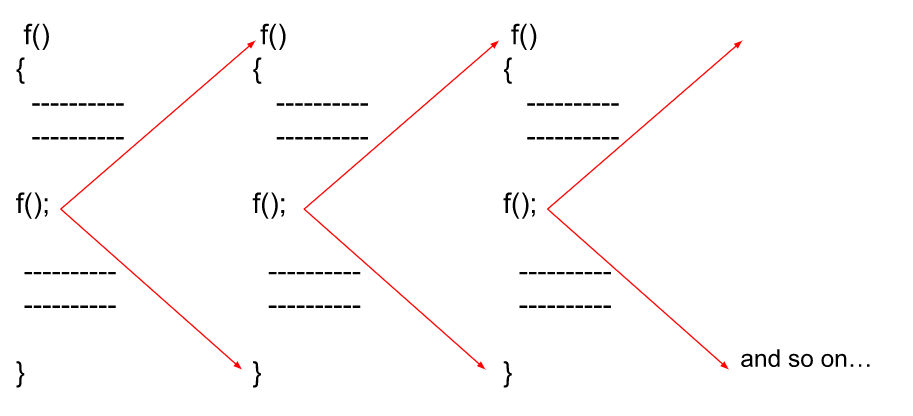
**Introduction to Recursion – Understand Recursion by printing something N times**

**Pre-requisite:**The learner must know how to write a basic function in any language and how to make a function call from the main function.

**What is Recursion?**

It is a phenomenon when a function calls itself indefinitely until a specified condition is fulfilled.

Let’s understand recursion with the help of an illustration :



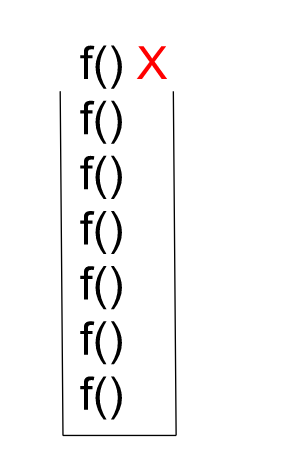
As we can see in the above image, a function is calling the same function inside its body. Since there is no condition to stop the recursive calls, the calls will run indefinitely until the stack runs out of memory ( stack overflow ).

**What is Stack Overflow in Recursion?**

Whenever recursion calls are executed, they’re simultaneously stored in a **recursion stack** where they wait for the completion of the recursive function. A recursive function can only be completed if a base condition is fulfilled and the control returns to the parent function.

But, when there is no base condition given for a particular recursive function, it gets called indefinitely which results in a Stack Overflow i.e, exceeding the memory limit of the recursion stack and hence the program terminates giving a Segmentation Fault error.

The illustration above also represents the case of a Stack Overflow as there is no terminating condition for recursion to stop, hence it will also result in a memory limit exceeded error.



**Base Condition**

It is the condition that is written in a recursive function in order for it to get completed and not to run infinitely. After encountering the base condition, the function terminates and returns back to its parent function simultaneously.

To get a better understanding of how the base condition is an integral part of recursive functions, let us see an example below :

Let’s say we have to print integers starting from 0 till 2 only, this will be how the pseudocode for it will look like

int count = 0;

void func(){

if(count == 3 ) return;

print(count);

count++;

func();

}

main()

{

print();

}

According to this pseudocode, the function will increment and print the value of count and then return when the base condition becomes true i.e, it will only print 0,1,2 and 3 and then execution gets completed.

**Recursive code for printing numbers from 0 to 3**:

static int cnt = 0;

static void print(){

// Base Condition.

if(cnt == 3 ) return;

System.out.println(cnt);

// Count incremented.

cnt = cnt+1;

print();

}

public static void main(String[] args) {

print();

}

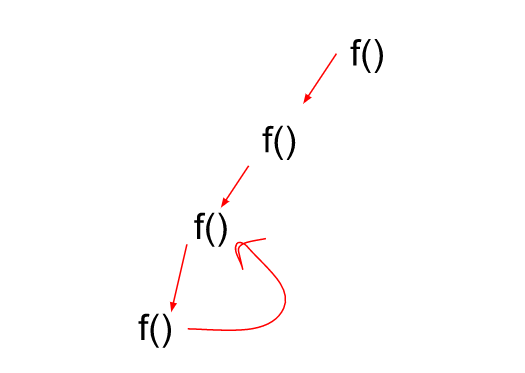
}

**Output**

0  
1  
2

**Recursive Tree**

A recursive tree is basically a representative form of recursion which depicts how functions are called and returned as a series of events happening consecutively. It is a pictorial description of the process of recursion as illustrated below :



When a recursive call gets completed, the control returns back to its parent function which is then further executed until the last function waiting in the recursive stack returns.

As a summary of the lecture, the basics of recursion such as the following were clear to us :

* What is Recursion
* Base Condition
* Stack Overflow / Stack Space
* Recursion Tree

**Print Name N times using Recursion**

**Prerequisite:**The learner must know how to write a simple function in any language with parameters.

Now, after understanding the basics of recursion, the recursion tree, and the base case of recursion we can solve some basic problems of recursion which would strengthen our concepts and make us understand how recursion functions at its core.

**Problem: Print your Name N times using recursion**

Since in this problem, there is no count that can be incremented each time we call a function, how can we keep a track of how many times we have printed the name?

For this problem, we’re going to be using a function along with parameters in which we can keep track of the number of times we’ve printed something. To understand this problem better, let us see the pseudocode below for this :

void func( i, n )

{

if(i>n) return;

print(“Raj”);

f( i+1,N );

}

main()

{

int n;

input(n);

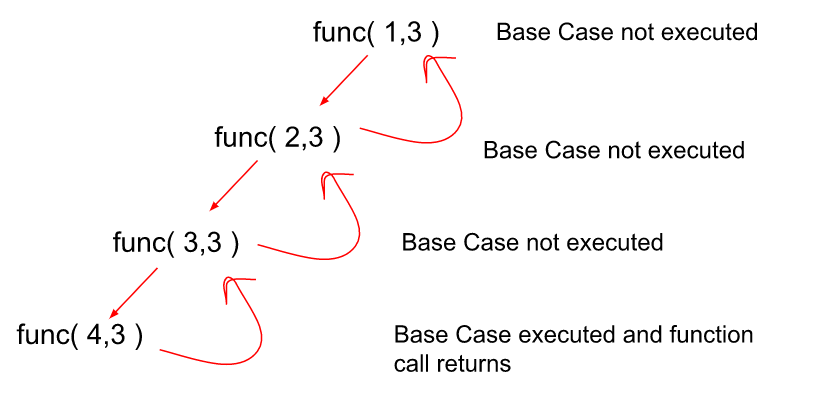
f(1,n);

}

We can clearly see in this pseudocode that we first call the function when the value of i is 1 and then print the name and increment i by 1 inside the parameter of the function and make a call again. But, we know that this will go on forever as i will be increasing continuously after every function call. So, to avoid this we put a base condition that if i exceeds n, then simply terminate the current recursive call and return to the previous call.

In this way, the text that we want to print would be printed n times and as soon as we exceed the count of printing by n, the function terminates.

**Recursion Tree for the following problem can be represented as follows :**



**Code**

class Recursion {

static void func(int i, int n){

// Base Condition.

if(i>n) return;

System.out.println("Raj");

// Function call to print till i increments.

func(i+1,n);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 4.

int n = 4;

func(1,n);

}

}

**Output**

Raj  
Raj  
Raj  
Raj

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

**Print 1 to N using Recursion**

**Prerequisite:**The learner must know how to write a simple function in any language with parameters.

Now, after understanding the basics of recursion, the recursion tree, and the base case of recursion we can solve some basic problems of recursion which would strengthen our concepts and make us understand how recursion functions at its core. In this article, we’ll print integers from 1 to N without using any Global Variable but by using function parameters.

**Problem: Print from 1 to N using Recursion**

Since in this problem, there is no global variable that can be incremented each time we call a function, how can we keep a track of the number of integers being printed on the output screen?

For this problem, we’re going to be using a function along with parameters that get incremented with each function call through which we can keep track of the integer count while printing. To understand this problem better, let us see the pseudocode below for this :

void func( i, n )

{

if(i>n) return;

print(i);

f( i+1,N );

}

main()

{

int n;

input(n);

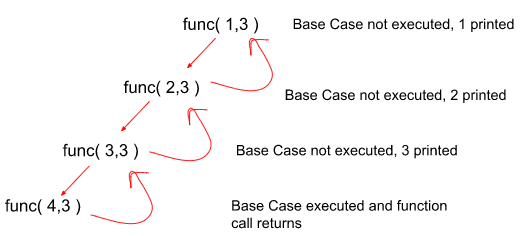
f(1,n);

}

We can clearly see in this pseudocode that we first call the function when the value of i is 1 and then print the value of i and increment i by 1 inside the parameter of the function and make a call again. But, we know that this will go on forever as i will be increasing continuously after every function call. So, to avoid this we put a base condition that if i exceeds n, then simply terminate the current recursive call and return to the previous call.

In this way, all the integers from 1 to N would get printed and as soon as we exceed the count of printing by n, the function terminates.

**Recursion Tree for the following problem can be represented as follows :**



**Code:**

class Recursion {

static void func(int i, int n){

// Base Condition.

if(i>n) return;

System.out.println(i);

// Function call to print i till i increments to n.

func(i+1,n);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 4.

int n = 4;

func(1,n);

}

}

**Output**

1  
2  
3  
4

**Alternate Approach for printing integers from 1 to N (using Backtracking)**

This is an alternative approach for printing the integers from 1 to N using recursion. In the previous approach, we used forward recursion but in this approach, we will be using backward recursion. The only change from the previous approach here will be that the print line would be kept after the function call inside the recursive function contrary to the previous approach. The function would be called for printing (i-1) integers and the nth integer would be printed.

To get a clear understanding of what this approach is like let’s see the pseudocode for this :

void func( i, n )

{

if(i<1) return;

f( i-1,N );

print(i);

}

main()

{

int n;

input(n);

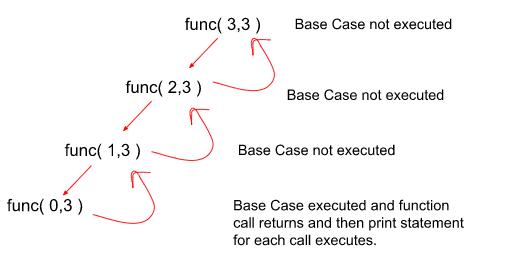
f(n,n);

}

We can clearly see in this pseudocode that we first call the function when the value of i is N and then make a call again inside this function for printing (n-1) integers and after this we print N. But, we know that this will go on forever as i will be decreasing continuously after every function call. So, to avoid this we put a base condition that if i is less than n, then simply terminate the current recursive call and return to the previous call.

In this way, all the integers from 1 to N would get printed and as soon as i becomes less than n, the function call terminates.

**Recursion Tree for the following approach can be represented as follows :**



**Code ( Alternate Approach )**

class Recursion {

static void func(int i){

// Base Condition.

if(i<1) return;

// Function call to print(n-1) integers.

func(i-1);

System.out.println(i);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 4.

int n = 4;

func(n);

}

}

**Output:**

1  
2  
3  
4

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

**Print N to 1 using Recursion**

**Prerequisite:**The learner must know how to write a simple function in any language with parameters.

Now, after understanding the basics of recursion, the recursion tree, and the base case of recursion we can solve some basic problems of recursion which would strengthen our concepts and make us understand how recursion functions at its core. In this article, we’ll print integers from N to 1 without using any Global Variable but by using function parameters.

**Problem: Print from N to 1 using Recursion**

Since in this problem, there is no global variable that can be decremented each time we call a function, how can we keep a track of the number of integers being printed on the output screen?

For this problem, we’re going to be using a function along with parameters that get decremented with each function call through which we can keep track of the integer count while printing. To understand this problem better, let us see the pseudocode below for this :

void func( i, n )

{

if(i<1) return;

print(i);

f( i-1,N );

}

main()

{

int n;

input(n);

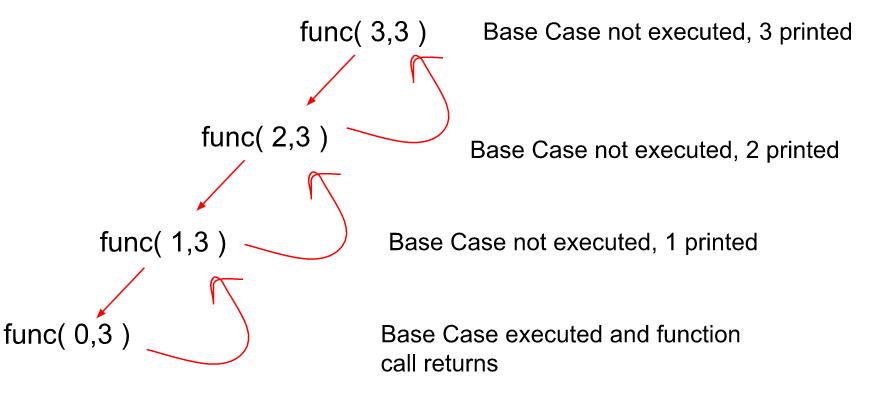
f(n,n);

}

We can clearly see in this pseudocode that we first call the function when the value of i is n and then print the value of i and decrement i by 1 inside the parameter of the function and make a call again. But, we know that this will go on forever as i will be decreasing continuously after every function call. So, to avoid this we put a base condition that if i is less than 1, then simply terminate the current recursive call and return to the previous call.

In this way, all the integers from N to 1 would get printed and as soon as the count becomes less than 1, the function terminates.

**Recursion Tree for the following problem can be represented as follows :**



**Code**

class Recursion {

static void func(int i, int n){

// Base Condition.

if(i<1) return;

System.out.println(i);

// Function call to print i till i decrements to 1.

func(i-1,n);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 4.

int n = 4;

func(n,n);

}

}

**Output**

4  
3  
2  
1

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

**Alternate Approach for printing integers from N to 1 (using Backtracking)**

This is an alternative approach for printing the integers from N to 1 using recursion. In the previous approach, we used forward recursion but in this approach, we will be using backward recursion. The only change from the previous approach here will be that the print line would be kept after the function call inside the recursive function contrary to the previous approach. The function would be called for printing (i+1) integers and the ith integer would be printed.

To get a clear understanding of what this approach is like let’s see the pseudocode for this :

void func( i, n )

{

if(i>n) return;

f( i+1,N );

print(i);

}

main()

{

int n;

input(n);

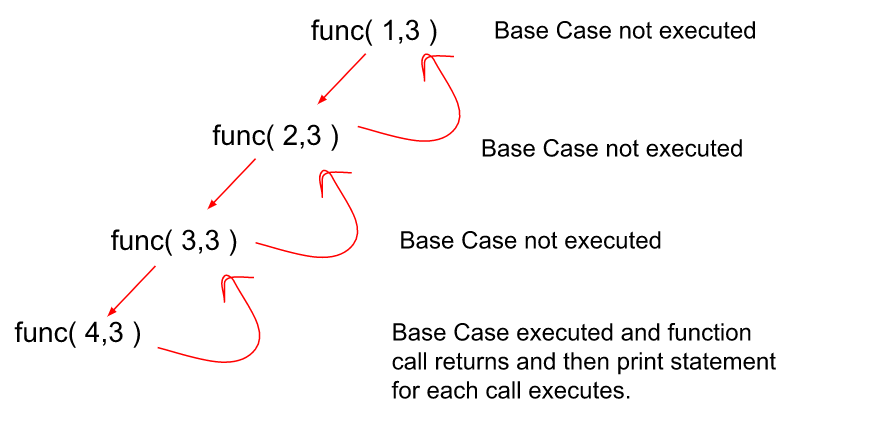
f(1,n);

}

We can clearly see in this pseudocode that we first call the function when the value of i is N and then make a call again inside this function for printing (n-1) integers and after this, we print N. But, we know that this will go on forever as i will be decreasing continuously after every function call. So, to avoid this we put a base condition that if i is less than n, then simply terminate the current recursive call and return to the previous call.

In this way, all the integers from 1 to N would get printed and as soon as i becomes less than n, the function call terminates.

**Recursion Tree for the following approach can be represented as follows :**



**Code ( Alternate Approach )**

class Recursion {

static void func(int i, int n){

// Base Condition.

if(i>n) return;

// Function call to print(i+1) integers.

func(i+1,n);

System.out.println(i);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 4.

int n = 4;

func(1,n);

}

}

**Output**

4  
3  
2  
1

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

# Sum of first N Natural Numbers

**Problem statement:**Given a number ‘N’, find out the sum of the first N natural numbers.

**Examples:**

**Example 1:**

**Input:** N=5

**Output:** 15

**Explanation:** 1+2+3+4+5=15

**Example 2:**

**Input:** N=6

**Output:** 21

**Explanation:** 1+2+3+4+5+6=15

### **Solution**

Disclaimer: Don’t jump directly to the solution, try it out yourself first.

**Solution1: Using Loop**

**Intuition:**We can simply add numbers one by one from 1 to N.

For eg. if N = 5, we can add 1+2+3+4+5=15.

We can use a for loop or while loop to achieve the goal.

**Approach:**

* Take a variable sum and initialize it as 0.
* Take a for loop and run from 1 to N.
* Save the result in sum.

**Code:**

import java.util.\*;

public class tuf {

public static void main(String[] args) {

solve(5);

solve(6);

}

public static void solve(int n) {

int sum = 0;

for (int i = 1; i <= n; i++) {

sum += i;

}

System.out.println("The sum of the first " + n + " numbers is: " + sum);

}

}

**Output:**

The sum of the first 5 numbers is: 15  
The sum of the first 6 numbers is: 21

**Time Complexity:** O(N)

**Space Complexity:** O(1)

**Solution 2: Using the formula**

**Intuition:**We can use the formula for the sum of N numbers, i.e**N(N+1)/2.**

For eg: N=5

5(5+1)/2 = 5(6)/2 = 15.

**Approach:**

* Take a variable sum.
* Initialize it with **N(N+1)/2,**where N is a given number.

**Code:**

import java.util.\*;

public class tuf {

public static void main(String[] args) {

solve(5);

solve(6);

}

public static void solve(int N) {

int sum = N \* (N + 1) / 2;

System.out.println("The sum of the first " + N + " numbers is: " + sum);

}

}

**Output:**

The sum of the first 5 numbers is: 15  
The sum of the first 6 numbers is: 21

**Time Complexity:** O(1)

**Space Complexity:** O(1)

**Recursive way of calculating the sum of first N Natural Numbers:**

* Parameterized Way
* Functional Way

**1. Parameterized way**

In this approach, instead of using a global variable for calculating the sum, we pass the sum in the parameters of the function each time we add an integer to it during the function call. The sum gets incremented by an ith integer and i get decremented by 1 in each function call. At the end when i becomes less than 1, we simply return the calculated sum until that point.

To understand this parameterized approach better, let us have a look at the pseudocode given below:

void func(i,sum)

{

if(i<1)

{

print(sum);

return;

}

func(i-1,sum+i);

}

main()

{

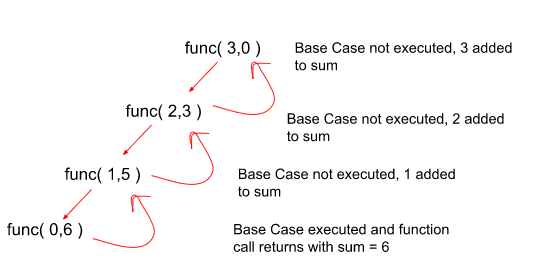
input(n);

func(n,0);

}

We can clearly see in this pseudocode that we first call the function when the value of sum is 0 and then we increment the value of sum by i (initially i is n) and decrement i by 1 inside the parameter of the function and make a call again. But, we know that this will go on forever as i will be decreasing continuously after every function call. So, to avoid this we put a base condition that if i is less than 1, then simply terminate the current recursive call and return the calculated sum.

