Recursion In Arrays

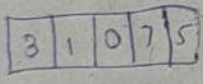
1. Display Array

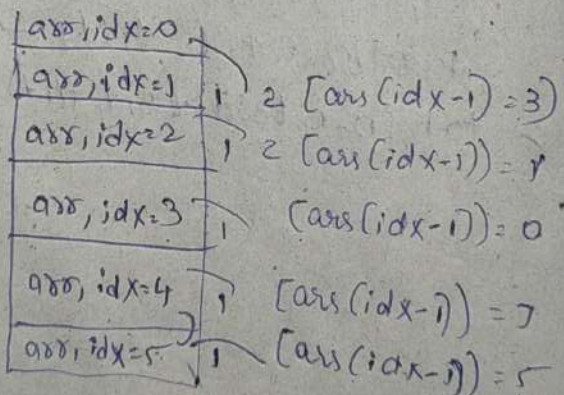
Input:	Output:	Constraints:
5	3	1 <= n <= 30
3	1	0 <= n1, n2.. n elements <= 1
1	0	
0	7	
7	5	
5		

Steps:

1. Take the inputs as required.
2. We will call the (idx-1) of function Assuming it has true till it reaches the base condition
3. Then Print the values

Dry Run:





Programm:

```
displayArr(int[]arr, int idx) {
if(idx) == 0
    return;
displayArr(arr, idx-1)        ——— 1
printInfo(arr(idx-1))         ——— 2
}
```

2. Display Array In Reverse

Input	Output	Constraints
5	5	1 <= n <= 30
3	7	0 <= n1, n2 ... nelements <= 10
1	0	
0	1	
7	3	
5		

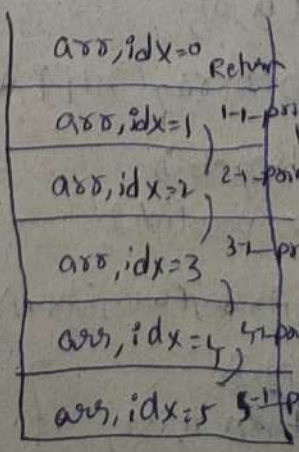
Steps:

1. Take the inputs as required.
2. We will print arr(idx-1) and call the recursive method, till the base condition idx==0

Program:

```
public static void displayArrReverse(int[] arr, int idx){
    if(idx==0)
        return;
    else
    println(arr[idx-1]); — (1)
    displayArrReverse(arr, idx-1); — (2)
}
}
```

Dry Run:



3. Max of an Array

Input

6
15
30
40
4
11
9

Output

40

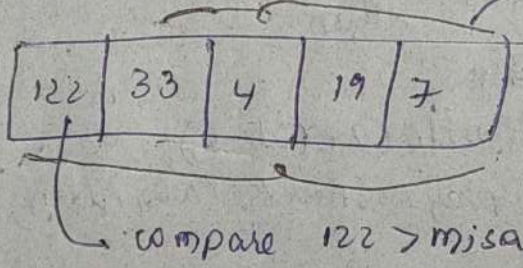
Constraints

$1 <= n <= 10^4$

$0 <= n_1, n_2 \ldots n$ elements $<= 10^9$

Steps:

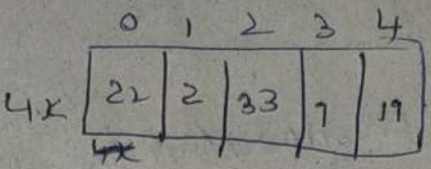
1. Pass arr, index=0 to the max function.
2. Expectation, max(arr, 0) → gives the Highest Element
3. Faith: max(arr, i) → to end gives the highest element and then compare with idx, which returns the maximum element.

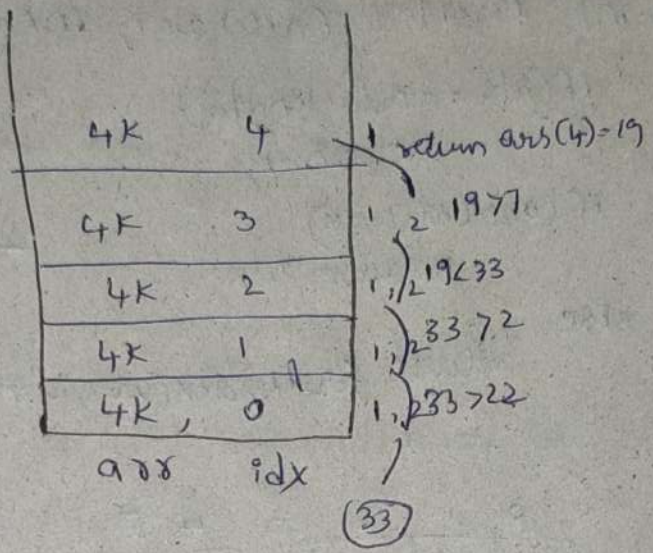


Program:

```
maxOfArray(a, 0);

public static int maxOfArray(int[] arr, int idx) {
    if(idx == arr.length-1)
        return arr[idx];
    int misa = maxOfArray(arr, idx+1);  ——— (1)
    if (misa > arr[idx])
        return misa;                    } ——— (2)
    else
        return arr[idx];
}
```

Dry Run:





4 First Index

Input

6 - n

15
11
40
4
4
9
elements

4 - x

Output

3

Constraints

1 <= n <= 10^4

0 <= n1, n2 ... nelements <= 10^3

0 <= x <= 10^3

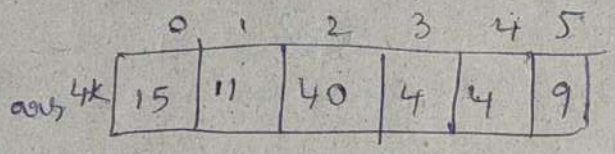
Steps:

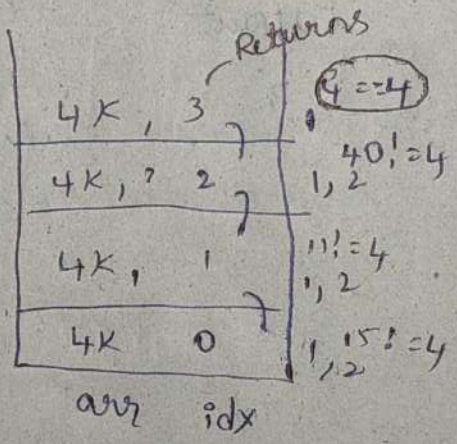
1. Pass arr, idx=0, x
2. If arr[idx]==x, then return idx
3. else Return, next element.
4. If it reaches end of Array and x not found, return -1.

