotating another string 2 places
class RotateString{
	public static void main(String[] args) {
		String s1="amazon";
		String s2="onamaz";
		boolean ans=check_substring(s1,s2);
		System.out.print(ans);


	}
	public static boolean check_substring(String s1,String s2){
		int l=s2.length();
		String clock_string="";
		clock_string+=s2.substring(l-2,l)+s2.substring(0,l-2);
		if(s1.equals(clock_string)){
			return true;
		}
		String anti_clock_string="";
		anti_clock_string+=s2.substring(2,l)+s2.substring(0,2);
		if(s1.equals(anti_clock_string)){
			return true;
		}
		return false;


	}


}
```

## Pair with given product | Set 1 (Find if any pair exists)

```
Input : arr[] = {10, 20, 9, 40};
          int x = 400;

 Output : Yes



 Input : arr[] = {10, 20, 9, 40};

          int x = 190;

 Output : No
```

```
import java.util.*;
class ProductofTwo{

	public static void main(String[] args) {
```

```
                int[] a={10, 20, 9, 40};
                int x=400;
                HashSet<Integer> hs=new HashSet<Integer>();
                for(int i=0;i<a.length;i++){
                        if(x%a[i]==0){
                                if(hs.contains(x/a[i])){
                                        System.out.println("Pair Exist:"+a[i]+"*"+x/a[i]+"="+x);
                                        break;
                                }
                                hs.add(a[i]);
                        }
                }
        }

}
```

## Position of rightmost set bit

Write a one line C function to return position of first 1 from right to left, in binary representation of an Integer.

```
I/P   18,   Binary Representation 010010

O/P   2

I/P   19,   Binary Representation 010011

O/P   1
```

```
class GFG {

    // function to find
    // the rightmost set bit
    static int PositionRightmostSetbit(int n)
    {
        // Position variable initialize
        // with 1 m variable is used to
        // check the set bit
        int position = 1;
        int m = 1;

        while ((n & m) == 0) {

            // left shift
            m = m << 1;
            position++;
        }
        return position;
    }

    // Driver Code
```

```
        public static void main(String[] args)
        {
                int n = 16;

                // function call
                System.out.println(PositionRightmostSetbit(n));
        }
}
```

## Replace all ‘0’ with ‘5’ in an input Integer

Given a integer as a input and replace all the ‘0’ with ‘5’ in the integer. Examples:

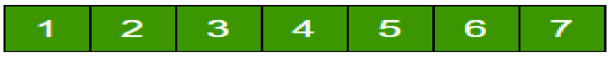
```
102 - 152

1020 - 1525
```

Use of array to store all digits is not allowed.

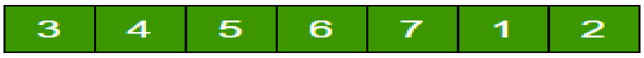
```
class Conver0to5digit{
        public static void main(String[] args) {
                int n=0000000;
                if(n==0){
                        System.out.println("5");
                }
                else{
                        System.out.println(Conversion(n));
                }
        }
        public static int Conversion(int n){
                if(n==0){
                        return 0;
                }
                int digit=n%10;
                if(digit==0){
                        digit=5;
                }
                return Conversion(n/10)*10+digit;
                //return Integer.parseInt(Integer.toString(n).replace('0','5'));
        }
}
```

## Program for array rotation

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.



Rotation of the above array by 2 will make array



```
 static int[] rotLeft(int[] a, int d) {
        int n=a.length;
        int[] b=new int[n];
        for(int i=0;i<n;i++){
            b[i]=a[i];
        }
        for(int i=0;i<n;i++){
            a[(n+i-d)%n]=b[i];
        }
        return a;
    }
```

## Calculate the angle between hour hand and minute hand

This problem is know as Clock angle problem where we need to find angle between hands of an analog clock at a given time.

Examples:

```
Input:  h = 12:00, m = 30.00

Output: 165 degree


Input:  h = 3.00, m = 30.00

Output: 75 degree
```

The idea is to take 12:00 (h = 12, m = 0) as a reference. Following are detailed steps.

1) Calculate the angle made by hour hand with respect to 12:00 in h hours and m minutes.

2) Calculate the angle made by minute hand with respect to 12:00 in h hours and m minutes.

3) The difference between two angles is the angle between two hands.

How to calculate the two angles with respect to 12:00?
The minute hand moves 360 degree in 60 minute(or 6 degree in one minute) and hour hand moves 360 degree in 12 hours(or 0.5 degree in 1 minute). In h hours and m minutes, the minute hand would move (h*60 + m)*6 and hour hand would move (h*60 + m)*0.5.

```
import java.util.*;
import java.io.*;
class ClockAngle{
	public static void main(String[] args) {
		Scanner in = new Scanner(new BufferedReader(new 
InputStreamReader(System.in)));
		int h=in.nextInt();
		int m=in.nextInt();
		System.out.println(Calculate_angle(h,m));
	}
	public static double Calculate_angle(int h,int m){

		double a1=(h*60+m)*0.5;
		double a2=6*m;
		double angle=Math.abs(a1-a2);
		return Math.min(360-angle,angle);
}
}
```

## Check if all bits of a number are set

Given a number **n**. The problem is to check whether every bit in the binary representation of the given number is set or not. Here 0 <= **n**.

Examples :

```
Input : 7

Output : Yes

(7)10 = (111)2
```

```
Input : 14

Output : No
```

```
import java.io.*;
import java.util.*;
class CheckBits{
	public static void main(String[] args) {
		Scanner in=new Scanner(new BufferedReader(new InputStreamReader(System.in)));

		System.out.println("ENter the number");
		int n=in.nextInt();
		System.out.println(check_validate(n));

	}
	public static String check_validate(int n){
		if(n==0){
			return "No";
		}
		while(n>0){
			if((n&1)==0){
				return "No";
			}
			n=n>>1;
		}
		return "YES";
	}
}
```

## Check if a given Binary Tree is SumTree

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Following is an example of SumTree.

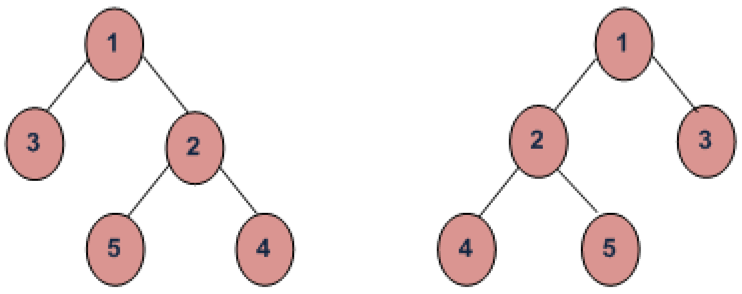
```
          26

        /   \

      10     3

    /    \     \
   4    6    3
```

```
import java.io.*;
import java.util.*;

class TreeNode{
        int data;
        TreeNode left;
        TreeNode right;
        TreeNode(int d){
                data=d;
                left=null;
                right=null;
        }
}

class SumTree{
        TreeNode root;
                public static void main(String[] args) {
                Scanner in=new Scanner(new BufferedReader(new InputStreamReader(System.in)));
                SumTree st=new SumTree();
                st.root=new TreeNode(26);
                st.root.left = new TreeNode(10);
      st.root.right = new TreeNode(3);
      st.root.left.left = new TreeNode(4);
      st.root.left.right = new TreeNode(6);
      st.root.right.right = new TreeNode(3);
      boolean flag=st.isSumTree(st.root);
      if(flag){
         System.out.println("The given tree is a sum tree");
      }
      else{
         System.out.println("The given tree is not a sum tree");
      }
   }
   public boolean isSumTree(TreeNode node)
        {

                int ls,rs;
                ls=rs=0;
                if(node==null || isLeaf(node)){
                        return true;
                }
                if(!isSumTree(node.left) && !isSumTree(node.right))
                {
                        if(node.left==null){
                                ls=0;
                        }
                        else if(isLeaf(node.left)){
                                ls=node.left.data;
                        }
                        else{
                                ls=2*node.left.data;
                        }

                        if(node.right==null){
                                rs=0;
                        }
                        else if(isLeaf(node.right)){
                                rs=node.right.data;
                        }
                        else{
                                rs=2*node.right.data;
                        }


                }
                if(node.data==ls+rs){
                                return true;
                        }
                        return false;

        }




        public boolean isLeaf(TreeNode node)
        {
                if(node == null)
                {
                        return false;
                }
                if(node.left==null && node.right==null)
                {
                        return true;
                }
                return false;
        }
}
```

```
class PowerofNum{
        public static void main(String[] args) {
                int n=128;
                System.out.println(power_of(n));
        }
        public static boolean power_of(int num){
                if(num==1){
                        return true;
                }
                int m=2;
                while(m<=Math.sqrt(num)){
                        int n=2;
                        while(n<=Math.sqrt(num)+1){
                                int x=(int)Math.pow(m,n);
                                if(x==num){
                                        return true;
                                }
                                n++;
                        }
                        m++;
                }
                return false;
        }
}
```

## Check if two trees are Mirror

Given two Binary Trees, write a function that returns true if two trees are mirror of each other, else false. For example, the function should return true for following input trees.



Mirror Trees

Recommended: Please solve it on "***PRACTICE*** " first, before moving on to the solution.

This problem is different from the problem discussed here.

For two trees ‘a’ and ‘b’ to be mirror images, the following three conditions must be true:

1.Their root node’s key must be same

2.Left subtree of root of ‘a’ and right subtree root of ‘b’ are mirror.

3.Right subtree of ‘a’ and left subtree of ‘b’ are mirror.

```
boolean areMirror(Node a, Node b)
    {
        /* Base case : Both empty */
        if (a == null && b == null)
            return true;

        // If only one is empty
        if (a == null || b == null)
            return false;

        /* Both non-empty, compare them recursively
           Note that in recursive calls, we pass left
           of one tree and right of other tree */
        return a.data == b.data
                && areMirror(a.left, b.right)
                && areMirror(a.right, b.left);
    }
```

## Count number of bits to be flipped to convert A to B

Given two numbers ‘a’ and b’. Write a program to count number of bits needed to be flipped to convert ‘a’ to ‘b’.

Example :

```
Input : a = 10, b = 20

Output : 4

Binary representation of a is 00001010
Binary representation of b is 00010100

We need to flip highlighted four bits in a

to make it b.



Input : a = 7, b = 10

Output : 3
```

```
Binary representation of a is 00000111
Binary representation of b is 00001010

We need to flip highlighted three bits in a

to make it b.
```

```
import java.io.*;
import java.util.*;

class FlipBit{
	public static void main(String[] args) {
		Scanner in=new Scanner(new BufferedReader(new InputStreamReader(System.in)));
		while(true){
			int a=in.nextInt();
		int b=in.nextInt();

		System.out.println(Count_bit_to_change(a^b));
		}
	}
	public static int Count_bit_to_change(int n){
		System.out.println(n);

		int flag=0;
		while(n!=0){
			flag+=n&1;
			n=n>>1;
		}
		return flag;
	}
}
```

## Count number of occurrences (or frequency) in a sorted array

Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[]. Expected time complexity is O(Logn)

Examples:

```
    Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 2

    Output: 4 // x (or 2) occurs 4 times in arr[]
```

```
Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 3

Output: 1



Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 1

Output: 2



Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 4

Output: -1 // 4 doesn't occur in arr[]
```

```
class CountOccurance{
	public static void main(String[] args) {
			int[] a={1,3,7,9,9,9,9,14,19,22};
			System.out.println(Count_Occurance(a,9));
		}

		public static int Count_Occurance(int[] a,int x){
			int m=a.length;
			int ind=Find_index(a,x,0,a.length-1);
			if(ind==-1){
				return 0;
			}
			int flag=1;
			int left=ind-1;
			while(left>=0 && a[left]==x){
				flag++;
				left--;
			}
			int right=ind+1;
			while(right<m && a[right]==x){
				flag++;
				right++;
			}
			return flag;
		}
		public static int Find_index(int[] a,int x,int start,int end){
			if(start>end){
				return -1;
			}
```

```
                int mid=start+(end-start)/2;
                if(a[mid]==x){
                        return mid;
                }
                if(a[mid]<x){
                        return Find_index(a,x,start,mid-1);
                }
                return Find_index(a,x,mid+1,end);
        }
}
```

## Count all possible paths from top left to bottom right of a mXn matrix

Note the count can also be calculated using the formula (m-1 + n-1)!/(m-1)!(n-1)!.

## Equilibrium index of an array

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an array A:

Example :

```
Input : A[] = {-7, 1, 5, 2, -4, 3, 0}

Output : 3

3 is an equilibrium index, because:

A[0] + A[1] + A[2]  =  A[4] + A[5] + A[6]
```

Write a function int equilibrium(int[] arr, int n); that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

```
class ArrayEquilibirium{
        public static void main(String[] args) {

                int[] a={-7, 1, 5, 2, -4, 3, 0};
                int ind=Find_index(a);
                System.out.println(ind);

        }
        public static int Find_index(int[] a){
                int sum=0;
                int left_sum=0;
                for(int i=0;i<a.length;i++){
                        sum+=a[i];
```

```
		}

		for(int i=0;i<a.length;i++){
			sum-=a[i];

			if(sum==left_sum){
				return i+1;
			}
			left_sum+=a[i];
		}
		return -1;
	}

}
```

## Find first and last positions of an element in a sorted array

Given a sorted array with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element x in the given array.

Examples:

```
Input : arr[] = {1, 3, 5, 5, 5, 5 ,67, 123, 125}

            x = 5

Output : First Occurrence = 2

              Last Occurrence = 5



Input : arr[] = {1, 3, 5, 5, 5, 5 ,7, 123 ,125 }

            x = 7

Output : First Occurrence = 6

              Last Occurrence = 6
```

```
class FirstAndLast{
	public static void main(String[] args) {
		int a[] = {1, 2, 2, 2, 2, 3, 4, 7, 8, 8};
		int x=2;
		int first=FindFirst(a,x,0,a.length-1,a.length);
		int last=FindLast(a,x,0,a.length-1,a.length);
		System.out.println(first+" "+last);
	}
	public static int FindFirst(int[] a,int x,int start,int end,int n){
		if(end>=start){
			int mid=start+(end-start)/2;
			if((mid==0 || x>a[mid-1]) && a[mid]==x){
				return mid;
			}
			else if(x>a[mid]){

				return FindFirst(a,x,mid+1,end,n);
			}
			else{
				return FindFirst(a,x,start,mid-1,n);
			}
		}
		return -1;
	}
	public static int FindLast(int[] a,int x,int start,int end,int n){
		if(end>=start){
			int mid=start+(end-start)/2;
			if((mid==n-1 || x<a[mid+1]) && a[mid]==x){
				return mid;
			}
			else if(x<a[mid]){

				return FindFirst(a,x,mid+1,end,n);
			}
			else{
				return FindFirst(a,x,mid+1,end,n);
			}
		}
		return -1;
	}

}
```

## Find four elements that sum to a given value | Set 1 (n^3 solution)

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X.
For example, if the given array is {10, 2, 3, 4, 5, 9, 7, 8} and X = 23, then your function should print “3 5 7 8” (3 + 5 + 7 + 8 = 23).