**Recursion Tree for the following problem can be represented as follows :**



**Code:**

class Recursion {

static void func(int i, int sum){

// Base Condition.

if(i<1){

System.out.println(sum);

return;

}

// Function call to increment sum by i till i decrements to 1.

func(i-1,sum+i);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 3.

int n = 3;

func(n,0);

}

}

**Output**

6

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

**2. Functional way**

This approach is a lot simpler than the parameterized recursion. We can visualize the sum of n natural numbers in the following way as shown below:

sumOfNaturalNumbers(N) = N + sumOfNaturalNumbers(N-1);

The Sum of N natural numbers would just be the Nth integer added to the Sum of (N-1) natural numbers. The base case can be visualized as if n decreases to 0, then we return 0 because the sum of 0 natural numbers is 0 only. Here, we’ve just broken the problem into 2 subparts and the answers of both these subparts would be added and stored in the Sum(n) function which would then be printed at last.

To understand this functional approach better, let us have a look at the pseudocode given below:

int func(n)

{

if(n == 0)

{

return 0;

}

return n + func(n-1);

}

main()

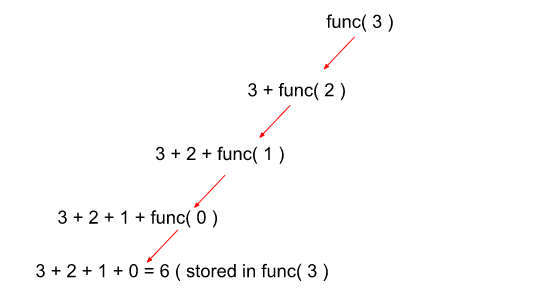
{

input(n);

func(n);

}

**Recursion Tree for the following problem can be represented as follows :**



**Code:**

class Recursion {

static int func(int n){

// Base Condition.

if(n == 0){

return 0;

}

// Problem broken down into 2 parts and then combined.

return n + func(n-1);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 3.

int n = 3;

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

}

}

**Output**

6

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

# Factorial of a Number : Iterative and Recursive

**Problem Statement:** Given a number **X,**  print its factorial.

To obtain the factorial of a number, it has to be multiplied by all the whole numbers preceding it. More precisely X! = X\*(X-1)\*(X-2) … 1.

Note: X  is always a positive number.

**Examples:**

**Example 1:**

**Input:** X = 5

**Output:** 120

**Explanation:** 5! = 5\*4\*3\*2\*1

**Example 2:**

**Input:** X = 3

**Output:** 6

**Explanation:** 3!=3\*2\*1

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Iterative**

**Approach**:  
  
Since the factorial of X  will be the product of the number itself and all its preceding numbers we can run loop i, from 1 to X. In every iteration current i, is multiplied with the product so far.

**Code:**

public class Main {

static int factorial(int X) {

int ans = 1;

for (int i = 1; i <= X; i++) {

ans = ans \* i;

}

return ans;

}

public static void main(String[] args) {

int X = 5;

int result = factorial(X);

System.out.println("The factorial of " + X + " is " + result);

}

}

**Output:** The factorial of 5 is 120

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**Solution 2: Recursive**

**Recursive way of calculating the factorial of first N Numbers (functional way):**

The Factorial of a number N can be calculated by multiplying all the natural numbers till the number N. Through this approach, we can visualize the factorial of n natural numbers in the following way as shown below:

factorial(N) = N \* factorial(N-1);

The Factorial of N natural numbers is the Nth integer multiplied by the Factorial of (N-1) natural numbers. The base case can be visualized as if n decreases to 0, then we return 1 because the factorial of 0 is 1 only. Here, we’ve just broken the problem into 2 subparts and the answers of both would be multiplied together and stored in the factorial(n) function which would then be printed at last.

To understand this functional approach better, let us have a look at the pseudocode given below:

int factorial(n)

{

if(n == 0)

{

return 1;

}

return n \* factorial(n-1);

}

main()

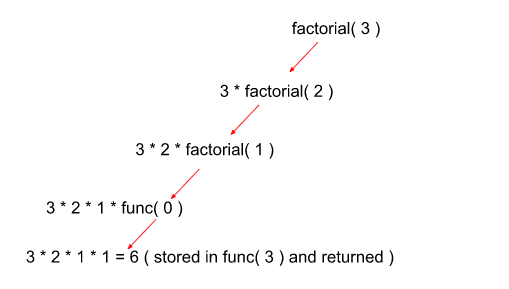
{

input(n);

factorial(n);

}

**Recursion Tree for the following problem can be represented as follows :**



class Recursion {

static int factorial(int n){

// Base Condition.

if(n == 0){

return 1;

}

// Problem broken down into 2 parts and then combined.

return n \* factorial(n-1);

}

public static void main(String[] args) {

// Here, let’s take the value of n to be 3.

int n = 3;

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

}

}

**Output**

6

**Time Complexity:** O(N) { Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N) }

**Space Complexity:** O(N) { In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space }.

# Reverse a given Array

**Problem Statement:** You are given an array. The task is to reverse the array and print it.

**Examples:**

**Example 1:**

**Input:** N = 5, arr[] = {5,4,3,2,1}

**Output:** {1,2,3,4,5}

**Explanation:** Since the order of elements gets reversed the first element will occupy the fifth position, the second element occupies the fourth position and so on.

**Example 2:**

**Input:** N=6 arr[] = {10,20,30,40}

**Output:** {40,30,20,10}

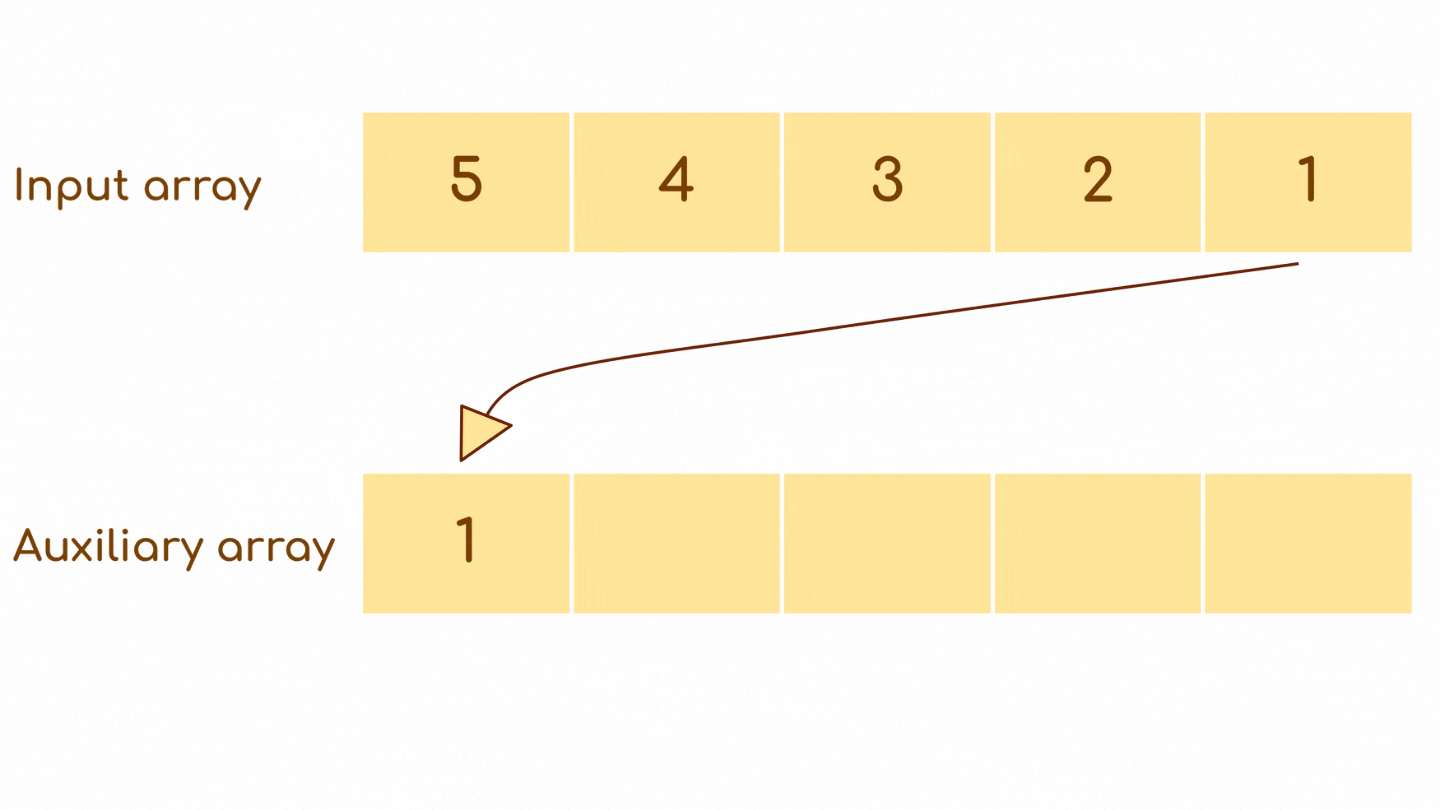
**Explanation:** Since the order of elements gets reversed the first element will occupy the fifth position, the second element occupies the fourth position and so on.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Using an extra array.**

**Approach**: Declare an array,ans[] of the same size as the input array. Iterate from the back of the input array while storing the elements in ans[]  in opposite direction.



**Code:**

public class Main {

//Function to print array

static void printArray(int ans[], int n) {

System.out.print("Reversed array is:- \n");

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

System.out.print(ans[i] + " ");

}

}

//Function to reverse array using an auxiliary array

static void reverseArray(int arr[], int n) {

int[] ans = new int[n];

for (int i = n - 1; i >= 0; i--) {

ans[n - i - 1] = arr[i];

}

printArray(ans, n);

}

public static void main(String[] args) {

int n = 5;

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

reverseArray(arr, n);

}

}

**Output:**

The reversed array is:-  
1 2 3 4 5

**Time Complexity: O(n),**single-pass for reversing array.

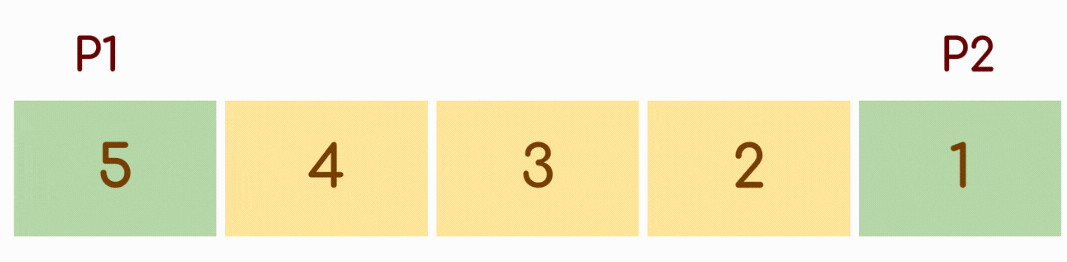
**Space Complexity: O(n),** for the extra array used.

**Solution 2: Space-optimized iterative method**

**Approach:**Unlike the previous method we use the same array to obtain the result. Follow the steps below.

* Keep a pointer p1  at the first index and another p2 at the last index of the array.
* Swap the elements pointed by p1 and p2, Post swapping increment p1 and decrement p2.
* This process is repeated for only the first n/2 elements where n is the length of array.

Note: Swapping all the n elements instead of n/2 elements leaves the array unaltered.



**Code:**

public class Main {

//Function to print array

static void printArray(int arr[], int n) {

System.out.print("Reversed array is:- \n");

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

System.out.print(arr[i] + " ");

}

}

//Function to reverse array

static void reverseArray(int arr[], int n) {

int p1 = 0, p2 = n - 1;

while (p1 < p2) {

int tmp = arr[p1];

arr[p1] = arr[p2];

arr[p2] = tmp;

p1++;

p2--;

}

printArray(arr, n);

}

public static void main(String[] args) {

int n = 5;

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

reverseArray(arr, n);

}

}

**Output:**

The reversed array is:-  
1 2 3 4 5

**Time Complexity: O(n)**, single-pass involved.

**Space Complexity: O(1)**

**Solution 3: Recursive method**

**Approach**: The recursive method has an approach almost similar to the iterative one. The approach has been broken down into some steps for simplicity.

1. Create a function that takes an array, start index, and end index of the array as parameters.
2. Swap the elements present  at the start and end index,
3. The portion of the array left to be reversed is arr[start+1,end-1]. Make a recursive call to reverse the rest of the array. While calling recursion pass **start +1**and**ends – 1**as parameters for the shrunk array. Repeat step 2.
4. Continue recursion as long as the ‘start < end’ condition is satisfied. This is the base case for our recursion.

**Code:**

public class Main {

//Function to print array

static void printArray(int arr[], int n) {

System.out.print("Reversed array is:- \n");

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

System.out.print(arr[i] + " ");

}

}

//Function to reverse array using recursion

static void reverseArray(int arr[], int start, int end) {

if (start < end) {

int tmp = arr[start];

arr[start] = arr[end];

arr[end] = tmp;

reverseArray(arr, start + 1, end - 1);

}

}

public static void main(String[] args) {

int n = 5;

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

reverseArray(arr, 0, n - 1);

printArray(arr, n);

}

}

**Output:**

The reversed array is:-  
1 2 3 4 5

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**Solution 4: Using library function (New Approach)**

**Approach**: C++ and Java have inbuilt functions to reverse an array.

**For C++:-**

The **std::reverse** function in C++ is predefined in a header file algorithm.

**Syntax: array\_name.reverse(BidirectionalIterator first, BidirectionalIterator last)**

Bidirectional iterators first and last indicate the starting and ending positions in the array. Elements that lie in this range are reversed. Note that this range includes the first but **excludes the last element**.

**For Java:-**

The reverse method in java can be imported from the Collections class present in java.util package.

Note:

1. Since this method expects an **object** as a parameter we have to convert the array into a list object using **asList()**.
2. Java collections require wrapper classes instead of primitive data types. In order to reverse an integer array use **Integer** instead of int.

**Syntax:** **Collections.reverse(class\_obj);**

**Code:**

import java.util.\*;

public class Main {

//Function to print array

static void printArray(Integer arr[], int n) {

System.out.print("Reversed array is:- \n");

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

System.out.print(arr[i] + " ");

}

}

//Reverse array using library function

static void reverseArray(Integer arr[]) {

//fetching array as list object

//reversing the fetched object

Collections.reverse(Arrays.asList(arr));

}

public static void main(String[] args) {

int n = 5;

Integer arr[] = {5,4,3,2,1};

reverseArray(arr);

printArray(arr, n);

}

}

**Output:**

The reversed array is:-  
1 2 3 4 5

**Time Complexity: O(n)**

**Space Complexity: O(1)**

# Check if the given String is Palindrome or not

**Problem Statement:** “Given a string, check if the string is palindrome or not.”  A string is said to be palindrome if the reverse of the string is the same as the string.

**Examples:**

**Example 1:**

**Input:** Str = “ABCDCBA”

**Output:** Palindrome

**Explanation:** String when reversed is the same as string.

**Example 2:**

**Input:** Str = “TAKE U FORWARD”

**Output:** Not Palindrome

**Explanation:** String when reversed is not the same as string.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Approach**:

Run a for loop till half the length of the string in order to check the first and last character of the string.

And check if the first and last elements of the string are equal. And then move both pointers first pointer forward and last pointer backward.

Here we are achieving this with the help of ‘i’ which moves with them for a loop. First element can be get by str[i] and last element by str[str.length() – i – 1]). If this condition gets executed then it is not palindrome and returns false.

If the loop ends after traversing elements till half of the length then, it is Palindrome and returns false.





**Code:**

import java.io.\*;

import java.util.Arrays;

class Test {

static private boolean isPalindrome(String s) {

int left = 0, right = s.length()-1;

while(left<right)

{

char l = s.charAt(left), r = s.charAt(right);

if(!Character.isLetterOrDigit(l))

left++;

else if(!Character.isLetterOrDigit(r))

right--;

else if(Character.toLowerCase(l)!=Character.toLowerCase(r))

return false;

else {

left++;

right--;

}

}

return true;

}

public static void main(String[] args) {

String str = "ABCDCBA";

boolean ans = isPalindrome(str);

if (ans == true) {

System.out.println("Palindrome");

} else {

System.out.println("Not Palindrome");

}

}

}

**Output:** Palindrome

**Time Complexity:  O(N)**

**Space Complexity: O(1)**

**Recursive Approach:**

* In this approach, we check the string using functional recursion where firstly, the letters on the two ends of the string (start, end) are compared to see if they’re the same or not.
* If they’re the same then we simply call recursion for the next elements (start+1, end-1) and so on until the start becomes greater than or equal to the end.
* If at any point the start and the end differ, we return false stating that the string is not a palindrome.
* Otherwise, if the base condition is reached, then the string is obviously a palindrome and we return true.

Let us understand it through a recursive code:

**Code:**

class Recursion {

static boolean palindrome(int i, String s){

// Base Condition

// If i exceeds half of the string, means all the elements

// are compared, we return true.

if(i>=s.length()/2) return true;

// If start is not equal to end, not palindrome.

if(s.charAt(i)!=s.charAt(s.length()-i-1)) return false;

// If both characters are same, increment i and check start+1 and end-1.

return palindrome(i+1,s);

}

public static void main(String[] args) {

// Example string.

String s = "madam";

System.out.println(palindrome(0,s));

}

}

**Output:**

True

**Time Complexity:**O(N) { Precisely, O(N/2) as we compare the elements N/2 times and swap them}.

**Space Complexity:** O(1) { The elements of the given array are swapped in place so no extra space is required}.

# Print Fibonacci Series up to Nth term

**Problem Statement:** Given an integer N. Print the Fibonacci series up to the Nth term.

**Examples:**

**Example 1:**

**Input:** N = 5

**Output:** 0 1 1 2 3 5

**Explanation:** 0 1 1 2 3 5 is the fibonacci series up to 5th term.(0 based indexing)

**Example 2:**

**Input:** 6

**Output:** 0 1 1 2 3 5 8

**Explanation:** 0 1 1 2 3 5 8 is the fibonacci series upto 6th term.(o based indexing)

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Naive approach**

**Intuition:** As we know fib(i) = fib(i-1) + fib(i-2).Simply iterate and go on calculating the ith term in the series.

**Approach**:

* Take an array say fib of size n+1.The 0th term and 1st term are 0 and 1 respectively.So fib(0)=0 and fib(1)=1.
* Now iterate from 2 to n and calculate fib(n).fib(n)=fib(n-1) + fib(n-2).
* Then print fib(0) + fib(1) + …………fib(n).

**Code:**

public class TUF {

public static void main(String args[]) {

int n = 5;

if (n == 0) {

System.out.println(0);

} else {

int fib[] = new int[n + 1];

fib[0] = 0;

fib[1] = 1;

for (int i = 2; i <= n; i++) {

fib[i] = fib[i - 1] + fib[i - 2];

}

System.out.println("The Fibonacci Series up to "+n+"th term:");

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

System.out.print(fib[i] + " ");

}

}

}

}

**Output:**

The Fibonacci Series up to 5th term:  
0 1 1 2 3 5

**Time Complexity:**O(n)+O(n), for calculating and printing the Fibonacci series.

**Space Complexity:**O(n), for storing Fibonacci series.

**Solution 2: Space optimized**

**Intuition**: For calculating the ith term we only need the last and second last term i.e (i-1)th and (i-2)th term, so we don’t need to maintain the whole array.

**Approach:**

* Take two variables last and secondLast for storing (i-1)th and (i-2)th term.
* Now iterate from 2 to n and calculate the ith term. ith term is last + secondLast term.
* Then update secondLast term to the last term and the last term to ith term as we iterate.

