# Recursion

Basics :

## Print 1 to N numbers

Using recursion method1

void f**(**int i **,** int N **){**

**if(**i**>** N **)**

**return** **;**

cout **<<** i **<<** endl **;**

f**(**i**+**1 **,** N **)** **;**

**}**

int main**()** **{**

init\_code**()** **;**

//Solution sol ;

f**(**1 **,** 5 **)** **;**

**return** 0 **;**

**}**

Using recursion method 2

void f**(**int i **,** int j **){**

**if(**j **<** i **)**

**return** **;**

f**(**i **,** j**-**1 **)** **;**

cout **<<** j **<<** endl **;**

**}**

int main**()** **{**

init\_code**()** **;**

//Solution sol ;

f**(** 1 **,** 5 **)** **;**

**return** 0 **;**

**}**

## Sum of the first N natural numbers

Parameterized recursion

void f**(**int N **,** int sum **){**

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

cout **<<** sum **;**

**return** **;**

**}**

f**(**N**-**1 **,** sum**+**N **)** **;**

**}**

int main**()** **{**

init\_code**()** **;**

//Solution sol ;

f**(** 5 **,** 0 **)** **;**

**return** 0 **;**

**}**

Functional recursion (return )

int f**(**int N **){**

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

**return** 0**;**

**}**

**return** N **+** f**(**N**-**1 **)** **;**

**}**

int main**()** **{**

init\_code**()** **;**

//Solution sol ;

cout **<<** f**(** 5 **)** **;**

**return** 0 **;**

**}**

## Reversing Array

Using two pointer

void rev**(**int l **,** int r **,** int arr**[]** **)** **{**

**if(**l **>=** r **)**

**return** **;**

swap**(**arr**[**l**]** **,** arr**[**r**])** **;**

rev**(**l**+**1 **,** r**-**1 **,** arr **)** **;**

**}**

int main**()** **{**

init\_code**()** **;**

//Solution sol ;

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

**for(**int i**=**0 **;** i**<**5 **;** i**++** **)**

cout **<<** arr**[**i**]** **<<** " " **;**

cout **<<** endl **;**

rev**(**0 **,** 4 **,** arr **)** **;**

**for(**int i**=**0 **;** i**<**5 **;** i**++** **)**

cout **<<** arr**[**i**]** **<<** " " **;**

cout **<<** endl **;**

**return** 0 **;**

**}**

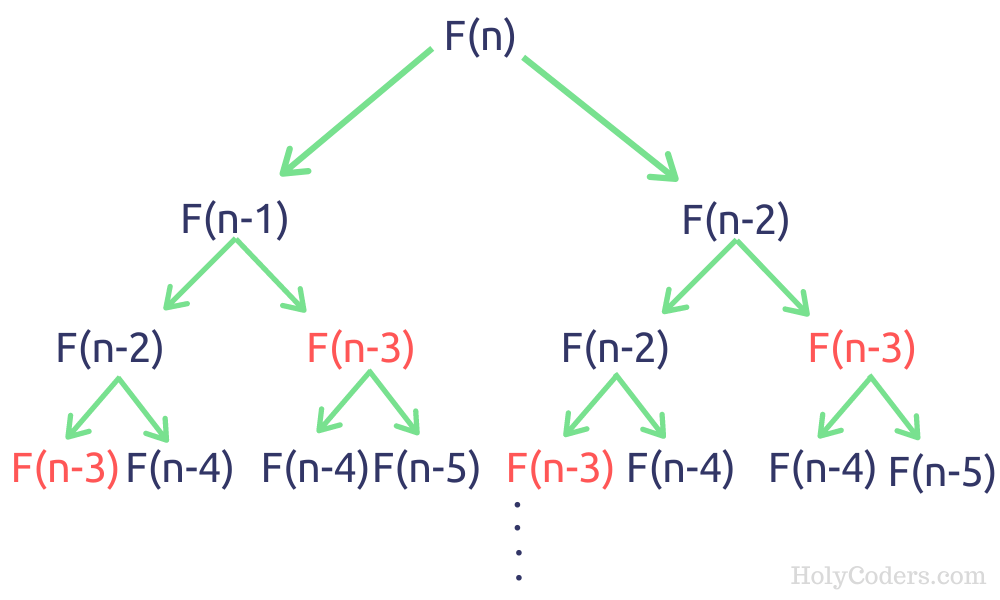
(We can consider slightly changing above two pointer to ‘i’ , ‘n-1-i’ , i>=n/2 will be the basecondition)