```
import java.util.*;
class FourthSum{
	public static void main(String[] args) {
		int[] a = {1, 4, 45, 6, 10, 12};
		int x=21;
		Find_El(a,x);

	}
	public static void Find_El(int[] a,int x){
		Arrays.sort(a);
		int n=a.length;
		for(int i=0;i<n-3;i++){
			for(int j=i+1;j<n-2;j++){
				int l=j+1;
				int r=n-1;
				while(l<=r){
					//System.out.println(a[i]+" "+a[j]+" "+a[l]+" "+a[r]);
					if(a[i]+a[j]+a[l]+a[r]==x){
						System.out.println(a[i]+" "+a[j]+" "+a[l]+" 
"+a[r]);

						l++;
						r--;
						break;
					}
					else if(a[i]+a[j]+a[l]+a[r]>x){
						r--;
					}
					else{
						l++;
					}
				}
			}
		}
	}
}
```

## k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

```
//k largest(or smallest) elements in an array | added Min Heap method
class KthLargest{
	public static void main(String[] args) {
		int[] a={1, 23, 12, 9, 30, 2, 50};
		int k=0;
		int j=0;
		int n=a.length;
		for(int i=0;i<n-1;i++){
			for(j=0;j<n-i-1;j++){
				if(a[j]>a[j+1]){
					int temp=a[j];
					a[j]=a[j+1];
					a[j+1]=temp;
				}

			}
```

```
                        if(k<3){
                                System.out.println(a[j]+" ");
                                k++;
                        }
                }

        }
}
```

## Find the index of first 1 in a sorted array of 0’s and 1’s

Given a sorted array consisting 0’s and 1’s. The problem is to find the index of first ‘1’ in the sorted array. It could be possible that the array consists of only 0’s or only 1’s. If 1’s are not present in the array then print “-1”.

Examples :

```
Input : arr[] = {0, 0, 0, 0, 0, 0, 1, 1, 1, 1}

Output : 6

The index of first 1 in the array is 6.


Input : arr[] = {0, 0, 0, 0}

Output : -1

1's are not present in the array.
```

```
class FirstOne{
        public static void main(String[] args) {
                int[] a={0, 0, 0, 0, 0, 0, 1, 1, 1, 1};
                int x=help(a);
                System.out.println(x);
        }
        public static int help(int[] a){
                int low=0;
                int high=a.length-1;
                while(low<high){
                        int mid=low+(high-low)/2;
                        if((a[mid]==1) && (mid==0 || a[mid-1]==1)){
                                return (int)mid;
                        }
                        else if(a[mid]==0){
                                low=mid+1;
                        }
                        else{
                                high=mid-1;
                }
            }
            return -1;
        }
}
```

## Find minimum difference between any two elements

Given an unsorted array, find the minimum difference between any pair in given array.

Examples :

```
Input  : {1, 5, 3, 19, 18, 25};

Output : 1

Minimum difference is between 18 and 19



Input  : {30, 5, 20, 9};

Output : 4

Minimum difference is between 5 and 9



Input  : {1, 19, -4, 31, 38, 25, 100};

Output : 5

Minimum difference is between 1 and -4
```

```
//Find minimum difference between any two elements
import java.util.*;
class MinDifference{
        public static void main(String[] args) {
            int a[]={1, 5, 3, 19, 18, 25};
            int n=a.length;
            Arrays.sort(a);
            int min=Integer.MAX_VALUE;
            for(int i=0;i<n-1;i++){
                min=Math.min(Math.abs(a[i]-a[i+1]),min);
            }
            System.out.println(min);
        }
}
```

# Count the number of possible triangles

Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).
For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.
As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Method 2 (Tricky and Efficient)
Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)
i) a + b > c
ii) b + c > a
iii) a + c > b

Following are steps to count triangle.

1. Sort the array in non-decreasing order.

2. Initialize two pointers ‘i’ and ‘j’ to first and second elements respectively, and initialize count of triangles as 0.

3. Fix ‘i’ and ‘j’ and find the rightmost index ‘k’ (or largest ‘arr[k]’) such that ‘arr[i] + arr[j] > arr[k]’. The number of triangles that can be formed with ‘arr[i]’ and ‘arr[j]’ as two sides is ‘k – j’. Add ‘k – j’ to count of triangles.
Let us consider ‘arr[i]’ as ‘a’, ‘arr[j]’ as b and all elements between ‘arr[j+1]’ and ‘arr[k]’ as ‘c’. The above mentioned conditions (ii) and (iii) are satisfied because ‘arr[i] < arr[j] < arr[k]'. And we check for condition (i) when we pick 'k'

4. Increment ‘j’ to fix the second element again.
Note that in step 3, we can use the previous value of ‘k’. The reason is simple, if we know that the value of ‘arr[i] + arr[j-1]’ is greater than ‘arr[k]’, then we can say ‘arr[i] + arr[j]’ will also be greater than ‘arr[k]’, because the array is sorted in increasing order.

5. If ‘j’ has reached end, then increment ‘i’. Initialize ‘j’ as ‘i + 1’, ‘k’ as ‘i+2’ and repeat the steps 3 and 4.

## Find a pair with maximum product in array of Integers

Given an array with both +ive and -ive integers, return a pair with highest product.

Examples :

```
Input: arr[] = {1, 4, 3, 6, 7, 0}

Output: {6,7}


Input: arr[] = {-1, -3, -4, 2, 0, -5}

Output: {-4,-5}
```

```
class MaxProd{
        public static void main(String[] args) {
                int[] a={-1, -3, -4, 2, 0, -5} ;
                int n=a.length;
                int maxa=Integer.MIN_VALUE;
                int maxb=Integer.MIN_VALUE;
                int mina=Integer.MAX_VALUE;
                int minb=Integer.MAX_VALUE;
                for(int i=0;i<n;i++){
                        if(a[i]>maxa){
                                maxb=maxa;
                                maxa=a[i];
                        }
                        else if(a[i]>maxb){
                                maxb=a[i];
                        }
                        if(a[i]<mina){
                                minb=mina;
                                mina=a[i];
                        }
                        else if(a[i]>minb){
                                minb=a[i];
                        }
                }
                System.out.println(Math.max(maxa*maxb,mina*minb));



        }
}
```

## Print all nodes that don’t have sibling

Given a Binary Tree, print all nodes that don’t have a sibling (a sibling is a node that has same parent. In a Binary Tree, there can be at most one sibling). Root should not be printed as root cannot have a sibling.

For example, the output should be “4 5 6” for the following tree.

```
class Node{
          int data;
          Node left,right;
          Node(int d){
                    data=d;
                    left=null;
                    right=null;
          }
}

class Sibling{
          Node root;



          public static void main(String[] args) {
                    Sibling tree = new Sibling();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.right = new Node(4);
        tree.root.right.left = new Node(5);
        tree.root.right.left.right = new Node(6);
        tree.printSingles(tree.root);
          }
          public static void printSingles(Node root){
                    if(root==null){
                              return;
                    }
                    if(root.left!=null && root.right!=null){
                              printSingles(root.left);
                              printSingles(root.right);
```

```
		}
		else if(root.left!=null){
			System.out.println(root.left.data);
			printSingles(root.left);
		}

		else if(root.right!=null){
			System.out.println(root.right.data);
			printSingles(root.right);
		}
}

}
```

## Reverse an array upto a given position

Given an array arr[] and a position in array, k. Write a function name reverse (a[], k) such that it reverses subarray arr[0..k-1]. Extra space used should be O(1) and time complexity should be O(k).

Example:

```
Input:
arr[] = {1, 2, 3, 4, 5, 6}
      k = 4


Output:
arr[] = {4, 3, 2, 1, 5, 6}
```

```
class SwapArrayPosition{
	public static void main(String[] args) {
		int[] a={1, 2, 3, 4, 5, 6};
		int k=4;
```

```
            for(int i=0;i<k/2;i++){
                int temp=a[i];
                a[i]=a[k-i-1];
                a[k-i-1]=temp;
            }
            for(int i=0;i<a.length;i++){
                System.out.print(a[i]+" ");
            }



        }
}
```

## Find the middle of a given linked list in C and Java

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

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

**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

Method 1:
Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

Method 2:
Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

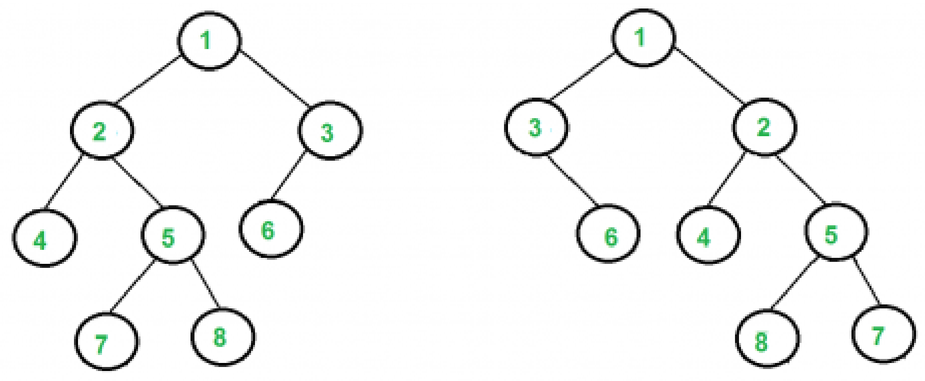
```
class FindMiddle{
	ListNode head;
	class ListNode{
		int data;
		ListNode next;
		ListNode(int d){
			data=d;
			next=null;
		}
	}
	public void push(int d){
		ListNode new_node=new ListNode(d);
		new_node.next=head;
		head=new_node;
	}
	public int FinaMid(){
		ListNode slow=head;
		ListNode fast=head;
		while(fast!=null && fast.next!=null){
			slow=slow.next;
			fast=fast.next.next;
		}
		return slow.data;
	}
	public static void main(String[] args) {
	FindMiddle f=new FindMiddle();
	f.push(1);
	f.push(2);
	f.push(3);
	f.push(4);
```

```
        f.push(5);
        f.push(6);
        f.push(7);
        System.out.print(f.FinaMid());
        }
}
```

## Tree Isomorphism Problem

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.



**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be n1 and n2 respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.
1) Data of n1 and n2 is same.
2) One of the following two is true for children of n1 and n2
……a) Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.
……b) Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.

```
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class TreeIsomorphism
{
    Node root1, root2;

    /* Given a binary tree, print its nodes in reverse level order */
    boolean isIsomorphic(Node n1, Node n2)
    {
        // Both roots are NULL, trees isomorphic by definition
        if (n1 == null && n2 == null)
            return true;

        // Exactly one of the n1 and n2 is NULL, trees not isomorphic
        if (n1 == null || n2 == null)
            return false;

        if (n1.data != n2.data)
            return false;

        // There are two possible cases for n1 and n2 to be isomorphic
        // Case 1: The subtrees rooted at these nodes have NOT been
        // "Flipped".
        // Both of these subtrees have to be isomorphic.
        // Case 2: The subtrees rooted at these nodes have been "Flipped"
        return (isIsomorphic(n1.left, n2.left) &&
                                    isIsomorphic(n1.right, n2.right))
        || (isIsomorphic(n1.left, n2.right) &&
                                    isIsomorphic(n1.right, n2.left));
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        TreeIsomorphism tree = new TreeIsomorphism();

        // Let us create trees shown in above diagram
        tree.root1 = new Node(1);
        tree.root1.left = new Node(2);
        tree.root1.right = new Node(3);
        tree.root1.left.left = new Node(4);
        tree.root1.left.right = new Node(5);
```

```
        tree.root1.right.left = new Node(6);
        tree.root1.left.right.left = new Node(7);
        tree.root1.left.right.right = new Node(8);

        tree.root2 = new Node(1);
        tree.root2.left = new Node(3);
        tree.root2.right = new Node(2);
        tree.root2.right.left = new Node(4);
        tree.root2.right.right = new Node(5);
        tree.root2.left.right = new Node(6);
        tree.root2.right.right.left = new Node(8);
        tree.root2.right.right.right = new Node(7);

        if (tree.isIsomorphic(tree.root1, tree.root2) == true)
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

## Majority Element

Write a function which takes an array and prints the majority element (if it exists), otherwise prints “No Majority Element”. A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Examples :

```
Input : {3, 3, 4, 2, 4, 4, 2, 4, 4}

Output : 4



Input : {3, 3, 4, 2, 4, 4, 2, 4}
```

```
Output : No Majority Element
```

```
import java.util.*;
class Majority{
	public static void main(String[] args) {
		int[] a= {3, 3, 4, 2, 4, 4, 2, 4, 4};
		HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();
		for(int i=0;i<a.length;i++){
			if(hm.containsKey(a[i]))
			{
				hm.put(a[i],hm.get(a[i])+1);
				if(hm.get(a[i])>a.length/2){
					System.out.println("Majority element is "+a[i]);
					break;
				}
			}
			else
			{
				hm.put(a[i],1);
			}
		}
	}
}
```

# Binary Tree to Binary Search Tree Conversion

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Examples.

```
Example 1

Input:

          10

         /  \

        2    7

       / \

      8   4

Output:

          8

         /  \

        4    10

       / \

      2   7



Example 2

Input:

          10

         /  \

        30   15

       /      \

      20       5

Output:
```

```
            15
           /  \
         10    20
        /       \
       5         30
```

```
import java.util.*;
class Node{
	int data;
	Node left;
	Node right;
	Node(int d){
		data=d;
	}
}
class ConverTree{
	public static void main(String[] args) {
		Node root=new Node(8);
		root.left = new Node(3);
		root.right = new Node(5);
		root.left.left = new Node(10);
		root.left.right = new Node(2);
		root.right.left = new Node(4);
		root.right.right = new Node(6);
		print(root);
		Set<Integer> hs = new TreeSet<>();
		constructHS(root,hs);
		Iterator<Integer> itr=hs.iterator();
		Conversion(root,itr);
		print(root);
	}
	public static void constructHS(Node root,Set<Integer> hs){
		if(root==null){
			return;
		}
		constructHS(root.left,hs);
		hs.add(root.data);
		constructHS(root.right,hs);
	}
	public static void Conversion(Node root,Iterator<Integer> itr){
		if(root==null){
			return;
		}
		Conversion(root.left,itr);
		root.data=itr.next();
		Conversion(root.right,itr);
	}
	public static void print(Node root){
		if(root==null){
			return;
		}
		print(root.left);
		System.out.println(root.data+"  ");
		print(root.right);
	}
}
```

# Find four elements a, b, c and d in an array such that a+b = c+d

Given an array of distinct integers, find if there are two pairs (a, b) and (c, d) such that a+b = c+d, and a, b, c and d are distinct elements. If there are multiple answers, then print any of them.