Program:

```
firstIndex(a,0,x);

Public static int firstIndex(int[] arr, int idx, int x) {
    if(idx == arr.length)
        return -1;
    if (arr[idx] == x)                    — (1)
        return idx;
    else
        return firstIndex(arr, idx+1, x); — (2)
}
```

DryRun:





5. Last Index:

Input	Output	Constraints
6 — n 15 11 40 4 4 9 } elements 4 — x	4	1 <= n <= 10^4 0 <= n1, n2 ... nelements <= 10^3 0 <= x <= 10^3

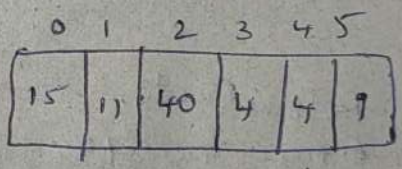
Steps:

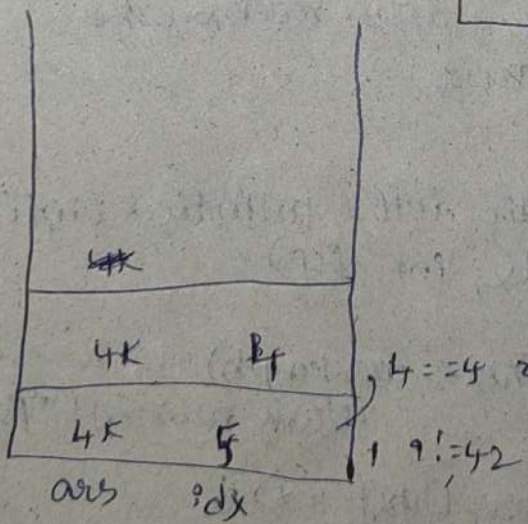
1) Pass arr, idx-1, x
2) Check if arr[idx]==x else do idx-1 and check
3) If it reaches the first element not found return -1;

Program:

```
lastIndex(a, n-1, x);

public static int lastIndex(int[] arr, int idx, int x) {
    if (idx==0)
        return -1;
    if (arr[idx]==x)                    ——(1)
        return idx;
    else
        return lastIndex(arr, idx-1, x);    ——(2)
}
```

Dry Run:





6. All Indices of Array

Sample Input

6 — n
15
11
40
4
4
9
4 — x

Sample Output

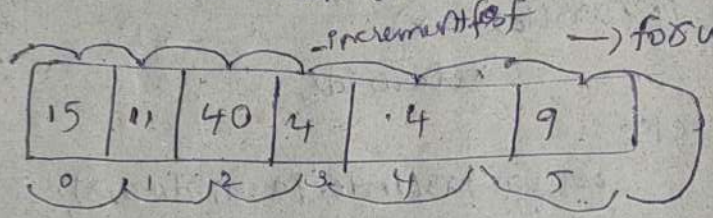
3
4

Constraints

$1 <= n <= 10^4$
$0 <= n_1, n_2 \ldots n_{elements} <= 10^3$
$0 <= x <= 10^3$

Steps;

1. fsf -> indicates the total number of 'x' element got repeated in Array and idx represents index of.

2. 

create array of fsf size.

| 3 | 4 |

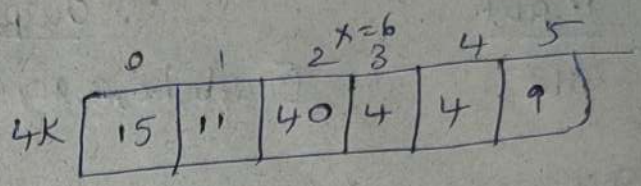
↓ return this Array

3. ~~Return~~ ~~Create~~ Create the array after reaching the end of elements.

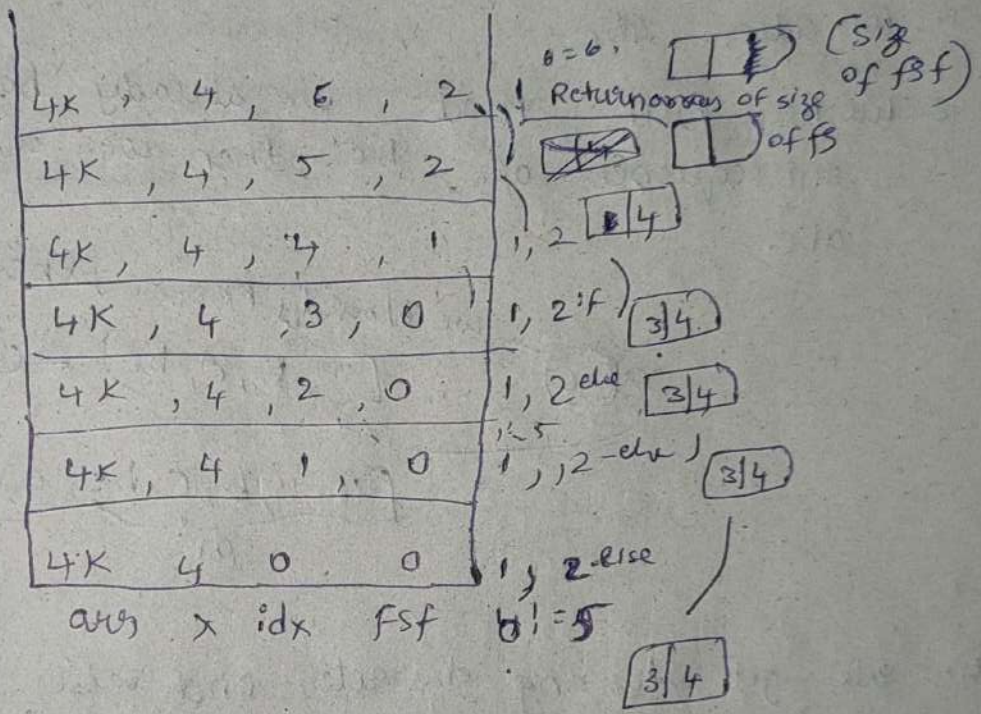
Program;

```
public static int[] allIndices(int[] arr, int x,
    int idx, int fsf)
{
    if (idx == arr.length)
        return new int[fsf];                    — (1)

    if (arr[idx] == x) {
        int[] ia = allIndices(arr, x, idx+1,
                              fsf+1);
        ia[fsf] = idx;
        return ia;
    }                                           — (2)
}
```

```
else {
    int[] ba = allIndices(ars, x, idx+1, fsf);
    return ba;
}
}
```



Dry Run:



Recursion With Array List

1. Get Subsequence

Input

abc

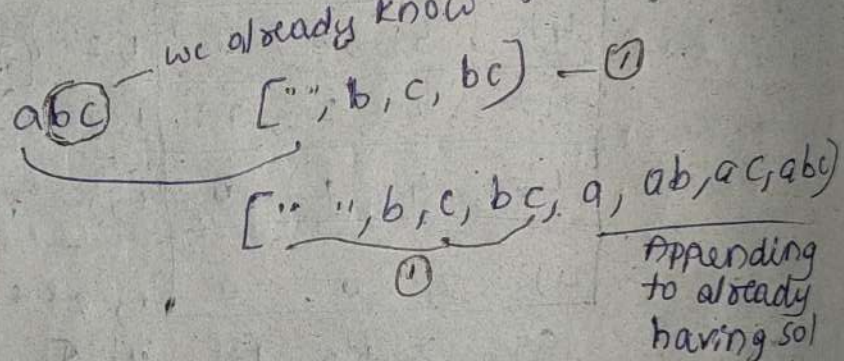
Output

[,c,b,bc,a,ac,ab,abc]

Constraints

0<=str.length<=

Steps:

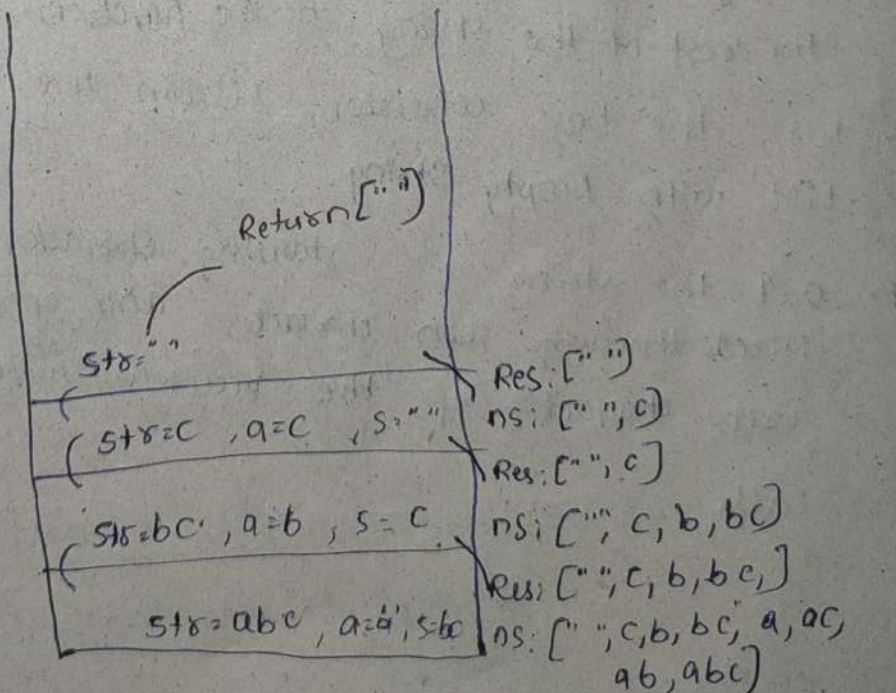
1. Consider abc
2. We keep faith that we already have subsequences of bc then we will find abc.
3. 

4. We get starting character and will pass the rest of string to the function
5. For the base condition, return the Array List with empty string.
6) For the new Array List, append the result and also concatenate the character with the result.

Program:

```
public static ArrayList<String> gss(String str) {
    if(str.length() == 0) {
        ArrayList<String> bs = new ArrayList<String>();
        bs.add("");
        return bs;
    }
    char a = str.charAt(0);
    String s = str.substring(1); ArrayList<String> result=gss(s);
ArrayList<String> ns=new ArrayList<String>();
    for(String i : result)
            ns.add(i);
    for(String i : result)
            ns.add(a+i);
    return ns;
}
```

Dry Run:



2. Get KPC

Input

78

Output

[tv, tw, tx, uv, uw, ux)

Constraints

0 <= Str.length <= 10

Str contains numbers only

Steps:

1. Store all the number keys in String Array
2. Consider 789
3. We keep faith that we already know the key values of 89, from that 789 can be computed.
4. Already know [vy, vz, wy, wz, xy, xz)

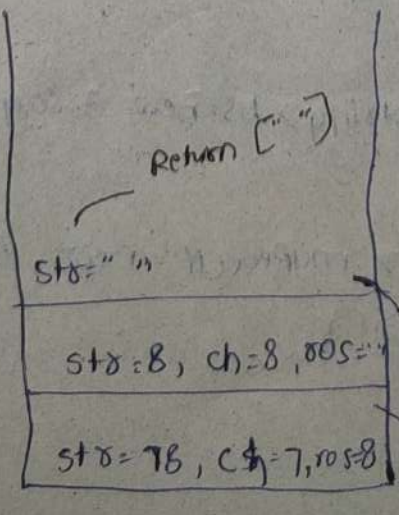
[tvy, tvz, twy, twz, txy, txz, uvy, uvz, uwy, uwz, uxy, uxz)

5. We get the starting character and will pass the rest of the string to the function.
6. For, the Base condition, return the Array List with Empty string.
7. Get the string of starting character and iterate through each character and append each character of the recursive function call.

Program:

```
static String[] codes = {".;", "abc", "def", "ghi", "jkl",
    "mno", "pqrs", "tu", "vwx", "yz"};

public static ArrayList<String> getKPC(String str) {
    if (str.length() == 0) {
        ArrayList<String> base = new ArrayList<String>();
        base.add("");
        return base;
    }
    char ch = str.charAt(0);
    String ros = str.substring(1);
    ArrayList<String> myres = getKPC(ros);
    ArrayList<String> res = new ArrayList<String>();
    String temp = codes[Character.getNumericValue(ch)];
    for (int i = 0; i < temp.length(); i++) {
        char tt = temp.charAt(i);
        for (String j : myres) {
            res.add(tt + j);
        }
    }
    return res;
}
```

Dry Run:



3. Get Stair Paths

Input:	Output	Constraints
3	[1,1,1, 1,2, 2,1, 3]	0 <= n <= 10

Steps:

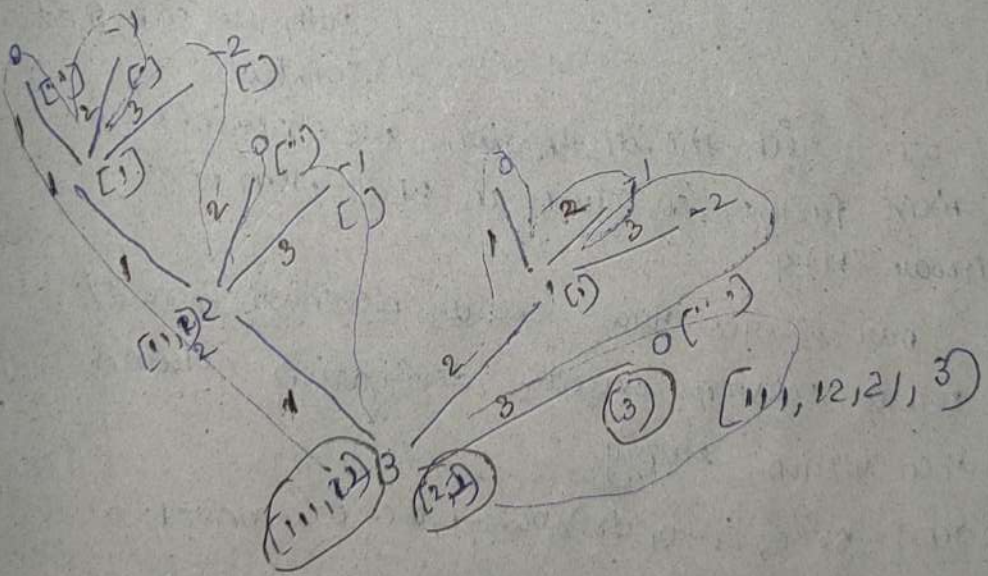
1. As per the statement, either 1 step / 2 steps / 3 steps at a time possible
2. We make three recursive calls for (n-1), (n-2), (n-3) and we will get the respect and append 1 step, 2 step, 3 step for respective recursive calls.
3. If n==0, no more steps possible, so a blank list is returned
4. If n<0, return Empty List

Program:

```
public static ArrayList<String> getStairPaths(int n){
    if(n==0){
        ArrayList<String> bs = new ArrayList<String>();
        bs.add("");
        return bs;
    }
    else if(n<0){
        ArrayList<String> bs = new ArrayList<String>();
    }
    ArrayList<String> myResult = new ArrayList
        <String>();
```

```
ArrayList<String> rp1 = getStairPaths(n-1); -①
for(String i: rp1)
  myResult.add("1" + i);
ArrayList<String> rp2 = getStairPaths(n-2); -②
for(String i: rp2)
  myResult.add("2"+i);
ArrayList<String> rp3 = getStairPaths(n-3); -③
for(String i: rp3)
  myResult.add("3"+i);
    return myResult;
}
```