## Time Complexity of the Fibonnaci Series :

It is basically how many recursion calls we are doing overall.

2^0+2^1+...+2^n = 2^n+1-1 , here basically there will be n-1 levels.

Simply , at each node in the tree there are 2 splits,hence it will be 2^n



# Printing Subsequences

Let us say given arr[] = {3 , 2 , 1 } , then all the subsequnces of that would be

{}

3

2

1

3 2

3 1

2 1

3 2 1

Recursion on Subsequnces

This can be achived using the take/not-take approach , in one recursion call we will add the current element and in the another recursion call we will not add that.

**Time complexity :** 2^n x n as for every element there are tow choices either to pick or not to pick , n is nearly for printing the answer in the for loop.

## Printing Subsequences whose sum is k

We can maintain the sum variable and have two options for the current element like pick or not pick , so that in case of pick the sum will be subtracted with picked element and when sum becomes zero we got out list.

## Printing any subsequences whose sum is k

Here we can use a boolean return type such that as soon as we find any one subsequnce we can return from there with true.

## Counting no subsequences whose sum is k

Take int as return type , whenever a base case is reached and the condition is satisfied we shall return 1.

# Combination Sum – 1

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.

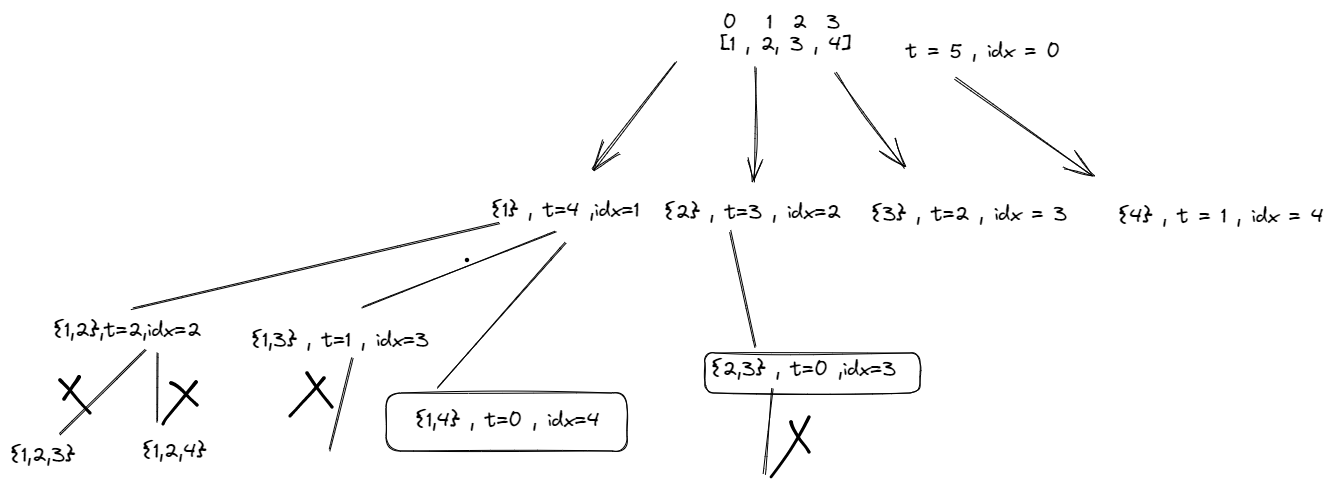
The same number may be chosen from the given array an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

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

If the question asked is like only one number can be picked only once,we shall go for the bounded knapsack approach.