**Code:**

public class TUF {

public static void main(String args[]) {

int n = 5;

if (n == 0) {

System.out.println("The Fibonacci Series up to "+n+"th term:");

System.out.print(0);

} else {

int secondLast = 0;

int last = 1;

System.out.println("The Fibonacci Series up to "+n+"th term:");

System.out.print(secondLast + " " + last + " ");

int cur;

for (int i = 2; i <= n; i++) {

cur = last + secondLast;

secondLast = last;

last = cur;

System.out.print(cur + " ");

}

}

}

}

**Output:**

The Fibonacci Series up to 5th term:  
0 1 1 2 3 5

**Time Complexity:**O(N).As we are iterating over just one for a loop.

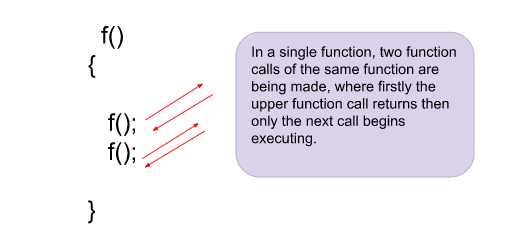
**Space Complexity:** O(1).

**Solution 3**

**Intuition:**

In this approach, instead of printing the Fibonacci series till N, we’re going to print the Nth Fibonacci number using functional recursion with multiple function calls.

One may wonder how multiple-function calls work. Let’s understand through an illustration below:



Similar kinds of multiple-function calls would be used in implementing the Fibonacci series where any Nth Fibonacci number can be written as a sum of (N-1)th and (N-2)th Fibonacci numbers. So, the function result would look like this:

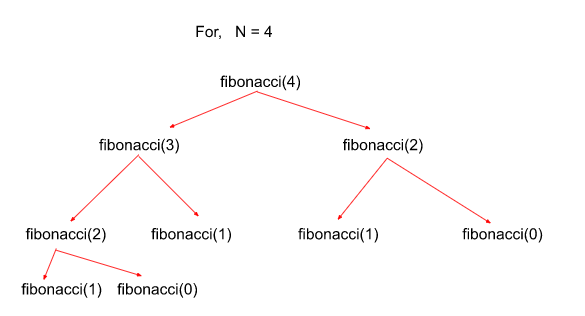
Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

Results from both the function calls would be summed and returned to the main function call.

**Approach:**

* Similar to all the recursion problems we’ve seen before, we need a base case in this problem too in order for recursion to not go infinitely. Here, we notice that the Fibonacci series start from N = 1, where we initialize its value as 1.
* Assume Fibonacci(0) = 0. So, Fibonacci(2) = 1+0 = 1 as the Nth Fibonacci number is the sum of the previous two Fibonacci numbers.
* Similarly, we call Fibonacci(N-1) and Fibonacci(N-2) and return their sum. Both the function calls Fibonacci(N-1) and Fibonacci(N-2) would be computed individually one by one until the base condition is reached for both and then they return back to the main function.

Let us see the recursion tree for the following problem to get an even better understanding:



**Code:**

class Recursion {

static int fibonacci(int N){

// Base Condition.

if(N <= 1){

return N;

}

// Problem broken down into 2 functional calls

// and their results combined and returned.

int last = fibonacci(N-1);

int slast = fibonacci(N-2);

return last + slast;

}

public static void main(String[] args) {

// Here, let’s take the value of N to be 4.

int N = 4;

System.out.println(fibonacci(N));

}

}

**Time Complexity: O(2^N)**{ This problem involves two function calls for each iteration which further expands to 4 function calls and so on which makes worst-case time complexity to be exponential in nature }.

**Space Complexity: O(N)**{ At maximum there could be N function calls waiting in the recursion stack since we need to calculate the Nth Fibonacci number for which we also need to calculate (N-1) Fibonacci numbers before it }.

# Power Set: Print all the possible subsequences of the String

**Problem Statement:** Given a string, find all the possible subsequences of the string.

**Examples:**

**Example 1:**

**Input:** str = "abc"

**Output:** a ab abc ac b bc c

**Explanation:** Printing all the 7 subsequence for the string "abc".

**Example 2:**

**Input:** str = "aa"

**Output:** a a aa

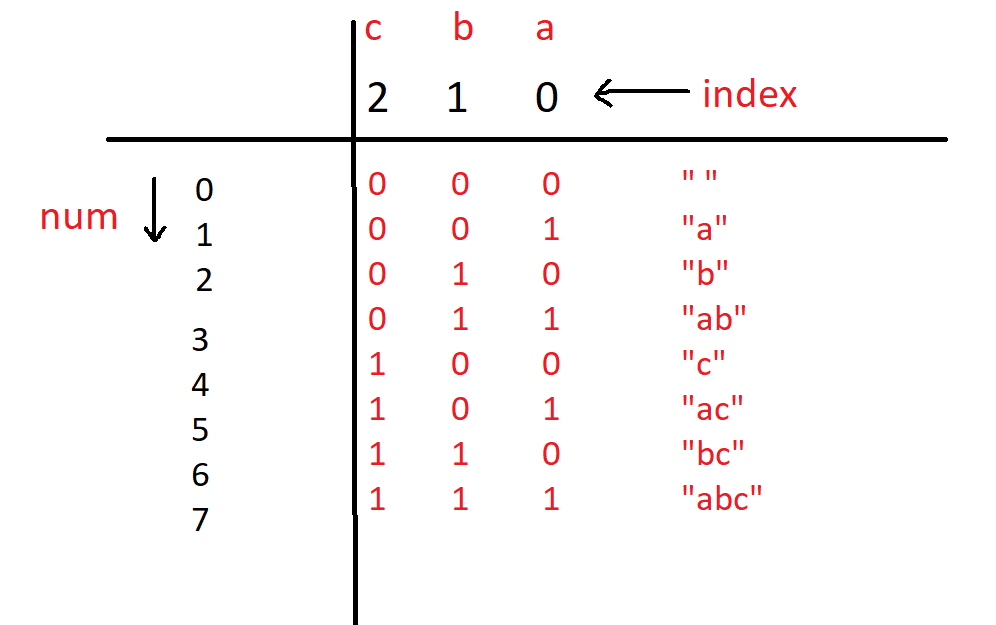
**Explanation:** Printing all the 3 subsequences for the string "aa"

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

### **Solution 1: Using bit manipulation**

**Approach**: Prerequisites: To check whether the ith bit is set or not.If n&(1<<i) != 0,then the ith bit is set.  
First, write down all the numbers from 0 to 2^(n)-1 and their bit representation.0 means I am not picking the character in my subsequence, and 1 means I am picking the character.



Basically, we traverse from 0 to 2^(n)-1 and check for every number if their bit is set or not. If the bit is set add that character to your subsequence.

**Code:**

import java.util.\*;

class TUF{

static ArrayList<String> AllPossibleStrings(String s) {

int n = s.length();

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

for (int num = 0; num < (1 << n); num++) {

String sub = "";

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

//check if the ith bit is set or not

if ((num & (1 << i))!=0) {

sub += s.charAt(i);

}

}

if (sub.length() > 0) {

ans.add(sub);

}

}

Collections.sort(ans);

return ans;

}

public static void main(String args[])

{

String s="abc";

ArrayList<String>ans = AllPossibleStrings(s);

//printint all the subsequence.

System.out.println("All possible subsequences are ");

for (String it : ans) {

System.out.print( it+" ");

}

}

}

**Output:**

All possible subsequences are  
a ab abc ac b bc c

**Time Complexity**: O(2^n \* n)

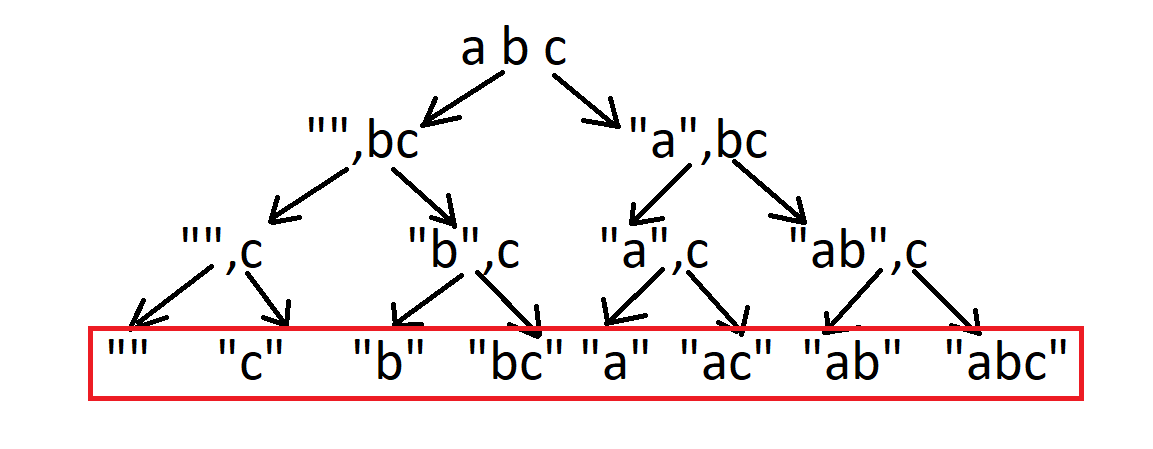
**Reason:** O(2^n) for the outer for loop and O(n) for the inner for loop.

**Space Complexity:** O(1)

### **Solution 2: Using recursion(Backtracking)**

**Intuition:**Since we are generating subsets two cases will be possible, either you can pick the character or you cannot pick the character and move to the next character.

**Approach:**

****

* Maintain a temp string (say f),which is empty initally.
* Now you have two options,either you can pick the character or not pick the character and move to the next index.
* Firstly we pick the character at ith index and then move to the next index.(f + s[i])
* If the base condition is hit,i.e i==s.length() ,then we print the temp string and return.
* Now while backtracking we have to pop the last character since now we have to implement the non-pick condition and then move to next index.

**Code:**

import java.util.\*;

class TUF{

static void solve(int i, String s, String f) {

if (i == s.length()) {

System.out.print(f+" ");

return;

}

//picking

//f = f + s.charAt(i);

solve(i + 1, s, f+s.charAt(i));

//poping out while backtracking

//f.pop\_back();

solve(i + 1, s, f);

}

public static void main(String args[]) {

String s = "abc";

String f = "";

System.out.println("All possible subsequences are: ");

solve(0, s, f);

}

}

**Output:**

All possible subsequences are:  
abc ab ac a bc b c

**Time Complexity:** O(2^n)

**Space Complexity:**O(n), recursion stack.

# Merge Sort Algorithm

**Problem**:  Given an array of size n, sort the array using **Merge Sort**.

**Examples:**

**Example 1:**

**Input:** N=5, arr[]={4,2,1,6,7}

**Output:** 1,2,4,6,7,

**Example 2:**

**Input:** N=7,arr[]={3,2,8,5,1,4,23}

**Output:** 1,2,3,4,5,8,23

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution :**

**Intuition:**

1. Merge Sort is a **divide and conquers algorithm**, it divides the given array into equal parts and then merges the 2 sorted parts.
2. There are 2 main functions :  
   **merge():**This function is used to merge the 2 halves of the array. It assumes that both parts of the array are sorted and merges both of them.  
   **mergeSort():** This function divides the array into 2 parts. low to mid and mid+1 to high where,

low = leftmost index of the array

high = rightmost index of the array

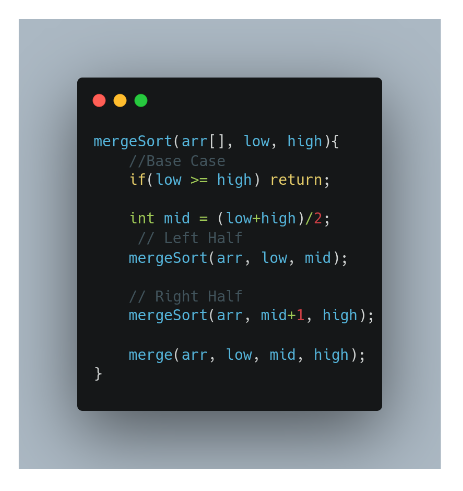
mid = Middle index of the array

1. We recursively split the array, and go from top-down until all sub-arrays size becomes 1.

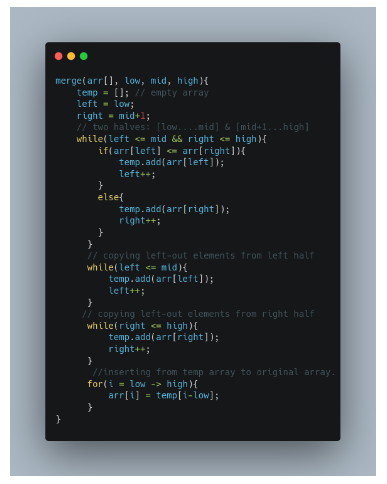
**Approach :**

**Approach:**

* We will be creating 2 functions mergeSort() and merge().
* **mergeSort(arr[], low, high):**
  1. In order to implement merge sort we need to first divide the given array into two halves. Now, if we carefully observe, we need not divide the array and create a separate array, but we will divide the array’s range into halves every time. For example, the given range of the array is 0 to 4(0-based indexing). Now on the first go, we will divide the range into half like (0+4)/2 = 2. So, the range of the left half will be 0 to 2 and for the right half, the range will be 3 to 4. Similarly, if the given range is low to high, the range for the two halves will be low to mid and mid+1 to high, where mid = (low+high)/2. This process will continue until the range size becomes.
  2. So, in mergeSort(), we will divide the array around the middle index(rather than creating a separate array) by making the recursive call :  
     1. mergeSort(arr,low,mid)   [Left half of the array]  
     2. mergeSort(arr,mid+1,high)[Right half of the array]  
     where low = leftmost index of the array, high = rightmost index of the array, and mid = middle index of the array.
  3. Now, in order to complete the recursive function, we need to write the base case as well. We know from step 2.1, that our recursion ends when the array has only 1 element left. So, the leftmost index, low, and the rightmost index high become the same as they are pointing to a single element.  
     **Base Case:**if(low >= high) return;
* **Pseudocode:**



* **merge(arr[], low, mid, high):**
  1. In the merge function, we will use a temp array to store the elements of the two sorted arrays after merging. Here, the range of the left array is low to mid and the range for the right half is mid+1 to high.
  2. Now we will take two pointers left and right, where left starts from low and right starts from mid+1.
  3. Using a while loop( while(left <= mid && right <= high)), we will select two elements, one from each half, and will consider the smallest one among the two. Then, we will insert the smallest element in the temp array.
  4. After that, the left-out elements in both halves will be copied as it is into the temp array.
  5. Now, we will just transfer the elements of the temp array to the range low to high in the original array. The pseudo code will look like the following:



**The red number shows the order of the steps in recursion calls.**

**Code:**

import java.util.\*;

class Solution {

private static void merge(int[] arr, int low, int mid, int high) {

ArrayList<Integer> temp = new ArrayList<>(); // temporary array

int left = low; // starting index of left half of arr

int right = mid + 1; // starting index of right half of arr

//storing elements in the temporary array in a sorted manner//

while (left <= mid && right <= high) {

if (arr[left] <= arr[right]) {

temp.add(arr[left]);

left++;

} else {

temp.add(arr[right]);

right++;

}

}

// if elements on the left half are still left //

while (left <= mid) {

temp.add(arr[left]);

left++;

}

// if elements on the right half are still left //

while (right <= high) {

temp.add(arr[right]);

right++;

}

// transfering all elements from temporary to arr //

for (int i = low; i <= high; i++) {

arr[i] = temp.get(i - low);

}

}

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

if (low >= high) return;

int mid = (low + high) / 2 ;

mergeSort(arr, low, mid); // left half

mergeSort(arr, mid + 1, high); // right half

merge(arr, low, mid, high); // merging sorted halves

}

}

public class tUf {

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

int n = 7;

int arr[] = { 9, 4, 7, 6, 3, 1, 5 };

System.out.println("Before sorting array: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

Solution.mergeSort(arr, 0, n - 1);

System.out.println("After sorting array: ");

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

}

**Output:**

Before Sorting Array:  
9 4 7 6 3 1 5  
After Sorting Array:  
1 3 4 5 6 7 9

**Time complexity:**O(nlogn)

Reason: At each step, we divide the whole array, for that logn and we assume n steps are taken to get a sorted array, so overall time complexity will be nlogn

**Space complexity:** O(n)

Reason: We are using a temporary array to store elements in sorted order.

# Quick Sort Algorithm

**Problem Statement:**  Given an array of n integers, sort the array using the **Quicksort** method.

**Examples:**

**Example 1:**

**Input:** N = 5 , Arr[] = {4,1,7,9,3}

**Output:** 1 3 4 7 9

**Explanation:** After sorting the array becomes 1, 3, 4, 7, 9

**Example 2:**

**Input:** N = 8 , Arr[] = {4,6,2,5,7,9,1,3}

**Output:** 1 2 3 4 5 6 7 9

**Explanation:** After sorting the array becomes 1, 3, 4, 7, 9

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

[Problem Link](https://www.codingninjas.com/codestudio/problems/quick-sort_983625?utm_source=youtube&utm_medium=affiliate&utm_campaign=striver_Arrayproblems).

**Intuition:**

Quick Sort is a divide-and-conquer algorithm like [the Merge Sort](https://takeuforward.org/data-structure/merge-sort-algorithm/). But unlike Merge sort, this algorithm does not use any extra array for sorting(though it uses an auxiliary stack space). So, from that perspective, Quick sort is slightly better than Merge sort.

This algorithm is basically a repetition of two simple steps that are the following:

* Pick a pivot and place it in its correct place in the sorted array.
* Shift smaller elements(i.e. Smaller than the pivot) on the left of the pivot and larger ones to the right.

Now, let’s discuss the steps in detail considering the array {4,6,2,5,7,9,1,3}:

**Step 1:**The first thing is to choose the pivot. **A pivot is basically a chosen element of the given array.** The element or the pivot can be chosen by our choice. So, in an array a pivot can be any of the following:

* The first element of the array
* The last element of the array
* Median of array
* Any Random element of the array

After choosing the pivot(i.e. the element), we should place it in its correct position(i.e. The place it should be after the array gets sorted) in the array. For example, if the given array is {4,6,2,5,7,9,1,3}, the correct position of 4 will be the 4th position.

**Note:** Here in this tutorial, we have chosen the first element as our pivot. You can choose any element as per your choice.

**Step 2:**In step 2, we will shift the smaller elements(i.e. Smaller than the pivot) to the left of the pivot and the larger ones to the right of the pivot. In the example, if the chosen pivot is 4, after performing step 2 the array will look like: {3, 2, 1, 4, 6, 5, 7, 9}.

From the explanation, we can see that after completing the steps, pivot 4 is in its correct position with the left and right subarray unsorted. Now **we will apply these two steps on the left subarray and the right subarray recursively**. And we will continue this process until the size of the unsorted part becomes 1(as an array with a single element is always sorted).

So, from the above intuition, we can get a clear idea that we are going to use recursion in this algorithm.

To summarize, the main intention of this process is to place the pivot, after each recursion call, at its final position, where the pivot should be in the final sorted array.

**Approach**:

Now, let’s understand how we are going to implement the logic of the above steps. In the intuition, we have seen that the given array should be broken down into subarrays. But while implementing, we are not going to break down and create any new arrays. Instead, we will specify the range of the subarrays using two indices or pointers(i.e. **low**pointer and **high**pointer) each time while breaking down the array.

The algorithm steps are the following for the **quickSort()** function:

1. Initially, the **low** points to the first index and the **high** points to the last index(as the range is n i.e. the size of the array).
2. After that, we will get the index(where the pivot should be placed after sorting) while shifting the smaller elements on the left and the larger ones on the right using a partition() function discussed later.  
   Now, this index can be called the **partition index** as it creates a partition between the left and the right unsorted subarrays.
3. After placing the pivot in the partition index(within the partition() function specified), we need to call the function quickSort() for the left and the right subarray recursively. So, **the range of the left subarray will be [low to (partition index – 1)]** and **the range of the right subarray will be [(partition index + 1) to high].**
4. This is how the recursion will continue until the range becomes 1.

