Ex.1   
**package** Recursion;

//1st program on 31 may 2024

**public** **class** A1Print1toNandNto1 {

//Recursion

//Que. Input:n i.e 5 and Print 5 to 1 using print fun and 1 to 5 using printFrom1 fun.

//and print both sequence using printBoth function

//Time complexity:O(n) and Space complexity:O(n);

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

**int** n=5;

*print*(n);

*printFrom1*(n);

*printBoth*(n);

}

**public** **static** **void** print(**int** n) {

**if**(n==1) { //Base case

System.***out***.println(n);

**return**;

}

System.***out***.println(n); //to print 5 to 1

*print*(n-1); //Recursion call

}

**public** **static** **void** printFrom1(**int** n) {

**if**(n==0) { //base call

**return**;

}

*printFrom1*(n-1); //Recursion call

System.***out***.println(n); //to print 1 to 5

}

**public** **static** **void** printBoth(**int** n) {

**if**(n==0) { //Base case

**return**;

}

System.***out***.println(n); //to print 5 to 1

*printBoth*(n-1); //Recursion call

System.***out***.println(n); //to print 1 to 5

}

}

Ex.2

**package** Recursion;

**public** **class** A2FactorialOfNo {

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

**int** num=3;

**int** factorialAns=*fact*(num);

System.***out***.println("Factorial of "+num+" is "+factorialAns); //Ans:6

}

**public** **static** **int** fact(**int** No) {

**if**(No<=1) { //Base case . we Know 1!=1 so we decide base case. Also 0!=1 we can write (No<=1)

**return** 1;

}

**return** No\**fact*(No-1); //We can devide factorial into sub part i.e recursive equation

}

}

Ex.3  
**package** Recursion;

**public** **class** A3SumOfNnumbers {

//Sum of a number n using recursion

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

**int** n=5;

**int** sumOfN=*sum*(n);

System.***out***.println("Sum of "+n+" no from 1 is "+sumOfN);

}

**public** **static** **int** sum(**int** No) {

**if**(No==1) {//Base case

**return** 1;

}

**return** No+*sum*(No-1); //recursive equation

}

}

Ex.4  
**package** Recursion;

**public** **class** A4SumOfDigits {

//Sum of digits of a number using normal and Recursion way

//For both way: Time complexity:O(No of digits)=O(log(n)) and for recursion way Space complexity: O(log(n));

// Here n is number and log(n) means no of digits of that number

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

**int** no=54;

**int** normalSumOfDigits=*normalWay*(no);

System.***out***.println("Normal way Sum of digits:"+normalSumOfDigits);//Ans:9

**int** recursionSumOfDigits=*recursionWay*(no);

System.***out***.println("Recursive way Sum of digits:"+recursionSumOfDigits);//Ans:9

}

**public** **static** **int** normalWay(**int** N) {

**int** sumOfDigits=0;

**while**(N>0) {

sumOfDigits += N%10; // N%10 means reminder(last digit of a number N)

N/=10; // N=N/10; We remove the last digit of a number N

}

**return** sumOfDigits;

}

**public** **static** **int** recursionWay(**int** N) {

**if**(N==0) { //base case

**return** 0;

} //IMP\*\* Note : Before solving recursive example always draw a equation using pen and paper

**return** N%10 + *recursionWay*(N/10); // Recursive equation draw on paper using pen

}

}

Ex.5  
**package** Recursion;

**public** **class** A5ProductOfDigitsOfNum {

//Product of a digits of a number using recursion

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

**int** n=1342;

**int** productOfDigits = *prod*(n);

System.***out***.println("Product of digits: "+productOfDigits);//Ans:24

}

**public** **static** **int** prod(**int** n) {

**if**(n==0) { //Base case or if(n%10==n) then return n;

**return** 1;

}

**return** (n%10)\**prod*(n/10); //recursive equation

}

}

Ex.6  
**package** Recursion;

**public** **class** A6ReverseNumber {

//Que. Reverse a number using normal way and using recursion

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

**int** num=3524;

**int** normalIterativeway=*revIterative*(num);

System.***out***.println("Reverse a num "+num+" using iterative way "+normalIterativeway);

//Using recursion

**int** rev=0;

**int** revRecursiveWay=*revRecursive*(rev,num);

System.***out***.println("Reverse of a num "+num+" using recursion is "+revRecursiveWay);

//Using Kunal method

**int** revNum= *helper*(num);

System.***out***.println("Reverse of a num "+num+" using no of digits/helper "+revNum);

}

**public** **static** **int** helper(**int** num) {

**int** noOfDigits = (**int**)Math.*log10*(num)+1; //Formula to count no of digits

**return** *revUsingNoOfDigits*(num,noOfDigits);

}

**public** **static** **int** revUsingNoOfDigits(**int** num, **int** noOfDigits) {

**if**(num%10==num) {//base case

**return** num;

}

**return** (num%10)\*(**int**)Math.*pow*(10, noOfDigits-1) + *revUsingNoOfDigits*(num/10,noOfDigits-1);//logic of Kunal Kushwaha

}

**public** **static** **int** revIterative(**int** n) {

**int** rev=0;

**while**(n>0) {

rev = rev\*10 + n%10; //formula for reverse

n/=10;

}

**return** rev;

}

//Recursive way imp

**public** **static** **int** revRecursive(**int** rev,**int** N) {

**if**(N==0){

**return** rev; //imp

}

rev=(rev\*10)+(N%10);

N/=10;

**return** *revRecursive*(rev,N);

}

}

Ex.7  
**package** Recursion;

**public** **class** A7PalindromNum {

//Check given no is palindrom or not using both normal and recursive way

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

**int** num=35653;

//Normal Way

**boolean** isPalindromNormal=*palindrom*(num);

System.***out***.println("The given num is palindrom: "+isPalindromNormal);

//Palindrom using Recursion way

**boolean** isPalinndromrecursion= *isPalind*(num);

System.***out***.println("The given no is palindrom using recursion: "+isPalinndromrecursion);

}

//Normal/Iterative way

**public** **static** **boolean** palindrom(**int** N) {

**int** rev=0;

**int** temp=N;

**while**(N>0) {

rev = rev\*10 + N%10;

N/=10;

}

**if**(temp==rev) {

**return** **true**;

}

**return** **false**;

}

//Recursive way

**public** **static** **boolean** isPalind(**int** N) {

**int** noOfDigits= (**int**)Math.*log10*(N)+1; //formula to find no of digits of a number

**int** temp=N;

**int** revOfNum= *revNum*(N,noOfDigits);

**if**(N==revOfNum) {

**return** **true**;

}

**return** **false**;

}

**public** **static** **int** revNum(**int** N,**int** noOfdigits) {

**if**(N%10==N) {//Base case means when remain last digit then return as it is

**return** N;

}

**return** N%10 \* (**int**)Math.*pow*(10, noOfdigits-1) + *revNum*(N/10,noOfdigits-1);

}

}

Ex.8  
**package** Recursion;

