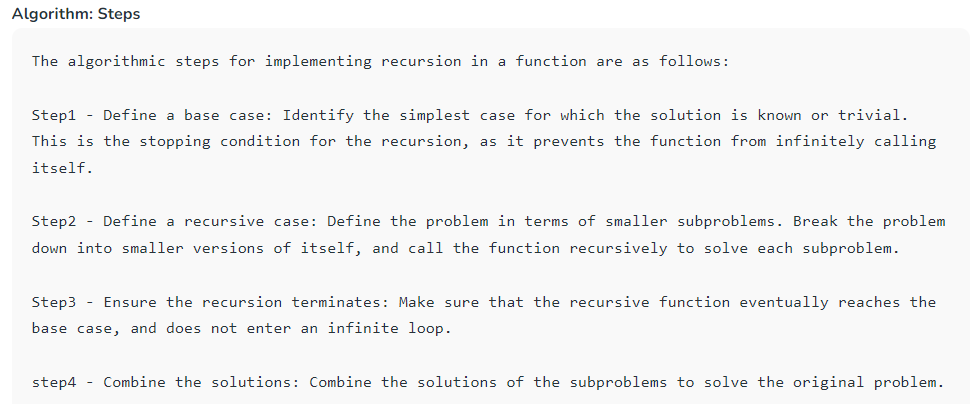
**Recursion**

**Theory :**

**What is Recursion?**   
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

**Base Case :**It is essential to know that we should provide a certain case in order to terminate this recursion process.



Eg: ***approach(2) – Recursive adding***

***f(n) = 1                  n=1***

***f(n) = n + f(n-1)    n>1***

**How are recursive functions stored in memory?**

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished.

**Why Stack Overflow error occurs in recursion?**   
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)

{

// wrong base case (it may cause

// stack overflow).

if (n == 100)

return 1;

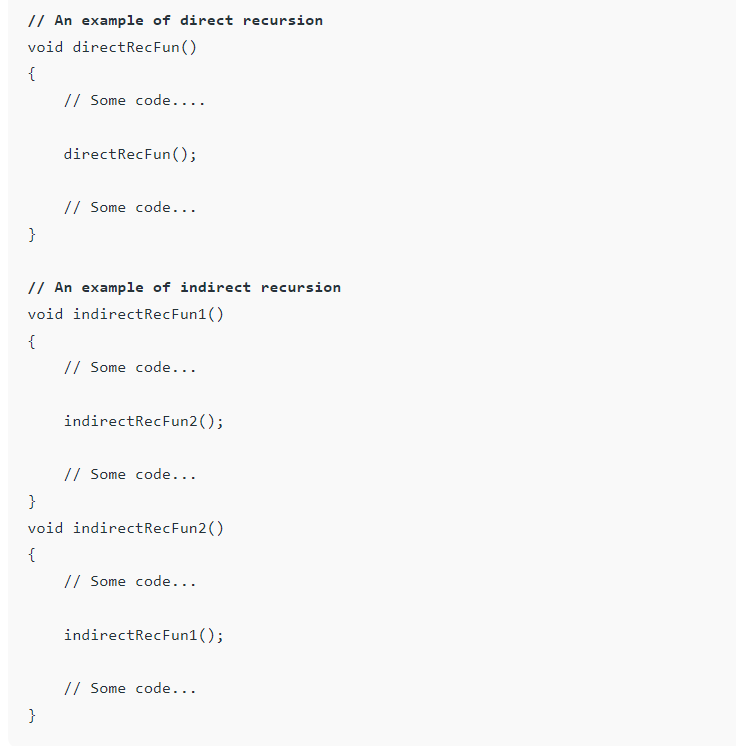
else

return n\*fact(n-1);

}

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

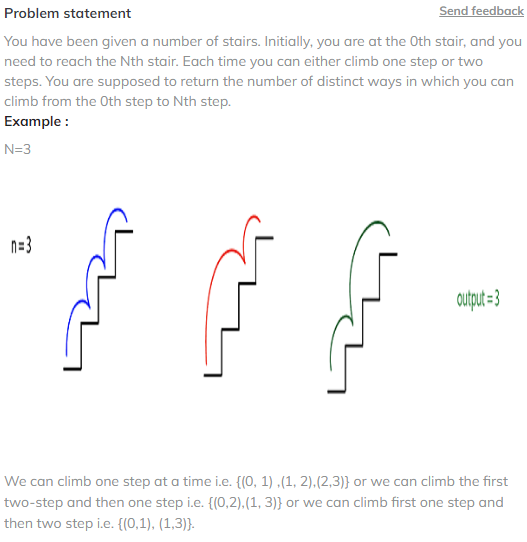
**What is the difference between direct and indirect recursion?**