**Pseudocode:**

So, the pseudocode will look like the following:

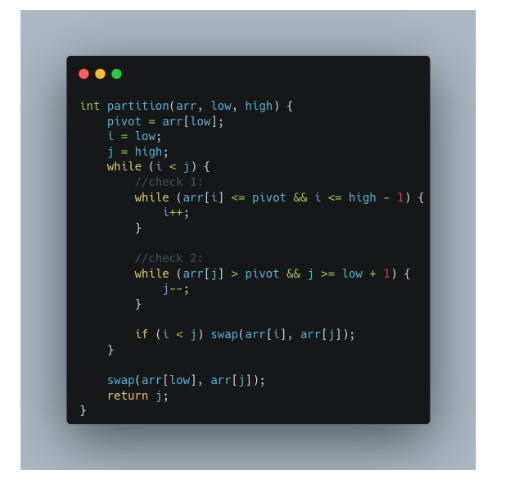


Now, let’s understand how to implement the partition() function to get the partition index.

1. Inside the function, we will first select the pivot(i.e. arr[low] in our case).
2. Now, we will again take two-pointers i and j. The i pointer points to low and the j points to high.
3. Now, the pointer i will move forward and find the first element that is greater than the pivot. Similarly, the pointer j will move backward and find the first element that is smaller than the pivot.  
   Here, we need to add some checks like i <= high-1 and j >= low+1. Because it might happen that i is standing at high and trying to proceed or j is standing at low and trying to exceed.
4. Once we find such elements i.e. arr[i] > pivot and arr[j] < pivot, and i < j, we will swap arr[i] and arr[j].
5. We will continue step 3 and step 4, until j becomes smaller than i.
6. Finally, we will swap the pivot element(i.e. arr[low]) with arr[j] and will return the index j i.e. the partition index.

**Pseudocode:**

So, the pseudocode will look like the following:



**Note:**In the function, we have kept the elements equal to the pivot on the left side. If you choose to place them on the right, check 1 will be arr[i] < pivot and check 2 will be arr[j] >= pivot.

**Note:**If you wish to see the dry run, refer to the video attached below.

**Code:**

* C++ Code
* Java Code

import java.util.\*;

class Solution {

static int partition(List<Integer> arr, int low, int high) {

int pivot = arr.get(low);

int i = low;

int j = high;

while (i < j) {

while (arr.get(i) <= pivot && i <= high - 1) {

i++;

}

while (arr.get(j) > pivot && j >= low + 1) {

j--;

}

if (i < j) {

int temp = arr.get(i);

arr.set(i, arr.get(j));

arr.set(j, temp);

}

}

int temp = arr.get(low);

arr.set(low, arr.get(j));

arr.set(j, temp);

return j;

}

static void qs(List<Integer> arr, int low, int high) {

if (low < high) {

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

qs(arr, low, pIndex - 1);

qs(arr, pIndex + 1, high);

}

}

public static List<Integer> quickSort(List<Integer> arr) {

// Write your code here.

qs(arr, 0, arr.size() - 1);

return arr;

}

}

public class tUf {

public static void main(String args[]) {

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

arr = Arrays.asList(new Integer[] {4, 6, 2, 5, 7, 9, 1, 3});

int n = arr.size();

System.out.println("Before Using insertion Sort: ");

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

System.out.print(arr.get(i) + " ");

}

System.out.println();

arr = Solution.quickSort(arr);

System.out.println("After insertion sort: ");

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

System.out.print(arr.get(i) + " ");

}

System.out.println();

}

}

**Output:**Before Using quick Sort:  
4 6 2 5 7 9 1 3  
After Using quick sort:  
1 2 3 4 5 6 7 9

**Time Complexity:**O(N\*logN), where N = size of the array.

**Reason:** At each step, we divide the whole array, for that logN and n steps are taken for partition() function, so overall time complexity will be N\*logN.

The following recurrence relation can be written for Quick sort :

F(n) = F(k) + F(n-1-k)

Here k is the number of elements smaller or equal to the pivot and n-1-k denotes elements greater than the pivot.

There can be 2 cases :

**Worst Case** – This case occurs when the pivot is the greatest or smallest element of the array. If the partition is done and the last element is the pivot, then the worst case would be either in the increasing order of the array or in the decreasing order of the array.

**Recurrence:**  
**F(n) = F(0)+F(n-1)  or  F(n) = F(n-1) + F(0)**

**Worst Case Time complexity: O(n2)**

Best Case – This case occurs when the pivot is the middle element or near to middle element of the array.  
Recurrence :  
F(n) = 2F(n/2)  
  
Time Complexity for the best and average case: O(N\*logN)

**Space Complexity:**O(1) + O(N) auxiliary stack space.

# Combination Sum – 1

**Problem Statement:**

Given an array of distinct integers and a**target**, you have to return the list of all unique combinations where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from the given array an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is guaranteed that the number of unique combinations that sum up to **target** is less than **150**combinations for the given input.

**Examples:**

**Example 1:**

**Input:** array = [2,3,6,7], target = 7

**Output:** [[2,2,3],[7]]

**Explanation:** 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

**Example 2:**

**Input:** array = [2], target = 1

**Output:** []

**Explaination:** No combination is possible.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution: Recursion**

**Intuition:**

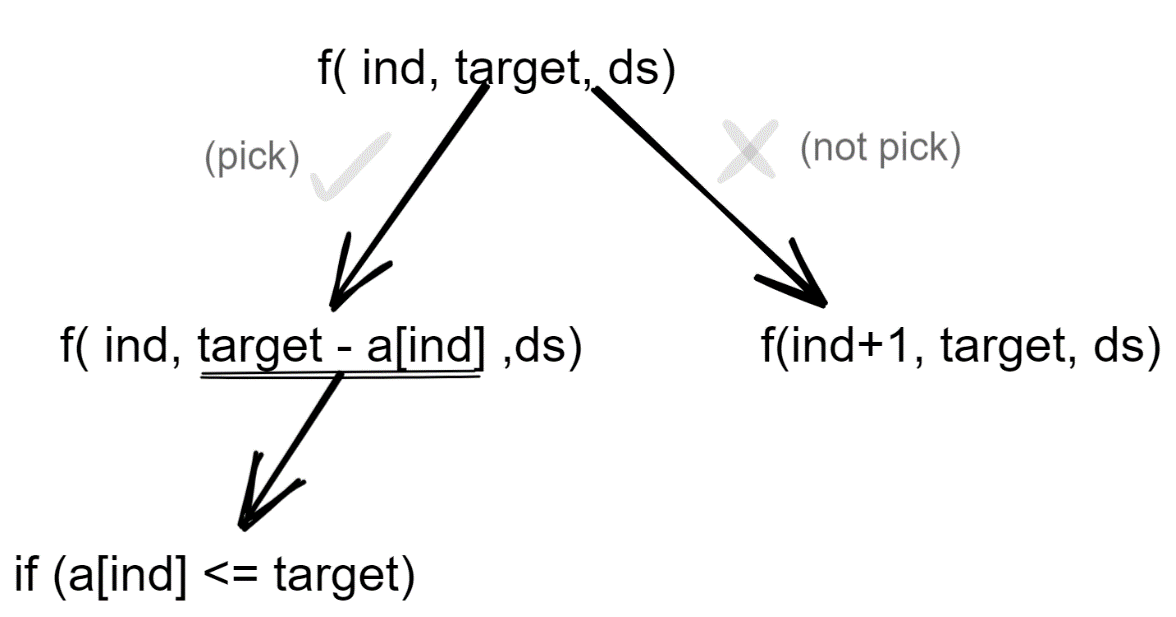
For questions like printing combinations or subsequences, the first thing that should strike your mind is recursion.

**How to think recursively?**

Whenever the problem is related to picking up elements from an array to form a combination, start thinking about the **“pick and non-pick”**approach.

**Approach**:

**Defining recursive function:**

****

Initially, the index will be 0, target as given and the data structure(vector or list) will be empty

Now there are 2 options viz to **pick or not pick**the current index element.

If you **pick** the element, again come back at the same index as multiple occurrences of the same element is possible so the target reduces to target – arr[index] (where target -arr[index]>=0)and also insert the current element into the data structure.

If you decide **not to pick** the current element, move on to the next index and the target value stays as it is. Also, the current element is not inserted into the data structure.

While backtracking makes sure to pop the last element as shown in the recursion tree below.

Keep on repeating this process while index < size of the array for a particular recursion call.

You can also stop the recursion when the target value is 0, but here a generalized version without adding too many conditions is considered.

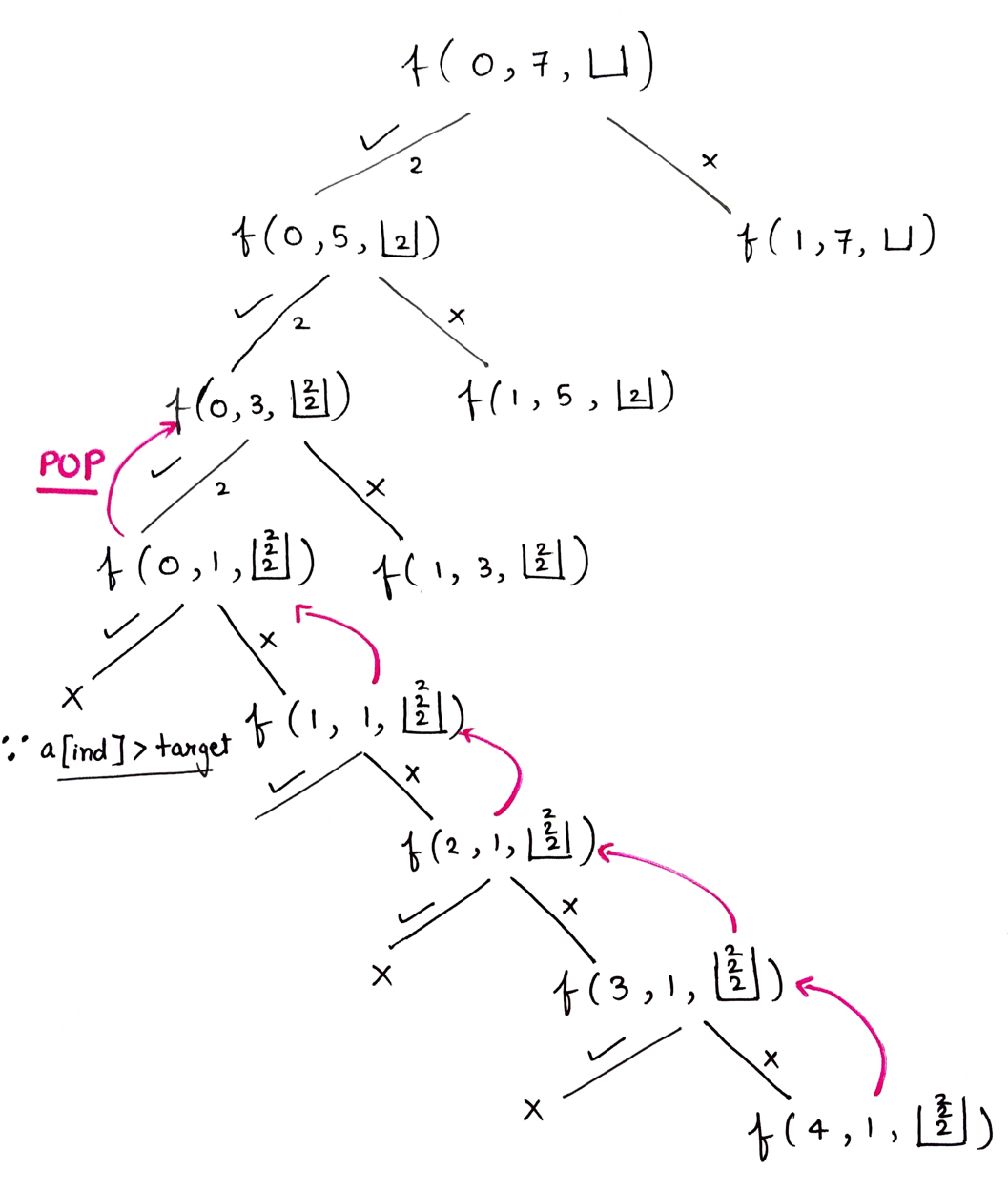
Using this approach, we can get all the combinations.

**Base condition**

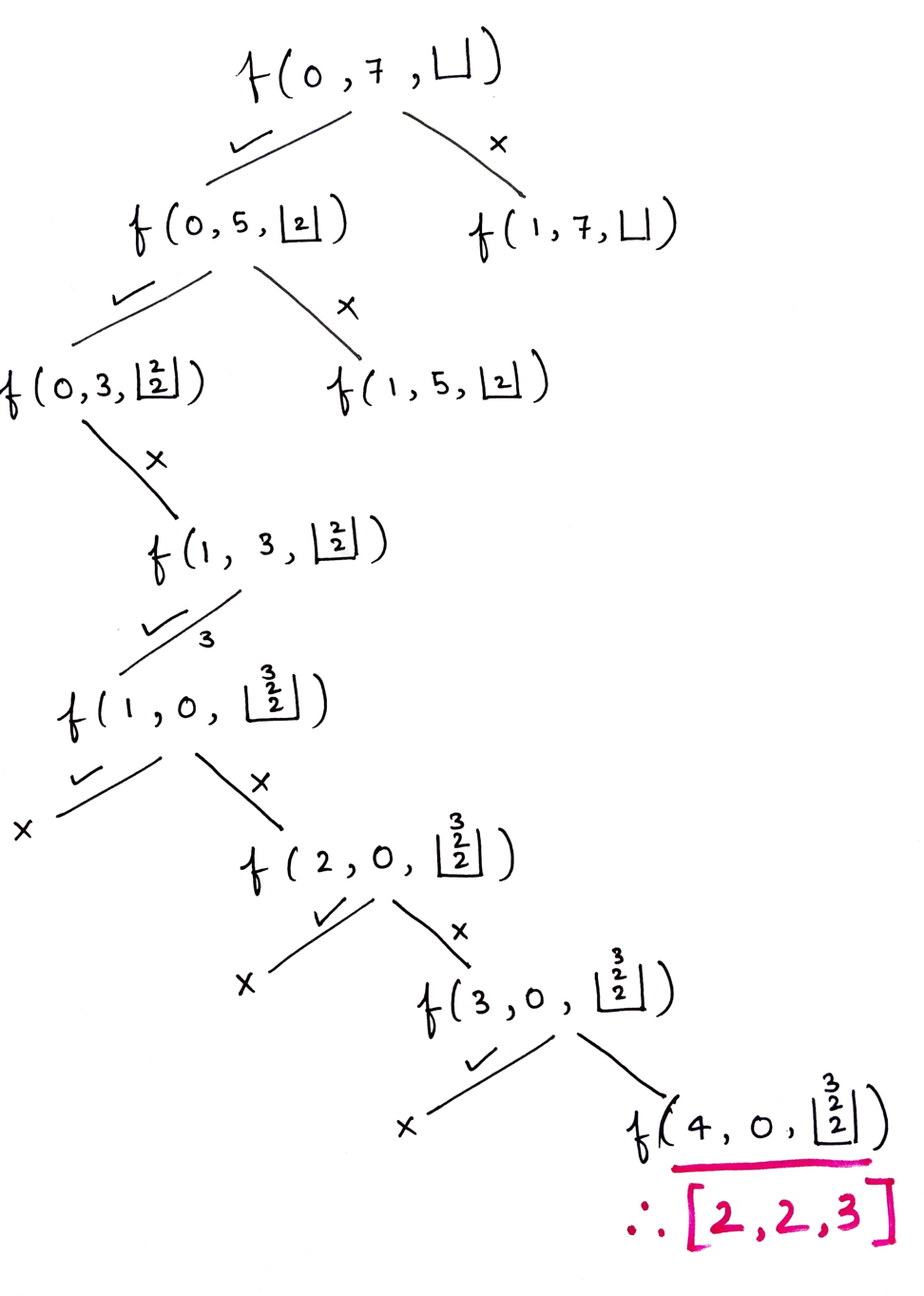
**If index== size of array and  target == 0 include the combination in our answer**

**Diagrammatic representation for Example 1:**

**Case 1:**



**Case 2:**



**Code:**

import java.io.\*;

import java.util.\*;

class Solution {

private void findCombinations(int ind, int[] arr, int target, List < List < Integer >> ans, List < Integer > ds) {

if (ind == arr.length) {

if (target == 0) {

ans.add(new ArrayList < > (ds));

}

return;

}

if (arr[ind] <= target) {

ds.add(arr[ind]);

findCombinations(ind, arr, target - arr[ind], ans, ds);

ds.remove(ds.size() - 1);

}

findCombinations(ind + 1, arr, target, ans, ds);

}

public List < List < Integer >> combinationSum(int[] candidates, int target) {

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

findCombinations(0, candidates, target, ans, new ArrayList < > ());

return ans;

}

}

public class Main {

public static void main(String[] args) {

int arr[] = {2,3,6,7};

int target = 7;

Solution sol = new Solution();

List < List < Integer >> ls = sol.combinationSum(arr, target);

System.out.println("Combinations are: ");

for (int i = 0; i < ls.size(); i++) {

for (int j = 0; j < ls.get(i).size(); j++) {

System.out.print(ls.get(i).get(j) + " ");

}

System.out.println();

}

}

}

**Output:**

Combinations are:  
2 2 3  
7

**Time Complexity: O(2^t \* k)**where t is thetarget, k is the average length

**Reason:** Assume if you were not allowed to pick a single element multiple times, every element will have a couple of options: pick or not pick which is 2^n different recursion calls, also assuming that the average length of every combination generated is k. (to put length k data structure into another data structure)

Why not (2^n) but (2^t) (where n is the size of an array)?

Assume that there is 1 and the target you want to reach is 10 so 10 times you can “pick or not pick” an element.

**Space Complexity: O(k\*x)**, k is the average length and x is the no. of combinations

# Combination Sum II – Find all unique combinations

In this article we will solve the most asked interview question “Combination Sum II – Find all unique combinations”.

**Problem Statement:**Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination.

**Note: The solution set must not contain duplicate combinations**.

**Examples:**

**Example 1:**

**Input:** candidates = [10,1,2,7,6,1,5], target = 8

**Output:**

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]]

**Explanation:** These are the unique combinations whose sum is equal to target.

**Example 2:**

**Input:** candidates = [2,5,2,1,2], target = 5

**Output:** [[1,2,2],[5]]

**Explanation:** These are the unique combinations whose sum is equal to target.

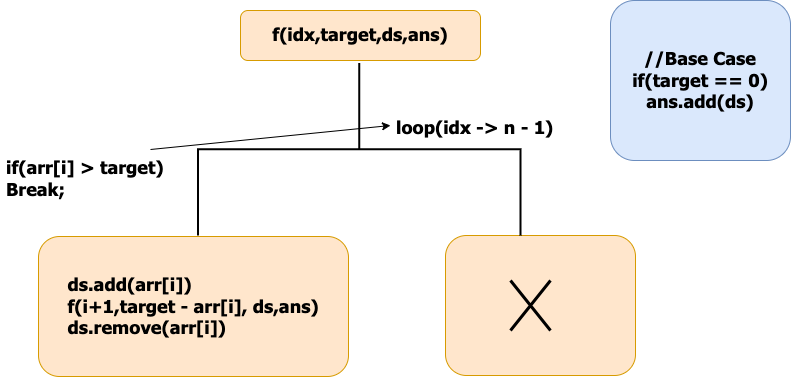
**Solution:**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Using extra space and time complexity**

**Approach:**

**Defining the Recursive Function:**



Before starting the recursive call make sure to sort the elements because the ans should contain the combinations in sorted order and should not be repeated.

Initially, We start with the index 0, At index 0 we have n – 1 way**to pick the first element of our subsequence.**

Check if the current index value can be added to our ds. If yes add it to the ds and move the index by 1. while moving the index skip the consecutive repeated elements because they will form duplicate sequences.