**public** **class** A8CountZeroInNumber {

//Que. Count zero's in a number using normal and recursive way

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

**int** num=10;

**int** countZeroNormalWay = *countZero*(num);

System.***out***.println("Total no of zeros in "+num+" is "+countZeroNormalWay);

//Recursive way

**int** totalZeros= *countZeroAns*(num,0);

System.***out***.println("Total no of zeros using recursion in no "+num+" is "+totalZeros);

}

//Normal way using iteration

**public** **static** **int** countZero(**int** num) {

**int** count=0;

**while**(num>0) {

**if**(num%10==0) {

count++;

}

num/=10;

}

**return** count;

}

//Recursive way

**public** **static** **int** countZeroAns(**int** num,**int** count) {

**if**(num==0) {

**return** count;

}

**if**(num%10==0) {

count++;

}

**return** *countZeroAns*(num/10,count);

}

}

Ex.9  
**package** Recursion;

**public** **class** A9NumberOfSteps {

//LeetCode Example

//Number of Steps to Reduce a Number to Zero

//Given an integer num, return the number of steps to reduce it to zero.

//In one step, if the current number is even, you have to divide it by 2, otherwise, you have to subtract 1 from it

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

**int** num=14;

System.***out***.println(*numberOfSteps*(num));//Ans:6

}

**public** **static** **int** numberOfSteps(**int** num) {

**if**(num==0){

**return** 0;

}

**if**(num%2==0){

**return** *numberOfSteps*(num/2) +1; //here +1 means we count the step; while returning the function

}**else**{

**return** *numberOfSteps*(num-1)+1; //here +1 means we count the step;

}

}

}

Ex.10  
**package** RecursionArray;

**public** **class** A1FindArrayisSorted {

//Que. Find the Array is sorted or not using normal and recursive way

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

**int** arr[]= {1,2,6,9,45,55};

//Normal way

**boolean** issorted= *isSortedArr*(arr);

System.***out***.println("Is array sorted: "+issorted);//true

//Recursive way

**boolean** sort= *isArraySorted*(arr,0);

System.***out***.println("Using recursion Is array sorted: "+sort);//true

}

//Normal/Iteration way

**public** **static** **boolean** isSortedArr(**int**[] arr) {

**int** flag=0;

**for**(**int** i=0;i<arr.length-1;i++) {

**if**(arr[i]>arr[i+1]) {

flag++;

**break**;

}

}

**if**(flag==0) {

**return** **true**;

}

**return** **false**;

}

//Using Recursion

**public** **static** **boolean** isArraySorted(**int**[] arr,**int** Index) {

**if**(Index==arr.length-1) { //base case

**return** **true**;

}

**if**(arr[Index]<arr[Index+1]) { //condition

**return** *isArraySorted*(arr,Index+1); //recursive call

}**else** {

**return** **false**;

}

}

}

Ex.11  
**package** RecursionArray;

**import** java.util.ArrayList;

**public** **class** A2LinearSearchRecursion {

//Find the given target/value's index in the array using normal and recursive way

**public** **static** ArrayList<Integer> *list*= **new** ArrayList<>();

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

**int** arr[]= {23,3,56,4,6,6,17};

**int** target=6;

//Normal way

**int** indexOfTarget=*findTarget*(arr,target);

System.***out***.println("target is at index: "+indexOfTarget);//Ans:4

//Recursive way

**int** ind=*findTargetIndex*(arr,target,0);

System.***out***.println("Using recursion Target at Index:"+ind);//Ans:4

//Recursive way find index from last

**int** lastIndex = *findIndexLast*(arr,target,arr.length-1);

System.***out***.println("Using recursion from last index of array, Index: "+lastIndex);//Ans:5

//If more than 1 time target occurs in array then use ArrayList and print them

*twoTargetinArray*(arr,target,0);

System.***out***.println(" Array List: "+*list*);//Ans:[4,5]

//Returning an ArrayList

ArrayList<Integer> list2=**new** ArrayList<>();

list2= *indArrayList*(arr,target,0,list2);

System.***out***.println("Return type ArrayList :"+list2);//Ans:[4,5]

//Arraylist created in each function call

System.***out***.println("ArrayList with local :"+*findIndex2*(arr,target,0));//Ans:[4,5]

}

//Normal way

**public** **static** **int** findTarget(**int**[] arr,**int** target) {

**int** index=-1;

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

**if**(arr[i]==target) {

index=i;

**break**;

}

}

**return** index;

}

//Recursive way

**public** **static** **int** findTargetIndex(**int**[] arr,**int** target,**int** index) {

//base condition

**if**(index==arr.length) {

**return** -1;

}

**if**(arr[index]==target) {

**return** index;

}

**return** *findTargetIndex*(arr,target,index+1);

}

//Recursive from last index of array

**public** **static** **int** findIndexLast(**int**[]arr,**int** target,**int** lastIndex) {

**if**(lastIndex<0) { //base condition

**return** -1;

}

**if**(arr[lastIndex]==target) {

**return** lastIndex;

}

**return** *findIndexLast*(arr,target,lastIndex-1);

}

//Store more than 1 index in ArrayList

**public** **static** **void** twoTargetinArray(**int**[]arr,**int** target,**int** index) {

**if**(index==arr.length) {

**return**;

}

**if**(arr[index]==target) {

*list*.add(index);

}

*twoTargetinArray*(arr,target,index+1);

}

//Returning an ArrayList

**public** **static** ArrayList<Integer> indArrayList(**int**[]arr,**int** target,**int** index,ArrayList<Integer> list2) {//return type is ArrayList<Integer>

**if**(index==arr.length) {

**return** list2; //we returning list

}

**if**(arr[index]==target) {

list2.add(index);

}

**return** *indArrayList*(arr,target,index+1,list2);

}

//Returning an ArrayList without passing the list in function

//Don't use this approach because object created again and again. Use above ArrayList approach

**public** **static** ArrayList<Integer> findIndex2(**int**[]arr,**int** target,**int** index) {

ArrayList<Integer> list3=**new** ArrayList<>(); //Arraylist created in each func call

**if**(index==arr.length) {

**return** list3;

}

**if**(arr[index]==target) {

list3.add(index); //this will contain answer for that call only

}

ArrayList<Integer> ansFromBelowCalls= *findIndex2*(arr,target,index+1);

list3.addAll(ansFromBelowCalls);//new type of recursion

**return** list3;

}

}

Ex.12  
**package** RecursionArray;

**public** **class** A3RotatedBinarySearch {

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

//This works only for Rotated Array.