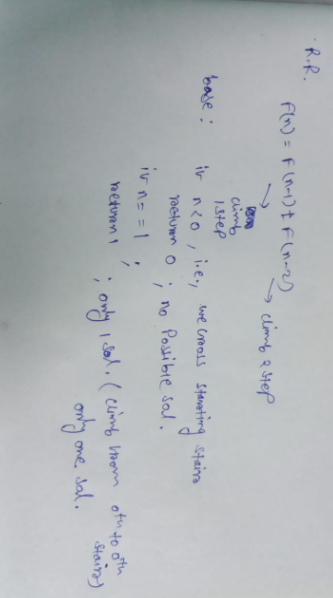
A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly.

**What is the difference between tailed and non-tailed recursion?**   
A recursive function is tail recursive when a recursive call is the last thing executed by the function.

**Used in :** Graph traversal,divide and conquer, sorting,backtracking

**Climb stairs**





https://github.com/Priyam-Mondal/DSA\_code\_notes/blob/main/Recursion/climb%20stairs

**Check if sorted array**

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Check%20if%20sorted%20array>

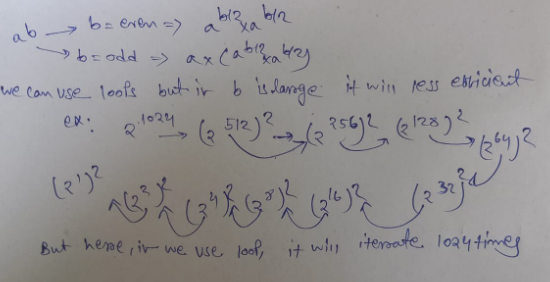
**Sum of array elements**

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Sum%20of%20array%20elements>

**String reverse with recursion**

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/String%20reverse%20with%20recursion>

**Exponent in efficient way**



<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Exponent%20in%20efficient%20way>

**Lecture 35: Merge sort using recursion**

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

-> Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.

- > Time complexity : O(N log(N))

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Merge%20sort%20using%20recursion>

**Inversion Count**

A[i] > a[j] when i < j

Add a variable to count inversion when merging is done and return count for each subarray to add for total array

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Inversion%20Count>

## **Lecture36: Quick Sort using Recursion**

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

-> to chose pivot element : start from left element and count number of elements smaller than first element. Swap first element with arr[count]

**Time Complexity:**

**Best Case:** Ω (N log (N)) : The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

**Worst Case:** O(N2) : The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element.

-> not efficient for small data set

-> not a stable algorithm

-> in-place sorting

//Function to sort an array using quick sort algorithm.

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

{

if(low>=high){

return;

}

int p = partition(arr,low,high);

quickSort(arr,low,p-1);

quickSort(arr,p+1,high);

}

static int partition(int arr[], int low, int high)

{

int pivotIndex = low;

int pivot = arr[low];

// Initialize the index to the left of the pivot

int i = low + 1;

// Iterate from the index next to the pivot till the end of the array

for (int j = low + 1; j <= high; j++) {

// If current element is less than the pivot, swap it with the element at index i

if (arr[j] < pivot) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

i++; // Move the index to the right

}

}

// Swap the pivot element with the element at index i - 1

int temp = arr[i - 1];

arr[i - 1] = arr[pivotIndex];

arr[pivotIndex] = temp;

// Return the index of the pivot element after partitioning

return i - 1;

}

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Quick%20Sort%20using%20Recursion>

**Why merge sort prefer for linked list and quick sort for arrays?**

https://www.prepbytes.com/blog/linked-list/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/