Dry Run;



4. Get Maze Paths

Input

3 → rows
3 → cols

Output

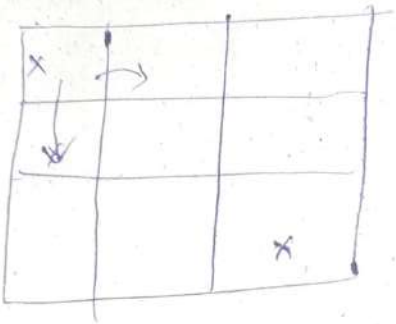
[hhvv, hvhv, hvvh, vhhv, vhvh, vvhh]

Constraints

$0 <= n <= 10$
$0 <= m <= 10$

Steps:

1. High level thinking, we assume we take one horizontal step and keep faith from remaining path we have solution



Similarly, if we take one vertical step and keep from remaining path we can find solution.

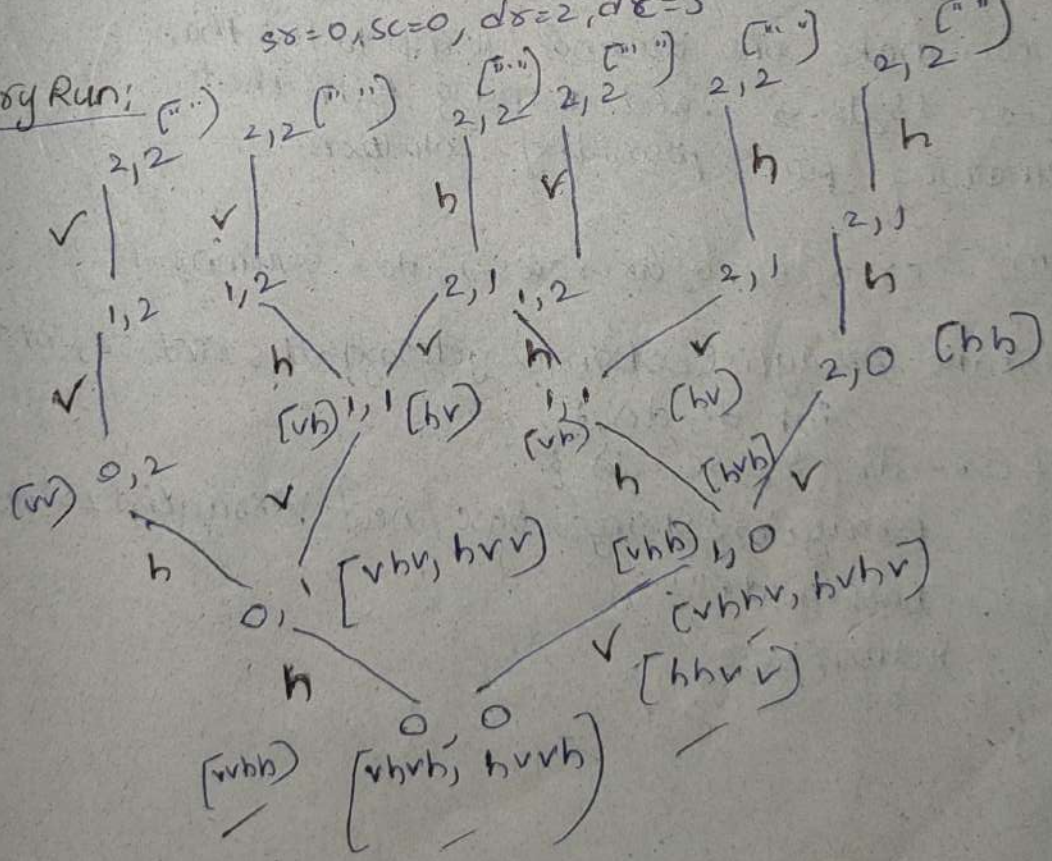
2. If we reach the destination, we no need to move further, so return an blank item of Array String
3. If our source row exceeds destination row/ source column exceeds destination column then return empty

Program: sr=0, sc=0, dr=rows-1, dc= columns-1

```
public static ArrayList<String> getMazePaths(int sr,
    int sc, int dr, int dc) {
        if(sr>dr || sc>dc){
            return new ArrayList<String>();
        }
}
```

```
if(sr==dr && sc==dc){
    ArrayList<String> base=new ArrayList<String>
        ();
    base.add("");
    return base;
    }

ArrayList<String> myResult=.new ArrayList<String>();
ArrayList<String> hr=getMazePaths(sr,sc+1,dr,dc);   [going horizontally] [h]
for(String i: hr)
        myResult.add("h"+i);

ArrayList<String> vc=getMazePaths(sr+1,sc,dr,dc);   [v]
                                  [going vertically]
for(String i: vc)
        myResult.add("v"+i);

return myResult;
}
```

Dry Run; sr=0, sc=0, dr=2, dc=3



5. Get Maze Path with Jumps

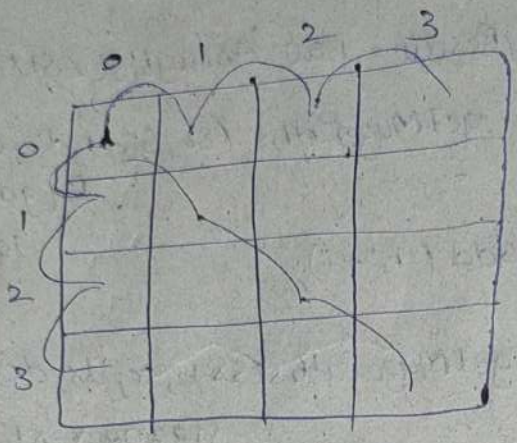
Input:
2 - rows
2 - columns

Output
[hi v1, v1 h1, d1]

Constraints
$0 <= n <= 10$
$0 <= m <= 10$

Steps:

1)



With High level understanding
We can make one horizontal jump to less than column no of jumps and keeping faith with remaining (we have solution) to end point.

2) We can make one vertical jump to less than number of rows and keeping faith that remaining path providing solution.

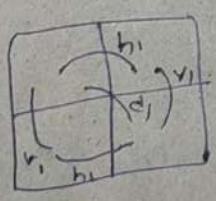
3) We can make one diagnol jump to less than $dc - sc$ && $dr - sr$ and keeping faith that remaining path providing solution.

