Competitive Programming

Dynamic Programming

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

What is Dynamic Programming?

- Dynamic Programming (DP) is a programming paradigm that can systematically and efficiently explore all possible solutions to a problem.

Characteristics

- The problem can be broken down into "overlapping subproblems" - smaller versions of the original problem that are re-used multiple times.
- The problem has an "optimal substructure" - an optimal solution can be formed from optimal solutions to the overlapping subproblems of the original problem.

Ways to implement a DP algorithm

- Bottom-up, also known as tabulation.
- Top-down, also known as memoization.

Bottom-up (Tabulation)

- Bottom-up is implemented with iteration and starts at the base cases.

Example

- The base cases for the Fibonacci sequence are F(0) = 0 and F(1) = 1.
- With bottom-up, we would use these base cases to calculate F(2), and then use that result to calculate F(3), and so on all the way up to F(n).

Pseudocode

```
F = array of length (n + 1)
F[0] = 0
F[1] = 1
for i from 2 to n:
        F[i] = F[i - 1] + F[i - 2]
```

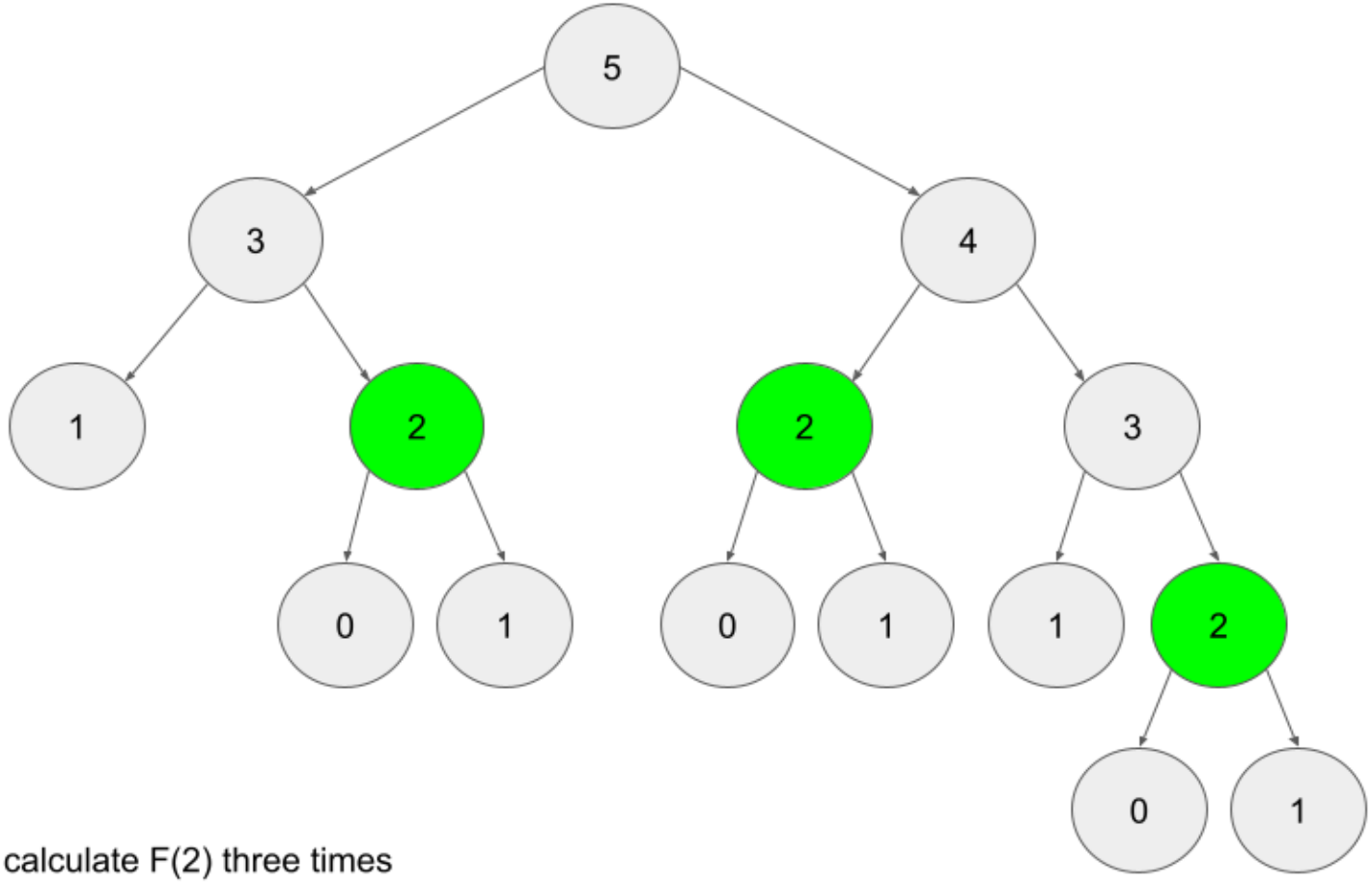
Top-down (Memoization)