Example:

```
Input:   {3, 4, 7, 1, 2, 9, 8}

Output:  (3, 8) and (4, 7)

Explanation: 3+8 = 4+7


Input:   {3, 4, 7, 1, 12, 9};

Output:  (4, 12) and (7, 9)

Explanation: 4+12 = 7+9


Input:  {65, 30, 7, 90, 1, 9, 8};

Output:  No pairs found
```

Expected Time Complexity: O(n2)

```
import java.util.*;
class Pair{
	int f;
	int s;
	Pair(int a, int b){
		f=a;
		s=b;
	}
}
```

```
class FindPair{
        public static void main(String[] args) {
                int[] a={3, 4, 7, 1, 2, 9, 8};
                FindPair(a);
        }
        public static void FindPair(int[] a){
                HashMap<Integer,Pair> hm=new HashMap<Integer,Pair>();
                for(int i=0;i<a.length;i++){
                        for(int j=i+1;j<a.length;j++){
                                int sum=a[i]+a[j];
                                if(hm.containsKey(sum)){
                                        Pair p=hm.get(sum);
                                        System.out.println(a[p.f]+" "+a[p.s]+" "+a[i]+" "+a[j]);
                                        break;
                                }
                                else{
                                        hm.put(sum,new Pair(i,j));
                                }
                        }
                }





        }
}
```

# For each element in 1st array count elements less than or equal to it in 2nd array

Given two unsorted arrays arr1[] and arr2[]. They may contain duplicates. For each element in arr1[] count elements less than or equal to it in array arr2[].

**Source:** Amazon Interview Experience | Set 354 (For SDE-2)

Examples:

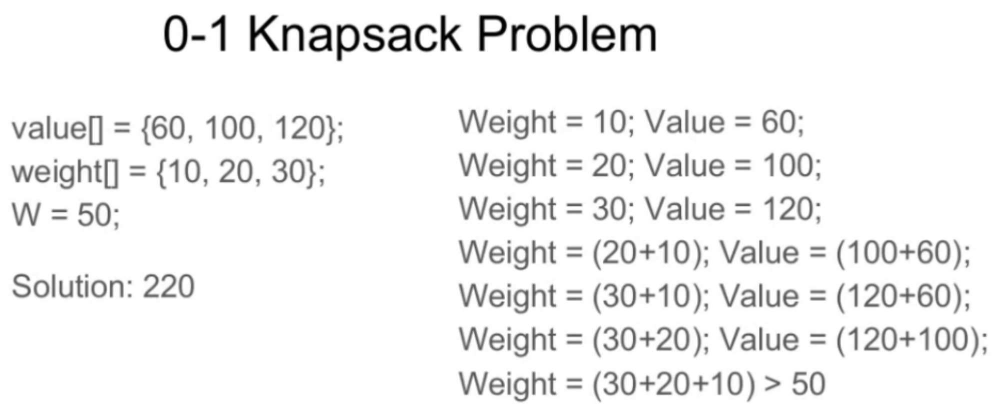
```
Input : arr1[] = [1, 2, 3, 4, 7, 9]
                arr2[] = [0, 1, 2, 1, 1, 4]
Output : [4, 5, 5, 6, 6, 6]


Input : arr1[] = [5, 10, 2, 6, 1, 8, 6, 12]
                arr2[] = [6, 5, 11, 4, 2, 3, 7]
Output : [4, 6, 1, 5, 0, 6, 5, 7]
```

```
import java.util.*;
class FindLessThanEqual{
        public static void main(String[] args) {
                int a[] = {1, 2, 3, 4, 7, 9};
        int b[] = {0, 1, 2, 1, 1, 4};
        CountELement(a,b);
        }
        public static void CountELement(int[] a,int[] b){
                Arrays.sort(b);
                for(int i=0;i<a.length;i++){
                        int find_index=binary_search(b,a[i])+1;
                        System.out.print(find_index+"  ");
                }
        }
```

```
	public static int binary_search(int[] b,int e){
		int low=0;
		int high=b.length-1;
		while(low<=high){
			int mid=low+(high-low)/2;
			if(b[mid]<=e){
				low=mid+1;
			}
			else{
				high=mid-1;
			}
		}
		return high;
	}
}
```

## 0-1 Knapsack Problem | DP-10

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).



## 0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

```
class KnapSackZeroOne{
        public static void main(String[] args) {
                int val[] = {60, 100, 120};
        int wt[] = {10, 20, 30};
        int  W = 50;
        System.out.println(Find_max(val,wt,W));
        }
        public static int Find_max(int[] val,int[] wt,int W){
                int n=val.length;
                int[][] dp=new int[n+1][W+1];
                for(int i=0;i<=n;i++){
                        for(int w=0;w<=W;w++){
                                if(i==0 || w==0){
                                        dp[i][w]=0;
                                }
                                else if(wt[i-1]<=w){
                                        dp[i][w]=Math.max(dp[i-1][w],dp[i-1][w-wt[i-1]]+val[i-1]);
                                }
                                else{
                                        dp[i][w]=dp[i-1][w];
                                }
                        }
                }
                return dp[n][W];
                /



        }
}
```

## Largest subarray with equal number of 0s and 1s

Given an array containing only 0s and 1s, find the largest subarray which contain equal no of 0s and 1s. Expected time complexity is O(n).

Examples:

```
Input: arr[] = {1, 0, 1, 1, 1, 0, 0}

Output: 1 to 6 (Starting and Ending indexes of output subarray)



Input: arr[] = {1, 1, 1, 1}

Output: No such subarray



Input: arr[] = {0, 0, 1, 1, 0}

Output: 0 to 3 Or 1 to 4
```

Method 1 (Simple)
A simple method is to use two nested loops. The outer loop picks a starting point i. The inner loop considers all subarrays starting from i. If size of a subarray is greater than maximum size so far, then update the maximum size.
In the below code, 0s are considered as -1 and sum of all values from i to j is calculated. If sum becomes 0, then size of this subarray is compared with largest size so far.

- C
- Java
- Python3

- C#
- PHP

```
// A simple program to find the largest subarray
// with equal number of 0s and 1s

#include <stdio.h>

// This function Prints the starting and ending
// indexes of the largest subarray with equal
// number of 0s and 1s. Also returns the size
// of such subarray.

int findSubArray(int arr[], int n)
{
    int sum = 0;
    int maxsize = -1, startindex;

    // Pick a starting point as i

    for (int i = 0; i < n-1; i++)
    {
        sum = (arr[i] == 0)? -1 : 1;

        // Consider all subarrays starting from i

        for (int j = i+1; j < n; j++)
        {
            (arr[j] == 0)? (sum += -1): (sum += 1);

            // If this is a 0 sum subarray, then
            // compare it with maximum size subarray
            // calculated so far

            if (sum == 0 && maxsize < j-i+1)
            {
                maxsize = j - i + 1;
                startindex = i;
            }
        }
    }
    if (maxsize == -1)
        printf("No such subarray");
    else
        printf("%d to %d", startindex, startindex+maxsize-1);

    return maxsize;
}
```

```
/* Driver program to test above functions*/

int main()
{
        int arr[] =  {1, 0, 0, 1, 0, 1, 1};
        int size = sizeof(arr)/sizeof(arr[0]);

        findSubArray(arr, size);
        return 0;
}
```

**Output:**

```
0 to 5
```

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Method 2 (Tricky)

Following is a solution that uses O(n) extra space and solves the problem in O(n) time complexity.

Let input array be arr[] of size n and maxsize be the size of output subarray.

1) Consider all 0 values as -1. The problem now reduces to find out the maximum length subarray with sum = 0.

2) Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i]. This can be done in O(n) time.

3) There are two cases, the output subarray may start from 0th index or may start from some other index. We will return the max of the values obtained by two cases.

4) To find the maximum length subarray starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.

5) Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we can create a hash table with size = max-min+1 where min is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. The idea is to hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because

there can be these many different possible values in sumleft[]. Initialize all values in hash as -1
6) To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.
7) To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

- C
- C++/STL
- Java
- C#

```
import java.util.HashMap;

class LargestSubArray1
{

    // Returns largest subarray with equal number of 0s and 1s

    int maxLen(int arr[], int n)
    {
        // Creates an empty hashMap hM

        HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();

        int sum = 0;     // Initialize sum of elements
        int max_len = 0; // Initialize result
        int ending_index = -1;
        int start_index = 0;

        for (int i = 0; i < n; i++)
        {
            arr[i] = (arr[i] == 0) ? -1 : 1;
        }

        // Traverse through the given array

        for (int i = 0; i < n; i++)
        {
            // Add current element to sum

            sum += arr[i];
```

```java

            // To handle sum=0 at last index

            if (sum == 0)
            {
                max_len = i + 1;
                ending_index = i;
            }

            // If this sum is seen before, then update max_len
            // if required

            if (hM.containsKey(sum + n))
            {
                if (max_len < i - hM.get(sum + n))
                {
                    max_len = i - hM.get(sum + n);
                    ending_index = i;
                }
            }
            else // Else put this sum in hash table
                hM.put(sum + n, i);
        }

        for (int i = 0; i < n; i++)
        {
            arr[i] = (arr[i] == -1) ? 0 : 1;
        }

        int end = ending_index - max_len + 1;
        System.out.println(end + " to " + ending_index);

        return max_len;
    }

    /* Driver program to test the above functions */

    public static void main(String[] args)
    {
        LargestSubArray1 sub = new LargestSubArray1();
        int arr[] = {1, 0, 0, 1, 0, 1, 1};
        int n = arr.length;

        sub.maxLen(arr, n);
    }
}
```

# Coin Change | DP-7

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

**Recommended: Please solve it on "*PRACTICE* " first, before moving on to the solution.**

1) Optimal Substructure
To count the total number of solutions, we can divide all set solutions into two sets.
1) Solutions that do not contain mth coin (or Sm).
2) Solutions that contain at least one Sm.
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

2) Overlapping Subproblems
Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
class CoinChangeProblem{
        public static void main(String[] args) {
                int a[] = {1, 2, 3};
                int m=4;
                Find_Change(a,a.length,m);
        }
        public static void Find_Change(int[] s,int n,int m){
                int[] dp=new int[m+1];
                dp[0]=1;
        for(int i=0;i<n;i++){
            System.out.println(" for loop "+i);
            for(int j=s[i];j<=m;j++){
                dp[j]+=dp[j-s[i]];
                System.out.print(dp[j]+" ");
            }
            System.out.println();
        }
        System.out.println(dp[m]);
    }
}
```

## Find the element that appears once

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.
Examples :

*Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}*
*Output: 2*
*In the given array all element appear three times except 2 which appears once.*

*Input: arr[] = {10, 20, 10, 30, 10, 30, 30}*
*Output: 20*
*In the given array all element appear three times except 20 which appears once.*

```
//Find the element that appears once
class FindbwThree{
    public static void main(String[] args) {
        int[]  a = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3};
        int f=0;
        int s=0;
        for(int i=0;i<a.length;i++){
            s=s|(f&a[i]);
            f=f^a[i];
            int n=~(f&s);
            f=f&n;
            s=s&n;
        }
        System.out.println(f);
    }
}
```

## Stock Buy Sell to Maximize Profit

The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

**Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.**

If we are allowed to buy and sell only once, then we can use following algorithm. Maximum difference between two elements. Here we are allowed to buy and sell multiple times. Following is algorithm for this problem.
1. Find the local minima and store it as starting index. If not exists, return.
2. Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
3. Update the solution (Increment count of buy sell pairs)
4. Repeat the above steps if end is not reached.

```
//Stock Buy Sell to Maximize Profit
class SellBuyStock{
        public static void main(String[] args) {
                int[] a={100, 180, 260, 310, 40, 535, 695};
                int max=0;
                for(int i=a.length-1;i>0;i--){
                        int x=a[i]-a[i-1];
                        if(x>0){
                                max+=x;
                        }
                }
                System.out.println(max);
        }
}
```

//Reverse words in a given string

Reverse words in a given string

Example: Let the input string be “i like this program very much”. The function should change the string to “much very program this like i”

reverse-words

```
import java.util.regex.Pattern;

class ReverseString{
        public static void main(String[] args) {
                String str="I love Java Programming";
                Pattern p=Pattern.compile("\\s");
                String[] temp=p.split(str);
                String result="";
                for(int i=0;i<temp.length;i++){
                        result=temp[i]+" "+result;
```

```
                    }
                    System.out.println(result);
          }
}
```

//Egg Dropping Puzzle

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

…..An egg that survives a fall can be used again.

…..A broken egg must be discarded.

…..The effect of a fall is the same for all eggs.

…..If an egg breaks when dropped, then it would break if dropped from a higher floor.

…..If an egg survives a fall then it would survive a shorter fall.

…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?

The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: Wiki for Dynamic Programming

Recommended: Please solve it on "PRACTICE " first, before moving on to the solution.

In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.

In the following solutions, we return the minimum number of trials in worst case; these solutions can be easily modified to print floor numbers of every trials also.

1) Optimal Substructure:

When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn’t break.

1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs

2) If the egg doesn’t break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.

Since we need to minimize the number of trials in worst case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

```
  k ==> Number of floors
  n ==> Number of Eggs
  eggDrop(n, k) ==> Minimum number of trials needed to find the critical
                    floor in worst case.
  eggDrop(n, k) = 1 + min{max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)):
                 x in {1, 2, ..., k}}
```

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
class EggDropping{
        public static void main(String[] args) {
                help_fun(36,2);
        }
        public static void help_fun(int floor,int egg){
                int[][] T=new int[egg+1][floor+1];
                for(int i=0;i<=floor;i++){
                        T[1][i]=i;
                }
                int c=0;
                for(int e=2;e<=egg;e++){
                        for(int f=2;f<=floor;f++){
                                T[e][f]=Integer.MAX_VALUE;
                                for(int k=1;k<=f;k++){
                                        c=1+Math.max(T[e-1][k-1],T[e][f-k]);
                                        if(c<T[e][f]){
                                                T[e][f]=c;
                                        }
                                }
                        }
                }
                System.out.println(T[egg][floor]);
        }
}
```

## Longest Common Subsequence | DP-4

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively. We also discussed one example problem in Set 3. Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

```
class LCS{
        public static void main(String[] args) {
                String s1="AGGTAB";
                String s2 ="GXTXAYB";
                char[] c1=s1.toCharArray();
                char[] c2=s2.toCharArray();
                int m=s1.length();
                int n=s2.length();
                int[][] dp=new int[m+1][n+1];
                for(int i=0;i<=m;i++){
                        for(int j=0;j<=n;j++){
                                if(i==0 || j==0){
                                        dp[i][j]=0;
                                }
                                else if(c1[i-1]==c2[j-1]){
                                        dp[i][j]=dp[i-1][j-1]+1;
                                }
                                else{
                                        dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);
                                }
                        }
                }
                System.out.println(dp[m][n]);
        }
```

}

## Edit Distance | DP-5

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

Examples:

```
Input:   str1 = "geek", str2 = "gesek"

Output:  1

We can convert str1 into str2 by inserting a 's'.


Input:   str1 = "cat", str2 = "cut"

Output:  1

We can convert str1 into str2 by replacing 'a' with 'u'.


Input:   str1 = "sunday", str2 = "saturday"

Output:  3

Last three and first characters are same.  We basically

need to convert "un" to "atur".  This can be done using

below three operations.

Replace 'n' with 'r', insert t, insert a
```

```
class Edit{
        public static void main(String[] args) {
                String s1 = "sunday";
        String s2 = "saturday";
        int m=s1.length();
        int n=s2.length();
```

```
    int[][] dp=new int[m+1][n+1];
    for(int i=0;i<=m;i++){
        dp[i][0]=i;
    }
    for(int i=0;i<=n;i++){
        dp[0][i]=i;
    }

    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(s1.charAt(i-1)==s2.charAt(j-1)){
                dp[i][j]=dp[i-1][j-1];
            }
            else{
                dp[i][j]=1+Math.min(dp[i-1][j-1],Math.min(dp[i-1][j],dp[i][j-1]));
            }
        }
    }
    System.out.println(dp[m][n]);
    }
}
```