Program: sr = 0, sc = 0, dr = rows - 1, dc = columns - 1

```
public static ArrayList<String> getMazePaths(int sr, int sc,
                                    int dr, int dc) {
    if (sr == dr && sc == dc) {
        ArrayList<String> base = new ArrayList<String>();
        base.add("");
        return base;
    }
}
```

```
// for horizontal jumps
for (int i=1; i<=dc-sc; i++){
    ArrayList<String> hr = getMazePaths(sr, sc+i, dr, dc);
    for(String j: hr)
        myResult.add("h"+i+j);
}

// for vertical jumps
for (int i=1; i<=dr-sr; i++) {
    ArrayList<String> vr = getMazePaths(sr+i, sc, dr, dc);
    for(String j: vr)
        myResult.add("v"+i+j);
}

// for diagonal jumps
for (int i=1; i<=dc-sc && i<=dr-sr; i++) {
    ArrayList<String> dg = getMazePaths(sr+i, sc+i, dr, dc);
    for(String j: dg)
        myResult.add("d"+i+j);
}
return myResult;
}
```

Dry Run; 2×2



h1 v1 , v1 h1 , d1

Recursion On the Way Up

1. Print Subsequence

Input:	Output:	Constraints
yVTA	yVTA	0 <= str.length <= 3
	yvT	
	yvA	
	yv	
	yTA	
	yT	
	yA	
	y	
	VTA	
	VT	
	VA	
	V	
	TA	
	T	
	A	

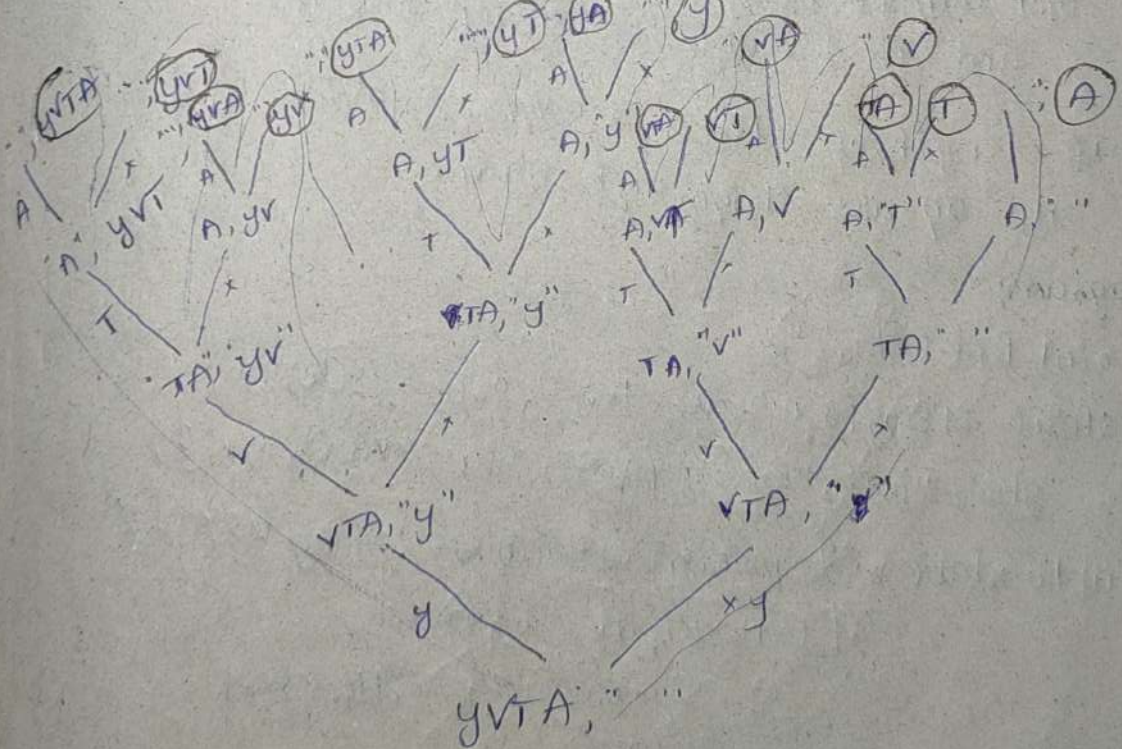
Steps;

1. We will have an empty string, we will get the first character and rest of the string goes to recursive calls.
2. Two cases; The first character will be considered or not
3. If the length of string reaches zero print answer and return.

Program:

```
printSS(str, "");
public static void printSS(String str, String ans) {
    if(str.length() == 0) {
        println(ans);
        return;
    }
    char ch = str.charAt(0);
    String ros = str.substring(1);
    printSS(ros, ans + ch);      // character considered
    printSS(ros, ans + " ");     // character not considered
}
}
```

Dry Run:



2. Print KPC

Input

78

Output

tv
tw
tx
uv
uw
ux

Constraints

0 <= str.length() <= 10
Str contains numbers only

Steps:

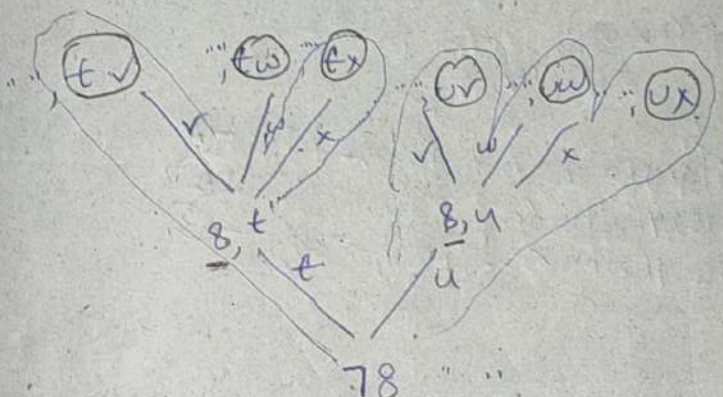
1. We will have an empty string, we will get the first character and rest of the string goes to recursive calls.
2. Two Cases: The first character will be considered and ~~appended to answer string~~ get the respective string from the codes Array.
3. Get each character and append to answer and go through the recursive calls.
4. If length of string reaches 0, print the answer string and return.

Program;

```
printKPC(str, "");
static String[] words = {".;", "abc", "def",
    "ghi", "jkl", "mno", "pqrs", "tu", "vwx", "yz"};
public static void printKPC(String str, String asf) {
    if (str.length() == 0) {
        System.out.println(asf);
        return;
    }
```

```
char ch = str.charAt(0);
String temp = str.substring(1);
String curr = words[ch-'0'];
for(int i=0; i<curr.length(); i++){
    char a = curr.charAt(i);
    PrintKPC(temp, asf+a);
}
```

Dry Run:



3. Print Stair Paths

Sample Input
3

Sample Output
```
111
12
21
3
```

Constraints
0<=n<=10

Steps:

1. As per the statement, we can make either 1/2/3 steps, so we will have 3 recursive calls.
2. If we take one step, append 1 to ans in Recursive call, similarly for 2 and 3 steps

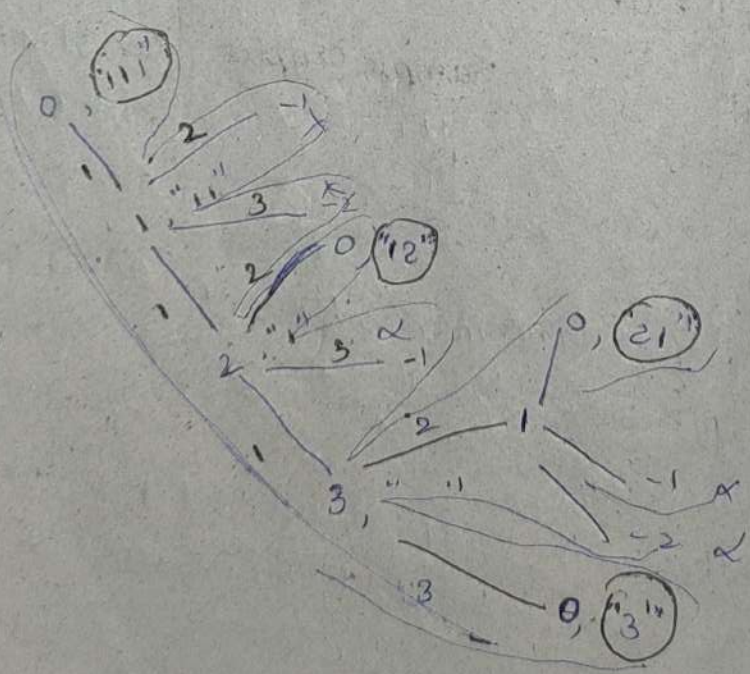
3. If we reach the destination, no need to move print ans. return