## **Lecture37: Recursion - Subsets / Subsequences of String**

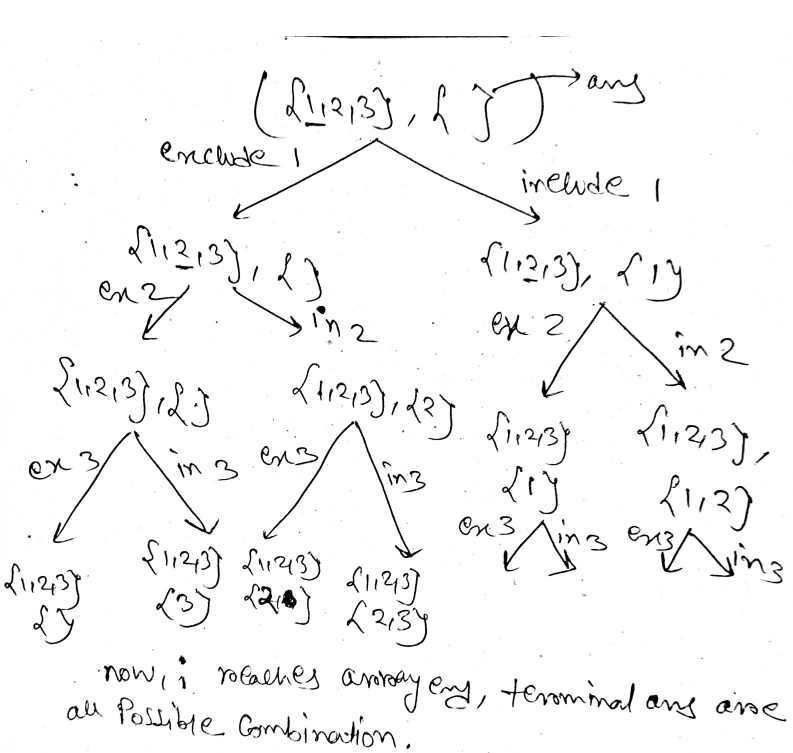
**Subset:**

Power set that is set of all subsets.

For set={1,2,3}

Power set will be = {[], {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}

If set has n elements then power set will have 2^n elements



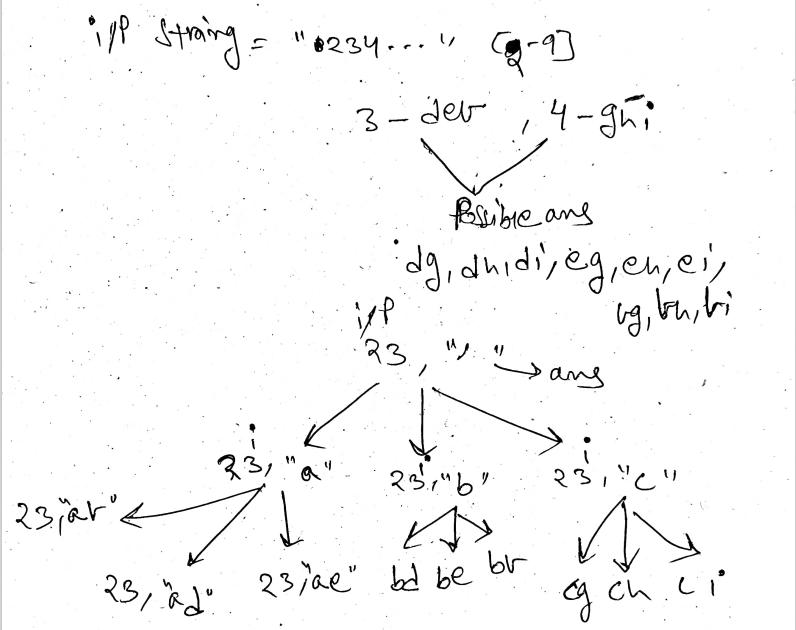
<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/subsets.java>

**Subsequences of String :**

Same recursion tree as subset problem

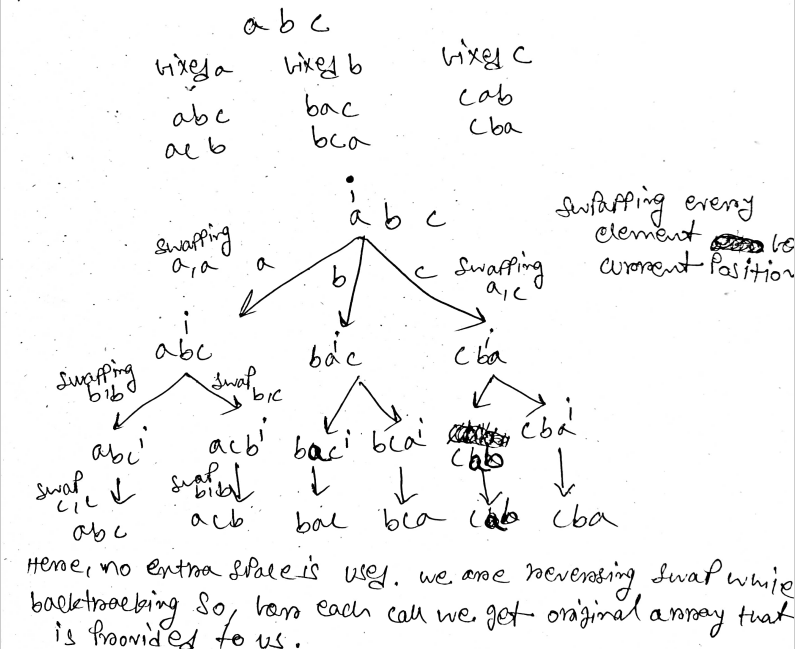
<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/Subsequences_of_String.java>

## **Lecture38: Phone Keypad Problem Recursion**



<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/letter_comb_phone_number.java>

## **Lecture39: Permutations of a String**



<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/permutations.java>

**Count Distinct Anagrams**

-> For each word, can you count the number of permutations possible if all characters are distinct?