# Sort a linked list of 0s, 1s and 2s

```
class LinkedList
{
        Node head;  // head of list

        /* Linked list Node*/
        class Node
        {
                int data;
                Node next;
                Node(int d) {data = d; next = null; }
        }

        void sortList()
        {
                // initialise count of 0 1 and 2 as 0
                int count[] = {0, 0, 0};

                Node ptr = head;

                /* count total number of '0', '1' and '2'
                  * count[0] will store total number of '0's
                  * count[1] will store total number of '1's
                  * count[2] will store total number of '2's  */
                while (ptr != null)
                {
                        count[ptr.data]++;
                        ptr = ptr.next;
                }

                int i = 0;
                ptr = head;

                /* Let say count[0] = n1, count[1] = n2 and count[2] = n3
                  * now start traversing list from head node,
                  * 1) fill the list with 0, till n1 > 0
                  * 2) fill the list with 1, till n2 > 0
                  * 3) fill the list with 2, till n3 > 0  */
                  while (ptr != null)
                  {
                        if (count[i] == 0)
                                i++;
                        else
                        {
                                ptr.data= i;
                                --count[i];
```

```
                    ptr = ptr.next;
                }
            }
        }


        /* Utility functions */

        /* Inserts a new Node at front of the list. */
        public void push(int new_data)
        {
                /* 1 & 2: Allocate the Node &
                                Put in the data*/
                Node new_node = new Node(new_data);

                /* 3. Make next of new Node as head */
                new_node.next = head;

                /* 4. Move the head to point to new Node */
                head = new_node;
        }

        /* Function to print linked list */
        void printList()
        {
                Node temp = head;
                while (temp != null)
                {
                     System.out.print(temp.data+" ");
                     temp = temp.next;
                }
                System.out.println();
        }

          /* Drier program to test above functions */
        public static void main(String args[])
        {
                LinkedList llist = new LinkedList();

                /* Constructed Linked List is 1->2->3->4->5->6->7->
                     8->8->9->null */
                llist.push(0);
                llist.push(1);
                llist.push(0);
                llist.push(2);
                llist.push(1);
                llist.push(1);
                llist.push(2);
                llist.push(1);
            llist.push(2);

            System.out.println("Linked List before sorting");
            llist.printList();

            llist.sortList();

            System.out.println("Linked List after sorting");
            llist.printList();
        }
}
```

Infix to postfix Notation

```
import java.util.Stack;

class InfixToPostFix
{
    // A utility function to return precedence of a given operator
    // Higher returned value means higher precedence
    static int Prec(char ch)
    {
        switch (ch)
        {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
```

```
        return 3;
    }
    return -1;
}

// The main method that converts given infix expression
// to postfix expression.
static String infixToPostfix(String exp)
{
    // initializing empty String for result
    String result = new String("");

    // initializing empty stack
    Stack<Character> stack = new Stack<>();

    for (int i = 0; i<exp.length(); ++i)
    {
        char c = exp.charAt(i);

         // If the scanned character is an operand, add it to output.
        if (Character.isLetterOrDigit(c))
            result += c;

        // If the scanned character is an '(', push it to the stack.
        else if (c == '(')
            stack.push(c);

        //  If the scanned character is an ')', pop and output from the stack
        // until an '(' is encountered.
        else if (c == ')')
        {
                while (!stack.isEmpty() && stack.peek() != '(')
                    result += stack.pop();

                if (!stack.isEmpty() && stack.peek() != '(')
                    return "Invalid Expression"; // invalid expression
                else
                    stack.pop();
            }
            else // an operator is encountered
            {
                while (!stack.isEmpty() && Prec(c) <= Prec(stack.peek()))
                    result += stack.pop();
                stack.push(c);
            }

        }

        // pop all the operators from the stack
        while (!stack.isEmpty())
            result += stack.pop();

        return result;
    }

    // Driver method
    public static void main(String[] args)
    {
        String exp = "a+b*(c^d-e)^(f+g*h)-i";
        System.out.println(infixToPostfix(exp));
    }
}
```

# Next higher palindromic number using the same set of digits

Given a palindromic number **num** having **n** number of digits. The problem is to find the smallest palindromic number greater than **num** using the same set of digits as in **num**. If no such number can be formed then print “Not Possible”.

The number could be very large and may or may not even fit into long long int.

Examples:

```
Input : 4697557964

Output :  4756996574


Input : 543212345

Output : Not Possible
```

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

**Approach:** Following are the steps:

1.If number of digits n <= 3, then print "Not Possible" and return.

2.Calculate **mid** = n/2 – 1.

3.Start traversing from the digit at index **mid** up to the 1st digit and while traversing find the index **i** of the rightmost digit which is smaller than the digit on its right side.

4.Now search for the smallest digit greater than the digit **num[i]** in the index range **i+1** to **mid**. Let the index of this digit be **smallest**.

5.If no such smallest digit found, then print “Not Possible”.

6.Else the swap the digits at index **i** and **smallest** and also swap the digits at index **n-i-1** and **n-smallest-1**. This step is done so as to maintain the palindromic property in **num**.

7.Now reverse the digits in the index range **i+1** to **mid**. Also If **n** is even then reverse the digits in the index range **mid+1** to **n-i-2** else if **n** is odd then reverse the digits in the index range **mid+2** to **n-i-2**. This step is done so as to maintain the palindromic property in **num**.

8.Print the final modified number **num**.

- C++
- Java
- Python
- C#
- PHP

filter_none

edit

play_arrow

brightness_4

```java
// Java implementation to find next higher
// palindromic number using the same set
// of digits
import java.util.*;

class NextHigherPalindrome
{
	// function to reverse the digits in the
	// range i to j in 'num'
	public static void reverse(char num[], int i,
											int j)
	{
		while (i < j) {
			char temp = num[i];
			num[i] = num[j];
			num[j] = temp;
			i++;
			j--;
		}
	}

	// function to find next higher palindromic
	// number using the same set of digits
	public static void nextPalin(char num[], int n)
	{
		// if length of number is less than '3'
		// then no higher palindromic number
		// can be formed
		if (n <= 3) {
			System.out.println("Not Possible");
			return;
}
char temp;

// find the index of last digit
// in the 1st half of 'num'
int mid = n / 2 - 1;
int i, j;

// Start from the (mid-1)th digit and
// find the the first digit that is
// smaller than the digit next to it.
for (i = mid - 1; i >= 0; i--)
	if (num[i] < num[i + 1])
		break;

// If no such digit is found, then all
// digits are in descending order which
// means there cannot be a greater
// palindromic number with same set of
// digits
if (i < 0) {
	System.out.println("Not Possible");
	return;
}

// Find the smallest digit on right
// side of ith digit which is greater
// than num[i] up to index 'mid'
int smallest = i + 1;
for (j = i + 2; j <= mid; j++)
	if (num[j] > num[i] &&
		num[j] < num[smallest])
		smallest = j;

// swap num[i] with num[smallest]
temp = num[i];
num[i] = num[smallest];
num[smallest] = temp;

// as the number is a palindrome,
// the same swap of digits should
// be performed in the 2nd half of
// 'num'
temp = num[n - i - 1];
num[n - i - 1] = num[n - smallest - 1];
num[n - smallest - 1] = temp;

// reverse digits in the range (i+1)
// to mid
```

```
            reverse(num, i + 1, mid);

            // if n is even, then reverse
            // digits in the range mid+1 to
            // n-i-2
            if (n % 2 == 0)
                reverse(num, mid + 1, n - i - 2);

            // else if n is odd, then reverse
            // digits in the range mid+2 to n-i-2
            else
                reverse(num, mid + 2, n - i - 2);

            // required next higher palindromic
            // number
            String result=String.valueOf(num);
            System.out.println("Next Palindrome: "+
                                                  result);
        }

        // Driver Code
        public static void main(String args[])
        {
            String str="4697557964";
            char num[]=str.toCharArray();
            int n=str.length();
            nextPalin(num,n);
        }
}

// This code is contributed by Danish Kaleem
```

**Output:**

```
Next Palindrome: 4756996574
```

Time Complexity: O(n)

Print cousins of a given node in Binary Tree | Single Traversal

Given a binary tree and a node, print all cousins of given node. Note that siblings should not be printed.

Examples:

Input : root of below tree

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

and pointer to a node say 5.

Output : 6, 7

**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**

Note that it is the same problem as given at Print cousins of a given node in Binary Treewhich consists of two traversals recursively. In this post, a single level traversal approach is discussed.
The idea is to go for level order traversal of the tree, as the cousins and siblings of a node can be found in its level order traversal. Run the traversal till the level containing the node is not found, and if found, print the given level.

How to print the cousin nodes instead of siblings and how to get the nodes of that level in the queue? During the level order, when for the parent node, if parent->left == Node_to_find, or parent->right == Node_to_find, then the children of this parent must not be pushed into the queue (as one is the node and other will be its sibling). Push the remaining nodes at the same level in the queue and then exit the loop. The current queue will have the nodes at the next level (the level of the node being searched, except the node and its sibling). Now, print the queue.
Following is the implementation of the above algorithm.

```
import java.util.*;
```

```
class Node{
	int data;
	Node left,right;
	Node(int d){
		data=d;
		left=null;
		right=null;
	}
}

class CznNode{
	public static void main(String[] args) {
		Node root = new Node(1);
	    root.left = new Node(2);
	    root.right = new Node(3);
	    root.left.left = new Node(4);
	    root.left.right = new Node(5);
	    root.left.right.right = new Node(15);
	    root.right.left = new Node(6);
	    root.right.right = new Node(7);
	    root.right.left.right = new Node(8);

	    help(root, root.left.right);
	}
	public static void help(Node root,Node x)
	{
		Node p=null;
		boolean flag=false;
		Queue<Node> q=new LinkedList<Node>();
		q.add(root);
```

```
while(!q.isEmpty() && flag==false){
        int q_size=q.size();
        while(q_size-->0){
                p=q.peek();
                q.remove();
                if(p.left==x || p.right==x){
                        flag=true;
                }
                else{
                        if(p.left!=null){
                                q.add(p.left);
                        }
                        if(p.right!=null){
                                q.add(p.right);
                        }
                }
        }
}
if(flag){
        int size=q.size();
        while(size>0){
                Node n=q.peek();
                q.poll();
                System.out.println(n.data+"  ");
                size--;
        }

}
else{
```

```
                        System.out.println("No Sibling found");
                   }








              }







         }
```

## Find total number of distinct years from a string

Given a string containing the words and dates, the task is to find the total number of distinct years mentioned in that string.

Note: Assuming that the date will be in ‘DD-MM-YYYY’ format.

Examples:

```
Input:  str = "UN was established on 24-10-1945.
                      India got freedom on 15-08-1947."

Output: 2
2 distinct years i.e. 1945 and 1947 have been referenced.


Input: str = "Soon after the world war 2 ended on 02-09-1945.
             The UN was established on 24-10-1945."

Output: 1
Only 1 Year, i.e 1945 has been referenced .
```

```
class DateString{
	public static void main(String[] args) {
		String str = "UN was established on 24-10-1945.India got freedom on 15-08-1947.";
		String s="";
		int c=0;
		int flag=0;
		for(int i=0;i<str.length();i++){
			if(Character.isDigit(str.charAt(i)) || str.charAt(i)=='-'){
				s+=str.charAt(i);
				c++;
				if(c==4){
					System.out.println(s);
					flag++;
				}
			}
			else {
				s="";
			}
			if(str.charAt(i)=='-'){
				c=0;
			}
		}
		System.out.println(flag);
	}
}
```

Reverse The String:

```
class ReverseStringWord{
	public static void main(String[] args) {
		String s="My name is X Y Z";
		StringBuilder result=new StringBuilder();
		StringBuilder str=new StringBuilder();
		for(int i=0;i<s.length();i++){
			if(s.charAt(i)==' '){
				result.insert(0,str+" ");
				str.setLength(0);
			}
			else{
				str.append(s.charAt(i));
				if(i==s.length()-1){
					result.insert(0,str+" ");
				}
			}
		}
		System.out.println(result);
	}
}
```

Sort120 element:

```
class Sort012{
	public static void main(String[] args) {
		int[] a={1,2,0,0,0,2,2,2,1,1,0,1,2};
		int[] c={0,0,0};
		for(int k=0;k<a.length;k++){
			c[a[k]]+=1;
		}

		int i=0;
```

```
                int m=0;
                int index=0;
                while(m<=a.length+1){

                        if(c[i]==0){
                                i++;
                        }
                        else{
                                a[index++]=i;
                                //System.out.print(a[i]+"  ");
                                c[i]-=1;
                        }
                        m++;

                }
                for(int k=0;k<a.length;k++){
                        System.out.print(a[k]+"  ");
                }
        }

}
```

Find Missing Two element:

```
class Missing2Element{
        public static void main(String[] args) {
                int[] a={1,2,3,4,7,8,9,10};
                boolean[] b=new boolean[a.length+3];
                for(int i=0;i<a.length;i++){
                        b[a[i]]=true;
                }
                for(int i=1;i<b.length;i++){
                        if(!b[i]){
                                System.out.println(i);
                        }
                }
        }
}
```

## Merge two sorted linked lists such that merged list is in reverse order

Given two linked lists sorted in increasing order. Merge them such a way that the result list is in decreasing order (reverse order).

Examples:

```
Input:  a: 5->10->15->40
           b: 2->3->20
Output: res: 40->20->15->10->5->3->2


Input:  a: NULL
           b: 2->3->20
Output: res: 20->3->2
```

**Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.**

A Simple Solution is to do following.

1) Reverse first list ‘a’.

2) Reverse second list ‘b’.

3) Merge two reversed lists.

Another Simple Solution is first Merge both lists, then reverse the merged list.

Both of the above solutions require two traversals of linked list.

How to solve without reverse, O(1) auxiliary space (in-place) and only one traversal of both lists?

The idea is to follow merge style process. Initialize result list as empty. Traverse both lists from beginning to end. Compare current nodes of both lists and insert smaller of two at the beginning of the result list.

```
1) Initialize result list as empty: res = NULL.

2) Let 'a' and 'b' be heads first and second lists respectively.

3) While (a != NULL and b != NULL)

    a) Find the smaller of two (Current 'a' and 'b')

    b) Insert the smaller value node at the front of result.

    c) Move ahead in the list of smaller node.

4) If 'b' becomes NULL before 'a', insert all nodes of 'a'

   into result list at the beginning.

5) If 'a' becomes NULL before 'b', insert all nodes of 'a'

   into result list at the beginning.
```

Below is the implementation of above solution.