4. If we make higher number of steps than the actual left over steps, return; [we cannot make solution in that path].

Program:

```
printStairPaths(n, "");
public static void printStairPaths(int n, String path){
    if(n==0){
        System.out.println(path);
        return;
    }
    if(n<0){
        return;
    }
    printStairPaths(n-1, path+"1"); -(1)
    printStairPaths(n-2, path+"2"); -(2)
    printStairPaths(n-3, path+"3"); -(3)
}
```

Dry Run:

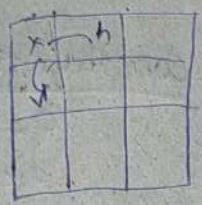


Print Maze Paths

Input	Output	Constraints
2 → rows	hv	0<=n<=10
2 → columns	vh	0<=m<=10

Steps:

1. We take an empty string psf (path so far)



→ We assume we take one horizontal step and assign to empty string and goes through iterations

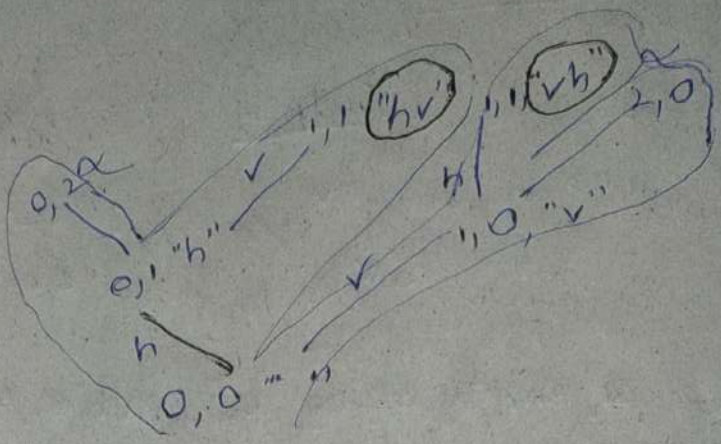
→ We assume we take one vertical step and assign to psf string and goes through iterations

2. If we reach the destination, print psf
3. If our source row exceeds destination row/ source column exceeds destination column then return empty.

Program:

```
sr=0, sc=0, dr=rows-1, dc=columns-1

printMazePaths(0,0,n-1,m-1,"");

public static void printMazePaths(int sr, int sc, int dr,
    int dc, String psf) {
        if(sr==dr && sc==dc){
            System.out.println(psf);
            return;
        }
        if(sr>dr || sc>dc){
            return;
        }
        printMazePaths(sr,sc+1,dr,dc,psf+"h");
        printMazePaths(sr+1,sc,dr,dc,psf+"v");
}
```

Dry Run;



5. Point Maze Paths with Jumps

Constraints

$0 <= n <= 5$

$0 <= m <= 5$

Input

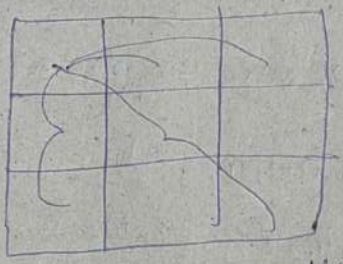
3 - rows

3 - columns

Output

h1h1v1v1	v1h1h1v1	d1h1v1
h1h1v2	v1h1v1h1	d1h1h1
h1v1h1v1	v1h1d1	d1d2
h1v1v1h1	v1h2v1	d2
h1v1d1	v1v1h1h1	
h1v2h1	v1v1h2	
h1d1v1	v1v1h2	
h2v1v1	v1d1h1	
h2v2	v2h1h1	
	v2h2	

Steps;



We can make one horizontal jump to dc-sc horizontal jumps and assign h and number to psf.

-> we can make one ~~horizontal~~ vertical jump to dr-sr and assign v and number to psf.

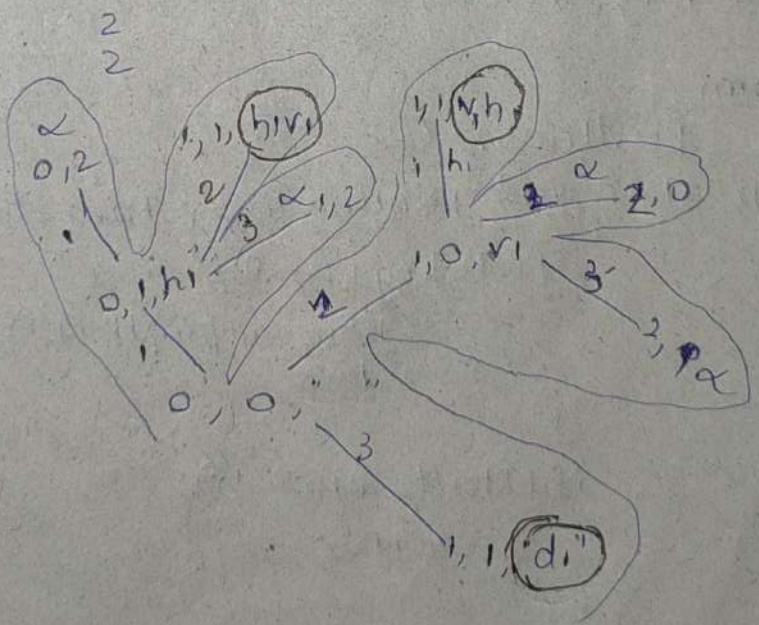
-> Similarly for diagonals $i <= dc-sc$ && $j <= dr-sr$ and assign diagonal and number to psf.

-> If you reach destination point, print psf, return

Program;

```
printMazePaths(0,0,n-1,m-1,"");
public static void printMazePaths(int sr, int sc, int dr,
int dc, String psf) {
    if(sr==dr && sc==dc){
        System.out.println(psf);
        return;
    }
// horizontal jump
    for(int i=1; i<=dc-sc; i++)
        printMazePaths(sr, sc+i, dr, dc, psf+"h"+i);

// vertical jump
    for(int i=1; i<=dr-sr; i++)
        printMazePaths(sr+i, sc, dr, dc, psf+"v"+i);

// diagonal jumps
    for(int i=1; i<=dc-sc && i<=dr-sr; i++)
        printMazePaths(sr+i, sc+i, dr, dc, psf+"d"+i);
}
```

Dry Run;



6. Print Encodings

Input	Output
655196	feeaif feesf

Constraints

0 <= str.length <= 10

Steps:

a - a = 0
b - a = 1 , b = a + 1 ⇒
c - a = 2 (c = a + 2 ⇒ c = 3 - 1 + a)

1. Take an Empty string
2. Get starting character from string, and pass rest of string, append [Get integer (from starting character) - 1 + 'a'] to asf (Recursively)
3. If length of string is ≥ 2, then check the first two characters are in the range If they are, then pass rest of string, append [Get integer (from starting 2 characters) - 1 + 'a'] to asf Recursively
4. If str.length() == 0, then return asf
5. If starting character is '0' then return.