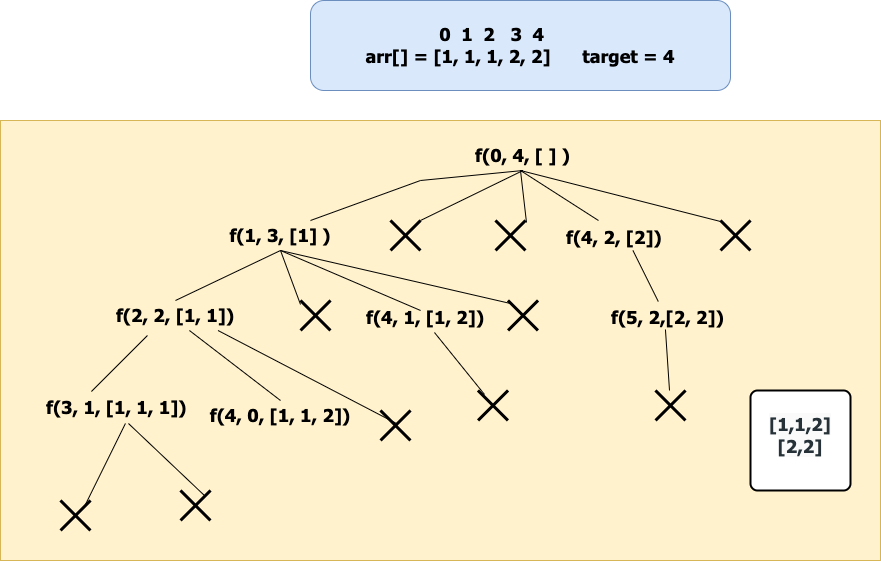
Reduce the target by arr[i],call the recursive call for f(idx + 1,target – 1,ds,ans) after the call make sure to pop the element from the ds.(By seeing the example recursive You will understand).

if(arr[i] > target) then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its**right will be in increasing order.**

**Base Condition:**

Whenever the target value is zero add the ds to the ans return.

**Representation of Recursive call for the example given below:**

****

**If we observe the recursive call for f(2,2,[1,1]) when it is returning the ds doesn’t include 1 so make sure to remove it from ds after the call.**

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF {

static void findCombinations(int ind, int[] arr, int target, List < List < Integer >> ans, List < Integer > ds) {

if (target == 0) {

ans.add(new ArrayList < > (ds));

return;

}

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

if (i > ind && arr[i] == arr[i - 1]) continue;

if (arr[i] > target) break;

ds.add(arr[i]);

findCombinations(i + 1, arr, target - arr[i], ans, ds);

ds.remove(ds.size() - 1);

}

}

public static List < List < Integer >> combinationSum2(int[] candidates, int target) {

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

Arrays.sort(candidates);

findCombinations(0, candidates, target, ans, new ArrayList < > ());

return ans;

}

public static void main(String args[]) {

int v[]={10,1,2,7,6,1,5};

List < List < Integer >> comb = combinationSum2(v, 8);

System.out.println(comb.toString().replace(",", " "));

}

}

**Output:**

[ [ 1 1 6 ][ 1 2 5 ][ 1 7 ][ 2 6 ] ]

**Time Complexity:O(2^n\*k)**

**Reason:**Assume if all the elements in the array are unique then the no. of subsequence you will get will be O(2^n). we also add the ds to our ans when we reach the base case that will take “k”//average space for the ds.

**Space Complexity:O(k\*x)**

**Reason:** if we have x combinations then space will be x\*k where k is the average length of the combination.

# Sudoku Solver

**Problem Statement:**

Given a 9×9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

         1. All the rows should be filled with numbers(1 – 9) exactly once.

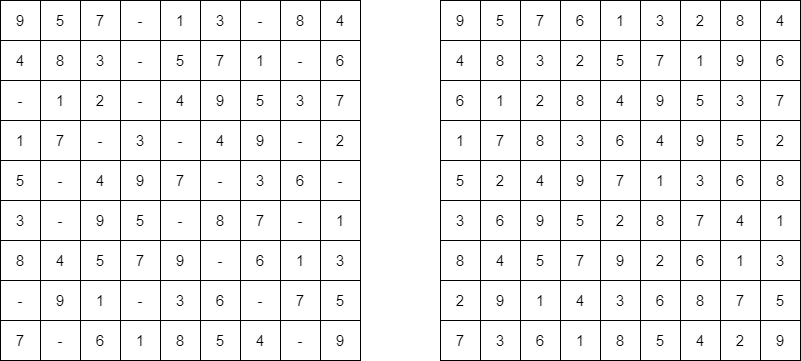
         2. All the columns should be filled with numbers(1 – 9) exactly once.

         3. Each 3×3 submatrix should be filled with numbers(1 – 9) exactly once.

**Note**: Character **‘.’** indicates empty cell.

**Example:**

**Input:** **Output:**



**Explanation:**

The empty cells are filled with the possible numbers. There can exist many such arrangements of numbers. The above solution is one of them. Let’s see how we can fill the cells below.

### **Solution**

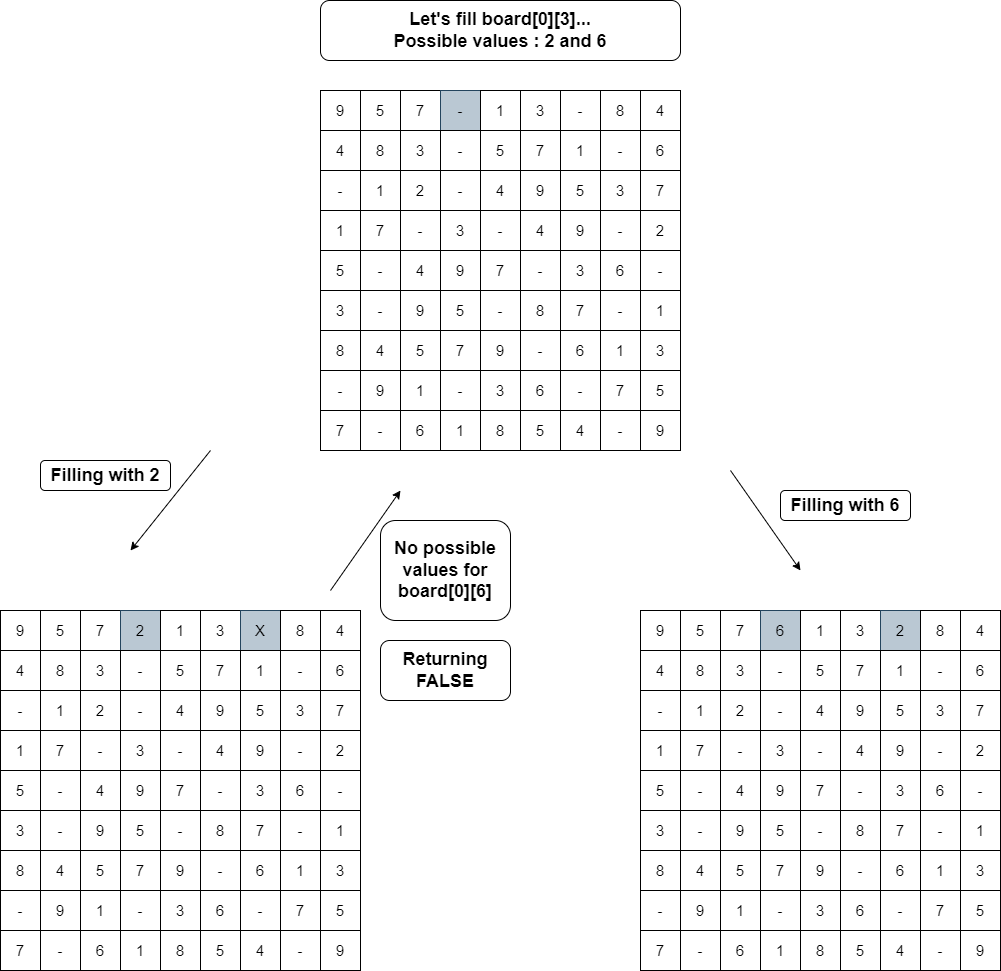
**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Intuition:**

Since we have to fill the empty cells with available possible numbers and we can also have multiple solutions, the main intuition is to try every possible way of filling the empty cells. And the more correct way to try all possible solutions is to use recursion. In each call to the recursive function, we just try all the possible numbers for a particular cell and transfer the updated board to the next recursive call.

**Approach**:

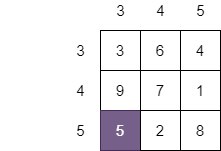
* Let’s see the step by step approach. Our main recursive function(solve()) is going to just do a plain matrix traversal of the sudoku board. When we find an empty cell, we pause and try to put all available numbers(1 – 9) in that particular empty cell.
* We need another loop to do that. But wait, we forgot one thing – the board has to satisfy all the conditions, right? So, for that we have another function(isValid()) which will check whether the number we have inserted into that empty cell will not violate any conditions.
* If it is violating, we try with the next number. If it is not, we call the same function recursively, but this time with the updated state of the board. Now, as usual it tries to fill the remaining cells in the board in the same way.
* Now we’ll come to the returning values. If at any point we cannot insert any numbers from 1 – 9 in a particular cell, it means the current state of the board is wrong and we need to backtrack. An important point to follow is, we need to return false to let the parent function(which is called this function) know that we cannot fill this way. This will serve as a hint to that function, that it needs to try with the next possible number. Refer to the picture below.



* If a recursive call returns true, we can assume that we found one possible way of filling and we simply do an **early return**.

**Validating Board**

* Now, let’s see how we are validating the sudoku board. After determining a number for a cell(at i’th row, j’th col), we try to check the validity. As we know, a valid sudoku needs to satisfy 3 conditions, we can use three loops. But we can do within a single loop itself. Let’s try to understand that.
* We loop from 0 to 8 and check the values – board[i][col](1st condition) and board[row][i](2nd condition), whether the number is already included. For the 3rd condition – the expression (3 \* (row / 3) + i / 3) evaluates to the row numbers of that 3×3 submatrix and the expression (3 \* (col / 3) + i % 3) evaluates to the column numbers.
* For eg, if row= 5 and col= 3, the cells visited are



It covers all the cells in the sub-matrix.

**Code:**

import java.util.\*;

class Solution {

public static boolean solveSudoku(char[][] board) {

for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++) {

if (board[i][j] == '.') {

for (char c = '1'; c <= '9'; c++) {

if (isValid(board, i, j, c)) {

board[i][j] = c;

if (solveSudoku(board))

return true;

else

board[i][j] = '.';

}

}

return false;

}

}

}

return true;

}

public static boolean isValid(char[][] board, int row, int col, char c) {

for (int i = 0; i < 9; i++) {

if (board[i][col] == c)

return false;

if (board[row][i] == c)

return false;

if (board[3 \* (row / 3) + i / 3][3 \* (col / 3) + i % 3] == c)

return false;

}

return true;

}

public static void main(String[] args) {

char[][] board= {

{'9', '5', '7', '.', '1', '3', '.', '8', '4'},

{'4', '8', '3', '.', '5', '7', '1', '.', '6'},

{'.', '1', '2', '.', '4', '9', '5', '3', '7'},

{'1', '7', '.', '3', '.', '4', '9', '.', '2'},

{'5', '.', '4', '9', '7', '.', '3', '6', '.'},

{'3', '.', '9', '5', '.', '8', '7', '.', '1'},

{'8', '4', '5', '7', '9', '.', '6', '1', '3'},

{'.', '9', '1', '.', '3', '6', '.', '7', '5'},

{'7', '.', '6', '1', '8', '5', '4', '.', '9'}

};

solveSudoku(board);

for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++)

System.out.print(board[i][j] + " ");

System.out.println();

}

}

}

**Output:**

9 5 7 6 1 3 2 8 4  
4 8 3 2 5 7 1 9 6  
6 1 2 8 4 9 5 3 7  
1 7 8 3 6 4 9 5 2  
5 2 4 9 7 1 3 6 8  
3 6 9 5 2 8 7 4 1  
8 4 5 7 9 2 6 1 3  
2 9 1 4 3 6 8 7 5  
7 3 6 1 8 5 4 2 9

**Time Complexity:** O(9(n ^ 2)), in the worst case, for each cell in the n2 board, we have 9 possible numbers.

**Space Complexity**: O(1), since we are refilling the given board itself, there is no extra space required, so constant space complexity.

# Subset Sum : Sum of all Subsets

**Problem Statement:** Given an array print all the sum of the subset generated from it, in the increasing order.

**Examples:**

**Example 1:**

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

**Output:** 0,1,2,3,5,6,7,8

**Explanation:** We have to find all the subset’s sum and print them.in this case the generated subsets are [ [], [1], [2], [2,1], [5], [5,1], [5,2]. [5,2,1],so the sums we get will be 0,1,2,3,5,6,7,8

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

**Output:** 0,1,2,3,3,4,5,6

**Explanation:** We have to find all the subset’s sum and print them.in this case the generated subsets are [ [], [1], [2], [2,1], [3], [3,1], [3,2]. [3,2,1],so the sums we get will be 0,1,2,3,3,4,5,6

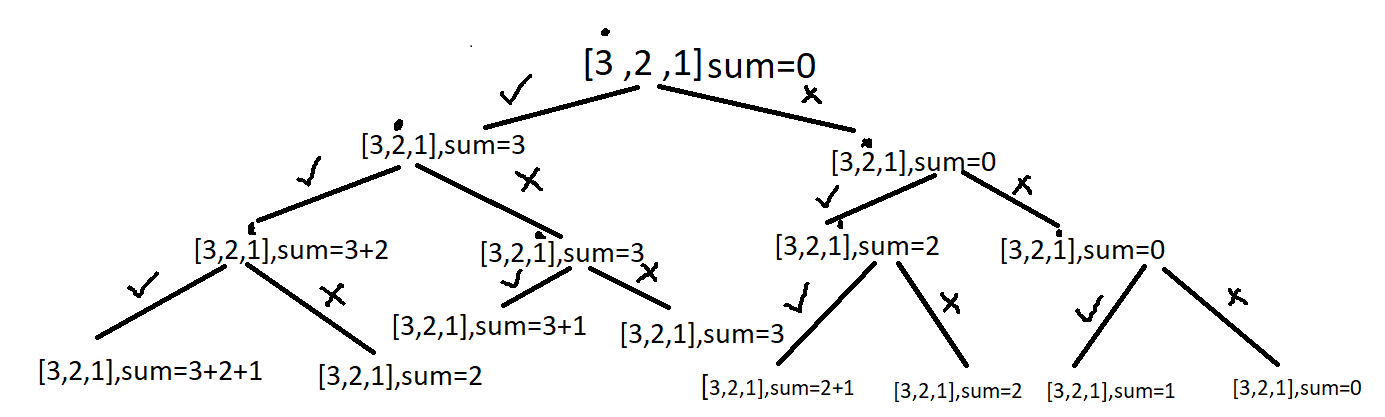
### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Using recursion**

**Intuition**: The main idea is that on every index you have two options either to select the element to add it to your subset(pick) or not select the element at that index and move to the next index(non-pick).

**Approach:**Traverse through the array and for each index solve for two arrays, one where you pick the element,i.e add the element to the sum or don’t pick and move to the next element, recursively, until the base condition. Here when you reach the end of the array is the base condition.



**Code:**

import java.util.\*;

class TUF {

static void subsetSumsHelper(int ind, int sum, ArrayList < Integer > arr, int N, ArrayList < Integer > sumSubset) {

if (ind == N) {

sumSubset.add(sum);

return;

}

// pick the element

subsetSumsHelper(ind + 1, sum + arr.get(ind), arr, N, sumSubset);

// Do-not pick the element

subsetSumsHelper(ind + 1, sum, arr, N, sumSubset);

}

static ArrayList < Integer > subsetSums(ArrayList < Integer > arr, int N) {

ArrayList < Integer > sumSubset = new ArrayList < > ();

subsetSumsHelper(0, 0, arr, N, sumSubset);

Collections.sort(sumSubset);

return sumSubset;

}

public static void main(String args[]) {

ArrayList < Integer > arr = new ArrayList < > ();

arr.add(3);

arr.add(1);

arr.add(2);

ArrayList < Integer > ans = subsetSums(arr, arr.size());

Collections.sort(ans);

System.out.println("The sum of each subset is ");

for (int i = 0; i < ans.size(); i++)

System.out.print(ans.get(i) + " ");

}

}

**Output:**

The sum of each subset is  
0 1 2 3 3 4 5 6

**Time Complexity**: O(2^n)+O(2^n log(2^n)). Each index has two ways. You can either pick it up or not pick it. So for n index time complexity for O(2^n) and for sorting it will take (2^n log(2^n)).

**Space Complexity:** O(2^n) for storing subset sums, since 2^n subsets can be generated for an array of size n.

# Subset – II | Print all the Unique Subsets

**Problem Statement:** Given an array of integers that **may contain duplicates** the task is to return all possible subsets. Return only **unique subsets**and they can be in any order.

**Examples:**

**Example 1:**

**Input:** array[] = [1,2,2]

**Output:** [ [ ],[1],[1,2],[1,2,2],[2],[2,2] ]

**Explanation:** We can have subsets ranging from length 0 to 3. which are listed above. Also the subset [1,2] appears twice but is printed only once as we require only unique subsets.

**Input:** array[] = [1]

**Output:** [ [ ], [1] ]

**Explanation:** Only two unique subsets are available

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Brute Force**

**Approach**:  
At every index, we make a decision whether to pick or not pick the element at that index. This will help us in generating all possible combinations but does not take care of the duplicates. Hence we will use a set to store all the combinations that will discard the duplicates.

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF {

static void printAns(List < String > ans) {

System.out.println("The unique subsets are ");

System.out.println(ans.toString().replace(",", " "));

}

public static void fun(int[] nums, int index, List < Integer > ds, HashSet < String > res) {

if (index == nums.length) {

Collections.sort(ds);

res.add(ds.toString());

return;

}

ds.add(nums[index]);

fun(nums, index + 1, ds, res);

ds.remove(ds.size() - 1);

fun(nums, index + 1, ds, res);

}

public static List < String > subsetsWithDup(int[] nums) {

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

HashSet < String > res = new HashSet < > ();

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

fun(nums, 0, ds, res);

for (String it: res) {

ans.add(it);

}

return ans;

}

public static void main(String args[]) {

int nums[]={1,2,2};

List < String > ans = subsetsWithDup(nums);

printAns(ans);

}

}

**Output:**

The unique subsets are  
[ [ ][ 1 ][ 1 2 ][ 1 2 2 ][ 2 ][ 2 2 ] ]

**Time Complexity: O( 2^n \*(k log (x) )). 2^n**for generating every subset and **k\*** **log( x)**  to insert every combination of average length k in a set of size x. After this, we have to convert the set of combinations back into a list of list /vector of vectors which takes more time.

**Space Complexity:  O(2^n \* k)**to store every subset of average length k. Since we are initially using a set to store the answer another **O(2^n \*k)** is also used.

**Solution 2: Optimal**

**Approach**:

In the previous method, we were taking extra time to store the unique combination with the help of a set.  To make the solution efficient we will have to decide on a method that will consider only the unique combinations without the help of additional data structure.

Lets  understand  with an example where arr = [1,2,2 ].

Initially start with an empty data structure. In the first recursion, call make a subset of size one, in the next recursion call a subset of size 2, and so on. But first, in order to make a subset of size one what options do we have?

We can pick up elements from either the first index or the second index or the third index. However, if we have already picked up two from the second index, picking up two from the third index will make another duplicate subset of size one. Since we are trying to avoid duplicate subsets we can avoid picking up from the third index. This should give us an intuition that whenever there are duplicate elements in the array we pick up only the first occurrence.

The next recursion calls will continue from the point the previous one ended.

Let’s summarize:-

* Sort the input array.Make a recursive function that takes the input array ,the current subset,the current index and  a list of list/ vector of vectors to contain the answer.
* Try to make a subset of size n during the nth recursion call and consider elements from every index while generating the combinations. Only pick up elements that are appearing for the first time during a recursion call to avoid duplicates.
* Once an element is picked up, move to the next index.The recursion will terminate when the end of array is reached.While returning backtrack by removing the last element that was inserted.