- C/C++
- Java

filter_none
edit
play_arrow
brightness_4

```
// Java program to merge two sorted linked list such that merged
// list is in reverse order

// Linked List Class
class LinkedList {

    Node head;  // head of list
```

```java
static Node a, b;

/* Node Class */
static class Node {

	int data;
	Node next;

	// Constructor to create a new node
	Node(int d) {
		data = d;
		next = null;
	}
}

void printlist(Node node) {
	while (node != null) {
		System.out.print(node.data + " ");
		node = node.next;
	}
}

Node sortedmerge(Node node1, Node node2) {

	// if both the nodes are null
	if (node1 == null && node2 == null) {
		return null;
	}

	// resultant node
	Node res = null;

	// if both of them have nodes present traverse them
	while (node1 != null && node2 != null) {

		// Now compare both nodes current data
		if (node1.data <= node2.data) {
			Node temp = node1.next;
			node1.next = res;
			res = node1;
			node1 = temp;
		} else {
			Node temp = node2.next;
			node2.next = res;
			res = node2;
			node2 = temp;
		}
	}
```

```
        // If second list reached end, but first list has
        // nodes. Add remaining nodes of first list at the
        // front of result list
        while (node1 != null) {
            Node temp = node1.next;
            node1.next = res;
            res = node1;
            node1 = temp;
        }

        // If first list reached end, but second list has
        // node. Add remaining nodes of first list at the
        // front of result list
        while (node2 != null) {
            Node temp = node2.next;
            node2.next = res;
            res = node2;
            node2 = temp;
        }

        return res;

    }

    public static void main(String[] args) {

        LinkedList list = new LinkedList();
        Node result = null;

        /*Let us create two sorted linked lists to test
         the above functions. Created lists shall be
         a: 5->10->15
         b: 2->3->20*/
        list.a = new Node(5);
        list.a.next = new Node(10);
        list.a.next.next = new Node(15);

        list.b = new Node(2);
        list.b.next = new Node(3);
        list.b.next.next = new Node(20);

        System.out.println("List a before merge :");
        list.printlist(a);
        System.out.println("");
        System.out.println("List b before merge :");
        list.printlist(b);

        // merge two sorted linkedlist in decreasing order
        result = list.sortedmerge(a, b);
```

```
                System.out.println("");
                System.out.println("Merged linked list : ");
                list.printlist(result);

        }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
List A before merge:

5 10 15

List B before merge:

2 3 20

Merged Linked List is:

20 15 10 5 3 2
```

## Compare two strings represented as linked lists

Given two linked lists, represented as linked lists (every character is a node in linked list). Write a function compare() that works similar to strcmp(), i.e., it returns 0 if both strings are same, 1 if first linked list is lexicographically greater, and -1 if second string is lexicographically greater.

Examples:

```
Input: list1 = g->e->e->k->s->a

            list2 = g->e->e->k->s->b

Output: -1



Input: list1 = g->e->e->k->s->a
```

list2 = g->e->e->k->s

Output: 1

Input: list1 = g->e->e->k->s

list2 = g->e->e->k->s

Output: 0

## A program to check if a binary tree is BST or not

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

• The left subtree of a node contains only nodes with keys less than the node’s key.

• The right subtree of a node contains only nodes with keys greater than the node’s key.

• Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

• Each node (item in the tree) has a distinct key.

```
import java.util.*;
class JsonObject{
        public static void main(String[] args) {
                String s = "b) (c) ()";
                int temp=0;
                int flag=0;
                for(int i=0;i<s.length();i++){
                        if(s.charAt(i)=='('){
                                temp++;
                                if(temp>flag){
                                        flag=temp;
                                }
                        }
                        else if(s.charAt(i)==')'){
                                temp--;
                        }
                }
                System.out.println(flag);
        }

}
```

## Print path from root to a given node in a binary tree

Given a binary tree with distinct nodes(no two nodes have the same have data values). The problem is to print the path from root to a given node **x**. If node **x** is not present then print “No Path”.

Examples:

```
Input :          1
                   /   \
                  2     3
                 / \   /  \
                4   5  6   7

                   x = 5

Output : 1->2->5
```

**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**

**Approach:** Create a recursive function that traverses the different path in the binary tree to find the required node **x**. If node **x** is present then it returns true and accumulates the path nodes in some array **arr[]**. Else it returns false.
Following are the cases during the traversal:

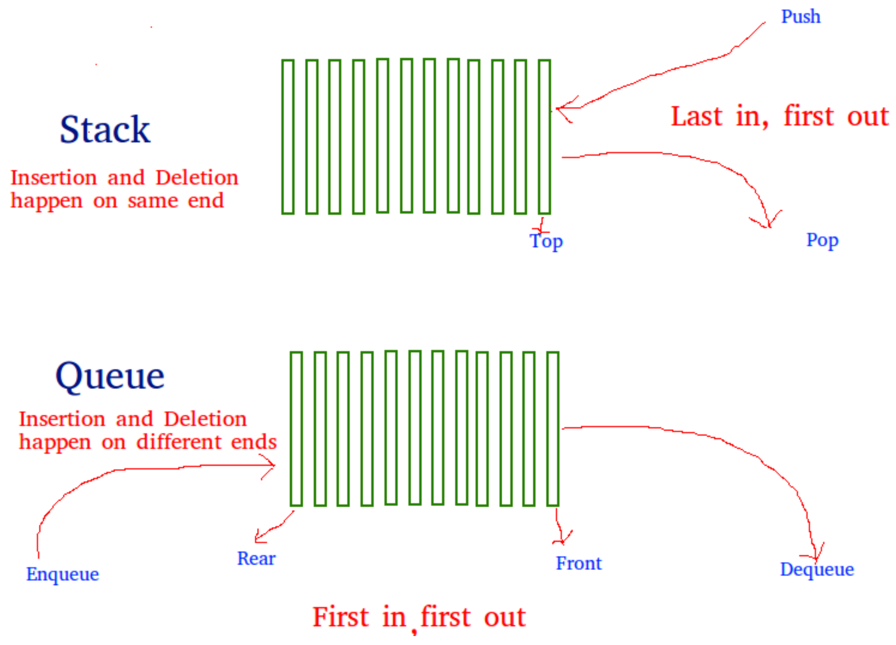
1. If **root = NULL**, return false.
2. push the root's data into **arr[]**.
3. if **root's data = x**, return true.
4. if node **x** is present in root's left or right subtree, return true.
5. Else remove root's data value from **arr[]** and return false.

```
import java.util.*;
class Node{
        int data;
        Node left, right;
        Node(int d){
                data=d;
                left=null;
                right=null;
        }



}

class RoottoNodePath{
        static ArrayList<Integer> ar=new ArrayList<Integer>();
        public static void main(String[] args) {
                Node root=new Node(1);
     root.left = new Node(2);
     root.right = new Node(3);
     root.left.left = new Node(4);
     root.left.right = new Node(5);
     root.right.left = new Node(6);
     root.right.right = new Node(7);
     int x=5;
     printPath(root, x);
        }
        public static void printPath(Node root,int x){
                if(hasPath(root,x,ar)){
                        for(int i=0;i<ar.size();i++){
                                System.out.print(ar.get(i));
                        }
                }
                else{
                                System.out.println("No path exist");
                        }
        }

        public static boolean hasPath(Node root,int x,ArrayList<Integer> ar){
                if(root==null){
                        return false;
                }
                ar.add(root.data);
                if(root.data==x){
                        return true;
                }

                if(hasPath(root.left,x,ar) || hasPath(root.right,x,ar)){
                        return true;
                }
```

```
                ar.remove(ar.size()-1);

                return false;




        }
```

## }Queue using Stacks

The problem is opposite of this post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.


Push
Stack
Insertion and Deletion happen on same end
Last in, first out
Top
Pop
Queue
Insertion and Deletion happen on different ends
Enqueue
Rear
Front
Dequeue
First in, first out

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

Method 1 (By making enQueue operation costly) This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

```
enQueue(q, x)
  1) While stack1 is not empty, push everything from stack1 to stack2.
  2) Push x to stack1 (assuming size of stacks is unlimited).
  3) Push everything back to stack1.
Here time complexity will be O(n)

deQueue(q)
  1) If stack1 is empty then error
  2) Pop an item from stack1 and return it
Here time complexity will be O(1)
```

- C++
- Java

filter_none
edit
play_arrow
brightness_4

```
// Java program to implement Queue using
// two stacks with costly enQueue()
import java.util.*;

class GFG
{
static class Queue
{
	static Stack<Integer> s1 = new Stack<Integer>();
	static Stack<Integer> s2 = new Stack<Integer>();

	static void enQueue(int x)
	{
		// Move all elements from s1 to s2
```

```
            while (!s1.isEmpty())
            {
                s2.push(s1.pop());
                //s1.pop();
            }

            // Push item into s1
            s1.push(x);

            // Push everything back to s1
            while (!s2.isEmpty())
            {
                s1.push(s2.pop());
                //s2.pop();
            }
        }

        // Dequeue an item from the queue
        static int deQueue()
        {
            // if first stack is empty
            if (s1.isEmpty())
            {
                System.out.println("Q is Empty");
                System.exit(0);
            }

            // Return top of s1
            int x = s1.peek();
            s1.pop();
            return x;
        }
    };

    // Driver code
    public static void main(String[] args)
    {
        Queue q = new Queue();
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);

        System.out.println(q.deQueue());
        System.out.println(q.deQueue());
        System.out.println(q.deQueue());
    }
    }

    // This code is contributed by Prerna Saini
```

# N Queen Problem | Backtracking-3

We have discussed Knight’s tour and Rat in a Maze problems in Set 1 and Set 2respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
{ 0,  1,  0,  0}

{ 0,  0,  0,  1}

{ 1,  0,  0,  0}

{ 0,  0,  1,  0}
```

```
public class
```

```
NQueenProblem {
                    class Position {
                    int row, col;
                    Position(int row, int col) {
                    this.row = row;
                    this.col = col;
                    }
                    }
                    public Position[] solveNQueenOneSolution(int n) {
                    Position[] positions = new Position[n];
                    boolean hasSolution = solveNQueenOneSolutionUtil(n, 0, positions);
                    if (hasSolution) {
                    return positions;
                    } else {
                    return new Position[0];
                    }
                    }
                    private boolean solveNQueenOneSolutionUtil(int n, int row, Position[]
                    positions) {
                    if (n == row) {
                    return true;
                    }
                    int col;
                    for (col = 0; col < n; col++) {
                    boolean foundSafe = true;
                    //check if this row and col is not under attack from any previous queen.
                    for (int queen = 0; queen < row; queen++) {
                    if (positions[queen].col == col || positions[queen].row - positions[queen].col
                    == row - col ||
                    positions[queen].row + positions[queen].col == row + col) {
                    foundSafe = false;
                    break;
                    }
                    }
                    if (foundSafe) {
                    positions[row] = new Position(row, col);
                    if (solveNQueenOneSolutionUtil(n, row + 1, positions)) {
                    return true;
                    }
                    }
                    }
                    return false;
                    }
```

# Find Two Missing Numbers | Set 1 (An Interesting Linear Time Solution)

Given an array of n unique integers where each element in the array is in range [1, n]. The array has all distinct elements and size of array is (n-2). Hence Two numbers from the range are missing from this array. Find the two missing numbers.

**Examples :**

```
Input  : arr[] = {1, 3, 5, 6}

Output : 2 4


Input : arr[] = {1, 2, 4}

Output : 3 5


Input : arr[] = {1, 2}

Output : 3 4
```

**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**

**Method 1 – O(n) time complexity and O(n) Extra Space**

**Step 1:** Take a boolean array *mark* that keeps track of all the elements present in the array.

**Step 2:** Iterate from 1 to n, check for every element if it is marked as true in the boolean array, if not then simply display that element.

•C++

•Python3

Method 2 – O(n) time complexity and O(1) Extra Space

The idea is based on this popular solution for finding one missing numbers. We extend the solution so that two missing elements are printed.

Let’s find out the sum of 2 missing numbers:

arrSum => Sum of all elements in the array

sum (Sum of 2 missing numbers) = (Sum of integers from 1 to n) - arrSum

= ((n)*(n+1))/2 – arrSum

avg (Average of 2 missing numbers) = sum / 2;

- •One of the numbers will be less than or equal to avg while the other one will be strictly greater then avg. Two numbers can never be equal since all the given numbers are distinct.
- •We can find the first missing number as sum of natural numbers from 1 to avg, i.e., avg*(avg+1)/2 minus the sum of array elements smaller than avg
- •We can find the second missing number as sum of natural numbers from avg+1 to n minus the sum of array elements greater than than avg

Consider an example for better clarification

Input : 1 3 5 6, n = 6

Sum of missing integers = n*(n+1)/2 - (1+3+5+6) = 6.

Average of missing integers = 6/2 = 3.

Sum of array elements less than or equal to average = 1 + 3 = 4

Sum of natural numbers from 1 to avg = avg*(avg + 1)/2

= 3*4/2 = 6

First missing number = 6 - 4 = 2

Sum of natural numbers from avg+1 to n

= n*(n+1)/2 - avg*(avg+1)/2

= 6*7/2 - 3*4/2

= 15

Sum of array elements greater than average = 5 + 6 = 11

Second missing number = 15 - 11 = 4

Below is implementation of above idea.

- C++
- Java
- Python3
- C#
- PHP

filter_none

edit

play_arrow

brightness_4

```java
// Java Program to find 2 Missing
// Numbers using O(1) extra space
import java.io.*;

class GFG
{

// Returns the sum of the array
static int getSum(int arr[], int n)
{
	int sum = 0;
	for (int i = 0; i < n; i++)
		sum += arr[i];
	return sum;
}

// Function to find two missing
// numbers in range [1, n]. This
// function assumes that size of
// array is n-2 and all array
// elements are distinct
static void findTwoMissingNumbers(int arr[],
								int n)
{
	// Sum of 2 Missing Numbers
	int sum = (n * (n + 1)) /
```

```
                                2 - getSum(arr, n - 2);

        // Find average of two elements
        int avg = (sum / 2);

        // Find sum of elements smaller
        // than average (avg) and sum of
        // elements greater than average (avg)
        int sumSmallerHalf = 0,
                sumGreaterHalf = 0;
        for (int i = 0; i < n - 2; i++)
        {
                if (arr[i] <= avg)
                        sumSmallerHalf += arr[i];
                else
                        sumGreaterHalf += arr[i];
        }

        System.out.println("Two Missing " +
                                                "Numbers are");

        // The first (smaller) element =
        // (sum of natural numbers upto
        // avg) - (sum of array elements
        // smaller than or equal to avg)
        int totalSmallerHalf = (avg *
                                                        (avg + 1)) / 2;
        System.out.println(totalSmallerHalf -
                                                    sumSmallerHalf);

        // The first (smaller) element =
        // (sum of natural numbers from
        // avg+1 to n) - (sum of array
        // elements greater than avg)
        System.out.println(((n * (n + 1)) / 2 -
                                                  totalSmallerHalf) -
                                                        sumGreaterHalf);
}

// Driver Code
public static void main (String[] args)
{
int arr[] = {1, 3, 5, 6};

// Range of numbers is 2
// plus size of array
int n = 2 + arr.length;

findTwoMissingNumbers(arr, n);
```

```
}
}

// This code is contributed by aj_36
```

**Output :**

```
Two Missing Numbers are

2 4
```

Difference between StringBuffer , StringBuilder and String in Java:

**Mutability Difference:**

`String` is **immutable**, if you try to alter their values, another object gets created, whereas `StringBuffer` and `StringBuilder` are **mutable** so they can change their values.

**Thread-Safety Difference:**

The difference between `StringBuffer` and `StringBuilder` is that `StringBuffer` is thread-safe. So when the application needs to be run only in a single thread then it is better to use `StringBuilder`. `StringBuilder` is more efficient than `StringBuffer`.

**Situations:**

- If your string is not going to change use a String class because a `String` object is immutable.
- If your string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a `StringBuilder` is good enough.
- If your string can change, and will be accessed from multiple threads, use a `StringBuffer`because `StringBuffer` is synchronous so you have thread-safety.

## How to create Immutable class in Java?

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like String, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:

- Class must be declared as final (So that child classes can’t be created)

• Data members in the class must be declared as final (So that we can’t change the value of it after object creation)

• A parameterized constructor

• Getter method for all the variables in it

• No setters(To not have option to change the value of the instance variable)

Example to create Immutable class

filter_none
edit
play_arrow
brightness_4

```
// An immutable class
public final class Student
{
	final String name;
	final int regNo;
	
	public Student(String name, int regNo)
	{
		this.name = name;
		this.regNo = regNo;
	}
	public String getName()
	{
		return name;
	}
	public int getRegNo()
	{
		return regNo;
	}
}

// Driver class
class Test
{
	public static void main(String args[])
	{
		Student s = new Student("ABC", 101);
		System.out.println(s.name);
		System.out.println(s.regNo);

		// Uncommenting below line causes error
		// s.regNo = 102;
```

```
        }
}
```

What is the difference betweeb wait and thread in Java:

Wait and sleep are two different things:

- In `sleep()` the thread stops working for the specified duration.
- In `wait()` the thread stops working until the object being waited-on is notified, generally by other threads.

One key difference not yet mentioned is that while sleeping a Thread does *not* release the locks it holds, while waiting releases the lock on the object that `wait()` is called on.