**Approach** : Recursion (un bounded knapsack model)

**Another Approach** : Use a for loop



We are allowed to pick the same element again.

**Time complexity :** O(2^t \* k) where t is the target, 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.however in the actual case this complexity can be much lesser than this as there are no duplicates in the array given.

**Space complexity :** we should not be considering the space used to return the answer.

The space complexity of the above recursive approach for the "Combination Sum" problem on LeetCode depends on the depth of the recursion tree and the size of the intermediate data structures that are used to store the current combination and the list of valid combinations.

At each level of the recursion tree, the algorithm adds one candidate to the current combination and calls the recursive function with a smaller target and the same current combination. Since the maximum depth of the recursion tree is limited to the target value, the space complexity of the recursion stack is O(target).

# 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.

Example :

arr[] = [1,1,1,2,2 ] target = 4

[1,1,2] [2,2]

**Solution :** Sort the array & Use the for loop to skip the duplicates.

**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.

# Subset Sum I

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

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

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

**Approach** : 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).

**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.

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

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

**Bruteforce Approach :** Generate all the possible subsets i.e 2^n and then consider removing the duplicates.

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.

**Best Approach :**

Use a for loop to skip the duplicates there by duplicate subsets.

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.

**Observations so far :**

|  |  |
| --- | --- |
| **When using for loop** | **When not using for loop** |
| No need to have condition like ***if(index == N)*** as the for loop condition ***for(int i=idx ; i<N ; i++ )*** going to take care of that. | Need to have the condition ***if(index == N)*** and the return statement in that , as this will be forming the basecase. |
| Mainly used when there is need to avoid or skip the duplicates . | Not possible to skip through the duplicates. |
| Sorting of the array is required( usecase specially designed for this . ) | Sorting of the array is not required. |
| Only one recursion call will be there inside the for loop statement and a datastructure will be passed by reference always in recursion calls will be used for pick(add) and non-pick(remove) | There will be two recursion calls explicity specified for pick and non-pick. |
| The order is always pick(add) and non-pick(remove) | This can be in any order of the recursion calls. |

# Print all permutations of a string/array

**Bruteforce Approach :**

* Initialize three additional arrays: a boolean array that keeps track of whether an element in the input array has been used or not, an array to store the current permutation being built, and an array to store all the computed permutations.
* Check whether the current permutation is complete (i.e., all elements in the input array have been used). If so, add it to the array of permutations and return.
* Loop through the elements in the input array and pick one that has not been used yet. Mark it as used, add it to the current permutation array, and recursively call the function with the updated state.
* Once the recursion is complete, remove the last element from the current permutation array and mark the element as unused again.