- Top-down is implemented with recursion and made efficient with memoization.

Example

- If we wanted to find the n^{th} Fibonacci number F(n), we try to compute this by finding F(n−1) and F(n−2).
- This defines a recursive pattern that will continue on until we reach the base cases F(0) = F(1) = 1.
- The problem with just implementing it recursively is that there is a ton of unnecessary repeated computation.

Recursion tree to find F(5)



We have to calculate F(2) three times

Example

- Notice that we need to calculate F(2) three times.
- This might not seem like a big deal, but if we were to calculate F(6), this entire image would be only one child of the root.
- Imagine if we wanted to find F(100) - the amount of computation is exponential and will quickly explode.
- The solution to this is to memoize results.

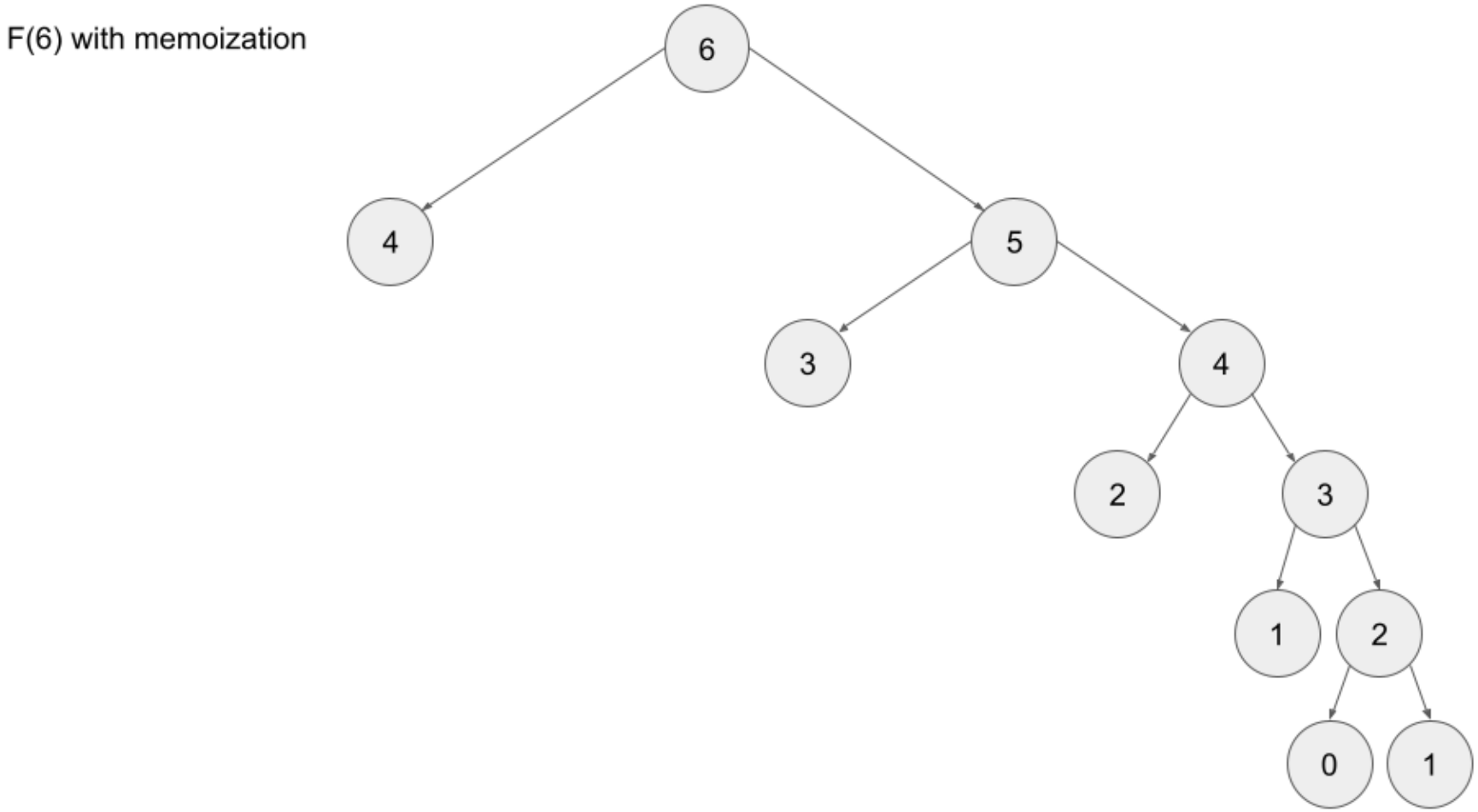
Memoizing

- Memoizing a result means to store the result of a function call, usually in a hashmap or an array, so that when the same function call is made again, we can simply return the memoized result instead of recalculating the result.