```
synchronized(LOCK) {
    Thread.sleep(1000); // LOCK is held
}
```

```
synchronized(LOCK) {
    LOCK.wait(); // LOCK is not held
}
```

Wait and Sleep both are time based system calls in Operating system. Purpose of them is slightly different.

Wait : This system call is used when user wants to halt the current process and **itreleases all the resources hold by the process** and waits for some other process to execute. We need to use notify to make this process aware that start execution again.
Sleep: This system call is used when user wants to halt the current process for sometime. **It keep the locks hold on resources till the sleep time is over** and again starts the execution of the process. Here process has control throughout the execution. e.g I executed some command on bash and want to sleep for some time since expecting some output from executed command which will be utilized in further execution of current process.

Object Clonning in Java:

**Why use `clone()` method ?**

The clone() **method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

**Advantage of Object cloning**

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

•You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.

•It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.

•clone() is the fastest way to copy array.

**Disadvantage of Object cloning**

Following is a list of some disadvantages of clone() method:

•To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone()etc.

•We have to implement Cloneable interface while it doesn't have any methods in it. This is just to tell the JVM that we can perform clone()on our object.

•Object.clone() is protected, so we have to provide our own clone()and indirectly call Object.clone() from it.

•Object.clone() doesn't invoke any constructor so we don't have any control over object construction.

•If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the **super**.clone() chain will fail.

•Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

## Serialization and Deserialization in Java with Example

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement
the java.io.Serializable interface.
The ObjectOutputStream class contains writeObject() method for serializing an Object.

```
public final void writeObject(Object obj)
                              throws IOException
```

The ObjectInputStream class contains readObject() method for deserializing an object.

```
public final Object readObject()
                          throws IOException,
                       ClassNotFoundException
```

Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.

Only the objects of those classes can be serialized which are implementing java.io.Serializable interface.

Serializable is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember

1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.
2. Only non-static data members are saved via Serialization process.
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.
4. Constructor of object is never called when an object is deserialized.
5. Associated objects must be implementing Serializable interface.

Example :

```
class A implements Serializable{


// B also implements Serializable

// interface.

B ob=new B();

}
```

SerialVersionUID

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an InvalidClassException. A Serializable class can declare its own UID explicitly by declaring a field name.
It must be static, final and of type long.
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;
If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification.
However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

serialver

The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.
You can run the following command to get serialVersionUID

```
serialver [-classpath classpath] [-show] [classname…]
```

Example 1:

filter_none
edit
play_arrow
brightness_4

```java
// Java code for serialization and deserialization
// of a Java object
import java.io.*;

class Demo implements java.io.Serializable
{
	public int a;
	public String b;

	// Default constructor
	public Demo(int a, String b)
	{
		this.a = a;
		this.b = b;
	}

}

class Test
{
	public static void main(String[] args)
	{
		Demo object = new Demo(1, "geeksforgeeks");
		String filename = "file.ser";

		// Serialization
		try
		{
			//Saving of object in a file
			FileOutputStream file = new FileOutputStream(filename);
			ObjectOutputStream out = new ObjectOutputStream(file);

			// Method for serialization of object
			out.writeObject(object);

			out.close();
			file.close();

			System.out.println("Object has been serialized");

		}

		catch(IOException ex)
```

```
        {
            System.out.println("IOException is caught");
        }


        Demo object1 = null;

        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object1 = (Demo)in.readObject();

            in.close();
            file.close();

            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }

    }
}
```

Output :

```
Object has been serialized

Object has been deserialized

a = 1

b = geeksforgeeks
```

# Find row number of a binary matrix having maximum number of 1s

Given a binary matrix (containing only 0 and 1) of order n*n. All rows are sorted already, We need to find the row number with maximum number of 1s. Also find number of 1 in that row.

Note: in case of tie, print smaller row number.

Examples :

```
Input : mat[3][3] = {0, 0, 1,
                     0, 1, 1,
                     0, 0, 0}
Output : Row number = 2, MaxCount = 2


Input : mat[3][3] = {1, 1, 1,
                     1, 1, 1,
                     0, 0, 0}
Output : Row number = 1, MaxCount = 3
```

**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**

**Basic Approach :** Traverse whole of matrix and for each row find the number of 1 and among all that keep updating the row number with maximum number of 1. This approach will result in O(n^2) time complexity.

Better Approach : We can perform a better if we try to apply binary search for finding position of first 1 in each row and as per that we can find the number of 1 from each row as each row is in sorted order. This will result in O(nlogn) time complexity.

**Efficient Approach :** Start with top left corner with index (1, n) and try to go left until you reach last 1 in that row (jth column), now if we traverse left to that row, we will find 0, so switch to the row just below, with same column. Now your position will be (2, j) again in 2nd row if jth element if 1 try to go left until you find last 1 otherwise in 2nd row if jth element is 0 go to next row. So Finally say if you are at any ith row and jth column which is index of last 1 from right in that row, increment i. So now if we have Aij = 0 again increment i otherwise keep decreasing j until you find last 1 in that particular row.

Sample Illustration :

**Algorithm :**

```
for (int i=0, j=n-1; i<n;i++)
{
    // find left most position of 1 in a row
    // find 1st zero in a row
    while (arr[i][j]==1)
```

```
        {
                row = i;
                j--;
        }
}
cout << "Row number =" << row+1;
cout << "MaxCount =" << n-j;
class FindMax1inMatrix{
        public static void main(String[] args) {
                int[][] m = { {0, 0, 0, 1},
                {0, 1, 1, 1},
                {1, 1, 1, 1},
                {0, 0, 0, 0}};
                int j=m[0].length-1;
                int row=0;
                int i;

                for(i=0;i<m.length;i++){
                        while(j>=0 && m[i][j]==1){
                                row=i;
                                j--;
                        }
                }
                System.out.print(row+1);



        }
}
```

# Pairwise swap elements of a given linked list

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

```
class Node{
                int data;
                Node next;
                Node(int d){
                        data=d;
                        next=null;
                }
        }
class SwapNode{
        static Node head;
        public static void main(String[] args) {
                SwapNode sn=new SwapNode();
                sn.head=new Node(1);
                sn.head.next=new Node(2);
                sn.head.next.next=new Node(3);
                sn.head.next.next.next=new Node(4);
                sn.head.next.next.next.next=new Node(5);
                sn.head.next.next.next.next.next=new Node(6);
                sn.head.next.next.next.next.next.next=new Node(7);
                sn.head.next.next.next.next.next.next.next=new Node(8);
                sn.head.next.next.next.next.next.next.next.next=new Node(9);
                help(head);
                Print(head);
        }
        public static void help(Node head){
```

```
        Node n=head;
        while(n!=null && n.next!=null){
            int temp=n.data;
            n.data=n.next.data;
            n.next.data=temp;
            n=n.next.next;
        }
    }
    public static void Print(Node head){
        Node temp=head;
        while(temp!=null){
            System.out.print(temp.data+" ");
            temp=temp.next;
        }
    }
}
```

perm_identity

# Find the row with maximum number of 1s

Given a boolean 2D array, where each row is sorted. Find the row with the maximum number of 1s.

Example:

Input matrix

```
0 1 1 1

0 0 1 1

1 1 1 1  // this row has maximum 1s

0 0 0 0
```

Output: 2

```
class FindMax1inMatrix{
        public static void main(String[] args) {
                int[][] m = { {0, 0, 0, 1},
                {0, 1, 1, 1},
                {1, 1, 1, 1},
                {0, 0, 0, 0}};
                int j=m[0].length-1;
                int row=0;
                int i;

                for(i=0;i<m.length;i++){
                        while(j>=0 && m[i][j]==1){
                                row=i;
                                j--;
                        }
                }
                System.out.print(row+1);



        }
}
```

## Cutting a Rod | DP-13

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
```

```
price    | 1   5   8   9  10  17  17  20
```

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 3   5   8   9  10  17  17  20
```

**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

```
class RodCutting{
        public static void main(String[] args) {
                int[] price = {1, 5, 8, 9, 10, 17, 17, 20};
                int t=8;
                System.out.println(helpdp(price,8));
        }

        public static int helpdp(int[] price,int n){
                int val[] = new int[n+1];
     val[0] = 0;

     // Build the table val[] in bottom up manner and return
     // the last entry from the table
     for (int i = 1; i<=n; i++)
     {
        int max_val = Integer.MIN_VALUE;
        for (int j = 0; j < i; j++)
           max_val = Math.max(max_val,price[j] + val[i-j-1]);
        val[i] = max_val;
     }
```

```
        return val[n];



          }
}
```

## Count occurrences of a word in string

You are given a string and a word your task is that count the number of the occurrence of the given word in the string and print the number of occurrence of the word.

Examples:

Input : string = "GeeksforGeeks A computer science portal for geeks"

word = "portal"

Output : Occurrences of Word = 1 Time

Input : string = "GeeksforGeeks A computer science portal for geeks"

word = "technical"

Output : Occurrences of Word = 0 Time

Approach :-

- First, we split the string by spaces in a
- Then, take a variable count = 0 and in every true condition we increment the count by 1
- Now run a loop at 0 to length of string and check if our string is equal to the word
- if condition true then we increment the value of count by 1 and in the end we print the value of count.

**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**

Below is the implementation of the above approach :

- Java
- Python 3
- C#
- PHP

filter_none

edit

play_arrow

brightness_4

```
// Java program to count the number
// of occurrence of a word in
// the given string given string
import java.io.*;

class GFG {

static int countOccurences(String str, String word)
{
	// split the string by spaces in a
	String a[] = str.split(" ");

	// search for pattern in a
	int count = 0;
	for (int i = 0; i < a.length; i++)
	{
	// if match found increase count
	if (word.equals(a[i]))
		count++;
	}

	return count;
}

// Driver code
public static void main(String args[])
{
	String str = "GeeksforGeeks A computer science portal for geeks ";
	String word = "portal";
	System.out.println(countOccurences(str, word));
}
}

/*This code is contributed by Nikita Tiwari.*/
```

Output:
1

## Check whether a given point lies inside a triangle or not

Given three corner points of a triangle, and one more point P. Write a function to check whether P lies within the triangle or not.

For example, consider the following program, the function should return true for P(10, 15) and false for P'(30, 15)

```
              B(10,30)
                / \
               /   \
              /     \
             /   P   \      P'
            /         \
     A(0,0) ----------- C(20,0)
```

Solution:

Let the coordinates of three corners be (x1, y1), (x2, y2) and (x3, y3). And coordinates of the given point P be (x, y)

1) Calculate area of the given triangle, i.e., area of the triangle ABC in the above diagram. Area A = [ x1(y2 – y3) + x2(y3 – y1) + x3(y1-y2)]/2

2) Calculate area of the triangle PAB. We can use the same formula for this. Let this area be A1.

3) Calculate area of the triangle PBC. Let this area be A2.

4) Calculate area of the triangle PAC. Let this area be A3.

5) If P lies inside the triangle, then A1 + A2 + A3 must be equal to A.

- C++
- Java
- Python
- C#
- PHP

filter_none

edit

play_arrow

brightness_4

```
// JAVA Code for Check whether a given point
// lies inside a triangle or not
import java.util.*;

class GFG {

    /* A utility function to calculate area of triangle
       formed by (x1, y1) (x2, y2) and (x3, y3) */
    static double area(int x1, int y1, int x2, int y2,
                                        int x3, int y3)
    {
       return Math.abs((x1*(y2-y3) + x2*(y3-y1)+
                                    x3*(y1-y2))/2.0);
    }

    /* A function to check whether point P(x, y) lies
       inside the triangle formed by A(x1, y1),
       B(x2, y2) and C(x3, y3) */
    static boolean isInside(int x1, int y1, int x2,
                int y2, int x3, int y3, int x, int y)
    {
       /* Calculate area of triangle ABC */
        double A = area (x1, y1, x2, y2, x3, y3);

       /* Calculate area of triangle PBC */
        double A1 = area (x, y, x2, y2, x3, y3);

       /* Calculate area of triangle PAC */
        double A2 = area (x1, y1, x, y, x3, y3);

       /* Calculate area of triangle PAB */
```

```
            double A3 = area (x1, y1, x2, y2, x, y);

           /* Check if sum of A1, A2 and A3 is same as A */
            return (A == A1 + A2 + A3);
        }

        /* Driver program to test above function */
        public static void main(String[] args)
        {
            /* Let us check whether the point P(10, 15)
                lies inside the triangle formed by
                A(0, 0), B(20, 0) and C(10, 30) */
           if (isInside(0, 0, 20, 0, 10, 30, 10, 15))
                System.out.println("Inside");
           else
                System.out.println("Not Inside");

        }
}
```

Q1: You are given an array in which you’ve to find a contiguous subarray such that the sum of elements in it is equal to zero. (I coded using hashtable in java)

```
;import java.util.*;

class SumZeroofSubArray{

        public static void main(String[] args) {

                int[] a={1, 4, -2, -2, 5, -4, 3};

                System.out.println(help(a));

        }

        public static boolean help(int[] a){

                int sum=0;

                HashSet<Integer> hs=new HashSet<Integer>();

                for(int i=0;i<a.length;i++){

                        sum+=a[i];

                        if(hs.contains(sum)){

                                return true;

                        }
```

```
                    else{
                            hs.add(sum);
                    }
            }
            return false;
    }
}
```

1. A string consists of parentheses and letters. Write a program to validate all the parentheses. Ignore the letters.
eg. ((alf)ls) – valid
)(dkk)() – invalid

```
import java.util.*;
class ValidParenthesis{
        public static void main(String[] args) {
            String s=")(dkk)() ";
            Stack<Character> st=new Stack<Character>();
            for(int i=0;i<s.length();i++){
                    if(s.charAt(i)=='('){
                            st.push('(');
                    }
                    else if(s.charAt(i)==')'){
                            try{
                                    st.pop();
                            }
                            catch(Exception e){
                                    st.push(')');
                                    System.out.print(false);
                                    break;
```

```
                        }
                }
        }
        if(st.isEmpty()){
                System.out.print(true);
        }
    }
}
```

## How to determine if a binary tree is height-balanced?

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

An empty tree is height-balanced. A non-empty binary tree T is balanced if:

1) Left subtree of T is balanced

2) Right subtree of T is balanced

3) The difference between heights of left subtree and right subtree is not more than 1.