**int** arr[]= {5,6,7,8,9,1,2,3};

**int** target=2;

**int** start=0;

**int** end=arr.length-1;

System.***out***.println(*findTargetIndex*(arr,target,start,end));//Ans:6

}

**public** **static** **int** findTargetIndex(**int** arr[],**int** target,**int** start,**int** end) {

// int index=-1;

**if**(start>end) {//base case

**return** -1;

}

//

**int** mid=(start+end)/2;

**if**(arr[mid]==target) {

**return** mid;

}

**if**(arr[start]<arr[mid]) {

**if**(target>arr[start] && target<arr[mid]) {

**return** *findTargetIndex*(arr,target,start,mid-1);

}**else** {

**return** *findTargetIndex*(arr,target,mid+1,end);

}

}

**if**(target>arr[mid]&&target<arr[end]) {

**return** *findTargetIndex*(arr,target,mid+1,end);

}

**else** {

**return** *findTargetIndex*(arr,target,start,mid-1);

}

}

}

Ex.13  
**package** recursionPattern;

**public** **class** A1PatternUsingRecursion {

//Que.Print a pattern using recursion

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

//My ans:

**int** row=4;

**int** col=4;

//pattern using recursion

*printPattern*(row,col);

//Kunal ans:

**int** r=3;

**int** c=0;

*priPattern*(r,c);

//Kunal's ans for Ascending \* pattern

*printPatternAsc*(r,c);

}

//My ans:

**public** **static** **void** printPattern(**int** row,**int** col) {

**if**(row==0) {//base case

**return**;

}

**if**(col>0) {

System.***out***.print("\*");

*printPattern*(row,col-1); //this is for printing columns

}**else** {

col=row;

System.***out***.println(); //new line after row

*printPattern*(row-1,col-1); //we change the row and col number

}

//output:

// \*\*\*\*

// \*\*\*

// \*\*

// \*

}

//Kunal Kushwaha Ans :

**public** **static** **void** priPattern(**int** r,**int** c) {

**if**(r==0) {

**return**;

}

**if**(c<r) {

System.***out***.print("\*");

*priPattern*(r,c+1);

}**else** {

System.***out***.println();//new line

*priPattern*(r-1,0);

}

//output:

// \*\*\*

// \*\*

// \*

}

//Kunal's ans for Ascending \* pattern

**public** **static** **void** printPatternAsc(**int** r,**int** c) {

**if**(r==0) {

**return**;

}

**if**(c<r) {

*printPatternAsc*(r,c+1);

System.***out***.print("\*");

}**else** {

*printPatternAsc*(r-1,0);

System.***out***.println();

}

}

//output:

// \*

// \*\*

// \*\*\*

}

Ex.14  
**package** recursionPattern;

**import** java.util.Arrays;

**public** **class** A2BubbleSortUsingRecursion {

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

**int** arr[]= {13,1,37,2,4,8,3};

//Bubble Sort -Normal way

**for**(**int** i=0;i<arr.length-1;i++) {//here we write arr.length-1 because when we sorted then last ele is already sort

**for**(**int** j=0;j<arr.length-1-i;j++) {//here arr.length-1-i in this we use -i because for each iteration of i last

// ele is sorted so we reduce size of arr

**if**(arr[j]>arr[j+1]) { //here we check the max value and swap them

**int** temp=arr[j+1];

arr[j+1]=arr[j];

arr[j]=temp;

}

}

}

**for**(**int** k=0;k<arr.length;k++) {

System.***out***.print(arr[k]+" "); //Ans:1 2 3 4 8 13 37

}

//Recursive way My ans:

**int** i=0;

**int** j=0;

System.***out***.println();

*bubbleSortArr*(arr,i,j);

//Recursive way : Kunal kushWaha ans:

**int** r=arr.length-1;

**int** c=0;

*bubble*(arr,c,r);

System.***out***.println(Arrays.*toString*(arr));//Ans:[1, 2, 3, 4, 8, 13, 37]

}

//Recursive way : My ans:

**public** **static** **void** bubbleSortArr(**int** arr[],**int** i,**int** j){

**if**(i==arr.length-1) { //base condition

**for**(**int** k=0;k<arr.length;k++) {

System.***out***.print(arr[k]+" ");//Ans:1 2 3 4 8 13 37

}

**return**; //return means stop the void type function call

}

**if**(j>arr.length-1-i) {//first iteration here i consider as row and j as column

**if**(arr[j]>arr[j+1]) {

**int** temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

*bubbleSortArr*(arr,i,j+1);

}**else** {//new iteration

*bubbleSortArr*(arr,i+1,0);

}

}

//Recursive way : Kunal kushwaha ans:

**public** **static** **void** bubble(**int** arr[],**int** r,**int** c) {

**if**(r==0) {

**return**;

}

**if**(c<r) {

**if**(arr[c]>arr[c+1]) {

**int** temp=arr[c];

arr[c]=arr[c+1];

arr[c+1]=temp;

}

*bubble*(arr,r,c+1);

}**else** {

*bubble*(arr,r+1,0);

}

}

}

Ex.15  
**package** recursionPattern;

**import** java.util.Arrays;

**public** **class** A3SelectionSortUsingRecursion {

//Selection sort using recursion

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

**int** arr[]= {2,45,5,7,23,3,34};

//In selection sort we place small element at first and then sort according to that from left to right

//Normal way using iteration

**for**(**int** i=0;i<arr.length-1;i++) {

**int** smallest=i;

**int** j;

**for**( j=i+1;j<arr.length;j++) {

**if**(arr[smallest]>arr[j]) {

smallest=j; //Instead of swap here we just assign index to max

}

}

//Swap

**int** temp=arr[i]; //we swap smallest element at start

arr[i]=arr[smallest];

arr[smallest]=temp;

}

System.***out***.println(Arrays.*toString*(arr));//Ans:[2, 3, 5, 7, 23, 34, 45]

//Recursion way : My ans

**int** i=0;

**int** j=i+1;

**int** small=i;

*selectionRecursion*(arr,i,j,small);

System.***out***.println("Recursion way: "+Arrays.*toString*(arr));//Ans:[2, 3, 5, 7, 23, 34, 45]

}

**public** **static** **void** selectionRecursion(**int**[] arr,**int** i, **int** j,**int** small) {

**if**(i==arr.length) {

**return**;

}

**if**(j<arr.length) {

**if**(arr[j]<arr[small]) {

small=j;

}

*selectionRecursion*(arr,i,j+1,small);

}**else** {

**int** temp=arr[small];

arr[small]=arr[i];

arr[i]=temp;

*selectionRecursion*(arr,i+1,i+2,i+1);

}

}

}

Ex.16  
**package** recursionQapnaClg;

**import** java.util.Arrays;

**public** **class** A1MergeSort {

//Merge Sort

//time complexity: O(nlogn);

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

**int** arr[]= {3,4,67,8,1,9};

**int** startIndex=0;

**int** endIndex=arr.length-1;

*divide*(arr,startIndex,endIndex);

System.***out***.println(Arrays.*toString*(arr));//[1, 3, 4, 8, 9, 67]