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF{

static void printAns(List <List<Integer>> ans) {

System.out.println("The unique subsets are ");

System.out.println(ans.toString().replace(","," "));

}

public static void findSubsets(int ind, int[] nums, List<Integer> ds, List<List<Integer>> ansList) {

ansList.add(new ArrayList<>(ds));

for(int i = ind;i<nums.length;i++) {

if(i!=ind && nums[i] == nums[i-1]) continue;

ds.add(nums[i]);

findSubsets(i+1, nums, ds, ansList);

ds.remove(ds.size() - 1);

}

}

public static List<List<Integer>> subsetsWithDup(int[] nums) {

Arrays.sort(nums);

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

findSubsets(0, nums, new ArrayList<>(), ansList);

return ansList;

}

public static void main(String args[]) {

int nums[]={1,2,2};

List < List<Integer>> ans = subsetsWithDup(nums);

printAns(ans);

}

}

**Output:**

The unique subsets are  
[ [ ][ 1 ][ 1 2 ][ 1 2 2 ][ 2 ][ 2 2 ] ]

**Time Complexity: O(2^n)**for generating every subset and**O(k)**to insert every subset in another data structure if the average length of every subset is**k.** Overall **O(k \* 2^n).**

**Space Complexity: O(2^n \* k)**to store every subset of average length k. Auxiliary space is **O(n)** if n is the depth of the recursion tree.

# Word Search – Leetcode

Given an **m x n** **grid of characters** board and a **string word**, return**true if the word exists** in the grid. The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

**Examples:**

**Example 1:**

**Input:**

[

["A", "B", "C", "E"],

["S", "F", "C", "S"],

["A", "D", "E", "E"]

]

word = "ABCCED"

**Output:** true

**Explanation:** We can easily find the given word in the matrix.

**Example 2:**

**Input:**

[

["A", "B", "C", "E"],

["S", "F", "C", "S"],

["A", "D", "E", "E"]

]

word = "ABCB"

**Output:** false

**Explanation:** There is no such word in the given matrix.

### **Solution:**

**Disclaimer:** Don’t jump directly to the solution, try it out yourself first.

**Prerequisite:**You should be aware of backtracking. If not then follow [Backtracking (Basics to Advanced)](https://www.youtube.com/watch?v=yVdKa8dnKiE&list=PLgUwDviBIf0rGlzIn_7rsaR2FQ5e6ZOL9)

**Approach:**  We are going to solve this by using backtracking, in this approach first we will linearly search the entire matrix to find the first letters matching our given string. If we found those letters then we can start backtracking in all four directions to find the rest of the letters of the given string.

**Step 1:** Find the first character of the given string.

**Step 2:** Start Backtracking in all four directions until we find all the letters of sequentially adjacent cells.

**Step 3:** At the end, If we found our result then return true else return false.

**Edge cases:** Now think about what will be our stopping condition, we can stop or return false if we **reach the end of the boundaries**of the matrix or the letter at which we are making recursive calls **is not the required letter**.

We will also return if **we found all the letters** of the given word i.e. we found the number of letters equal to the length of the given word.

NOTE: Do not forget that we cannot reuse a cell again.

That is, we have to somehow keep track of our position so that we cannot find the same letter again and again.

In this approach, we are going to mark visited cells with some random character that will prevent us from revisiting them again and again.

**Code:**

* C++ Code
* Java Code

import java.io.\*;

import java.lang.\*;

class Solution {

public static void main(String[] args) {

char[][] board = {{'A','B','C','E'},

{'S','F','C','S'},

{'A','D','E','E'}};

String word = "ABCCED";

Solution sol = new Solution();

boolean res = sol.exist(board, word);

System.out.println(res);

}

public boolean exist(char[][] board, String word) {

int m = board.length;

int n = board[0].length;

int index = 0;

// First search the first character

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

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

if (board[i][j] == word.charAt(index)) {

if (searchNext(board, word, i, j, index, m, n))

return true;

}

}

}

return false;

}

private boolean searchNext(char[][] board, String word, int row, int col,

int index, int m, int n) {

// if index reaches at the end that means we have found the word

if (index == word.length())

return true;

// Checking the boundaries if the character at which we are placed is not

//the required character

if (row < 0 || col < 0 || row == m || col == n || board[row][col] !=

word.charAt(index) || board[row][col] == '!')

return false;

// this is to prevent reusing of the same character

char c = board[row][col];

board[row][col] = '!';

// top direction

boolean top = searchNext(board, word, row - 1, col, index + 1, m, n);

// right direction

boolean right = searchNext(board, word, row, col + 1, index + 1, m, n);

// bottom direction

boolean bottom = searchNext(board, word, row + 1, col, index + 1, m, n);

// left direction

boolean left = searchNext(board, word, row, col - 1, index + 1, m, n);

board[row][col] = c; // undo change

return top || right || bottom || left;

}

}

**Output:**True

**Time Complexity:  O(m\*n\*4^k)**, where “K” is the length of the word. And we are searching for the letter m\*n times in the worst case. Here 4 in 4^k is because at each level of our decision tree we are making 4 recursive calls which equal 4^k in the worst case.

**Space Complexity: O(K)**, Where k is the length of the given words.

# N Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

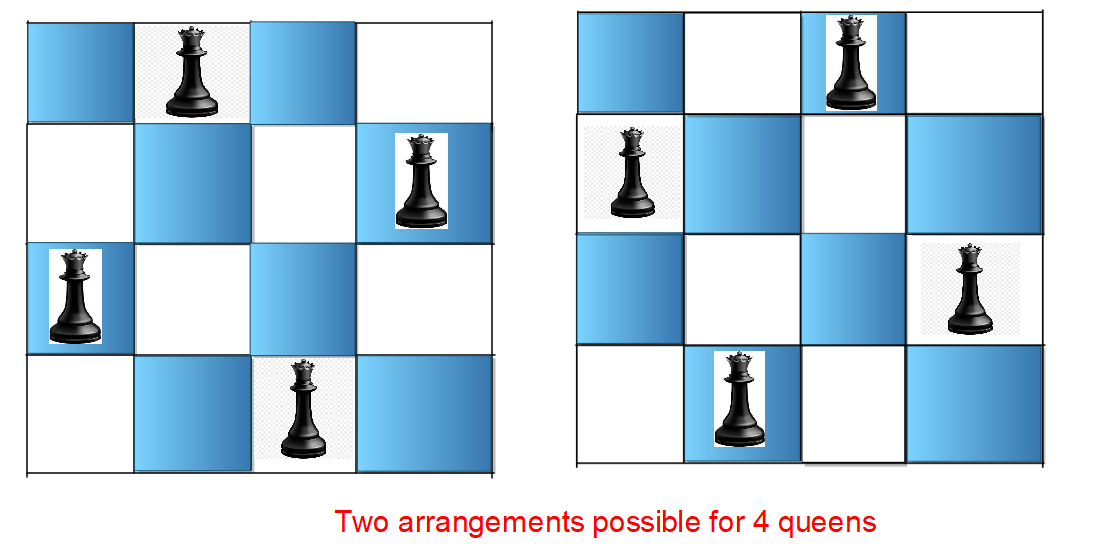
**Problem Statement:** The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n -queens puzzle. Each solution contains a distinct boards configuration of the queen’s placement, where ‘Q’ and ‘.’ indicate queen and empty space respectively.

**Examples:**

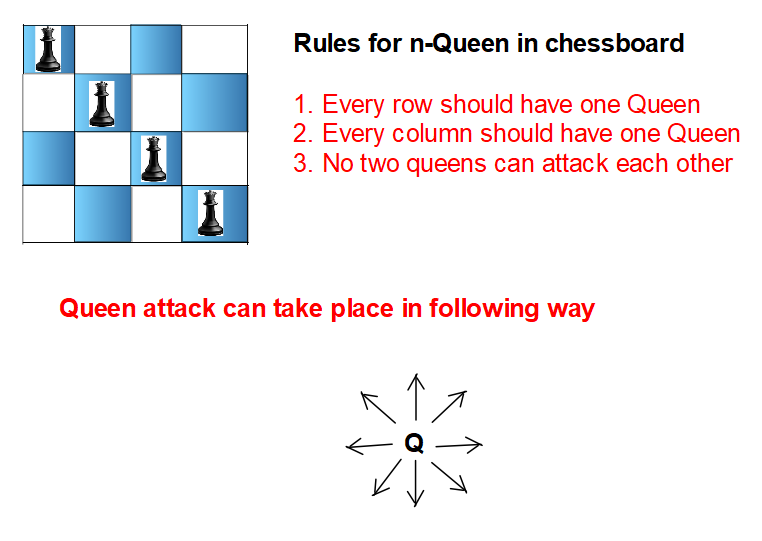
**Input:** n = 4

**Output:** [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

**Explanation:** There exist two distinct solutions to the 4-queens puzzle as shown below



**Let us first understand how can we place queens in a chessboard so that no attack on either of them can take place.**

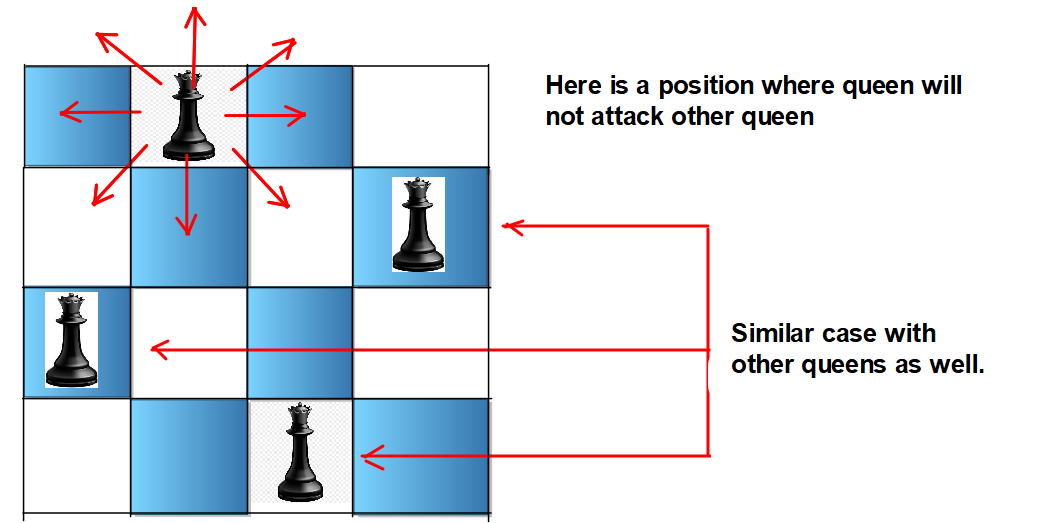


### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

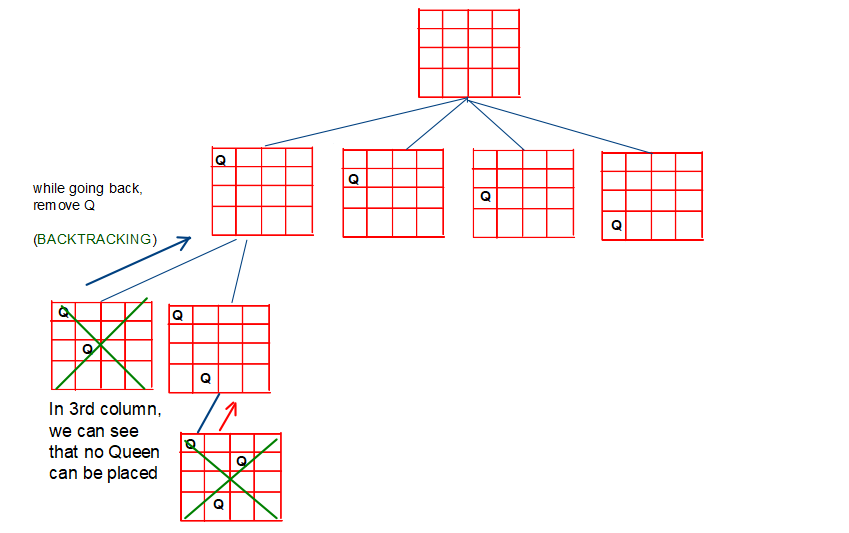
**Solution 1:**

**Intuition:**Using the concept of Backtracking, we will place Queen at different positions of the chessboard and find the right arrangement where all the n queens can be placed on the n\*n grid.

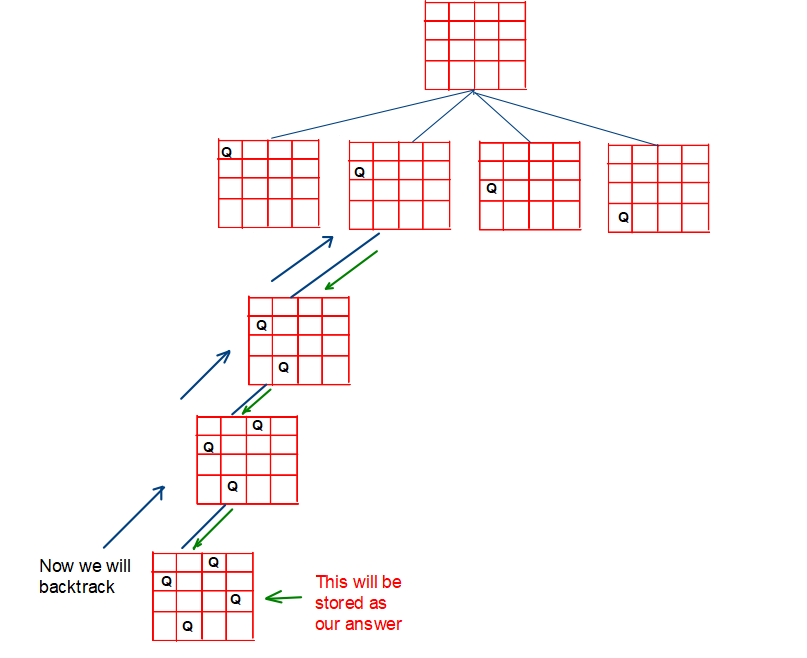


**Approach**:

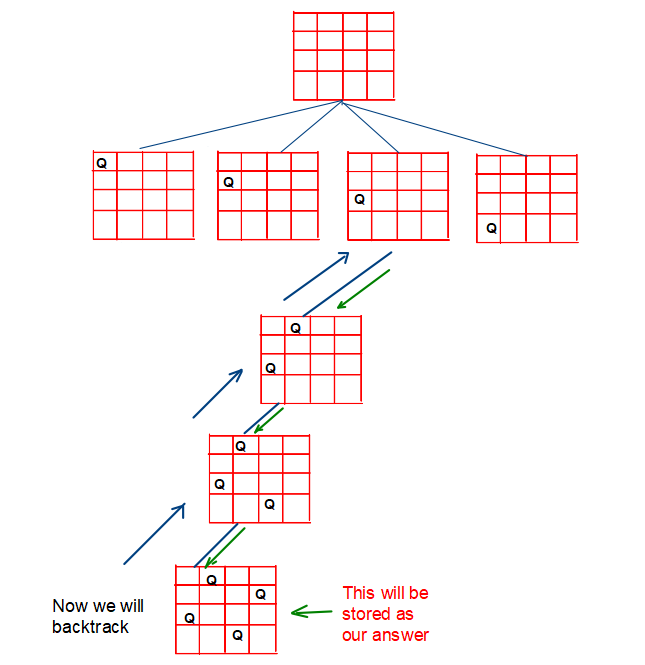
**Ist position**: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 3rd column, the Queen will be killed at all possible positions of row.



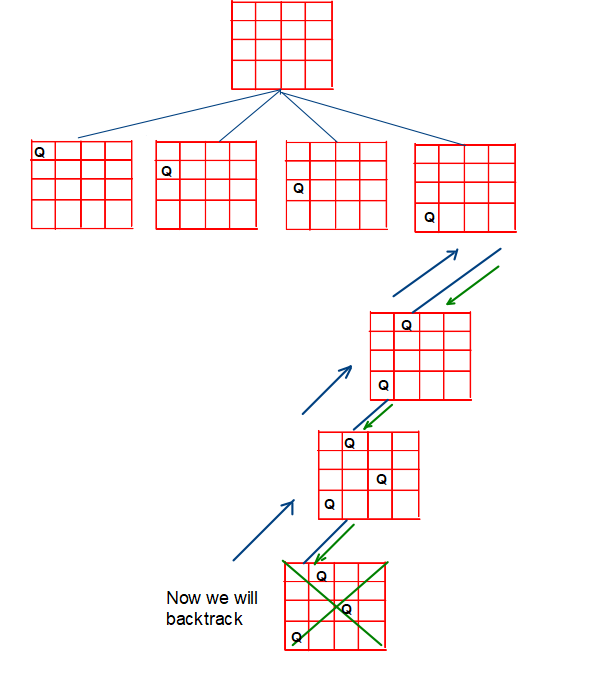
**2nd position:**One of the correct possible arrangements is found. So we will store it as our answer.

****

**3rd position:**One of the correct possible arrangements is found. So we will store it as our answer.

****

**4th position:**This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 4th column, the Queen will be killed at all possible positions of row.

****

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF {

public static List < List < String >> solveNQueens(int n) {

char[][] board = new char[n][n];

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

for (int j = 0; j < n; j++)

board[i][j] = '.';

List < List < String >> res = new ArrayList < List < String >> ();

dfs(0, board, res);

return res;

}

static boolean validate(char[][] board, int row, int col) {

int duprow = row;

int dupcol = col;

while (row >= 0 && col >= 0) {

if (board[row][col] == 'Q') return false;

row--;

col--;

}

row = duprow;

col = dupcol;

while (col >= 0) {

if (board[row][col] == 'Q') return false;

col--;

}

row = duprow;

col = dupcol;

while (col >= 0 && row < board.length) {

if (board[row][col] == 'Q') return false;

col--;

row++;

}

return true;

}

static void dfs(int col, char[][] board, List < List < String >> res) {

if (col == board.length) {

res.add(construct(board));

return;

}

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

if (validate(board, row, col)) {

board[row][col] = 'Q';

dfs(col + 1, board, res);

board[row][col] = '.';

}

}

}

static List < String > construct(char[][] board) {

List < String > res = new LinkedList < String > ();

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

String s = new String(board[i]);

res.add(s);

}

return res;

}

public static void main(String args[]) {

int N = 4;

List < List < String >> queen = solveNQueens(N);

int i = 1;

for (List < String > it: queen) {

System.out.println("Arrangement " + i);

for (String s: it) {

System.out.println(s);

}

System.out.println();

i += 1;

}

}

}

**Output:**

Arrangement 1  
..Q.  
Q…  
…Q  
.Q..

Arrangement 2  
.Q..  
…Q  
Q…  
..Q.

