CP- EITC

Lecture 6

CP Cell Leads:
Idriss EL ABIDI
Souhail LOUZI

Dynamic Programming (DP)

February , 2025

1. Introduction to DP
2. Top-Down Approach
3. Bottom Up Approach
4. Practice

1 Introduction to DP

2 Top-Down Approach

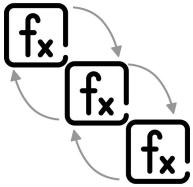
3 Bottom Up Approach

4 Practice

What is Dynamic Programming?

Definition :

Dynamic programming is a problem-solving paradigm where we try to find a solution by breaking a large problem into subproblems, and exploit the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.



you can start by assuming that DP is a kind of intelligent and faster recursive backtracking.

What is Dynamic Programming

Dynamic programming:

- Split the problem into overlapping subproblems
- Solve each subproblem recursively
- Combine the solutions to subproblems into a solution for the given problem
- Dont compute the answer to the same subproblem more than once

A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

Uses of Dynamic Programming?

There are two uses for dynamic programming:

- **Finding an optimal solution:** We want to find a solution that is as large as possible or as small as possible.
- **Counting the number of solutions:** We want to calculate the total number of possible solutions

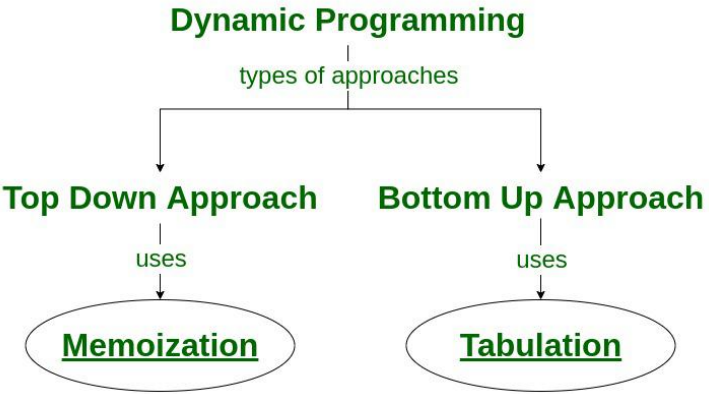
So if you encounter a problem that says **"minimize this"** or **"maximize that"** or **"count how many ways"**, then there is a chance that it is a DP problem.

Dynamic programming formulation

- 1. Formulate the problem in terms of smaller versions of the problem (recursively)
- 2. Turn this formulation into a recursive function
- 3. Memoize the function (remember results that have been computed)

```
map<problem, value> memory;

value dp(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, dp(Q));
    }

    memory[P] = result;
    return result;
}
```

Dynamic programming formulation



1 Introduction to DP

2 Top-Down Approach

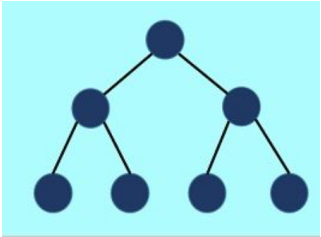
3 Bottom Up Approach

4 Practice

Top-Down Approach

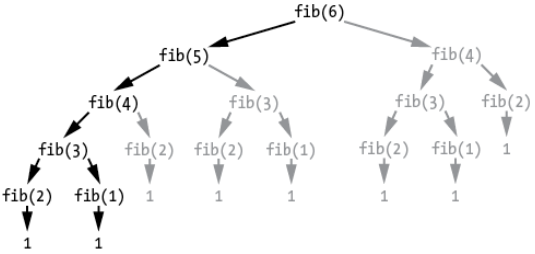
Definition :

A top-down DP approach tries to solve the bigger problem by recursively finding the solution to smaller problems while also storing those local solutions in a look-up table to prevent them from being computed each time. This technique is called **Memoization**.



Example

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$



Solution

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to calculate the nth Fibonacci number using memoization
int nthFibonacciUtil(int n, vector<int>& memo) {

    // Base case: if n is 0 or 1, return n
    if (n <= 1) {
        return n;
    }

    // Check if the result is already in the memo table
    if (memo[n] != -1) {
        return memo[n];
    }

    // Recursive case: calculate Fibonacci number
    // and store it in memo
    memo[n] = nthFibonacciUtil(n - 1, memo)
                + nthFibonacciUtil(n - 2, memo);

    return memo[n];
}

// Wrapper function that handles both initialization
// and Fibonacci calculation
int nthFibonacci(int n) {

    // Create a memoization table and initialize with -1
    vector<int> memo(n + 1, -1);