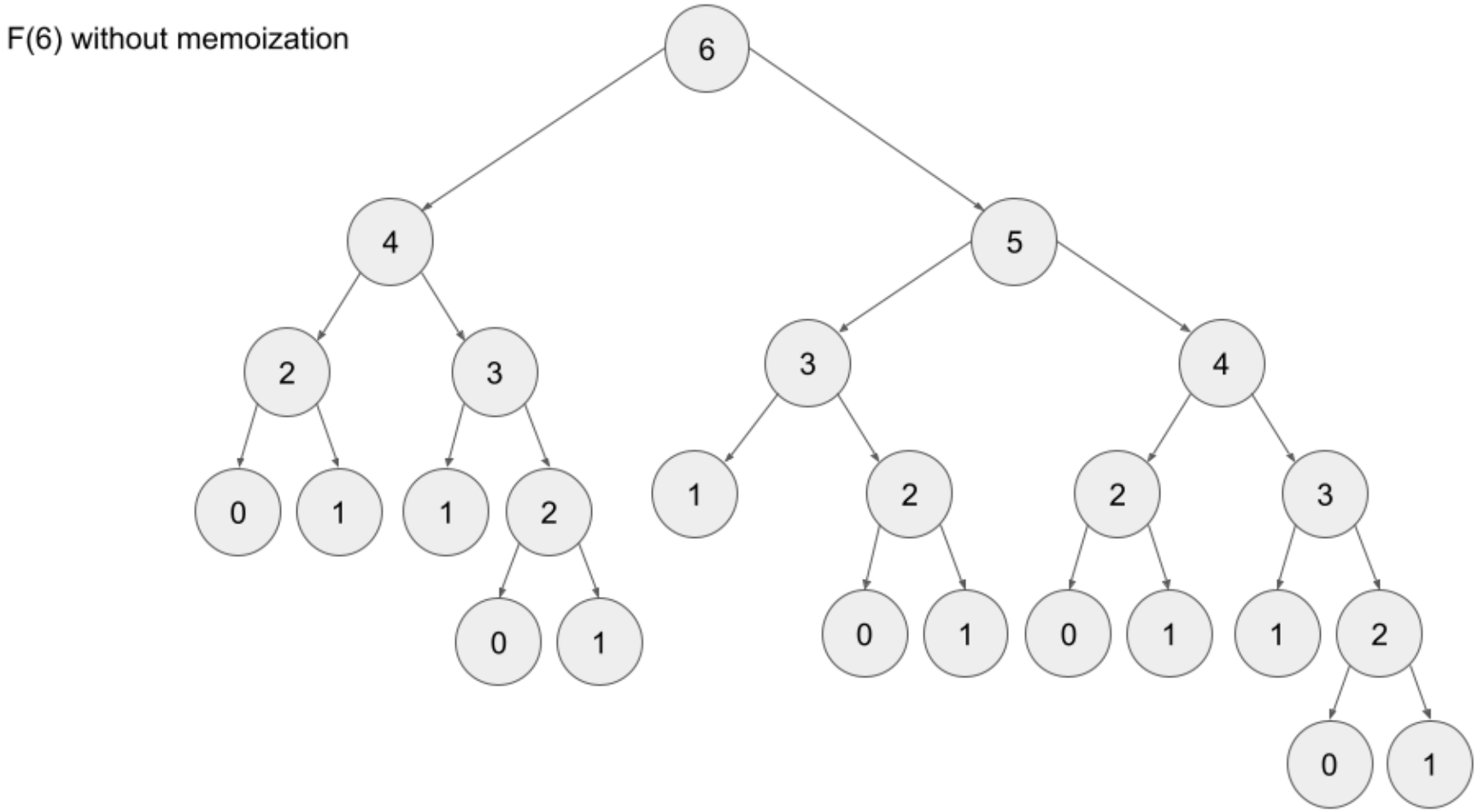
Example

- After we calculate F(2), let's store it somewhere (typically in a hashmap), so in the future, whenever we need to find F(2), we can just refer to the value we already calculated instead of having to go through the entire tree again.

Recursion tree for F(6) with memoization



Recursion tree for F(6) without memoization



Pseudocode

```
memo = hashmap
Function F(integer i):
        if i is 0 or 1:
                return i
        if i doesn't exist in memo:
                memo[i] = F(i - 1) + F(i - 2)
        return memo[i]
```

Which is better?

- Any DP algorithm can be implemented with either method, and there are reasons for choosing either over the other.
- However, each method has one main advantage that stands out:
  - A bottom-up implementation's runtime is usually faster, as iteration does not have the overhead that recursion does.
  - A top-down implementation is usually much easier to write. This is because with recursion, the ordering of subproblems does not matter, whereas with tabulation, we need to go through a logical ordering of solving subproblems.

When to Use DP?

- The problem can be broken down into "overlapping subproblems" - smaller versions of the original problem that are re-used multiple times.
- The problem has an "optimal substructure" - an optimal solution can be formed from optimal solutions to the overlapping subproblems of the original problem.

First Characteristic

- The first characteristic that is common in DP problems is that the problem will ask for the optimum value (maximum or minimum) of something, or the number of ways there are to do something.
- For example:
  - What is the minimum cost of doing...
  - What is the maximum profit from...
  - How many ways are there to do...
  - What is the longest possible...
  - Is it possible to reach a certain point...

Note

- Not all DP problems follow this format, and not all problems that follow these formats should be solved using DP.
- However, these formats are very common for DP problems and are generally a hint that you should consider using dynamic programming.

First Characteristic

- When it comes to identifying if a problem should be solved with DP, this first characteristic is not sufficient.
- Sometimes, a problem in this format (asking for the max/min/longest etc.) is meant to be solved with a greedy algorithm.
- The next characteristic will help us determine whether a problem should be solved using a greedy algorithm or dynamic programming.

Second Characteristic

- The second characteristic that is common in DP problems is that future "decisions" depend on earlier decisions.
- Deciding to do something at one step may affect the ability to do something in a later step.
- This characteristic is what makes a greedy algorithm invalid for a DP problem - we need to factor in results from previous decisions.
- Admittedly, this characteristic is not as well defined as the first one, and the best way to identify it is to go through some examples.

Examples

- House Robber
- Longest Increasing Subsequence

Climbing Stairs

- 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 step + 1 step
    - 2 steps

Example 2

- Input:
  - n = 3
- Output:
  - 3
- Explanation:
  - There are three ways to climb to the top.
    - 1 step + 1 step + 1 step
    - 1 step + 2 steps
    - 2 steps + 1 step