```
import java.util.*;
class Node{
        int data;
        Node left;
        Node right;
        Node(int d){
                left=null;
                right=null;
        }
}

class IsBalance{
        static Node root;
        public static void main(String[] args) {
                root=new Node(1);
                root.left=new Node(1);
                root.left.left=new Node(1);
                root.left.right=new Node(1);
                root.right=new Node(1);
                root.right.left=new Node(1);
                root.right.right=new Node(1);
                System.out.println(help(root));
        }
        public static boolean help(Node node){
                if(node==null){
                        return true;
                }

                int ls=Height(node.left);
                int rs=Height(node.right);

                if(Math.abs(ls-rs)<=1 && help(node.left) && help(node.right)){
                        return true;
                }
                return false;
        }


        public static int Height(Node root){
            if(root==null ){
                  return 0;
            }



            return 1+Math.max(Height(root.left),Height(root.right));




      }
}
```

## Print a Binary Tree in Vertical Order | Set 2 (Map based Method)

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.

```
           1
        /    \
       2      3
     / \   /   \
    4   5  6   7
               /  \
              8   9
```

The output of print this tree vertically will be:

4

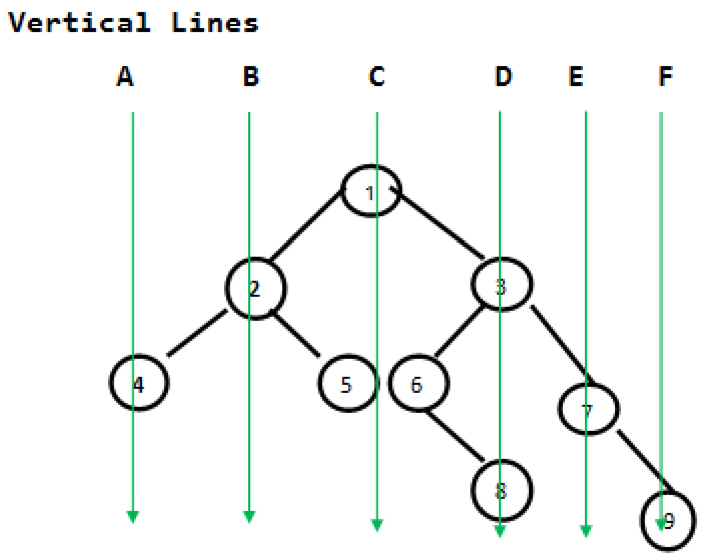
2

1 5 6

3 8

7

9



**Vertical order traversal is:**

**A- 4**

**B- 2**

**C- 1 5 6**

**D- 3 8**

**E- 7**

**F- 9**

```
import java.util.*;
class Node{
        int data;
        Node left;
        Node right;
        Node(int d){
                data=d;
                left=null;
                right=null;
        }
}
class VerticalOrder{
        static Node root;
        static TreeMap<Integer,ArrayList<Integer>> tm=new
TreeMap<Integer,ArrayList<Integer>>();
        public static void main(String[] args) {
                root = new Node(1);
       root.left = new Node(2);
       root.right = new Node(3);
       root.left.left = new Node(4);
       root.left.right = new Node(5);
       root.right.left = new Node(6);
       root.right.right = new Node(7);
       root.right.left.right = new Node(8);
       root.right.right.right = new Node(9);
       help(root,0);
       Print();
        }

        public static void help(Node node, int hd){
                if(node==null){
                        return;
                }

                if(tm.containsKey(hd)){
                        ArrayList<Integer> al=tm.get(hd);
                        al.add(node.data);
                tm.put(hd,al);
            }
            else{
                ArrayList<Integer> al=new ArrayList<Integer>();
                al.add(node.data);
                tm.put(hd,al);
            }
            help(node.left,hd-1);
            help(node.right,hd+1);
        }
        public static void Print(){
            for(Map.Entry m:tm.entrySet()){
                System.out.println(m.getKey()+" "+m.getValue());

        }
        }



}
```

## Find zeroes to be flipped so that number of consecutive 1’s is maximized

Given a binary array and an integer m, find the position of zeroes flipping which creates maximum number of consecutive 1’s in array.

Examples :

```
Input:   arr[] = {1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1}

           m = 2

Output:  5 7

We are allowed to flip maximum 2 zeroes. If we flip
```

arr[5] and arr[7], we get 8 consecutive 1's which is

maximum possible under given constraints

```
Input:   arr[] = {1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1}
           m = 1
Output:  7
```

We are allowed to flip maximum 1 zero. If we flip

arr[7], we get 5 consecutive 1's which is maximum

possible under given constraints.

```
Input:   arr[] = {0, 0, 0, 1}
           m = 4
Output:  0 1 2
```

Since m is more than number of zeroes, we can flip

all zeroes.

```
class ZeroCount{
        public static void main(String[] args) {
                int[] a={1, 0, 0, 1, 1, 0, 1, 0, 1, 1};
                int wl=0;
                int wr=0;
                int bl=0;
                int bw=0,zc=0;

                while(wr<a.length){
                        if(zc<=1){
                                if(a[wr]==0){
                                        zc++;
                                }
                wr++;
            }
            if(zc>1){
                if(a[wl]==0){
                    zc--;
                }
                wl++;
            }

            if(wr-wl>bw && zc<=1){
                //System.out.println(wr+"  "+wl);
                bw=wr-wl;
                bl=wl;
            }
        }

        for(int i=0;i<bw;i++){
            if(a[bl+i]==0){
                System.out.println((bl+i)+" ");
            }
        }
    }


}

/*
Find maximum level sum in Binary Tree
Given a Binary Tree having positive and negative nodes, the task is to find
maximum sum level in it.
```

```
Examples:

Input :               4
                   /   \
                  2    -5
                / \    /\
              -1   3 -2  6
Output: 6
Explanation :
Sum of all nodes of 0'th level is 4
Sum of all nodes of 1'th level is -3
Sum of all nodes of 0'th level is 6
Hence maximum sum is 6

Input :          1
              /   \
            2       3
          /  \       \
         4    5       8
                   /   \
                  6     7
Output :  17



*/

import java.util.*;
class Node{
        int data;
        Node left;
        Node right;
```

```
          Node(int d){
                    data=d;
                    left=null;
                    right=null;
          }
}
class MaxLevelSum{
          static Node root1;
          public static void main(String[] args) {
                    root1 = new Node(2);
        root1.right = new Node(3);
        root1.right.right = new Node(3);
        root1.left = new Node(10);
        root1.left.left = new Node(6);
        root1.left.left.right = new Node(3);
        root1.left.right = new Node(6);
        help(root1);
          }

          public static void help(Node n){
                    Queue<Node> q=new LinkedList<Node>();
                    q.add(n);
                    int max=Integer.MIN_VALUE;
                    while(!q.isEmpty()){
                              int sum=0;
                              int size=q.size();
                              while(size-->0){
                                        Node temp=q.poll();
                                        sum+=temp.data;
                                        if(sum>max){
                                                  max=sum;
                    }
                    if(temp.left!=null){
                            q.add(temp.left);
                    }

                    if(temp.right!=null){
                            q.add(temp.right);
                    }
                }
            }
            System.out.println(max);



        }


}
```

## Find maximum level sum in Binary Tree

Given a Binary Tree having positive and negative nodes, the task is to find maximum sum level in it.

Examples:

```
Input :          4
                / \
               2   -5
              / \  /\
                  -1   3 -2  6
```

Output: 6

Explanation :

Sum of all nodes of 0'th level is 4

Sum of all nodes of 1'th level is -3

Sum of all nodes of 0'th level is 6

Hence maximum sum is 6

```
Input :        1
              /   \
            2       3
          /  \       \
        4    5       8
                    /   \
                  6     7
```

Output :  17

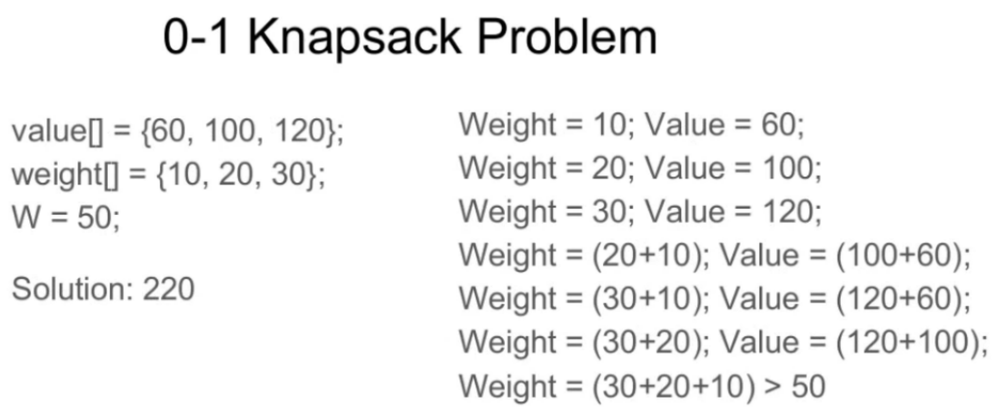
```
class Node{
        int data;
        Node left;
        Node right;
        Node(int d){
                data=d;
                left=null;
                right=null;
        }
```

```
    }

class CheckSubTree{



    static Node root1,root2;
    public static void main(String[] args) {
            root1 = new Node(26);
        root1.right = new Node(3);
        root1.right.right = new Node(3);
        root1.left = new Node(10);
        root1.left.left = new Node(4);
        root1.left.left.right = new Node(30);
        root1.left.right = new Node(6);

        // TREE 2
        /* Construct the following tree
           10
         /    \
         4      6
          \
          30  */

        root2 = new Node(10);
        root2.right = new Node(6);
        root2.left = new Node(4);
        root2.left.right = new Node(30);
        System.out.println(help(root1,root2));
    }

    public static boolean help(Node n1,Node n2){
```

```
		if(n2==null){
			return true;
		}

		if(n1==null){
			return false;
		}

		if(isIdentical(n1,n2)){
			return true;
		}

		return help(n1.left,n2) || help(n1.right,n2);
	}

	public static boolean isIdentical(Node n1,Node n2){
		if(n1==null && n2==null){
			return true;
		}

		if(n1!=null && n2!=null){

			return (n1.data==n2.data) && isIdentical(n1.left,n2.left) &&
isIdentical(n1.right,n2.right);
		}

		return false;
	}
}
```

# 0-1 Knapsack Problem | DP-10

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).



0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

**Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.**

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

- C/C++
- Java
- Python
- C#
- PHP

filter_none

edit

play_arrow

brightness_4

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                            knapSack(W, wt, val, n-1)
```

```c
                    );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Output:

```
220
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, K() refers to knapSack().  The two parameters indicated in the following recursion tree are n and W.

The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

```
                      K(3, 2)     ---------> K(n, W)
                  /       \
                /           \
            K(2,2)          K(2,1)
          /    \           /   \
        /        \       /      \
       K(1,2)   K(1,1)    K(1,1)   K(1,0)
       /  \       /  \      /  \
        /     \    /      \     /      \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
```

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
class KnapSackZeroOne{
        public static void main(String[] args) {
                int val[] = {60, 100, 120};
        int wt[] = {10, 20, 30};
        int  W = 50;
        System.out.println(Find_max(val,wt,W));
        }
        public static int Find_max(int[] val,int[] wt,int W){
                int n=val.length;
                int[][] dp=new int[n+1][W+1];
                for(int i=0;i<=n;i++){
                        for(int w=0;w<=W;w++){
                                if(i==0 || w==0){
                                        dp[i][w]=0;
                                }
                                else if(wt[i-1]<=w){
                                        dp[i][w]=Math.max(dp[i-1][w],dp[i-1][w-wt[i-1]]
+val[i-1]);
                                }
                                else{
```

```
                        dp[i][w]=dp[i-1][w];
                    }
                }

            }
            return dp[n][W];




        }
    }
```

perm_identity

## Find Excel column name from a given column number

MS Excel columns has a pattern like A, B, C, … ,Z, AA, AB, AC,…. ,AZ, BA, BB, … ZZ, AAA, AAB ….. etc. In other words, column 1 is named as “A”, column 2 as “B”, column 27 as “AA”.

Given a column number, find its corresponding Excel column name. Following are more examples.

| Input | Output |
|---|---|
| 26 | Z |
| 51 | AY |
| 52 | AZ |
| 80 | CB |
| 676 | YZ |
| 702 | ZZ |
| 705 | AAC |

```
class ExcelColumnName{
	public static void main(String[] args) {
		int x=705;
		System.out.println(help(x));
	}
	public static String help(int x){
		StringBuilder sb=new StringBuilder();
		while(x>0){
			int rem=x%26;
			if(rem==0){
				sb.append('Z');
				x=x/26-1;
			}
			else{

				sb.append((char)((rem-1)+'A'));
				x=x/26;
			}
		}
		return sb.reverse().toString();
	}
}

/*perm_identity
Level order traversal in spiral form
Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.
spiral_order*/
import java.util.*;
class Node{
	int data;
```

```
        Node left;
        Node right;
        Node(int d){
                data=d;
                left=null;
                right=null;
        }
}
class SpiralOrderTraversal{
        public static void main(String[] args) {
                Node root = new Node(1);
       root.left = new Node(2);
       root.right = new Node(3);
       root.left.left = new Node(7);
       root.left.right = new Node(6);
       root.right.left = new Node(5);
       root.right.right = new Node(4);
       help(root);
        }
        public static void help(Node n){
                Stack<Node> st1=new Stack<Node>();
                Stack<Node> st2=new Stack<Node>();
                st1.push(n);
                while(!st1.isEmpty() || !st2.isEmpty()){
                        while(!st1.isEmpty()){
                                Node temp=st1.peek();
                                st1.pop();
                                System.out.print(temp.data+" ");

                                if(temp.right!=null){
                                        st2.push(temp.right);
                                }
                                if(temp.left!=null){
                                        st2.push(temp.left);
                                }


                        }
                        while(!st2.isEmpty()){
                                Node temp=st2.peek();
                                st2.pop();
                                System.out.print(temp.data+" ");
                                if(temp.left!=null){
                                        st1.push(temp.left);
                                }
                                if(temp.right!=null){
                                        st1.push(temp.right);
                                }
                        }
                }
        }
}
```

/*perm_identity

Recursively remove all adjacent duplicates

Given a string, recursively remove adjacent duplicate characters from string. The output string should not have any adjacent duplicates. See following examples.

Input:  azxxzy

Output: ay

First "azxxzy" is reduced to "azzy".

The string "azzy" contains duplicates,

so it is further reduced to "ay".