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

System.***out***.print(arr[i]+" "); //Ans:1 3 4 8 9 67

}

}

**public** **static** **void** divide(**int** arr[],**int** si,**int** ei) {

**if**(si>=ei) {//base case

**return**;

}

**int** mid=si + (ei-si)/2;

*divide*(arr,si,mid);

*divide*(arr,mid+1,ei); //divide is for divide the array into single element

*conquer*(arr,si,mid,ei); //conquer is for combine the array

}

**public** **static** **void** conquer(**int**[] arr,**int** si,**int** mid,**int** ei) {

**int**[] merge = **new** **int**[ei-si+1];

**int** idx1=si; //first index of first array

**int** idx2=mid+1; //first index of second array

**int** x=0; //first index of merge array

//O(n) complexity for conquer part and O(logn) for divide part so combine =O(nlogn)

**while**(idx1<=mid && idx2<=ei) {

**if**(arr[idx1]<arr[idx2]) {

merge[x++]=arr[idx1++]; //here ++ is used for going towards next index

}**else** {

merge[x++]=arr[idx2++];

}

}

//this is for remainig/extra element of an array

**while**(idx1<=mid) {

merge[x++]=arr[idx1++];

}

**while**(idx2<=ei) {

merge[x++]=arr[idx2++];

}

**for**(**int** i=0,j=si;i<merge.length;i++,j++) { //here j=si is important

arr[j]=merge[i];

}

}

}

Ex.17  
**package** recursionQapnaClg;

**public** **class** A2QuickSort {

//Quick Sort for sorting the array using pivot and partition

//time complexity: O(n^2) when last element is smaller or larger

//but good thing is it not take extra space as merge sort for creating new array

**public** **static** **int** partition(**int** arr[],**int** low,**int** high) {

**int** pivot=arr[high]; //here we consider as last element as pivot

**int** i=low-1; //here index is below the array that is -1

**for**(**int** j=low;j<arr.length;j++) {

**if**(arr[j]<pivot) {

i++;

//swap

**int** temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

//for swapping the pivot

i++;

**int** temp=arr[i];

arr[i]=arr[high]; //arr[high] means pivot here we conside last element as pivot

arr[high]=temp;

**return** i; //here i is the index of pivot after partition.

}

**public** **static** **void** quickSort(**int** arr[],**int** low, **int** high) {

**if**(low<high) {

**int** pivot = *partition*(arr,low,high);

*quickSort*(arr,low,pivot-1); //here pivot works as mid

*quickSort*(arr,pivot+1,high);

}

}

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

**int** arr[]= {6,37,1,8,34,13,24};

**int** n=arr.length;

*quickSort*(arr,0,n-1); //low means 0 arr[0]=6 and high means 6 arr[6]=24

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

System.***out***.print(arr[i]+" "); //Ans:1 6 8 13 24 34 37

}

}

}

Ex.18  
**package** recursionSorting;

**import** java.util.Arrays;

**public** **class** A1MergeSort {

//Kunal Kushwaha ans

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

**int** arr[]= {4,13,3,76,7,5,8};

arr=*mergeSort*(arr);

System.***out***.println(Arrays.*toString*(arr));//Ans:[3, 4, 5, 7, 8, 13, 76]

}

**public** **static** **int**[] mergeSort(**int**[] arr) {

**if**(arr.length==1) {//base case

**return** arr;

}

**int**[] left;

**int**[] right;

**int** mid=(arr.length)/2;

left=*mergeSort*(Arrays.*copyOfRange*(arr, 0, mid));//this gives left part

right=*mergeSort*(Arrays.*copyOfRange*(arr,mid,arr.length));//this gives right part

**return** *mix*(left,right);

}

**public** **static** **int**[] mix(**int** first[],**int** second[]) {

**int** i=0; //1st index of first array

**int** j=0; //1st index of second array

**int** k=0; //1st index of new combine array

**int**[] combine=**new** **int**[first.length+second.length];

**while**(i<first.length && j<second.length) {

**if**(first[i]<second[j]) {

combine[k]=first[i];

i++;

}**else** {

combine[k]=second[j];

j++;

}

k++;

}

**while**(i<first.length) {

combine[k]=first[i];

i++; k++;

}

**while**(j<second.length) {

combine[k]=second[j];

j++; k++;

}

**return** combine;

}

}

Ex.19  
**package** recursionString;

**public** **class** A1removeAfromString {

//Question given a string str=baccad remove a from that means newStr=bccd using recursion

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

String str="abaccada";

String newStr="";

String newStr2="";

//Normal way

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

**if**(str.charAt(i)=='a') {

**continue**;

}**else** {

newStr+=str.charAt(i);

}

}

System.***out***.println("New String are: "+newStr); //Ans:bccd

//Recursion way

newStr=*removeAlla*(str,0,"");

System.***out***.println("New String using recursion: "+newStr); //Ans:bccd

//Recursion way passing only one argument in function call

newStr2=*removeAfromStr*(str);//here we pass only one argument that is str

System.***out***.println("Recursion by passing one argument: "+newStr2); //Ans:bccd

}

**public** **static** String removeAlla(String str,**int** index,String newStr) {

**if**(str.length()==index) {

//System.out.println("New string using recursion: "+newStr);

**return** newStr;

}

**if**(str.charAt(index)!='a') {

newStr+=str.charAt(index);

}

**return** *removeAlla*(str,index+1,newStr);

}

**public** **static** String removeAfromStr(String str) {

**if**(str.isEmpty()) { //str.length()==0

**return** "";

}

**if**(str.charAt(0)=='a') {

**return** *removeAfromStr*(str.substring(1));

}**else** {

**return** str.charAt(0)+ *removeAfromStr*(str.substring(1));

}

}

}

Ex.20

**package** recursionString;

**public** **class** A2skipStringFromString {

//Que. Skip a String(apple) from a String bdapplefg using recursion

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

String str="bdapplelfg";

String newStr= *skipStr*(str);

System.***out***.println("String after a skip is :"+newStr);//Ans:bdlfg

//Skip app when not apple : means if appjhle then skip app and when apple then don't skip app

System.***out***.println("String after skip app when not apple is : "+*skipAppNotApple*("dfapphfgbdapple"));

}

**public** **static** String skipStr(String str) {

**if**(str.isEmpty()){

**return** "";

}

**if**(str.startsWith("apple")) {

**return** *skipStr*(str.substring(5));

}**else** {

**return** str.charAt(0) + *skipStr*(str.substring(1));

}

}

**public** **static** String skipAppNotApple(String str) {

**if**(str.isEmpty()){

**return** "";

}

**if**(str.startsWith("app") && !str.startsWith("apple")) { //here when apple then it not skip app

**return** *skipAppNotApple*(str.substring(3));

}**else** {

**return** str.charAt(0) + *skipAppNotApple*(str.substring(1));

}

}

}

Ex.21  
**package** recursionString;