Constraints

- 1 <= n <= 45

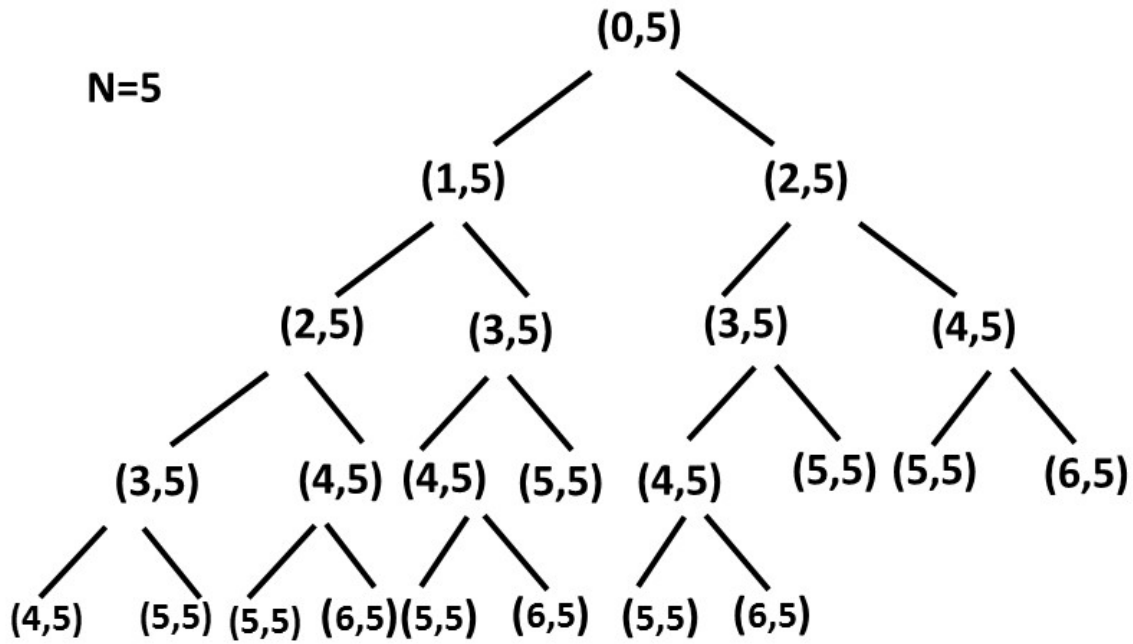
Brute Force

```
int climb_Stairs(int i, int n)
{
        if (i > n)
                return 0;
        if (i == n)
                return 1;
        return climb_Stairs(i + 1, n) + climb_Stairs(i + 2, n);
}
```

Complexity Analysis

- Time complexity : $O(2^n)$.
- Size of recursion tree will be 2^n.
- Space complexity : O(n).
- The depth of the recursion tree can go upto n.

Recursion tree for n=5



Number of Nodes = $O(2^n)$

Recursion with Memoization

```
int climb_Stairs(int i, int n, int memo[])
{
        if (i > n)
                return 0;
        if (i == n)
                return 1;
        if (memo[i] > 0)
                return memo[i];
        memo[i] = climb_Stairs(i + 1,n,memo) + climb_Stairs(i + 2,n,memo);
        return memo[i];
}
```

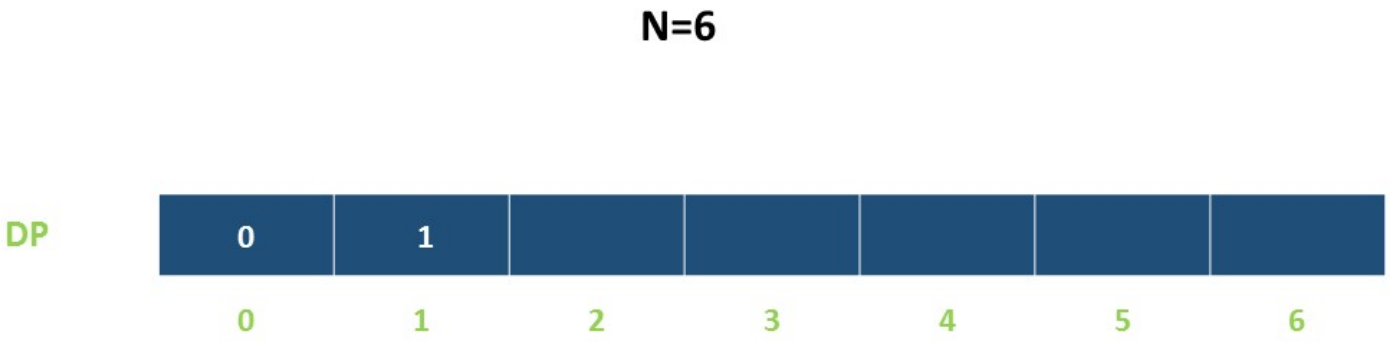
Complexity Analysis

- Time complexity : O(n).
- Size of recursion tree can go upto n.
- Space complexity : O(n).
- The depth of recursion tree can go upto n.