```
Input: geeksforgeeg
Output: gksfor
First "geeksforgeeg" is reduced to
"gksforgg". The string "gksforgg"
contains duplicates, so it is further
reduced to "gksfor".

Input: caaabbbaacdddd
Output: Empty String

Input: acaaabbbacdddd
Output: acac*/

class RecursiveRemove{
	public static void main(String[] args) {
		String s="acaaabbbacdddd";
		System.out.println(help(s));
	}

	public static String help(String s){
		StringBuilder sb=new StringBuilder();
		boolean flag=true;
		char prev=s.charAt(0);
		for(int i=1;i<s.length();i++){
			if(prev!=s.charAt(i)){
				if(flag){
					sb.append(prev);
				}
				else{
					flag=true;
```

```
                                }


                        }
                        else{
                                flag=false;
                        }
                        if(sb.length()>=2 && sb.charAt(sb.length()-
1)==sb.charAt(sb.length()-2))
                        {
                                sb.deleteCharAt(sb.length()-1);
                                sb.deleteCharAt(sb.length()-1);
                        }
                        prev=s.charAt(i);



                }
                return sb.toString();
        }
}
```

## Reverse Level Order Traversal

We have discussed level order traversal of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.

The method 2 of normal level order traversal can also be easily modified to print level order traversal in reverse order. The idea is to use a stack to get the reverse level order. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print contents of stack, we get “5 4 3 2 1” for above example tree, but output should be “4 5 2 3 1”. So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to stack, then left subtree.

```
/*Reverse Level Order Traversal

import java.util.*;
class Node{
        int data;
        Node left;
        Node right;
        Node(int d){
                data=d;
                left=null;
                right=null;
        }
}

class ReverseLevelOrder{
        public static void main(String[] args) {
                Node root = new Node(1);
       root.left = new Node(2);
       root.right = new Node(3);
       root.left.left = new Node(7);
       root.left.right = new Node(6);
       root.right.left = new Node(5);
       root.right.right = new Node(4);
       help(root);
        }
        public static void help(Node n){
                Stack<Integer> st=new Stack<Integer>();
                Queue<Node> q=new LinkedList<Node>();
                q.add(n);
                while(!q.isEmpty()){
                        int size=q.size();
                        while(size-->0){
                                Node temp=q.poll();
                                st.push(temp.data);
                                if(temp.left!=null){
                                        q.add(temp.left);
                                }
                                if(temp.right!=null){
                                        q.add(temp.right);
                                }
                        }
                        st.push(-1);
                }
                while(!st.isEmpty()){
                        if(st.peek()!=-1){
                                System.out.print(st.peek()+" ");
                        }
                        else{
                                System.out.println();
                        }
                        st.pop();
                }




        }
}
```

/*Leaders in an array

Write a program to print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. For example int the array {16, 17, 4, 3, 5, 2}, leaders are 17, 5 and 2.

Let the input array be arr[] and size of the array be size.*/

```
class LeadersInArray{
	public static void main(String[] args) {
		int[] a={16, 17, 4, 3, 5, 2};
		int n=a.length;
		int ll=a[n-1];
		String s=String.valueOf(a[n-1]+" ");
		for(int i=n-2;i>=0;i--){
			if(ll<a[i]){
				s=String.valueOf(a[i])+"  "+s;
				ll=a[i];
			}
		}
		System.out.println(s);
	}
}
```

/*Find the number of islands | Set 1 (Using DFS)

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

Example:

```
Input : mat[][] = {{1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1}
Output : 5
```

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”.*/

```
class MatrixIsland{
        public static void main(String[] args) {
                int[][] a = {{1, 1, 0, 0, 0},
                  {0, 1, 0, 0, 1},
                  {1, 0, 0, 1, 1},
                  {0, 0, 0, 0, 0},
                  {1, 0, 1, 0, 1}};
       System.out.println(help(a));


        }
        public static int help(int[][] a){
                int isLand=0;
                for(int i=0;i<a.length;i++){
                        for(int j=0;j<a[0].length;j++){
                                if(a[i][j]==1){
                                        isLand++;
                                        destroy(a,i,j);
                                }
                        }
                }
                return isLand;
        }
        public static void destroy(int[][] a,int i,int j){
                a[i][j]=0;
                if(i<a.length-1 && a[i+1][j]==1){
                        destroy(a,i+1,j);
                }
                if(i>0 && a[i-1][j]==1){
                        destroy(a,i+1,j);
```

```
			}
			if(j<a[0].length-1 && a[i][j+1]==1){
				destroy(a,i,j+1);
			}
			if(j>0 && a[i][j-1]==1){
				destroy(a,i,j-1);
			}
		}
	}
```

/*perm_identity

Remove BST keys outside the given range

Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].

BinaryTree1

The given tree should be changed to following. Note that all keys outside the range [-10, 13] are removed and modified tree is BST.

*/

```
class Node{
			int data;
			Node left;
			Node right;
			Node(int d){
				data=d;
				left=null;
				right=null;
			}
		}
```

```
class RemoveOutsideRange{

	 public static void main(String[] args) {
		 Node root = new Node(1);
	root.right = new Node(10);
	root.left = new Node(-10);
	root.left.left = new Node(-27);
	root.left.right = new Node(-5);
	root.right.left = new Node(5);
	root.right.right = new Node(40);
	Print(root);
	System.out.println();
	Node result=Remove(root,1,10);
	Print(result);
	 }

	 public static Node Remove(Node root,int min,int max){
		 if(root==null){
			 return null;
		 }
		 if(root.data<min){
			 return Remove(root.right,min,max);
		 }
		 if(root.data>max){
			 return Remove(root.left,min,max);
		 }

		 root.left=Remove(root.left,min,max);
		 root.right=Remove(root.right,min,max);
		 //System.out.println(root.data);
		return root;
	}

	public static void Print(Node root){
		if(root!=null){
			Print(root.left);
			System.out.print(root.data+"  ");
			Print(root.right);
		}
	}
}
```

/*Reverse Level Order Traversal

We have discussed level order traversal of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.

The method 2 of normal level order traversal can also be easily modified to print level order traversal in reverse order. The idea is to use a stack to get the reverse level order. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print contents of stack, we get “5 4 3 2 1” for above example tree, but output should be “4 5 2 3 1”. So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to stack, then left subtree.*/

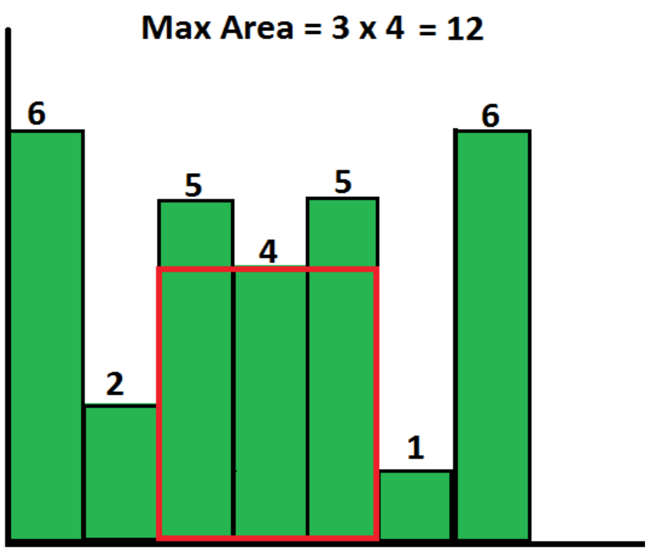
```
import java.util.*;
class Node{
	int data;
	Node left;
	Node right;
	Node(int d){
		data=d;
		left=null;
		right=null;
	}
```

```
}

class ReverseLevelOrder{
         public static void main(String[] args) {
                  Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(7);
        root.left.right = new Node(6);
        root.right.left = new Node(5);
        root.right.right = new Node(4);
        help(root);
         }
         public static void help(Node n){
                  Stack<Integer> st=new Stack<Integer>();
                  Queue<Node> q=new LinkedList<Node>();
                  q.add(n);
                  while(!q.isEmpty()){
                           int size=q.size();
                           while(size-->0){
                                    Node temp=q.poll();
                                    st.push(temp.data);
                                    if(temp.left!=null){
                                             q.add(temp.left);
                                    }
                                    if(temp.right!=null){
                                             q.add(temp.right);
                                    }
                           }
                           st.push(-1);
                  }
        while(!st.isEmpty()){
            if(st.peek()!=-1){
                System.out.print(st.peek()+" ");
            }
            else{
                System.out.println();
            }
            st.pop();
        }



    }
}
```

## Largest Rectangular Area in a Histogram | Set 1

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 2, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



```
/*Largest Rectangular Area in a Histogram | Set 1

Find the largest rectangular area possible in a given histogram where the
largest rectangle can be made of a number of contiguous bars. For simplicity,
assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5,
4, 5, 2, 6}. The largest possible rectangle possible is 12 (see the below figure,
the max area rectangle is highlighted in red)


histogram*/

import java.util.*;

class MaxHistogram{

	public static void main(String[] args) {

		int[] a={6, 1, 5, 4, 5, 2, 6};

		System.out.println(MaxHistoGram(a));



	}

	public static int MaxHistoGram(int[] a){

		Stack<Integer> st=new Stack<Integer>();

		int i=0;

		int max=0,area=0;
```

```
while(i<a.length){
        if(st.isEmpty() || a[st.peek()]<=a[i]){
                st.push(i++);
        }
        else{
                int top=st.peek();
                st.pop();
                if(st.isEmpty()){
                        area=a[top]*i;
                }
                else{
                        area=a[top]*(i-st.peek()-1);
                }
                if(area>max){
                        max=area;
                }
        }
        System.out.println(area);
}
while(st.isEmpty()){
        int top=st.peek();
                st.pop();
                if(st.isEmpty()){
                        area=a[top]*i;
                }
                else{
                        area=a[top]*(i-st.peek()-1);
                }
                if(area>max){
                        max=area;
                }
```

```
                        System.out.println(area);
                }
                return max;
        }



}

/*Rearrange characters in a string such that no two adjacent are same*/

import java.util.*;
import java.io.*;

class CharFrequency{
        char ch;
        int fq;
        CharFrequency(char c,int f){
                ch=c;
                fq=f;
        }
}



class KeyComparator implements Comparator<CharFrequency>{
        public int compare(CharFrequency c1,CharFrequency c2){
                if(c1.fq<c2.fq){
                        return 1;
                }
                else if(c1.fq>c2.fq){
```

```
                return -1;
            }
            return 0;
        }
}




class ReArrangeString{
        public static void main(String[] args) {
            String s="aaabc";
            System.out.println(help(s));
        }

        public static String help(String s){
            String str="";
            char[] ca=new char[256];

            PriorityQueue<CharFrequency> pq = new
PriorityQueue<CharFrequency>(new KeyComparator());



            for(int i=0;i<s.length();i++){
                ca[(int)s.charAt(i)]++;
            }
            for(char c='a';c<='z';c++){
                int val=ca[(int)c];
                if(val>0){
                    pq.add(new CharFrequency(c,val));
                }
            }
```

```
		CharFrequency prev=new CharFrequency('#',-1);
		while(pq.size()>0){
			CharFrequency k=pq.peek();
			pq.poll();
			str+=k.ch;
			if(prev.fq>0){
				pq.add(prev);
			}

			(k.fq)--;
			prev=k;
		}
		return str;
	}
}
```

/*Given a range [L,R] find the count of numbers having prime number of set bits in their binary representation. [This hint was included in O/P section. Only even numbers should be checked within the L,R range]*/

```
class CountPrimeBit{
	public static void main(String[] args) {
		int m=6;
		int n=10;
		int count=0;
		for(int i=m;i<=n;i++){
			int bc=Integer.bitCount(i);
			if(isPrime(bc)){
				count++;
			}
```

```
                }
                //System.out.println(count);


        }
        public static boolean isPrime(int n){
                System.out.println(n);
                        if(n<=1){
                                return false;
                        }
                        for(int i=3;i<=Math.sqrt(n);i++){
                                if(n%i==0){
                                        return false;
                                }
                        }
                        return true;



                }
}
```

/*Find if two rectangles overlap

Given two rectangles, find if the given two rectangles overlap or not.

Note that a rectangle can be represented by two coordinates, top left and bottom right. So mainly we are given following four coordinates.

l1: Top Left coordinate of first rectangle.

r1: Bottom Right coordinate of first rectangle.

l2: Top Left coordinate of second rectangle.

r2: Bottom Right coordinate of second rectangle.

rectanglesOverlap

We need to write a function bool doOverlap(l1, r1, l2, r2) that returns true if the two given rectangles overlap.

Recommended: Please solve it on “PRACTICE ” first, before moving on to the solution.

Note : It may be assumed that the rectangles are parallel to the coordinate axis.

One solution is to one by one pick all points of one rectangle and see if the point lies inside the other rectangle or not. This can be done using the algorithm discussed here.

Following is a simpler approach. Two rectangles do not overlap if one of the following conditions is true.

1) One rectangle is above top edge of other rectangle.

2) One rectangle is on left side of left edge of other rectangle.

We need to check above cases to find out if given rectangles overlap or not. Following is the implementation of the above approach.*/

```
class RectangleOverlap{
    static class P{
        int x,y;
        P(int a,int b){
            x=a;
            y=b;
        }
    }
    public static void main(String[] args) {
        P l1=new P(0,10);//top left 1
        P r1=new P(10,0);//botton right 1
        P l2=new P(5,5);//top left 2
        P r2=new P(15,0);//bottom right 2
```

```
		System.out.println(help(l1,r1,l2,r2));
	}
	public static boolean help(P l1,P r1,P l2,P r2){
		if(l1.x>r2.x || l2.x>r1.x || l1.y<r2.y || l2.y<r1.y){
			return false;
		}
		return true;


	}
}
```

/*Find minimum number of coins that make a given value

Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?

Examples:

Input: coins[] = {25, 10, 5}, V = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, V = 11

Output: Minimum 2 coins required

We can use one coin of 6 cents and 1 coin of 5 cents*/

```
class MinCoin{
	public static void main(String[] args) {
		int[] a={9,6,5,1};
		int v=11;
		int[] dp=new int[v+1];
		dp[0]=0;
```

```
                for(int i=1;i<=v;i++){
                        dp[i]=Integer.MAX_VALUE;
                }
                for(int i=1;i<=v;i++){
                        for(int j=0;j<a.length;j++){
                                if(a[j]<=i){
                                        int sub_res=dp[i-a[j]];
                                        if(sub_res!=Integer.MAX_VALUE &&
sub_res+1<dp[i]){
                                                dp[i]=sub_res+1;
                                        }
                                }
                        }
                }
                System.out.println(dp[v]);
        }
}
```