**import** java.util.ArrayList;

**public** **class** A3SubSequenceOfString {

//Que.Find the sub sequences of a string "abc" that is a,b,c,ab,ac,bc,abc

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

String up="abc"; //up means unprocessed string

String p=""; //p means processed string means new string

*subSequence*(p,up);//Normal void recursive way

//return array list way

ArrayList<String> list =**new** ArrayList<>();

list=*subSequenceRet*(p,up,list);

System.***out***.println("Array list return type "+list);//Ans:[abc, ab, ac, a, bc, b, c, ]

//return array list created in function

ArrayList<String> list2= *subSeqRetArrayList*(p,up);

System.***out***.println("Return ArrayList created locally: "+*subSeqRetArrayList*(p,up));//Ans: [abc, ab, ac, a, bc, b, c, ]

}

**public** **static** **void** subSequence(String p,String up) {

**if**(up.isEmpty()) {

System.***out***.println(p);//Ans:abc, ab, ac, a, bc, b, c,

**return**;

}

**char** ch=up.charAt(0);

//to be add

*subSequence*(p+ch, up.substring(1));

//to be not add

*subSequence*(p, up.substring(1));

}

**public** **static** ArrayList<String> subSequenceRet(String p, String up, ArrayList<String> list){

**if**(up.isEmpty()) {

list.add(p);

**return** list;

}

**char** ch=up.charAt(0);

//to be add

*subSequenceRet*(p+ch,up.substring(1),list);

//to be not add

*subSequenceRet*(p,up.substring(1),list);

**return** list;

}

**public** **static** ArrayList<String> subSeqRetArrayList(String p, String up){

**if**(up.isEmpty()) {

ArrayList<String> list=**new** ArrayList<>();

list.add(p);

**return** list;

}

**char** ch=up.charAt(0);

//to be add

ArrayList<String> list1= *subSeqRetArrayList*(p+ch,up.substring(1));

ArrayList<String> list2= *subSeqRetArrayList*(p,up.substring(1));

list1.addAll(list2);

**return** list1;

}

}

Ex.22  
**package** recursionString;

**public** **class** A4ASCIIValueOfChar {

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

String up="abc";

String p="";

*subSeqASCII*(p,up);

}

**public** **static** **void** subSeqASCII(String p, String up) {

**if**(up.isEmpty()) {

System.***out***.println(p);

**return**;

}

**char** ch=up.charAt(0);

//to be add

*subSeqASCII*(p+ch,up.substring(1));

//to be not add

*subSeqASCII*(p,up.substring(1));

//to be add ASCII value of char

*subSeqASCII*(p+(ch+0),up.substring(1));

}

}

Ex.23  
package recursionString;

import java.util.List;

import java.util.ArrayList;

public class A5SubSet {

//Que.SubSet of array using iteration

public static void main(String[] args) {

int arr[]= {1,2,3};

subSet(arr);

System.out.println("SubSet of above array is :"+subSet(arr));

}

public static List<List<Integer>> subSet(int[] arr){ //here List<List<Integer>> is a datatype

List<List<Integer>> outer=new ArrayList<>();

outer.add(new ArrayList<>()); //one empty list is added in outer array

for(int ele:arr) {

int n=outer.size(); //first time n is 1 because we added empty list in outer

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

List<Integer> internal = new ArrayList<>(outer.get(i)); //here outer.get(i) gives a copy and store into internal

internal.add(ele);

outer.add(internal);

}

}

return outer;

}

}

Ex.24  
**package** recursionString;

**import** java.util.\*;

**public** **class** A6SubSetDuplicate {

//Subset duplicate

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

// **TODO** Auto-generated method stub

**int** arr[]= {1,2,2};

List<List<Integer>> ans= *subsetDublicate*(arr);

**for** (List<Integer> list : ans) {

System.***out***.println(list);

}

}

**public** **static** List<List<Integer>> subsetDublicate(**int**[] arr){

Arrays.*sort*(arr);//build in method for Sort the array in ascending order

List<List<Integer>> outer = **new** ArrayList<>();

outer.add(**new** ArrayList<>());

**int** start=0;

**int** end=0;

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

start=0;

//if current and previous element is same then s=e+1;

**if**(i>0 && arr[i]==arr[i-1]) {

start=end +1;

}

end=outer.size()-1;

**int** n=outer.size();

**for**(**int** j=start;j<n;j++) {

List<Integer> inner= **new** ArrayList<>(outer.get(j));

inner.add(arr[i]);

outer.add(inner);

}

}

**return** outer;

}

}

Ex.25  
**package** recursionString;

**import** java.util.\*;

**public** **class** A7SubSetDuplicateKunalsCode {

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

// **TODO** Auto-generated method stub

**int**[] arr = {1, 2, 2};

List<List<Integer>> ans = *subsetDuplicate*(arr);

**for** (List<Integer> list : ans) {

System.***out***.println(list);

}

}

**public** **static** List<List<Integer>> subsetDuplicate(**int**[] arr) {

Arrays.*sort*(arr);

List<List<Integer>> outer = **new** ArrayList<>();

outer.add(**new** ArrayList<>());

**int** start = 0;

**int** end = 0;

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

start = 0;

// if current and previous element is same, s = e + 1

**if** (i > 0 && arr[i] == arr[i-1]) {

start = end + 1;

}

end = outer.size() - 1;

**int** n = outer.size();

**for** (**int** j = start; j < n; j++) {

List<Integer> internal = **new** ArrayList<>(outer.get(j));

internal.add(arr[i]);

outer.add(internal);

}

}

**return** outer;

}

}

Ex.26  
**package** recursionString;

**import** java.util.\*;

**public** **class** A8PermutationsOfabc {

//Que.Find all the permutations of abc that is abc,acb,bac,bca,cab,cba

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

// **TODO** Auto-generated method stub

String up="abc";//here up means unprocessed

String p=""; //here p means processed

*permutations*(p,up);

//returning ans as a ArrayList

System.***out***.println("Permutaions Array List :"+*permutationsList*(p,up));//Ans:[cba, bca, bac, cab, acb, abc]

//returning ans as a count

System.***out***.println("Permutations count: "+*permutationsListCount*(p,up));//Ans:6

}

**public** **static** **void** permutations(String p, String up) {

**if**(up.isEmpty()) {

System.***out***.println(p); //Ans:cba, bca, bac, cab, acb, abc

**return**;

}

**char** ch=up.charAt(0);

//to be add

**for**(**int** i=0; i<=p.length();i++) { //here we use <= because when p is 1 then we need to add 2 times

String first=p.substring(0,i); //here we divide string and then we add ch

String last=p.substring(i,p.length());

*permutations*(first+ch+last,up.substring(1)); //here first+ch+last is imp

}

}

//returning ans as Array list

**public** **static** ArrayList<String> permutationsList(String p, String up){

**if**(up.isEmpty()) {

ArrayList<String> list=**new** ArrayList<>();

list.add(p);