**Time Complexity:**Exponential in nature since we are trying out all ways, to be precise its O(N! \* N).

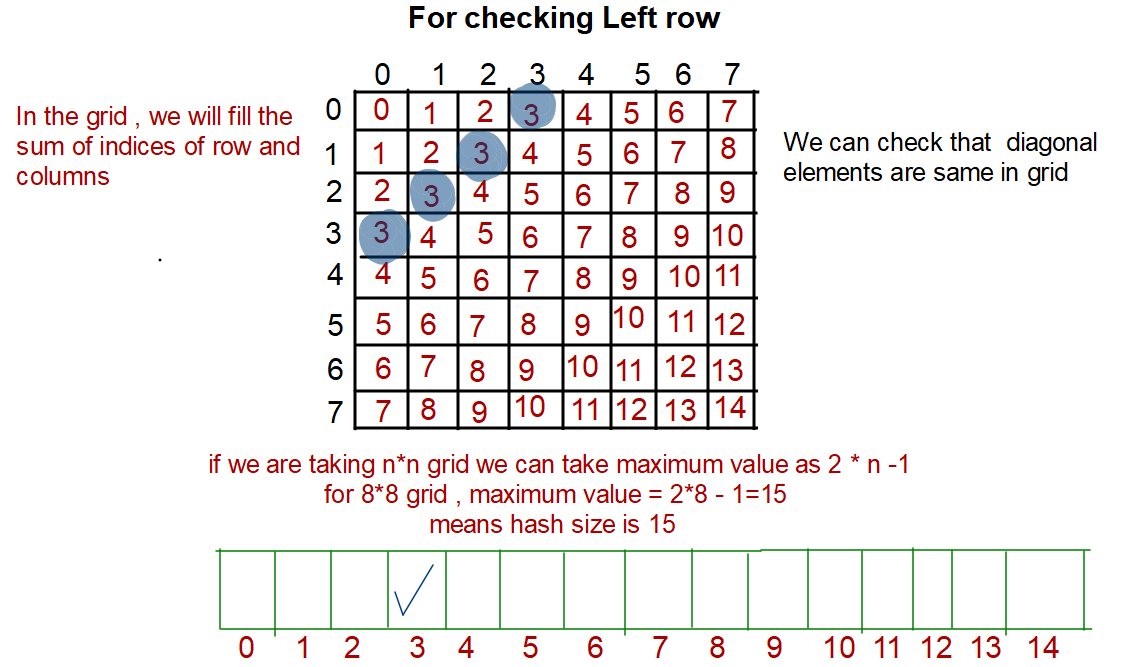
**Space Complexity:**O( N2)

**Solution 2:**

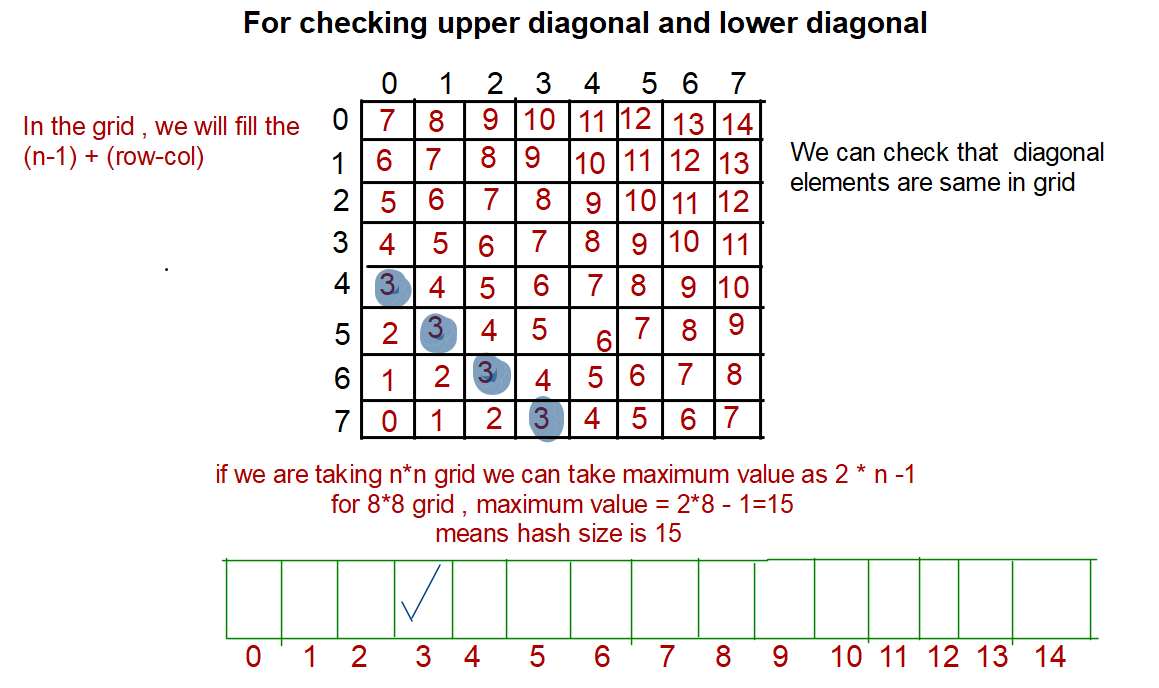
**Intuition:**This is the optimization of the issafe function. In the previous issafe function, we need o(N) for a row, o(N) for the column, and o(N) for the diagonal. Here, we will use hashing to maintain a list to check whether that position can be the right one or not.

**Approach:**

**For checking Left row elements**



**For checking upper diagonal and lower diagonal**



**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF {

public static List < List < String >> solveNQueens(int n) {

char[][] board = new char[n][n];

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

for (int j = 0; j < n; j++)

board[i][j] = '.';

List < List < String >> res = new ArrayList < List < String >> ();

int leftRow[] = new int[n];

int upperDiagonal[] = new int[2 \* n - 1];

int lowerDiagonal[] = new int[2 \* n - 1];

solve(0, board, res, leftRow, lowerDiagonal, upperDiagonal);

return res;

}

static void solve(int col, char[][] board, List < List < String >> res, int leftRow[], int lowerDiagonal[], int upperDiagonal[]) {

if (col == board.length) {

res.add(construct(board));

return;

}

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

if (leftRow[row] == 0 && lowerDiagonal[row + col] == 0 && upperDiagonal[board.length - 1 + col - row] == 0) {

board[row][col] = 'Q';

leftRow[row] = 1;

lowerDiagonal[row + col] = 1;

upperDiagonal[board.length - 1 + col - row] = 1;

solve(col + 1, board, res, leftRow, lowerDiagonal, upperDiagonal);

board[row][col] = '.';

leftRow[row] = 0;

lowerDiagonal[row + col] = 0;

upperDiagonal[board.length - 1 + col - row] = 0;

}

}

}

static List < String > construct(char[][] board) {

List < String > res = new LinkedList < String > ();

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

String s = new String(board[i]);

res.add(s);

}

return res;

}

public static void main(String args[]) {

int N = 4;

List < List < String >> queen = solveNQueens(N);

int i = 1;

for (List < String > it: queen) {

System.out.println("Arrangement " + i);

for (String s: it) {

System.out.println(s);

}

System.out.println();

i += 1;

}

}

}

**Output:**

Arrangement 1  
..Q.  
Q…  
…Q  
.Q..

Arrangement 2  
.Q..  
…Q  
Q…  
..Q.

**Time Complexity:**Exponential in nature since we are trying out all ways, to be precise its O(N! \* N).

**Space Complexity:**O(N)

# M – Coloring Problem

**Problem Statement:** Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.

**Examples:**

**Example 1:**

**Input:**

N = 4

M = 3

E = 5

Edges[] = {

(0, 1),

(1, 2),

(2, 3),

(3, 0),

(0, 2)

}

**Output:** 1

**Explanation:** It is possible to colour the given graph using 3 colours.

**Example 2:**

**Input:**

N = 3

M = 2

E = 3

Edges[] = {

(0, 1),

(1, 2),

(0, 2)

}

**Output:** 0

**Explanation:** It is not possible to color.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Backtracking**

**Approach**:

Basically starting from vertex 0 color one by one the different vertices.

**Base condition:**

If I have colored all the N nodes return true.

**Recursion:**

Trying every color from 1 to m with the help of a for a loop.

Is safe function returns true if it is possible to color it with that color i.e none of the adjacent nodes have the same color.  
  
In case this is an algorithm and follows a certain intuition, please mention it.

Color it with color i then call the recursive function for the next node if it returns true we will return true.

And If it is false then take off the color.

Now if we have tried out every color from 1 to m and it was not possible to color it with any of the m colors then return false.

**Code:**

import java.util.\*;

class TUF {

public static boolean graphColoring(List < Integer > [] G, int[] color, int i, int C) {

// Your code here

int n = G.length;

if (solve(i, G, color, n, C) == true) return true;

return false;

}

private static boolean isSafe(int node, List < Integer > [] G, int[] color, int n, int col) {

for (int it: G[node]) {

if (color[it] == col) return false;

}

return true;

}

private static boolean solve(int node, List < Integer > [] G, int[] color, int n, int m) {

if (node == n) return true;

for (int i = 1; i <= m; i++) {

if (isSafe(node, G, color, n, i)) {

color[node] = i;

if (solve(node + 1, G, color, n, m) == true) return true;

color[node] = 0;

}

}

return false;

}

public static void main(String[] args) {

int N = 4, M = 3;

List < Integer > [] G = new ArrayList[N];

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

G[i] = new ArrayList < > ();

}

G[0].add(1);

G[1].add(0);

G[1].add(2);

G[2].add(1);

G[2].add(3);

G[3].add(2);

G[3].add(0);

G[0].add(3);

G[0].add(2);

G[2].add(0);

int[] color = new int[N];

boolean ans = graphColoring(G, color, 0, M);

if (ans == true)

System.out.println("1");

else

System.out.println("0");

}

}

**Output:** 1

**Time Complexity: O( N^M) (n raised to m)**

**Space Complexity: O(N)**

# Palindrome Partitioning

**Problem Statement:** You are given a string s, partition it in such a way that every substring is a palindrome. Return all such palindromic partitions of s.

**Note:**A palindrome string is a string that reads the same backward as forward.

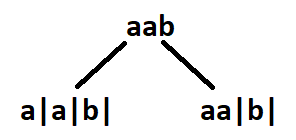
**Examples:**

**Example 1:**

**Input:** s = “aab”

**Output:** [ ["a","a","b"], ["aa","b"] ]

**Explanation:** The first answer is generated by making three partitions. The second answer is generated by making two partitions.

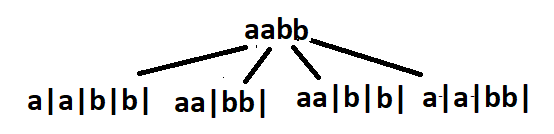


**Example 2:**

**Input:** s = “aabb”

**Output:** [ [“a”,”a”,”b”,”b”], [“aa”,”bb”], [“a”,”a”,”bb”], [“aa”,”b”,”b”] ]

**Explanation:** See Figure



### **Solution**

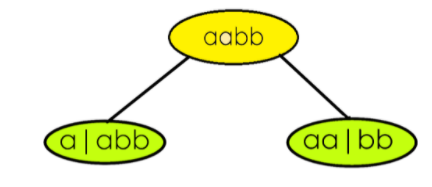
**Disclaimer**: Don’t jump directly to the solution, try it out yourself

**Approach**: The initial idea will be to make partitions to generate substring and check if the substring generated out of the partition will be a palindrome. Partitioning means we would end up generating every substring and checking for palindrome at every step. Since this is a repetitive task being done again and again, at this point we should think of recursion. The recursion continues until the entire string is exhausted. After partitioning, every palindromic substring is inserted in a data structure When the base case has reached the list of palindromes generated during that recursion call is inserted in a vector of vectors/list of list.

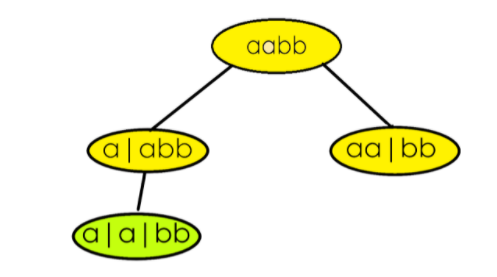
We have already discussed the initial thought process and the basic outline of the solution. The approach will get clearer with an example.

Say s = “aabb” and assume indexes of string characters to be 0-based. For a better understanding, we have divided recursion into some steps.

**STEP 1**: We consider substrings starting from the 0th index.[0,0] is a palindrome, so partition right after the 0th index.[0,1] is another palindrome, make a partition after 1st index. Beyond this point, other substrings starting from index 0 are “aab”  and “aabb”. These are not palindromes, hence no more. partitions are possible. The strings remaining on the right side of the partition are used as input to make recursive calls.

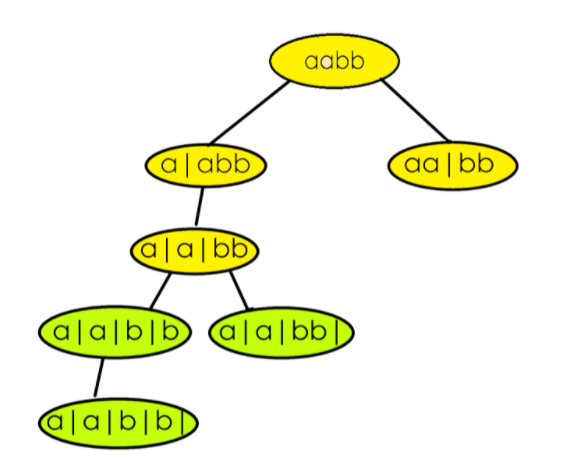


**STEP 2**: Consider the recursive call on the left(refer to image)  where “abb” is the input.**[1,1]**is a palindrome, make a partition after it.[1,2] and [1,3] are not palindromes.



**STEP 3**: Here “bb” is the input.**[2,2]**as well as **[2,3]**  are palindromes. Make one partition after the 2nd index and one after the 3rd index The entire string is exhausted after the 3rd index, so the right recursion ends here. Palindromes generated from the right recursion are inserted in our answer.

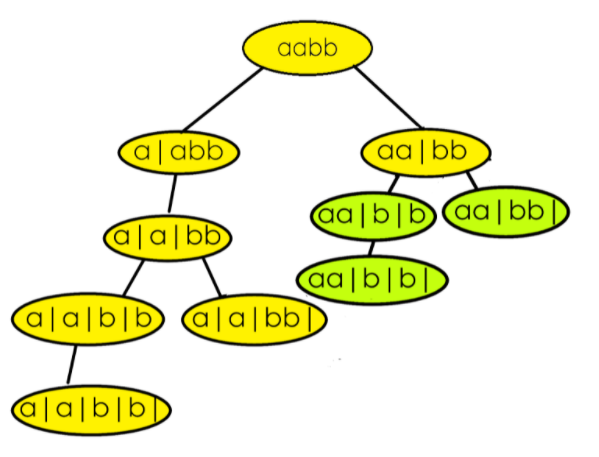
**Our answer at this point :[ [“a”,”**a”,” **bb”] ]**



The left recursion will continue with  “b” as its input.**[3,3]** is a palindrome so one last partition for the left recursion is made after the 3rd index. Insert the palindromes.

**ans = [  [“a”,”a”,”bb”], [ “a”,”a”,”b”,”b”] ]**

STEP 4: After the list of palindromic substrings are returned from the left recursive call, continue the same process for the call on the right that was left to recur. The right recursion is having “bb” as input, something we have already encountered in step 3. Hence we will repeat the same task which was done in step 3 onwards.



**Final answer : [  [“a”,”a”,”bb”], [ “a”,”a”,”b”,”b”] ,[“aa”,”b”,”b”], [“aa”,”bb”] ]**

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

class TUF {

public static List < List < String >> partition(String s) {

List < List < String >> res = new ArrayList < > ();

List < String > path = new ArrayList < > ();

partitionHelper(0, s, path, res);

return res;

}

static void partitionHelper(int index, String s, List < String > path, List < List < String >> res) {

if (index == s.length()) {

res.add(new ArrayList < > (path));

return;

}

for (int i = index; i < s.length(); ++i) {

if (isPalindrome(s, index, i)) {

path.add(s.substring(index, i + 1));

partitionHelper(i + 1, s, path, res);

path.remove(path.size() - 1);

}

}

}

static boolean isPalindrome(String s, int start, int end) {

while (start <= end) {

if (s.charAt(start++) != s.charAt(end--))

return false;

}

return true;

}

public static void main(String args[]) {

String s = "aabb";

List < List < String >> ans = partition(s);

int n = ans.size();

System.out.println("The Palindromic partitions are :-");

System.out.print(" [ ");

for (int i = 0; i < ans.size(); i++) {

System.out.print("[ ");

for (int j = 0; j < ans.get(i).size(); j++) {

System.out.print(ans.get(i).get(j) + " ");

}

System.out.print("] ");

}

System.out.print("]");

}

}

**Output:**

The Palindromic partitions are :-  
[ [ a a b b ] [ a a bb ] [ aa b b ] [ aa bb ] ]

**Time Complexity: O( (2^n) \*k\*(n/2) )**

**Reason: O(2^n)**to generate every substring and**O(n/2)** to check if the substring generated is a palindrome. O(k) is for inserting the palindromes in another data structure, where k  is the average length of the palindrome list.

**Space Complexity: O(k \* x)**

**Reason:**The space complexity can vary depending upon the length of theanswer. k is the average length of the list of palindromes and if we have x such list of palindromes in our final answer. The depth of the recursion tree is n, so the auxiliary space required is equal to the O(n).

# Rat in a Maze

**Rat in a Maze**

Consider a rat placed at **(0, 0)** in a square matrixof order **N \* N**. It has to reach the destination at **(N – 1, N – 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **‘U'(up)**, **‘D'(down)**, **‘L’ (left)**, **‘R’ (right)**. Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it while value 1 at a cell in the matrix represents that rat can travel through it.

**Note**: In a path, no cell can be visited more than one time.

Print the answer in lexicographical(sorted) order

**Examples:**

**Example 1:**

**Input:**

N = 4

m[][] = {{1, 0, 0, 0},

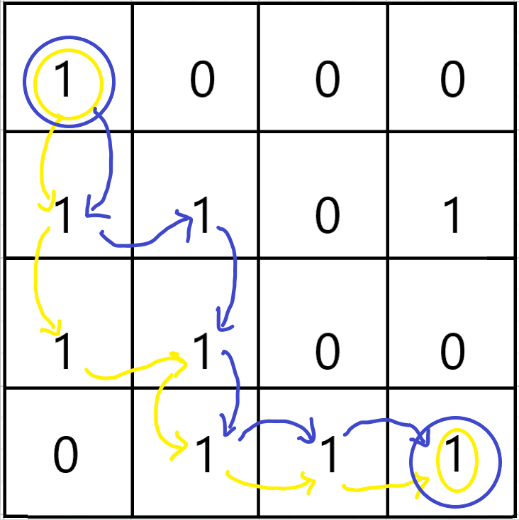
{1, 1, 0, 1},

{1, 1, 0, 0},

{0, 1, 1, 1}}

**Output:** DDRDRR DRDDRR

**Explanation:**



The rat can reach the destination at (3, 3) from (0, 0) by two paths - DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR DRDDRR.

**Example 2:**

**Input:** N = 2

m[][] = {{1, 0},

{1, 0}}

**Output:**

No path exists and the destination cell is blocked.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1: Recursion**

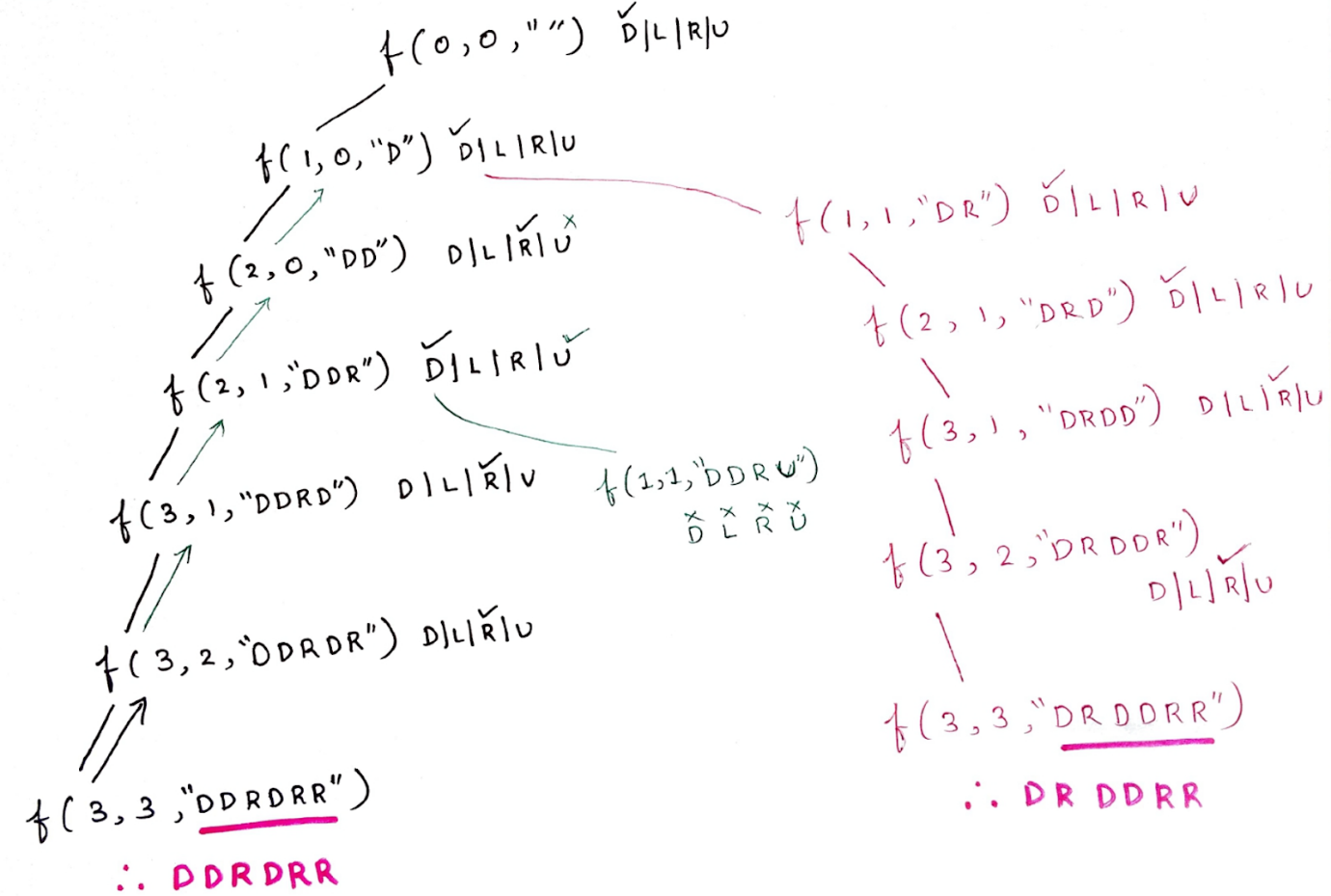
**Intuition:**

The best way to solve such problems is using recursion.

