**Dynamic Programming by Alvin from Structy (FreeCodeCamp) :**

Dynamic programming is a method to solve complex coding problems by breaking them into simpler sub problems.

LeetCode Problems , Interview Questions , etc are often solved using DP.

**Problem 1 : Fibonacci problem :-**

FIB(n) : 0 1 1 2 3 5 8 13 21 34 55 89 …………

We can solve this iteratively but will do recursively

FIB(6) 🡪 FIB(5) + FIB(4)

Here ‘6’ is the root node and both ‘5’ & ‘4’ are child nodes of this node , which is turn have more child nodes.

FIB(n) 🡪 FIB(n-1) + FIB(n-2)

**Complexity ?**

There are several recursive calls made in the visual tree. Since after , every level recursive calls get doubled. There are n levels in the tree , so the time complexity is estimated to be 2^n. It can be less than this based on the values plugged in to the code. So , time complexity is O(2^n).

Now , space complexity is going to be the number of stack frames for recursive calls.

So , space complexity be O(n).

But when solving this recursively , we have to solve many duplicate problems which then takes the shape of the **Brute Force recursion.**

So , to avoid this duplicacy problem we can remove duplicate versions of the problem.

We can do this by **memorization** ,which means storing some additional information inside a memo. Memo would be a hashed data structure ( HashMap in C++ / Java , Dictionary in Python ).

We can store previously computed values in our memo and then retrieve them when required.

Now we calculate the runtime , so now time complexity is also linear , i.e. , O(n).

**Source code:**

import java.util.HashMap; //for memoization , a fast look Data structure

class Fibonacci

{

    //Recursion + Memoization

    //Memoization is used to eliminate brute force recursion

    public static int fib(int n){

        return fib(n , new HashMap<>());

    }

    // An overloaded function

    public static int fib(int n, HashMap<Integer, Integer> memo)

    {

        if ( n == 0 || n == 1 ) return n ; // base cases

        if(memo.containsKey(n))

        {

            return memo.get(n);

        }

        int result =  fib(n-1 , memo) + fib(n-2,memo) ;

        memo.put(n,result);

        return result ;

    }

    public static void main(String[] args)

    {

        System.out.println(fib(100));

    }

}

**Problem 2 : Tribonacci Problem :-**

Sequence :- 0 0 1 1 2 4 7 13 24 44 ….

FIB(n) 🡪 FIB(n-1) + FIB(n-2) + FIB(n-3)

We can also solve this like previous problem.

Since every parent node here has 3 child nodes.

Time complexity :- O(3^n)

Space complexity :- O(n)

**NOTE :- Dynamic Programming (** a type or problem solving where we utilize any overlapping sub problems right in order , the information is stored first time the problem is solved and reused later on when encountering the same problem. **)**

Memoization is a dynamic programming strategy.

**Source Code :**

import java.util.HashMap;

public class Tribonacci {

    // using Brute force recursion + memoization

    public static int trib(int n)

    {

        return trib(n , new HashMap<>());

    }

    public static int trib(int n , HashMap<Integer , Integer> memo )

    {

        // 0 0 1 1 2 4 7 13 24 44 ......

        if( n == 0 || n == 1) return 0;

        if( n == 2 || n == 3) return 1;

        if(memo.containsKey(n))

        {

            return memo.get(n);

        }

        int result = trib(n-1 , memo) + trib(n-2 , memo) + trib(n-3 , memo);

        memo.put(n , result);

        return result;

    }

    public static void main(String[] args){

        System.out.println(trib(50));

    }

}

**Problem 3 : Generating a target amount from the numbers given in the array .**

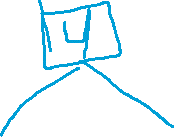
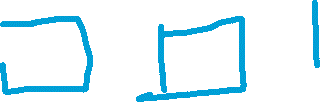
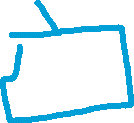
Example :

Target Amount : - 4

Array :- [1,2,3]



Firstly , we will create a decision tree to create all the possible decisions we can make from the numbers present in the Array to add up to the target amount.



This is the Brute force method to solve the problem. If child nodes have remaining quantities and solving them out will make the quantities negative. The child nodes will return false which in turn will make the parent nodes false and so on.

Child nodes return True 🡪 It’s parent nodes will return True

**Brute force complexity ?**

a = amount

n = length of numbers

Time = O(n^a)

Space Complexity should also be very high as recursive calls will be made and stored.

Memoization can be done to store some values and some cache.

**Complexity while memoization ?**

a = amount

n = length of numbers

time : O(a\*n)

space : O(n)