**return** list;

}

**char** ch=up.charAt(0);

ArrayList<String> list2=**new** ArrayList<>();

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

String first=p.substring(0,i);

String last=p.substring(i,p.length());

list2.addAll(*permutationsList*(first+ch+last,up.substring(1)));

}

**return** list2;

}

//returning ans as count

**public** **static** **int** permutationsListCount(String p, String up){

**if**(up.isEmpty()) {

ArrayList<String> list=**new** ArrayList<>();

list.add(p);

**return** 1;

}

**char** ch=up.charAt(0);

**int** count=0;

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

String first=p.substring(0,i);

String last=p.substring(i,p.length());

count = count + *permutationsListCount*(first+ch+last,up.substring(1));

}

**return** count;

}

}

Ex.27  
**package** recursionString;

**import** java.util.\*;

//Que.Letter combinations of phone number where 2 means abc , 3 means def

//Always use Kunal Kushwaha (p,up) trick for solving the problems and first solve using pen and paper

**public** **class** A9LetterCombinationOfPhoneNo {

**static** String[] *arr*= {"."," ","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

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

String str="23";

ArrayList<String> ans = *lettersCombinations*(str);

System.***out***.println("Combinations are: "+ans);//Ans: [ad, ae, af, bd, be, bf, cd, ce, cf]

}

**public** **static** ArrayList<String> lettersCombinations(String digits){

**return** *lettersComb*("",digits);

}

**public** **static** ArrayList<String> lettersComb(String p,String up){

**if**(up.isEmpty()) {

ArrayList<String> list=**new** ArrayList<>();

list.add(p);

**return** list;

}

**char** ch= up.charAt(0);

String str = *arr*[ch-'0'];//here ch-'0' means substracting char from char gives integer this convert '2' into 2

ArrayList<String> list2 = **new** ArrayList<>();

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

list2.addAll(*lettersComb*(p+str.charAt(i),up.substring(1)));

}

**return** list2;

}

}

Ex.28  
**package** recursionString;

**import** java.util.\*;

**public** **class** B1DiceThrow {

//we know dice=[1,2,3,4,5,6]

//Que.Find the maximum combination to form target in dice like for target 4 = (112),(121),(22),(31),(4),(211),(1111),(13)

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

**int** target=4;

String p="";//here p means processed and target means unprocessed

//printing the ans

*dice*(p,target);

//returning the ans through ArrayList

ArrayList<String> list = **new** ArrayList<>();

list=*diceRet*(p,target);

System.***out***.println("Returning type ArrayList: "+list);//Ans:[1111, 112, 121, 13, 211, 22, 31, 4]

//Count no of ans and return

System.***out***.println("Count Ans: "+*diceRetCount*(p,target));//Ans:8

}

//printing the ans

**public** **static** **void** dice(String p, **int** target) {

**if**(target==0) {

System.***out***.print(p+" ");//Ans:1111 112 121 13 211 22 31 4

**return**;

}

**for**(**int** i=1;i<=target&&i<=6;i++) {

*dice*(p+i,target-i);

}

}

//Returning the ans

**public** **static** ArrayList<String> diceRet(String p, **int** target){

**if**(target==0) {

ArrayList<String> list2=**new** ArrayList<>();

list2.add(p);

**return** list2;

}

ArrayList<String> list3=**new** ArrayList<>();

**for**(**int** i=1;i<=target&&i<=6;i++) { //here i=1 because dice starts from 1 and to 6.

list3.addAll(*diceRet*(p+i,target-i));

}

**return** list3;

}

//Returning the no of ans

**public** **static** **int** diceRetCount(String p, **int** target) {

**if**(target==0) {

**return** 1;

}

**int** count =0;

**for**(**int** i=1;i<=target&&i<=6;i++) {

count = count + *diceRetCount*(p+i,target-i);

}

**return** count;

}

}

//BackTracking start

Ex.29  
**package** Y1BackTracking;

**public** **class** A1MazeProblem {

//Note: Always follow the diagram given by the Kunal kushwaha before solving the problem or draw the diagram using pen and paper

//Que. Print the possible ways to reach towards the destination(1,1) from (3,3). we can move only right and down direction

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

**int** ans = *count*(3,3); //here first 3 means row and second three means column

System.***out***.println("Total no of ways from (3,3) to (1,1) is "+ans);

}

**public** **static** **int** count(**int** row, **int** col) {

**if**(row==1 || col==1) { //base condition when col or row become 1 then only path to return is 1

**return** 1;

}

**int** left=*count*(row-1,col); //when we move towards downward direction

**int** right=*count*(row,col-1); //when we move towards right direction

**return** left+right;

}

}

Ex.30

**package** Y1BackTracking;

**import** java.util.ArrayList;

**public** **class** A2MazeProblemPathprint {

//Que. Print the path of maze problem to reach (1,1) from (3,3)

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

//before solving the problem always draw a problem diagram using pen and paper as drawn by kunal kushwaha

//By seeing the diagram try to code

*countPathPrint*("",3,3); //here "" consider as processed(p) and 3,3 as unprocessed (up) as used in permutations example

//Return ans using ArrayList

ArrayList<String> list = **new** ArrayList<>();

*pathReturn*("",3,3,list);

System.***out***.println("Ans in ArrayList:"+list);//Ans: [DDRR, DRDR, DRRD, RDDR, RDRD, RRDD]

//If allow to move diagonally also means 3 directions right,down,diagonally

System.***out***.println("When allow to move Diagonally also :"+*pathDiagonally*("",3,3));

//Ans: [DDRR, DRDR, DRRD, DRX, DXR, RDDR, RDRD, RDX, RRDD, RXD, XDR, XRD, XX]

}

**public** **static** **void** countPathPrint(String p,**int** row,**int** col) {

**if**(row==1 && col==1) { //Base condition

System.***out***.print(p+ " ");// Ans: DDRR DRDR DRRD RDDR RDRD RRDD

**return**;

}

//when we move towards down

**if**(row!=1) {

*countPathPrint*(p+"D",row-1,col);

}

//when we move towards right

**if**(col!=1) {

*countPathPrint*(p+"R",row,col-1);

}

}

//Return ans using ArrayList

**public** **static** ArrayList<String> pathReturn(String p, **int** row, **int** col, ArrayList<String> list){ //here we pass list as a argument/parameter

**if**(row==1 && col==1) { //here when row and col is 1 means we reach the destination then that path we want

list.add(p);

**return** list;

}

**if**(row>1) {

*pathReturn*(p+"D",row-1,col,list); //here D means Downward direction

}

**if**(col>1) {

*pathReturn*(p+"R",row,col-1,list); //here R means Right direction

}

**return** list;

}

//if allow to move diagonally also use pen and paper with diagram for understanding

**public** **static** ArrayList<String> pathDiagonally(String p,**int** row,**int** col){

**if**(row==1 && col==1) {

ArrayList<String> list=**new** ArrayList<>();

list.add(p);