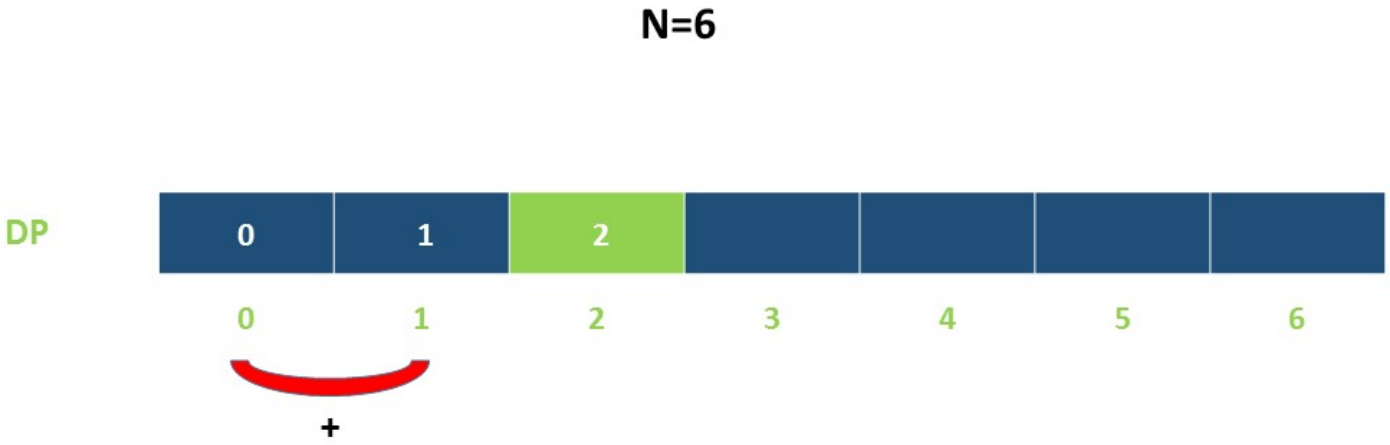
Dynamic Programming

- One can reach i^{th} step in one of the two ways:
  - Taking a single step from $(i-1)^{th}$ step.
  - Taking a step of 2 from $(i-2)^{th}$ step.
- So, the total number of ways to reach i^{th} is equal to sum of ways of reaching $(i-1)^{th}$ step and ways of reaching $(i-2)^{th}$ step.
- Let dp[i] denotes the number of ways to reach on i^{th} step:
  - dp[i] = dp[i − 1] + dp[i − 2]