    // Call the utility function
    return nthFibonacciUtil(n, memo);
}

int main() {
    int n = 5;
    int result = nthFibonacci(n);
    cout << result << endl;

    return 0;
}
```

1 Introduction to DP

2 Top-Down Approach

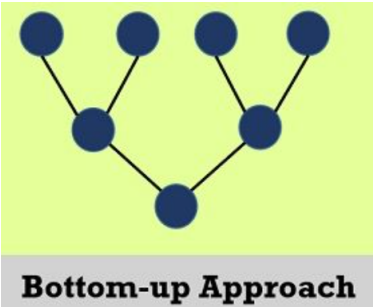
3 Bottom Up Approach

4 Practice

Top-Down Approach

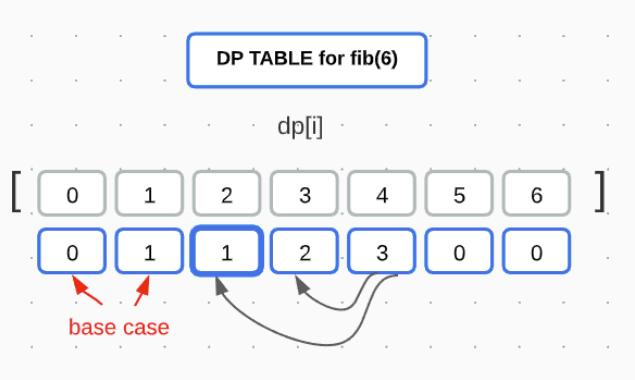
Definition :

This technique is the opposite of the former, and we start from the smallest solution and go up to the required solution.



Example

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

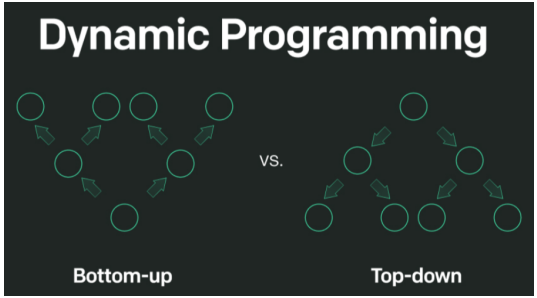


Solution

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to calculate the nth Fibonacci number using recursion
int nthFibonacci(int n){
    // Handle the edge cases
    if (n <= 1)
        return n;

    // Create a vector to store Fibonacci numbers
    vector<int> dp(n + 1);

    // Initialize the first two Fibonacci numbers
    dp[0] = 0;
    dp[1] = 1;

    // Fill the vector iteratively
    for (int i = 2; i <= n; ++i){

        // Calculate the next Fibonacci number
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    // Return the nth Fibonacci number
    return dp[n];
}

int main(){
    int n = 5;
    int result = nthFibonacci(n);

    cout << result << endl;

    return 0;
}
```

Comparaison

Iterative (bottom up) approach produces a shorter code. In addition to that, the complexity of the algorithm is apparent from the code, and it is usually faster than the recursive (top-down) counterpart. However, thinking of a bottom up solution is less intuitive, and their base cases are often trickier than the recursive base cases.



Comparaison

	Memoization	Tabulation
Memory Initialization	Initializes memo array with -1 to mark unfilled states.	Initializes dp array with 0 as base values.
Solution Building Direction	Starts from n and recursively breaks down to smaller subproblems. Also, Only computes states that are actually needed.	Systematically builds solution from smallest subproblem (length 1) to largest (length n). Computes all possible states in a predetermined order.
Handling Base Cases	Uses conditional checking to handle base cases.	Base cases are pre-filled during initialization.
Space Usage	Additional space for recursion stack (O(n) in worst case)	Only requires the DP array O(n)

Comparaison

Top-Down	**Bottom-Up**
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to recursive calls overhead (usually this is not penalized in programming contests) 2. If there are M states, it can use up to $O(M)$ table size which can lead to Memory Limit Exceeded (MLE) for some harder problems	Cons: 1. For some programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

1 Introduction to DP

2 Top-Down Approach

3 Bottom Up Approach

4 Practice

Fibonacci Number

Link

https://leetcode.com/problems/fibonacci-number/description/

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Climbing Stairs

Link

https://leetcode.com/problems/climbing-stairs/description/

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Minimum Path Sum

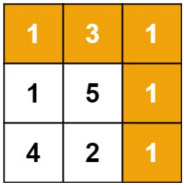
Link

https://leetcode.com/problems/minimum-path-sum/description/

Given a `m x n grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:




```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
```

Unique Paths

Link

https://leetcode.com/problems/fibonacci-number/description/

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

Example 1:


```
Input: m = 3, n = 7
Output: 28
```

Combination Sum

Link

https://leetcode.com/problems/combination-sum-iv/description/

Example 1:

```
Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as
different combinations.
```

Example 2:

```
Input: nums = [9], target = 3
Output: 0
```

Coin Change |

Link

https://leetcode.com/problems/coin-change/description/

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Coin Change ||

Link

https://leetcode.com/problems/coin-change-ii/description/

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the number of combinations that make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `0`.

You may assume that you have an infinite number of each kind of coin.

The answer is **guaranteed** to fit into a signed **32-bit** integer.

Thank you for listening !

Idriss El Abidi
Souhail Louzi