**return** list;

}

ArrayList<String> list2=**new** ArrayList<>();

**if**(row>1) {

list2.addAll(*pathDiagonally*(p+"D",row-1,col));//here D means Down

}

**if**(col>1) {

list2.addAll(*pathDiagonally*(p+"R",row,col-1)); //here R means Right

}

**if**(row>1 && col>1) {

list2.addAll(*pathDiagonally*(p+"X",row-1,col-1)); //here X means Diagonally

}

**return** list2;

}

}

Ex.31  
**package** Y1BackTracking;

**import** java.util.\*;

**public** **class** A3MazeProblemWithRestriction {

//Que. Display possible ways when middle part is restricted

//here we move from (0,0) till the (maze.length-1, maze.length-1) here maze is 3 into 3 2-D Array

//In this example we move from (0,0) of Array here 1st element is (0,0); and destination is (2,2)

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

**boolean** maze[][] = {{**true**,**true**,**true**},

{**true**,**false**,**true**},

{**true**,**true**,**true**}};

ArrayList<String> list=**new** ArrayList<>();

list= *pathRestrictions*("",maze,0,0);

System.***out***.println("Ans: "+list);//Ans:[VVHH, HHVV]

}

**public** **static** ArrayList<String> pathRestrictions(String p,**boolean** maze[][],**int** row,**int** col){

**if**(row==maze.length-1 && col==maze[0].length-1) { //base case

ArrayList<String> list2=**new** ArrayList<>(); //this created for each solution path

list2.add(p);

**return** list2;

}

**if**(!maze[row][col]) { //this is for restriction purpose

ArrayList<String> list4=**new** ArrayList<>();

**return** list4; //here we just returning back that is just for syntax purpose we returning list4

}

ArrayList<String> list3=**new** ArrayList<>();

**if**(row<maze.length-1) {

list3.addAll( *pathRestrictions*(p+"V",maze,row+1,col)); //here V means Vertical

}

**if**(col<maze[0].length-1) {

list3.addAll(*pathRestrictions*(p+"H",maze,row,col+1)); //here H means Horizontal

}

**return** list3;

}

}

Ex.32  
**package** Y1BackTracking;

**public** **class** A4BackTrackingEx1 {

//BAckTracking means make a change and reverse that change when that process is finished

//BackTracking Example

//Note: Always follow the diagram using pen and paper

//here we move in all 4 direction that is Down(D), Right(R), Up(U), Left(L)

//while moving we make cell as False

//while coming back means when function execute then make that cell as true

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

**boolean** maze[][]= {{**true**,**true**,**true**},

{**true**,**true**,**true**},

{**true**,**true**,**true**}}; //Initially all the cells are true

*backTrackingEx*("",maze,0,0);

}

**public** **static** **void** backTrackingEx(String p,**boolean** maze[][],**int** row,**int** col) {

**if**(row==maze.length-1 && col==maze[0].length-1) {

System.***out***.print(p+" ");//Ans: DDRR DDRURD DDRUURDD DRDR DRRD DRURDD RDDR RDRD RDLDRR RRDD RRDLDR RRDLLDRR

**return**;

}

**if**(!maze[row][col]) { //this return when cell of Matrix is false

**return**; //this not allow to move towards the false cell

}

//while moving we make cell as false

// i am considering this block in my path

maze[row][col]=**false**;

//move towards down

**if**(row<maze.length-1) { //see the matrix diagram it allows till the rows of array which starts from 0

*backTrackingEx*(p+"D",maze,row+1,col);

}

//move towards right

**if**(col<maze[0].length-1) {

*backTrackingEx*(p+"R",maze,row,col+1);

}

//move towards Up

**if**(row>0) {

*backTrackingEx*(p+"U",maze,row-1,col);

}

//move towards Left

**if**(col>0) {

*backTrackingEx*(p+"L",maze,row,col-1);

}

//While returning back we make the cells as true

// this line is where the function will be over

// so before the function gets removed, also remove the changes that were made by that function

maze[row][col]=**true**; //this is a backtracking

}

}

Ex.33  
**package** Y1BackTracking;

**import** java.util.Arrays;

**public** **class** A5BackTrackingPrintMatrixAndPath {

//Print the Matrix of path from 1 to that number to achieve goal and also print path

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

**boolean** maze[][] = {{**true**,**true**,**true**},

{**true**,**true**,**true**},

{**true**,**true**,**true**}};

**int**[][] path=**new** **int**[maze.length][maze[0].length];

*printPathAll*("",maze,0,0,path,1);

}

**public** **static** **void** printPathAll(String p,**boolean**[][]maze, **int** row, **int** col, **int**[][]path,**int** step) {

**if**(row==maze.length-1 && col==maze.length-1) { //base case

path[row][col]=step;//last cell of matrix also required for printing the last element

**for**(**int**[]arr:path) {

System.***out***.println(Arrays.*toString*(arr));//to printing the row of 2D Array

}

System.***out***.println(p+" ");

**return**;

}

**if**(!maze[row][col]) {

**return**;

}

maze[row][col]=**false**;

path[row][col]=step;

**if**(row<maze.length-1) {

*printPathAll*(p+"D",maze,row+1,col,path,step+1);

}

**if**(col<maze[0].length-1) {

*printPathAll*(p+"R",maze,row,col+1,path,step+1);

}

**if**(row>0) {

*printPathAll*(p+"U",maze,row-1,col,path,step+1);

}

**if**(col>0) {

*printPathAll*(p+"L",maze,row,col-1,path,step+1);

}

//here while moving towards back we removes the assigned value

maze[row][col]=**true**;

path[row][col]=0;

}

}

Ex.34  
**package** Y1BackTracking;

//N-Queens problem of BackTracking

**public** **class** A6NQueens {

//follow the diagram while solving

//Time complexity=O(n!);

//Space complexity:O(n\*n);

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

**boolean** board[][] =**new** **boolean**[4][4];

System.***out***.println(*queens*(board,0)); //here 0 is row

}

**public** **static** **int** queens(**boolean**[][] board,**int** row) {

**if**(row==board.length) {//base case

*display*(board);

**return** 1;

}

**int** count=0;

//placing the queen and checking for every row and column

**for**(**int** col=0;col<board[0].length;col++) {

//place the queen if it is safe

**if**(*isSafe*(board,col,row)) {

board[row][col]=**true**;

count+=*queens*(board,row+1);

board[row][col]=**false**; //Backtracking

}

}

**return** count;

}

**public** **static** **boolean** isSafe(**boolean**[][] board,**int** col,**int** row) {

//check vertically

**for**(**int** i=0; i<row;i++) {

**if**(board[i][col]) {

**return** **false**;

}

}

//diagonal left

**for**(**int** i=1;i<=Math.*min*(row, col);i++) {

**if**(board[row-i][col-i]) {

**return** **false**;

}

}

//Diagonal right

**for**(**int** i=1;i<=Math.*min*(row, board.length-col-1);i++) {

**if**(board[row-i][col+i]) {

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** display(**boolean**[][] board) {