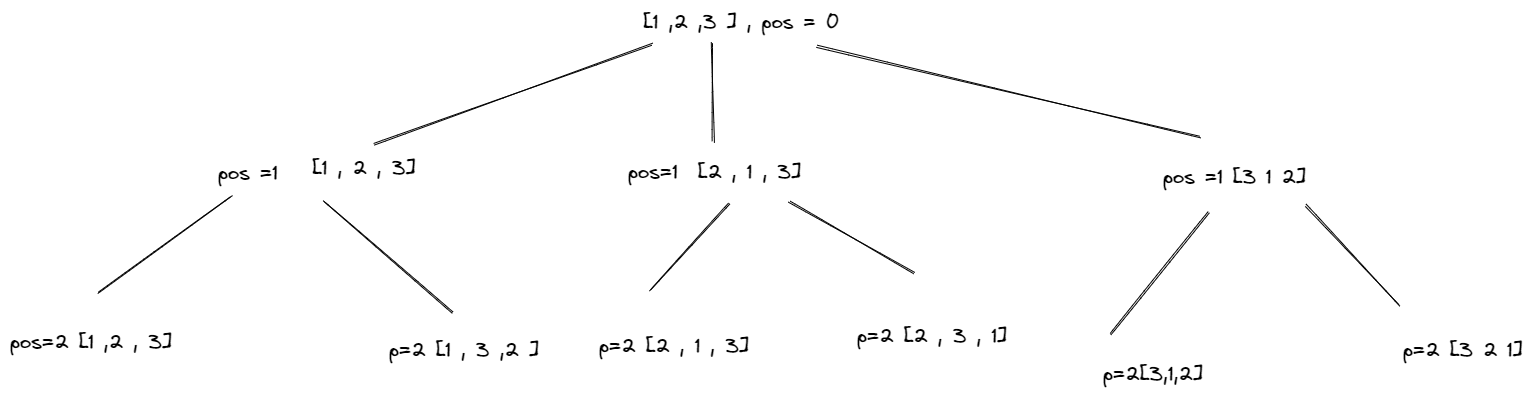
**TimeComplexity :** Each recursive call generates a new permutation by choosing an unused element of the input array and adding it to the current permutation. Since there are n elements to choose from on the first recursive call, n-1 on the second, n-2 on the third, and so on, the total number of possible permutations is n \* (n-1) \* (n-2) \* ... \* 1 = n!.

Each function call takes O(n) time to build the permutation because the algorithm loops through all n elements of the input array in order to find an unused element to add to the current permutation. This loop takes O(n) time because in the worst case, every element of the input array needs to be examined.

**Approach 2 :**

The idea is to fix one position of the array, then recursively generate all permutations of the remaining elements. We do this by swapping each element with the fixed element in the position.

**TC :** same as above.



We can see that the **pos** is fixed at each level , which will be swapped starting from that index to other.

# N-Queens

**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.

**Approach :** 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.

**Time Complexity :**

Since there are N queens to be placed on the board, the number of possible ways to place them is N! (i.e. N \* (N-1) \* (N-2) \* ... \* 1)

**Little Optimization:**

We can take a hash of the all the placed queens in all directions to determinw the next safe position.

# Sudoku Solver

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.

**Approach :**

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.

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

# 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.

**Approach :**

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

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.

If I have colored all the N nodes return true.

Time Complexity: O( N^M) (n raised to m) , because for every node of n nodes , we can possibly color with the any of the given m colors.

# Rat in a Maze

Consider a rat placed at (0, 0) in a square matrix of 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.

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

# 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.

**Intution :** We start by considering all possible partition lengths, starting from the first character of the string. For each partition length, we check if the substring from the start index to the current index is a palindrome. If it is, we add it to the current path and recursively explore all possible partitions starting from the next index.

If we reach the end of the string, it means we have found a valid partition, so we add the current path to the result vector. If we encounter a substring that is not a palindrome, we backtrack by removing the last element from the current path and continue exploring other possible partitions.

**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

# 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.

**Approach 1 : Brute Force**

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.

TC : (n! x log (n!) )

**Approach 2 : Optimal**

The algorithm is based on the observation that the kth permutation of the numbers 1 to n can be constructed by selecting the first digit, then the second digit, and so on.

To select the first digit, we divide k by (n-1)! to get the index of the digit in the remaining n-1 digits. The (n-1)! factor represents the number of permutations of the remaining digits. We then select the digit at this index from the remaining digits, and remove it from the set of available digits.

We repeat this process for the second digit, using the remaining n-2 digits and the updated value of k. We continue until we have selected all n digits of the permutation.

The implementation of this algorithm involves calculating factorials to determine the indices of the digits to select. This can be done iteratively or recursively. In both cases, we use a vector to keep track of the available digits and remove the selected digits as we go.

**Time Complexity :**

The time complexity of the algorithm is O(n^2) since we need to perform n iterations of the loop and removing an element from a vector takes O(n) time.