-> use HashSet to store unique permutation

-> multiply number of permutations for each word to get total number

But using normal recursion will result in TLE, we have to use some modulo factorial methods. Which I cant implement for this problem.

<https://github.com/Priyam-Mondal/DSA_code_notes/blob/main/Recursion/count_anagrams.java>

**275\_Rat in a Maze**

1. Declare a 2D array visited to keep track of visited cells and an ArrayList result to store the paths.
2. path Function:
   1. If the current cell (x, y) is the bottom-right corner, add the dir to the result list and return.
   2. If the current cell is blocked or already visited, return.
   3. Mark the current cell as visited.
   4. Recursively explore the cells above, left, below, and right if they are within bounds.
   5. After exploration, mark the current cell as unvisited to backtrack
3. Initialize the visited array with zeros.
4. Clear the result list.
5. Check if the top-left or bottom-right cell is blocked. If blocked, returns an empty result.
6. Call the path function to start finding paths from the top-left corner.
7. Sort the result list lexicographically.
8. Return the sorted result list.

public static boolean isSafe(int x, int y, int[][] arr, int n, int[][] visited){

        if((x<n && x>=0 && y>=0 && y<n) && arr[x][y]==1 && visited[x][y]==0){

            return true;

        }

        else{

            return false;

        }

    }

    public static void solve(int[][] arr, int n, int srcX, int srcY,

     ArrayList<String> ans, int[][] visited, StringBuilder path)

    {

        if(srcX==n-1 && srcY==n-1){

            ans.add(new String(path));

            return;

        }

        visited[srcX][srcY]=1;

        // 4 choices - U, D, L, R

        // up

        int newX=srcX-1;

        int newY=srcY;

        if(isSafe(newX,newY,arr,n,visited)){

            path.append("U");

            solve(arr,n,newX,newY,ans,visited,path);

            // deleting last char for backtrack

            path.deleteCharAt(path.length()-1);

        }

        // down

        newX=srcX+1;

        newY=srcY;

        if(isSafe(newX,newY,arr,n,visited)){

            path.append("D");

            solve(arr,n,newX,newY,ans,visited,path);

            path.deleteCharAt(path.length()-1);

        }

        // left

        newX=srcX;

        newY=srcY-1;

        if(isSafe(newX,newY,arr,n,visited)){

            path.append("L");

            solve(arr,n,newX,newY,ans,visited,path);

            path.deleteCharAt(path.length()-1);

        }

        // right

        newX=srcX;

        newY=srcY+1;

        if(isSafe(newX,newY,arr,n,visited)){

            path.append("R");

            solve(arr,n,newX,newY,ans,visited,path);

            path.deleteCharAt(path.length()-1);

        }

        // resetting visited for backtrack

        visited[srcX][srcY]=0;

    }

    public static ArrayList < String > findSum(int[][] arr, int n) {

        ArrayList<String> ans = new ArrayList<>();

        if(arr[0][0]==0){

            return ans;

        }

        int srcX=0, srcY=0;

        StringBuilder path = new StringBuilder();

        int[][] visited = new int[n][n];

        // making visited 0

        for(int i=0; i<n; i++){

            for(int j=0; j<n; j++){

                visited[i][j]=0;

            }

        }

        solve(arr,n,srcX,srcY,ans,visited,path);

        Collections.sort(ans);

        return ans;

    }

**Time Complexity:**

The time complexity is determined by the number of recursive calls made by the solve function. In the worst case, each cell of the matrix is visited, and for each cell, four recursive calls are made (up, down, left, right). Therefore, the time complexity is O(4^(n^2)), where 'n' is the size of the matrix.

**Space Complexity:**

The space complexity is determined by the space required for the recursive call stack and the additional space used for the visited matrix and the path StringBuilder. In the worst case, the maximum depth of the recursive call stack is n^2, and for each recursive call, a constant amount of space is used for the visited matrix and path. Therefore, the space complexity is O(n^2).