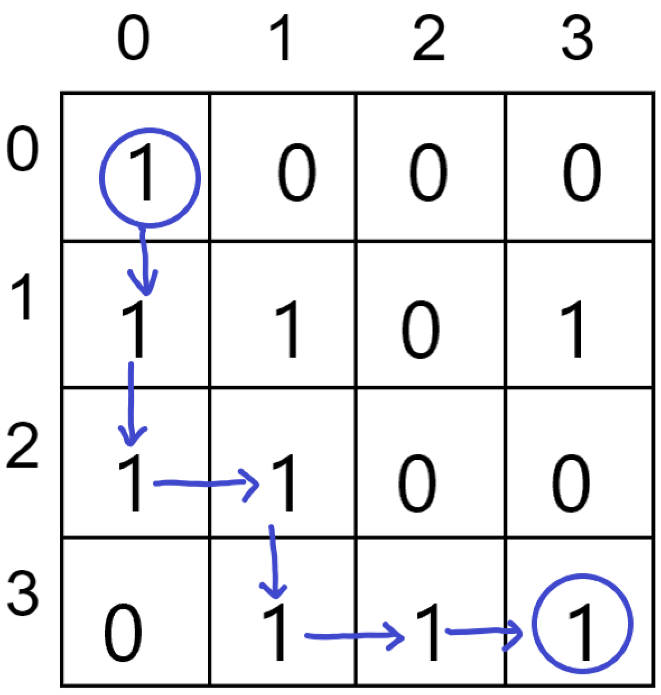
**Approach**:

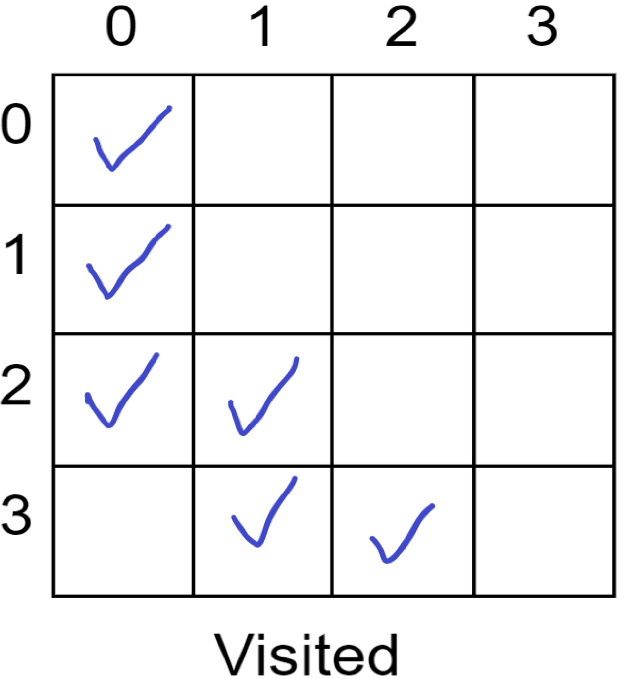
* Start at the source(0,0) with an empty string and try every possible path i.e upwards**(U)**, downwards**(D)**, leftwards**(L)** and rightwards**(R)**.
* As the **answer**should be in lexicographical order so it’s better to try the **directions**in lexicographical order i.e (D,L,R,U)
* Declare a 2D-array named visited because the question states that a single cell should be included only once in the path,so it’s important to keep track of the visited cells in a particular path.
* If a cell is in path, mark it in the visited array.
* Also keep a check of the**“out of bound”**conditions while going in a particular direction in the matrix.
* Whenever you reach the destination**(n,n)** it’s very important to get back as shown in the recursion tree.
* While getting back, keep on unmarking the visited array for the respective direction.Also check whether there is a different path possible while getting back and if yes, then mark that cell in the visited array.

**Recursive tree:**



**For  “DDRDRR” :**

****

****

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

// m is the given matrix and n is the order of matrix

class Solution {

private static void solve(int i, int j, int a[][], int n, ArrayList < String > ans, String move,

int vis[][]) {

if (i == n - 1 && j == n - 1) {

ans.add(move);

return;

}

// downward

if (i + 1 < n && vis[i + 1][j] == 0 && a[i + 1][j] == 1) {

vis[i][j] = 1;

solve(i + 1, j, a, n, ans, move + 'D', vis);

vis[i][j] = 0;

}

// left

if (j - 1 >= 0 && vis[i][j - 1] == 0 && a[i][j - 1] == 1) {

vis[i][j] = 1;

solve(i, j - 1, a, n, ans, move + 'L', vis);

vis[i][j] = 0;

}

// right

if (j + 1 < n && vis[i][j + 1] == 0 && a[i][j + 1] == 1) {

vis[i][j] = 1;

solve(i, j + 1, a, n, ans, move + 'R', vis);

vis[i][j] = 0;

}

// upward

if (i - 1 >= 0 && vis[i - 1][j] == 0 && a[i - 1][j] == 1) {

vis[i][j] = 1;

solve(i - 1, j, a, n, ans, move + 'U', vis);

vis[i][j] = 0;

}

}

public static ArrayList < String > findPath(int[][] m, int n) {

int vis[][] = new int[n][n];

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

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

vis[i][j] = 0;

}

}

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

if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis);

return ans;

}

}

class TUF {

public static void main(String[] args) {

int n = 4;

int[][] a = {{1,0,0,0},{1,1,0,1},{1,1,0,0},{0,1,1,1}};

Solution obj = new Solution();

ArrayList < String > res = obj.findPath(a, n);

if (res.size() > 0) {

for (int i = 0; i < res.size(); i++)

System.out.print(res.get(i) + " ");

System.out.println();

} else {

System.out.println(-1);

}

}

}

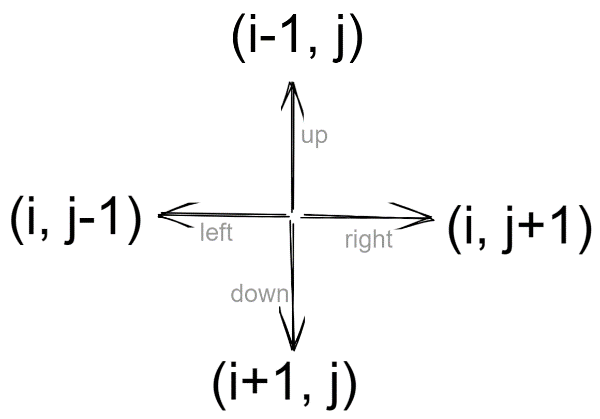
**Output:**

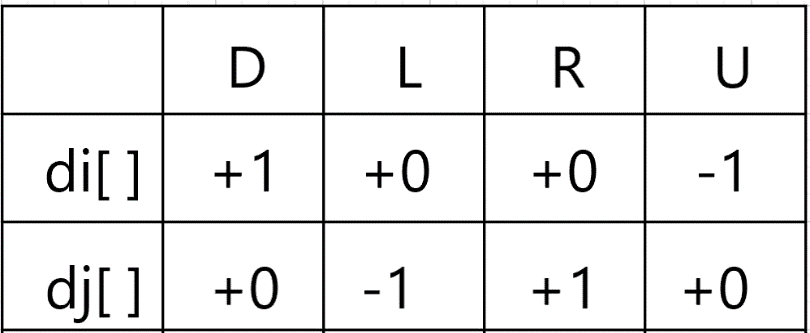
DDRDRR DRDDRR

**Time Complexity: O(4^(m\*n)),**becauseon every cell we need to try 4 different directions.

**Space Complexity:  O(m\*n),**Maximum Depth of the recursion tree(auxiliary space).

**But, writing an individual code for every direction is a lengthy process therefore we truncate the 4 “if statements” into a single for loop using the following approach.**





* C++ Code
* Java Code
* Python Code

import java.util.\*;

// m is the given matrix and n is the order of matrix

class Solution {

private static void solve(int i, int j, int a[][], int n, ArrayList < String > ans, String move,

int vis[][], int di[], int dj[]) {

if (i == n - 1 && j == n - 1) {

ans.add(move);

return;

}

String dir = "DLRU";

for (int ind = 0; ind < 4; ind++) {

int nexti = i + di[ind];

int nextj = j + dj[ind];

if (nexti >= 0 && nextj >= 0 && nexti < n && nextj < n &&

vis[nexti][nextj] == 0 && a[nexti][nextj] == 1) {

vis[i][j] = 1;

solve(nexti, nextj, a, n, ans, move + dir.charAt(ind), vis, di, dj);

vis[i][j] = 0;

}

}

}

public static ArrayList < String > findPath(int[][] m, int n) {

int vis[][] = new int[n][n];

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

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

vis[i][j] = 0;

}

}

int di[] = {

+1,

0,

0,

-1

};

int dj[] = {

0,

-1,

1,

0

};

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

if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis, di, dj);

return ans;

}

}

class TUF {

public static void main(String[] args) {

int n = 4;

int[][] a = {{1,0,0,0},{1,1,0,1},{1,1,0,0},{0,1,1,1}};

Solution obj = new Solution();

ArrayList < String > res = obj.findPath(a, n);

if (res.size() > 0) {

for (int i = 0; i < res.size(); i++)

System.out.print(res.get(i) + " ");

System.out.println();

} else {

System.out.println(-1);

}

}

}

**Output:**

DDRDRR DRDDRR

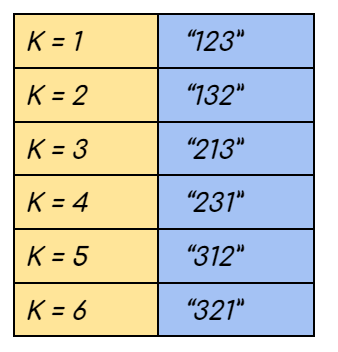
**Time Complexity: O(4^(m\*n)),**becauseon every cell we need to try 4 different directions.

**Space Complexity:  O(m\*n),** Maximum Depth of the recursion tree(auxiliary space).

# Find K-th Permutation Sequence

**Problem Statement:**Given **N** and **K**, where N is the sequence of numbers from **1 to N([1,2,3….. N])** find the**Kth permutation sequence**.

For N = 3  the 3!  Permutation sequences in order would look like this:-



**Note**: 1<=K<=N! Hence for a given input its Kth permutation always exists

**Examples:**

**Example 1:**

**Input:** N = 3, K = 3

**Output:** “213”

**Explanation:** The sequence has 3! permutations as illustrated in the figure above. K = 3 corresponds to the third sequence.

**Example 2:**

**Input:** N = 3, K = 5

**Result:** “312”

**Explanation:** The sequence has 3! permutations as illustrated in the figure above. K = 5 corresponds to the fifth sequence.

### **Solution**

**Disclaimer**: Don’t jump directly to the solution, try it out yourself first.

**Solution 1:** Brute Force Solution

**Approach**:

The extreme naive solution is to generate all the possible permutations of the given sequence.  This is achieved using recursion and every permutation generated is stored in some other data structure (here we have used a vector). Finally, we sort the data structure in which we have stored all the sequences and return the Kth sequence from it.

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

public class Main {

static void swap(char s[], int i, int j) {

char ch = s[i];

s[i] = s[j];

s[j] = ch;

}

static void permutationHelper(char s[], int index, ArrayList < String > res) {

if (index == s.length) {

String str = new String(s);

res.add(str);

return;

}

for (int i = index; i < s.length; i++) {

swap(s, i, index);

permutationHelper(s, index + 1, res);

swap(s, i, index);

}

}

static String getPermutation(int n, int k) {

String s = "";

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

for (int i = 1; i <= n; i++) {

s += i;

}

permutationHelper(s.toCharArray(), 0, res);

Collections.sort(res);

return res.get(k);

}

public static void main(String args[]) {

int n = 3, k = 3;

String ans = getPermutation(n, k);

System.out.println("The Kth permutation sequence is " + ans);

}

}

**Output:**

The Kth permutation sequence is 213

**Time complexity**: **O(N! \* N) +O(N! Log N!)**

Reason:  The recursion takes O(N!)  time because we generate every possible permutation and another O(N)  time is required to make a deep copy and store every sequence in the data structure. Also, O(N! Log N!)  time required to sort the data structure

**Space complexity: O(N)**

Reason: Result stored in a vector, we are auxiliary space taken by recursion

**Solution 2:(Optimal Approach)**

Say we have N = 4  and K = 17. Hence the number sequence is {1,2,3,4}.

**Intuition:**

Since this is a permutation we can assume that there are four positions that need to be filled using the four numbers of the sequence. First, we need to decide which number is to be placed at the first index. Once the number at the first index is decided we have three more positions and three more numbers.  Now the problem is shorter. We can repeat the technique that was used previously until all the positions are filled. The technique is explained below.

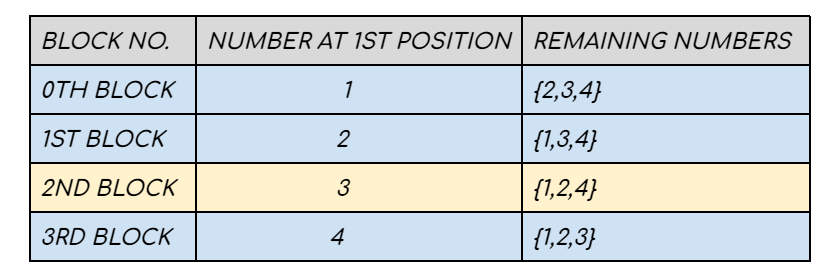
**Approach**:

**STEP 1:**

Mathematically speaking there can be 4 variations while generating the permutation. We can have our permutation starting with either 1 or 2 or 3 or 4. If the first position is already occupied by one number there are three more positions left. The remaining three numbers can be permuted among themselves while filling the 3 positions and will generate 3! = 6 sequences. Hence each block will have 6 permutations adding up to a total of 6\*4 = 24 permutations. If we consider the sequences as 0-based and in the sorted form we observe:-

* The **0th – 5th permutation** will start with 1
* The **6th – 11th permutation** will start with 2
* The **12th – 17th permutation** will start with 3
* The **18th – 23rd permutation** will start with 4.

 (For better understanding refer to the picture below.)



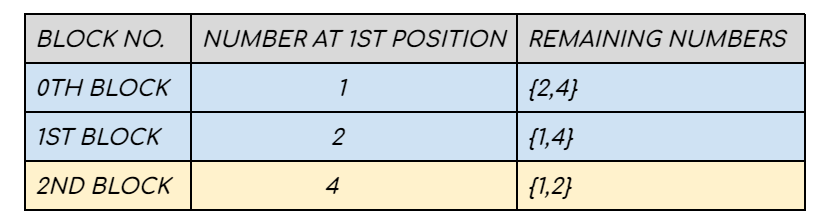
We make **K = 17-1 considering 0-based indexing**. Since each of the four blocks illustrated above comprises 6 permutations, therefore, the 16th permutation will lie in (16 / 6 ) = 2nd block, and our answer is the (16 % 6) = 4th sequence from the 2nd block. Therefore 3 occupies the first position of the sequence and **K = 4.**



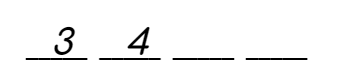
**STEP 2:**

Our new search space comprises three elements {1,2,4} where K = 4 . Using the previous technique we can consider the second position to be occupied can be any one of these 3 numbers. Again one block can start with 1, another can start with 2 and the last one can start with 4 . Since one position is fixed, the remaining two numbers of each block can form **2! = 2  sequences.** In sorted order :

* The **0th – 1st sequence**starts with 1
* The **2nd – 3rd sequence** starts with 2
* The **4th – 5th sequence**starts with 4



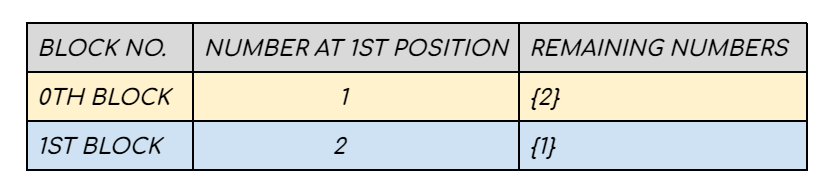
The 4th permutation will lie in **(4/2) = 2nd block** and our answer is the**4%2 = 0th sequence from the 2nd block**. Therefore **4**occupies the second position and **K = 0.**



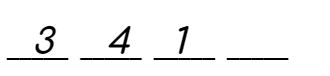
**STEP 3:**

The new search space will have two elements **{1 ,2}**and **K = 0**. One block starts with 1 and the other block starts with 2. The other remaining number can form only one **1! = 1 sequence**. In sorted form –

* The  **0th sequence** starts with 1
* The  **1st sequence**. starts with 2

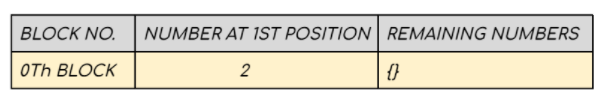


The 0th permutation will lie in the **(0/1) = 0th block** and our answer is the 0%1 = **0th sequence from the 0th block**. Therefore **1** occupies the 3rd position and**K = 0.**

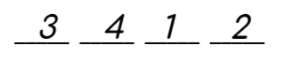


**STEP 4:**

Now the only block has 2 in the first position and no**remaining number is present**.



This is the point where we place 2 in the last position and stop.



**The final answer is 3412.**

**Code:**

* C++ Code
* Java Code
* Python Code

import java.util.\*;

public class Main {

static String getPermutation(int n, int k) {

int fact = 1;

ArrayList < Integer > numbers = new ArrayList < > ();

for (int i = 1; i < n; i++) {

fact = fact \* i;

numbers.add(i);

}

numbers.add(n);

String ans = "";

k = k - 1;

while (true) {

ans = ans + "" + numbers.get(k / fact);

numbers.remove(k / fact);

if (numbers.size() == 0) {

break;

}

k = k % fact;

fact = fact / numbers.size();

}

return ans;

}

public static void main(String args[]) {

int n = 3, k = 3;

String ans = getPermutation(n, k);

System.out.println("The Kth permutation sequence is " + ans);

}

}

**Output:**

The Kth permutation sequence is 213.

**Time Complexity: O(N) \* O(N) = O(N^2)**

Reason: We are placing N numbers in N positions. This will take O(N) time. For every number, we are reducing the search space by removing the element already placed in the previous step. This takes another O(N) time.

**Space Complexity: O(N)**

Reason: We are storing  the numbers in a data structure(here vector)