**for**(**boolean** row[]: board) {

**for**(**boolean** ele:row) {

**if**(ele==**true**) {

System.***out***.print("Q ");

}**else** {

System.***out***.print("X ");

}

}

System.***out***.println();

}

System.***out***.println();

}

}

Ex.35  
**package** Y1BackTracking;

**public** **class** A7NQueensPractice {

//N-Queens practice

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

**boolean**[][] board = **new** **boolean**[5][5];

**int** row=0;

**int** ans = *nQueens*(board,row);

System.***out***.println("No of Count: "+ans);

}

**public** **static** **int** nQueens(**boolean**[][] board, **int** row) {

**if**(row==board.length) {

*display*(board); //this fun we call for just printing the matrix

**return** 1; //return 1 is for calculating the no of ans that is count

}

**int** count=0;

**for**(**int** col=0;col<board[0].length;col++) {

**if**(*isSafe*(board,row,col)) {

board[row][col]=**true**;

count += *nQueens*(board,row+1);

board[row][col]=**false**;

}

}

**return** count;

}

**public** **static** **void** display(**boolean**[][] board) {

**for**(**boolean**[] row:board) {

**for**(**boolean** ele:row) {

**if**(ele==**true**) {

System.***out***.print("Q ");

}**else** {

System.***out***.print("x ");

}

}

System.***out***.println();

}

System.***out***.println();

}

**public** **static** **boolean** isSafe(**boolean**[][] board,**int** row,**int** col) {

//check vertically that is there any Queen is present or not

**for**(**int** i=0;i<row;i++) {

**if**(board[i][col]) {

**return** **false**;

}

}

//check upper left diagonally

**for**(**int** i=1;i<=Math.*min*(row, col);i++) {

**if**(board[row-i][col-i]) {

**return** **false**;

}

}

//check upper right diagonally

**for**(**int** i=1;i<=Math.*min*(row, board[0].length-col-1);i++) {

**if**(board[row-i][col+i]) {

**return** **false**;

}

}

**return** **true**;

}

}

Ex.36  
**package** Y1BackTracking;

**public** **class** A8KNights {

//here we have to place 4 horses in chess board of 4\*4 size successfully so that each horse not cut to each other

//Note: Always use diagram or draw diagram using pen and paper while solving the problem

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

**int** n=4;

**int** row=0;

**int** col=0;

**int** target=4;

**boolean**[][] board=**new** **boolean**[n][n];

*kNights*(board,row,col,target);

}

**public** **static** **void** kNights(**boolean**[][] board,**int** row,**int** col,**int** target) {

//base condition

**if**(target==0) {

*display*(board);

**return**;

}

**if**(row==board.length-1 && col==board.length) {

**return**;

}

**if**(col==board[0].length) {

*kNights*(board,row+1,0,target);

**return**;

}

**if**(*isSafe*(board,row,col)) {

board[row][col]=**true**;

*kNights*(board,row,col+1,target-1);

board[row][col]=**false**;

}

*kNights*(board,row,col+1,target);

// kNights(board,row+1,0,target);

// board[row][col]=false;

// target+=1;

}

**public** **static** **boolean** isSafe(**boolean**[][] board,**int** row,**int** col) {

**if**(row>=1 && col<board[0].length-2) {

**if**(board[row-1][col+2]) {

**return** **false**;

}

}

**if**(row>=1 && col>=2) {

**if**(board[row-1][col-2]) {

**return** **false**;

}

}

**if**(row>=2 && col<board[0].length-1) {

**if**(board[row-2][col+1]) {

**return** **false**;

}

}

**if**(row>=2 && col>=1){

**if**(board[row-2][col-1]) {

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** display(**boolean**[][] board) {

**for**(**boolean**[] row: board) {

**for**(**boolean** ele: row) {

**if**(ele) {

System.***out***.print("K ");

}**else** {

System.***out***.print(". ");

}

}

System.***out***.println();

}

System.***out***.println();

}

}

Ex.37  
**package** Y1BackTracking;

//Sudoko solver

//Always draw the diagram using pen and paper then try to solve

**public** **class** A9SudokoSolver {

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

**int**[][] board = **new** **int**[][] {

{3, 0, 6, 5, 0, 8, 4, 0, 0},

{5, 2, 0, 0, 0, 0, 0, 0, 0},

{0, 8, 7, 0, 0, 0, 0, 3, 1},

{0, 0, 3, 0, 1, 0, 0, 8, 0},

{9, 0, 0, 8, 6, 3, 0, 0, 5},

{0, 5, 0, 0, 9, 0, 6, 0, 0},

{1, 3, 0, 0, 0, 0, 2, 5, 0},

{0, 0, 0, 0, 0, 0, 0, 7, 4},

{0, 0, 5, 2, 0, 6, 3, 0, 0}

};

**if**(*sudoko*(board)) {

*display*(board);

}

System.***out***.println();

System.***out***.print("Ans: "+*sudoko*(board));

}

**public** **static** **boolean** sudoko(**int**[][] board) {

**int** n=board.length;

**int** row=-1;

**int** col=-1;

**boolean** isEmptyLeftSide=**true**;

//we travel from each ele of 2D Array

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

**for**(**int** j=0;j<board[0].length;j++) {

**if**(board[i][j]==0) {

row=i;

col=j;

isEmptyLeftSide=**false**;

**break**;

}

}

**if**(isEmptyLeftSide==**false**) {

**break**;

}

}

**if**(isEmptyLeftSide==**true**) {

**return** **true**; //Sudoko is solved

}

//through this we try to place value where we found zero

**for**(**int** num=1;num<=9;num++) {

**if**(*isSafe*(board,row,col,num)) {

board[row][col]=num;

**if**(*sudoko*(board)) {

**return** **true**;

}**else** {

//BackTracking

board[row][col]=0;

}

}

}

**return** **false**;

}

**public** **static** **boolean** isSafe(**int**[][] board,**int** row ,**int** col, **int** num) {

//checking row

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

**if**(board[row][i]==num) {

**return** **false**;

}

}

//checking column

**for**(**int**[] rowArr : board) {

**if**(rowArr[col]==num) {

**return** **false**;

}

}

//checking grid matrix imp

**int** sqrt=(**int**)Math.*sqrt*(board.length);

**int** startRow= row - row % sqrt;

**int** startCol= col- col % sqrt;

**for**(**int** i=startRow;i<startRow+sqrt;i++) {

**for**(**int** j=startCol;j<startCol+sqrt;j++) {

**if**(board[i][j]==num) {

**return** **false**;

}

}

}

**return** **true**;

}

**public** **static** **void** display(**int**[][] board) {

**for**(**int**[] rowArr: board) {

**for**(**int** ele:rowArr) {

System.***out***.print(ele+" ");

}

System.***out***.println();

}

} }