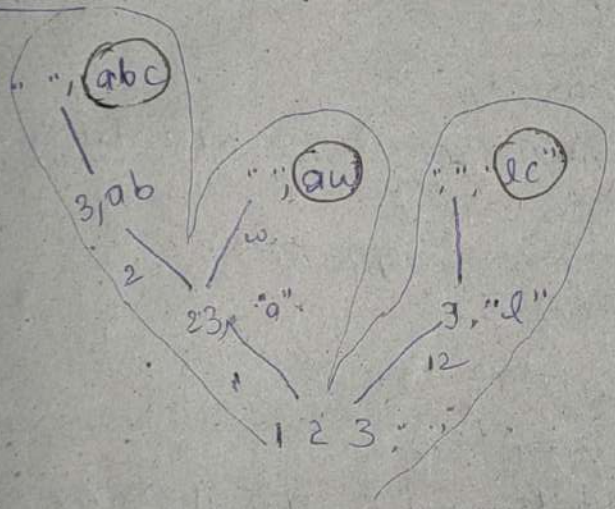
Program:

```
printEncodings("str", "");
public static void printEncodings(String str, String asf){
    if(str.length() == 0){
        System.out.println(asf);
        return;
    }
    if(str.charAt(0) == '0'){
        return;
    }
}
```

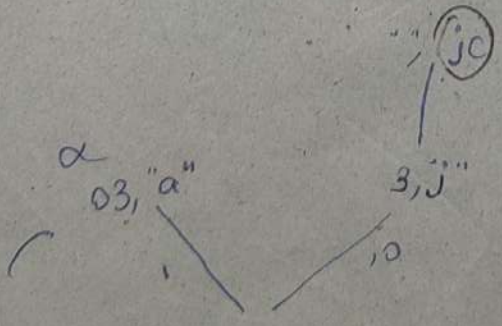
```
String s = str.substring(0,1);
String ros = str.substring(1);
printEncodings(ros, asft(char)(Integer.parseInt(s)-1+'a'));
if(str.length() >= 2){
    String s1 = str.substring(0,2);
    String ros1 = str.substring(2);
    if(Integer.parseInt(s1) <= 26){
printEncodings(ros1, asft(char)(Integer.parseInt(s1)-1+'a'));
    }
}
```

Dry Run:

① 123



② 103



Starting with 0 not accepted

7. Print Permutations

Sample Input	Sample Output	Constraints
abc	abc acb bac bca cab cba	0 <= str.length <= 7

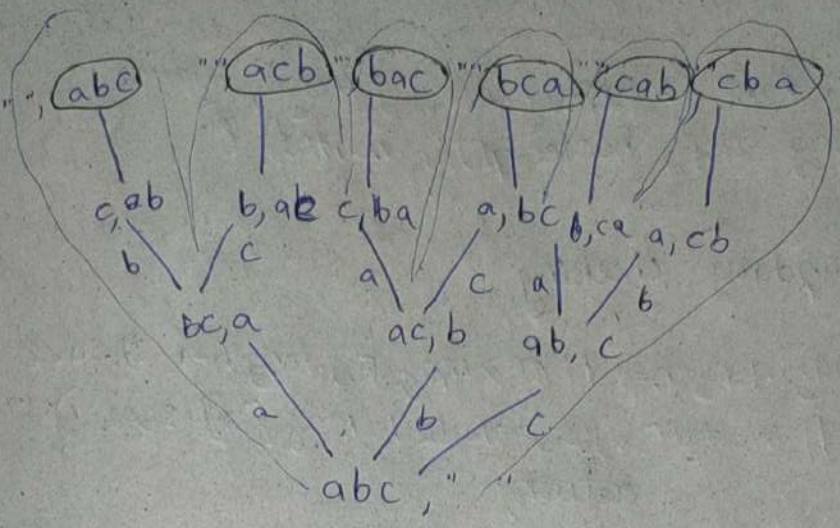
Steps:

1. Take an empty string,
2. Iterate through the string, get first character, assign the ros → (by deleting the first character), append asf+ch → pass Recursively.
3. If length of string reaches 0, print the asf

Program;

```
printPermutations(str, "");
public static void printPermutations(String str, String asf){
        if(str.length()==0){
            System.out.println(asf);
            return;
        }
    for(int i=0; i<str.length(); i++){
        char ch = str.charAt(i);
        String ros = str.substring(0, i)+str.substring
                    (i+1, str.length());
        printPermutations(ros, asf+ch);
    }
}
```

ry Run;



Recursion Backtracking

Flood Fill

Sample Input	Sample Output	Constraints
3 3 0 0 0 1 0 1 0 0 0	rddr	1 <= n <= 10 1 <= m <= 10 $e_1 e_2 \ldots$ n*m elements belongs to set (0,1)

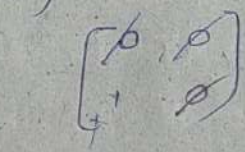
Steps:

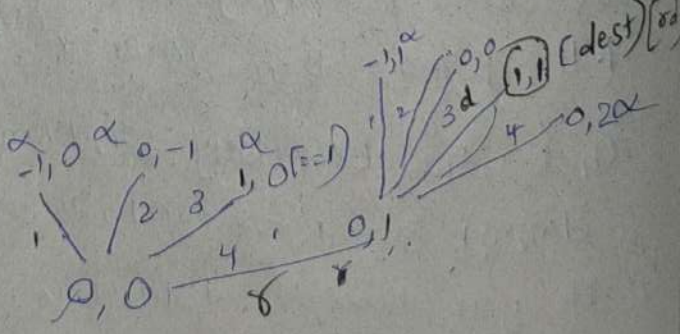
1. As per the statement we need to use operations in order of t, l, d, r (top, left, down, right)
2. We keep track an visited Array, if element is visited, path found make true else after performing all recursive calls. no path found then make visited = false
3. Check the boundary conditions and if element is 1 -> return, already visited -> return
4. If you reach the destination, print ans -> return

Program:

```
Public static void floodfill(int[][] maze, int sr,
int sc, String asf, boolean[][] visited) {

    // boundary checking
    if (sr > maze.length-1 || sc > maze[0].length-1 || sr < 0 ||
        sc < 0 || visited[sr][sc] == true || maze[sr][sc] == 1)
        return;

    if (sr == maze.length-1 && sc == maze[0].length-1) {
        System.out.println(asf);
        return;
    }

    visited[sr][sc] = true;
    floodfill(maze, sr-1, sc, asf+"t", visited); -1
    floodfill(maze, sr, sc-1, asf+"l", visited); -2
    floodfill(maze, sr+1, sc, asf+"d", visited); -3
    floodfill(maze, sr, sc+1, asf+"r", visited); -4
    visited[sr][sc] = false;
}
```

Dry Run:





2. Target Sum Subsets

Input

5 — n
10
20
30
40
50 } elements
60 — sum

Output

10,20,30,.
10,50,.
20,40.,

Constraints

0 <= n <= 30
0 <= n1, n2 ... n element <= 20
~~0 <= target <= 50~~

Steps:

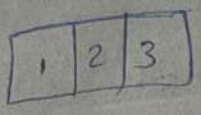
1. Consider the element of the array and add to asf, and targetsum // Don't add the element to asf and targetsum
2. If targetsum exceeds target return,
3. If all elements are covered and targetsum == target, print asf → Return.

targetSum=0, idx=0, asf=""; target=60

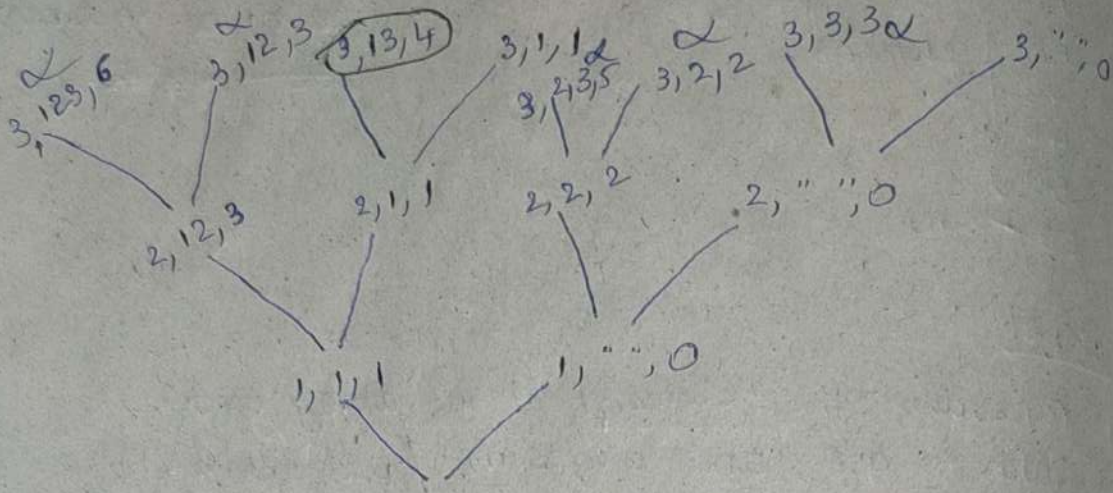
Program:

```
public static void printTargetSumSubset (int[] arr,
int idx, String asf, int targetsum, int target) {
        if (targetSum > target)
            return;
        if (idx == arr.length) {
            if (targetsum == target {
                println(asf+".");
            return;
        }
    printTargetSumSubset (arr, idx+1, asf+arr[idx]+" ", target
        Sum+arr[idx], target);
    printTargetSumSubset (arr, idx+1, asf, targetsum, target);
}
```

Dry Run;



target=4
targetsum=0



idx, asf, targetsum=0
0 " "

3. NQueens

Input	Output	Constraints
4	0-1, 1-3, 2-0, 3-2, 0-2, 1-0, 2-3, 3-1,	1<=n<=10

Steps;

1. Go through the number of columns
2. Check placing the queen is safe or not with that col and row.
3. Place the Queen
4. Make Recursive Call with row+1
5. If placed queen, doesn't make solution, we need to do Backtrack, unplace the queen.

for Queen Safe: check in the ~~row wise, and~~ column wise, diagonal wise (both), if element found return false.

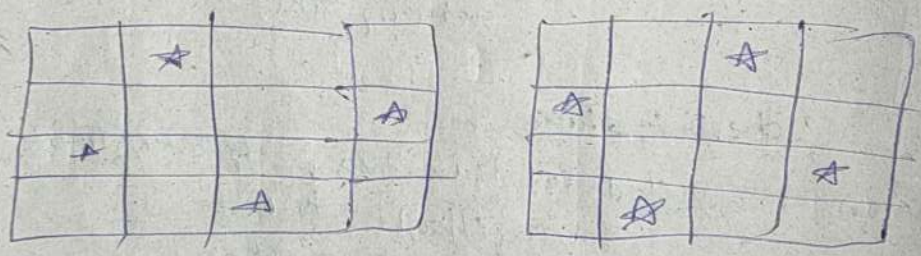
Program:

```
boolean[][] chess = new boolean[n][n];
printNQueens(chess, "", 0);

// check Queen is safe to place
public static boolean isQueenSafe(boolean[][] chess, int
row, int col) {
    // check in col
    for (int i = row-1; i >= 0; i--) {
        if (chess[i][col] == true)
            return false;
    }
    // check in diag1
    for (int i = row, j = col-1; i >= 0 && j >= 0; i--, j--) {
        if (chess[i][col] == true)
            return false;
    }
    // check in diag2
    for (int i = row-1, j = col+1; i >= 0 && j < chess.length-1; i--, j++) {
        if (chess[i][col] == true)
            return false;
    }
    return true;
}
```

```
Public static void printNQueens(boolean[][] chess,
    String qsf, int row) {
        if(row == chess.length) {
            System.out.println(qsf + ".");
            return;
        }
    for(int col=0; col<chess.length; col+1) {
        if(isQueenSafe(chess, row, col)==true) {
            //place Queen
            chess[row][col] = true;
            //make Recursive Call
            printNQueens(chess, qsf + row+"-"+col+", ",
                row+1);
            // unplace the Queen
            chess[row][col] = false;
        }
    }
}
```

Dry Run; ⌐ Visualize by Recursion Tree

For n=4 → 2 possible solutions



4. Knights Tour

Sample Input	Sample Output;	Constraints
5	25 2 13 8 23	n=5
2	12 7 24 3 14	0<=row<n
0	1 18 15 22 9	0<=col<n
	6 11 20 17 4	
	19 16 5 10 21	

(continues with all patterns)

Steps;

1. Take initially stepNo=1, and assign it to starting position.
2. Go through all 8 directions, mentioned in statement.
3. If next element not covered, backtrack, and make visited element=0
4. If stepno equal to ArrayLength → then assign left over position to stepno, print the board, and reassign to 0 => to get other possible solutions => return

Program;

```
printKnightsTour(chess, r, c, 1);
Public static void printKnightsTour (int[][] chess, int r, int c,
  int stepno) {
    if (r<0 || c<0 || r>=chess.length || c>=chess.
        length || chess[r][c]!=0)
        return;
    if (stepno == chess.length * chess.length) {
        chess[r][c]=stepno;
        displayBoard(chess);
        chess[r][c]=0;
        return;
    }
}
```

```
chess[r][c] = stepno;
printKnightsTour(chess, r-2, c+1, stepno+1);
printKnightsTour(chess, r-1, c+2, stepno+1);
printKnightsTour(chess, r+1, c+2, stepno+1);
printKnightsTour(chess, r+2, c+1, stepno+1);
printKnightsTour(chess, r+2, c-1, stepno+1);
printKnightsTour(chess, r+1, c-2, stepno+1);
printKnightsTour(chess, r-1, c-2, stepno+1);
printKnightsTour(chess, r-2, c-1, stepno+1);
chess[r][c] = 0;
}
```

Dry Run:

Eg for 1 pattern : 2,0 starting position

	0	1	2	3	4
0	25	2 (3)	13 (3)	8	23 (6)
1	12 (2)	7 (4)	24 (3)	3 (4)	14 (5)
2	1 (1)	18	15 (1)	22 (1)	9 (5)
3	6	11 (6)	20 (6)	17 (6)	4 (6)
4	19	16 (1)	5 (1)	10 (7)	21