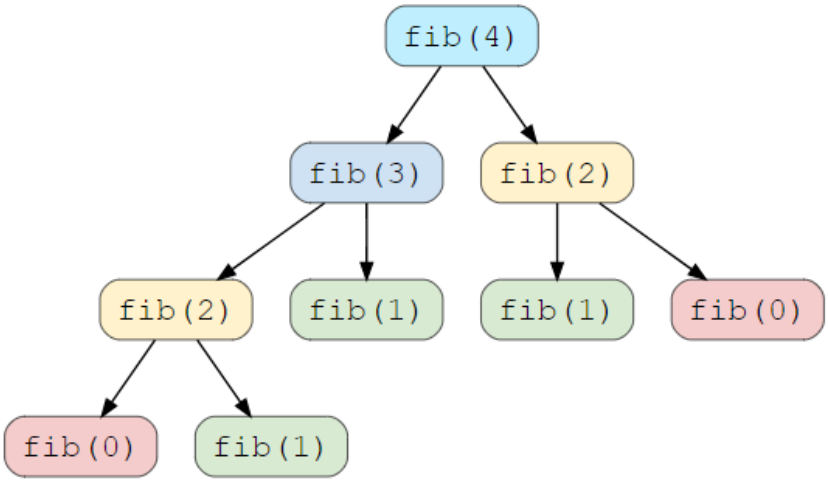
Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique for solving an **optimization problem** by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Characteristics of DP

1) **Overlapping Subproblems:** Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves **solving the same subproblem multiple times**.



Recursion tree for calculating Fibonacci numbers

2) **Optimal Substructure Property:** Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its subproblems.

For Fibonacci numbers,

Recursive formula: $fib(n) = \begin{cases} 0;\ n = 0 \\ 1;\ n = 1 \\ fib(n-1) + fib(n-2); n > 1 \end{cases}$

DP Methods

1) **Top-down with Memorization:** In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called **Memorization**.

Recursive Implementation	Top-down with Memorization

```
int fib(int term){
    if(term==0){
        return 0; ///base condition 1
    }
    else if(term==1){
        return 1; ///base condition 2
    }
    else{
        ///recursive subproblems
        int myresult=fib(term-1)
                    +fib(term-2);
        return myresult;
    }
}
```

```
int cache[100]={0};

int memfib(int term){
    if(term==0){
        return 0; ///base condition 1
    }
    else if(term==1){
        return 1; ///base condition 1
    }
    else{
        ///checking the cache memory
        if(cache[term]!=0){
            ///found
            return cache[term];
        }
        else{
            ///not found
            cache[term]=memfib(term-1)
                        +memfib(term-2);

            return cache[term];
        }
    }
}
```

2) **Bottom-up with Tabulation:** Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem "bottom-up" (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Recursive Implementation	Bottom-up with Tabulation

```
int fib(int term){
    if(term==0){
        return 0; ///base condition 1
    }
    else if(term==1){
        return 1; ///base condition 2
    }
    else{
        ///recursive subproblems
        int myresult=fib(term-1)
                    +fib(term-2);
        return myresult;
    }
}
```

```
int dptable[100];

int dpfib(int term){
    dptable[0]=0; ///base condition 1
    dptable[1]=1; ///base condition 2

    ///filling up the table
    for(int t=2;t<=term;t++){
        dptable[t]=dptable[t-1]+dptable[t-2];
    }

    return dptable[term];
}
```

Practice 1 – Staircase problem

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.

Ref: https://www.geeksforgeeks.org/count-ways-reach-nth-stair/

Practice 2 – Tiling problem

Given a "2 x n" board and tiles of size "2 x 1", count the number of ways to tile the given board using the 2 x 1 tiles. A tile can either be placed horizontally i.e., as a 1 x 2 tile or vertically i.e., as 2 x 1 tile.

Ref: https://www.geeksforgeeks.org/tiling-problem/

Practice 3 – Friends pairing problem

Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.

Ref: https://www.geeksforgeeks.org/friends-pairing-problem/

Practice 4 – House thief

There are n houses build in a line, each of which contains some value in it. A thief is going to steal the maximal value of these houses, but he can't steal in two adjacent houses because the owner of the stolen houses will tell his two neighbors left and right side. What is the maximum stolen value?

Ref: https://www.geeksforgeeks.org/find-maximum-possible-stolen-value-houses/

Practice 5 – Minimum jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, they cannot move through that element. If the end isn't reachable, return -1.

Ref: https://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/

Problem 6 – Catalan Number

Find out the n^{th} Catalan number.
First few Catalan numbers for n = 0, 1, 2, 3, 4, … are 1, 1, 2, 5, 14, 42, … etc.

Recursive formula: $C(n) = \begin{cases} 1; n = 0 \\ \sum_{i=0}^{n-1} C(i) * C(n-1-i) \end{cases}$

Ref: https://www.geeksforgeeks.org/program-nth-catalan-number/

Practice 7 – Binomial coefficient

Write a function that takes two parameters n, r and returns the value of Binomial Coefficient C(n, r) or, nCr.

Ref: https://www.geeksforgeeks.org/binomial-coefficient-dp-9/

Practice 8 – Permutation coefficient

Write a function that takes two parameters n, r and returns the value of nPr.

Ref: https://www.geeksforgeeks.org/permutation-coefficient/

Practice 9 – Subset sum

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Ref: https://www.geeksforgeeks.org/subset-sum-problem-dp-25/

Practice 10 – 0/1 Knapsack problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Ref: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Practice 11 – Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", .. etc are subsequences of "abcdefg".

Ref: https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

Practice 12 – Edit Distance

Given two strings str1 and str2 and 3 operations (Insert, Replace, Delete) that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'. All of the above operations are of equal cost.

Ref: https://www.geeksforgeeks.org/edit-distance-dp-5/