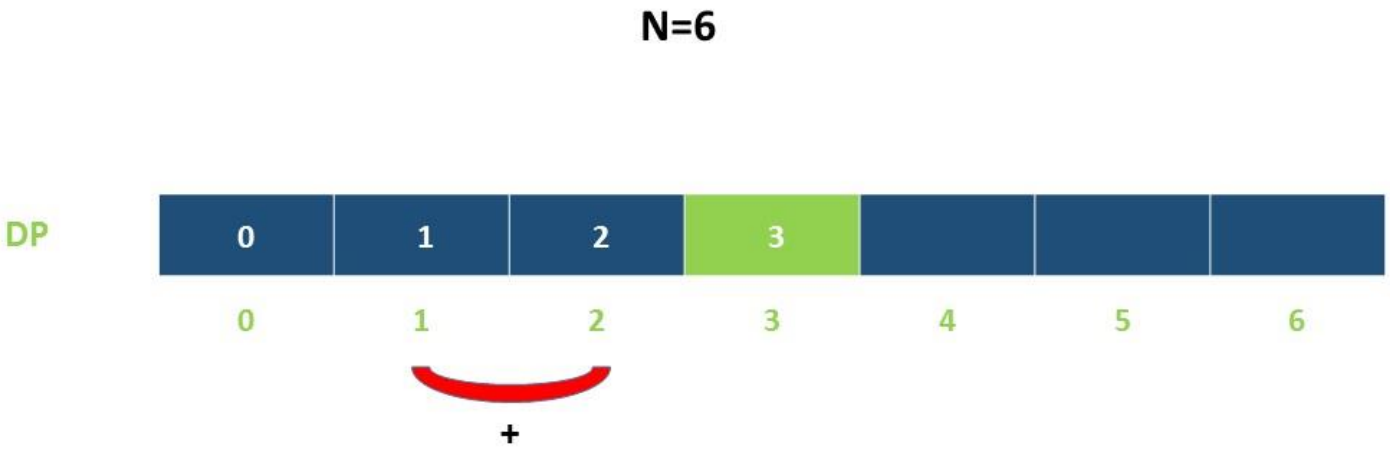
Example



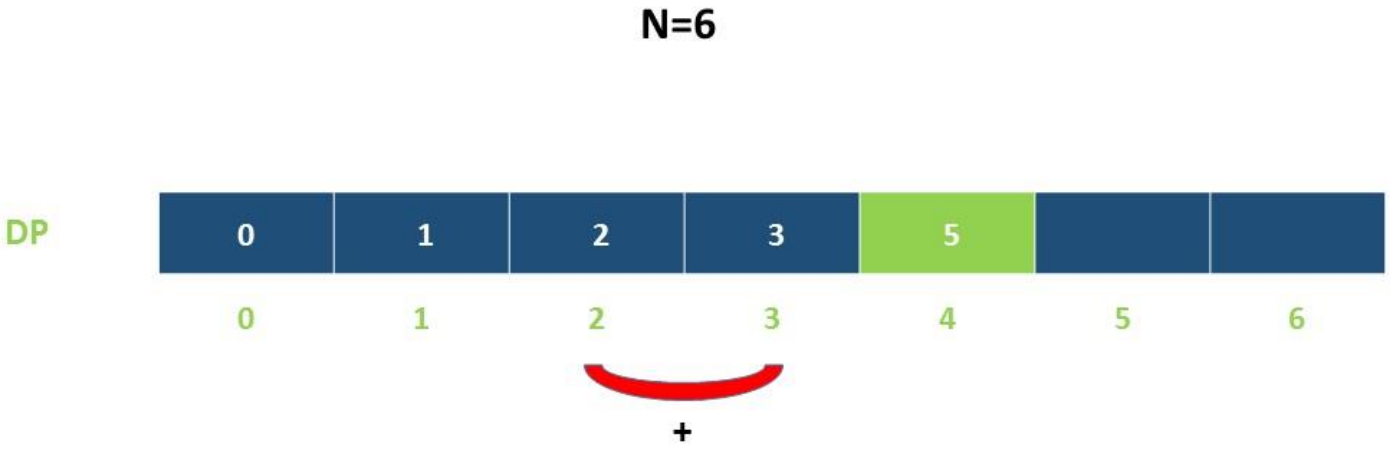
Example



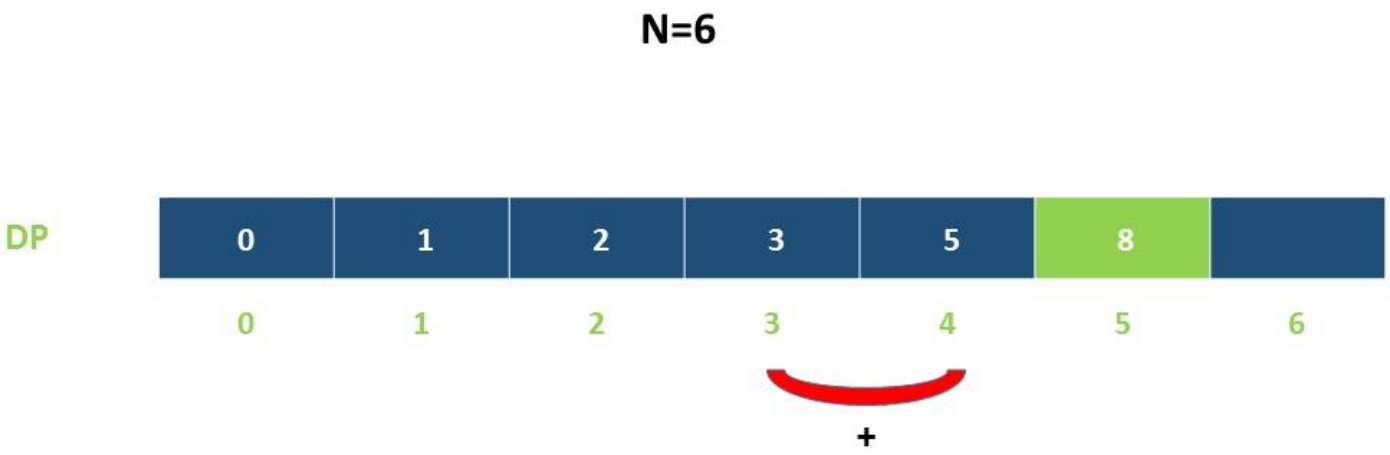
Example



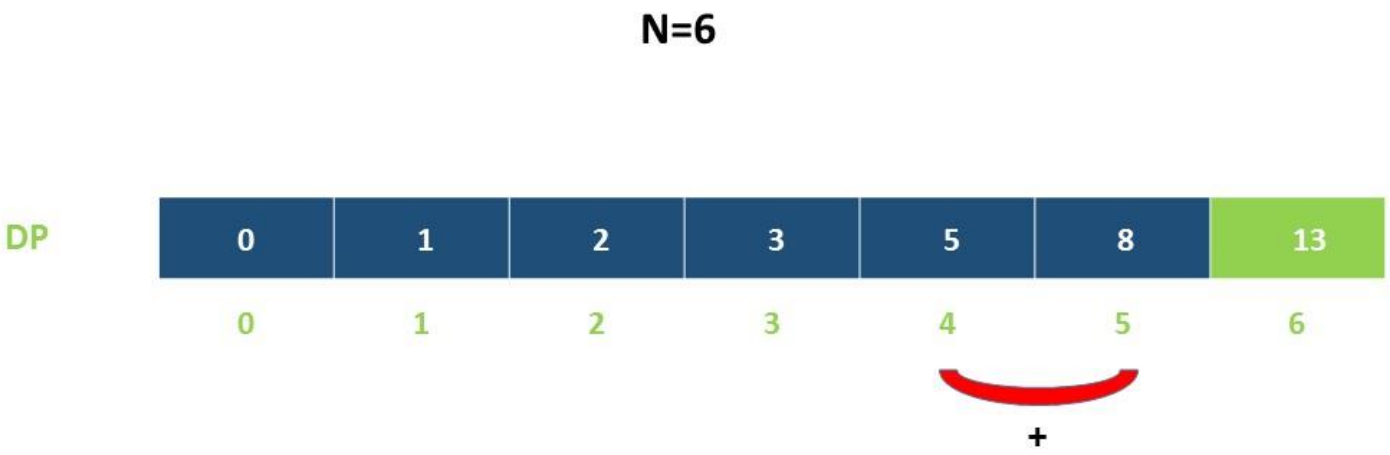
Example



Example



Example



Example

Dynamic Programming

```
int climbStairs(int n)
{
        int i;
        int dp[45] = { 0 };
        if (n == 1)
                return 1;
        dp[1] = 1;
        dp[2] = 2;
        for (i = 3; i <= n; i++)
                dp[i] = dp[i - 1] + dp[i - 2];
        return dp[n];
}
```

Complexity Analysis

- Time complexity : O(n).
- Single loop upto n.
- Space complexity : O(n).
- dp array of size n is used.

Fibonacci Number

- In the above approach we have used dp array where dp[i]=dp[i−1]+dp[i−2]. It can be easily analysed that dp[i] is nothing but i^{th} fibonacci number.
- Fib(n) = Fib(n − 1) + Fib(n − 2)
- Now we just have to find n^{th} number of the fibonacci series having 1 and 2 their first and second term respectively, i.e. Fib(1)=1 and Fib(2)=2.

Fibonacci Number

```
int climbStairs(int n)
{
        int i, first, second, third;
        if (n == 1)
                return 1;
        first = 1;
        second = 2;
        for (i = 3; i <= n; i++)
        {
                third = first + second;
                first = second;
                second = third;
        }
        return second;
}
```

Complexity Analysis

- Time complexity : O(n).
- Single loop upto n is required to calculate n^{th} fibonacci number.
- Space complexity : O(1).
- Constant space is used.

House Robber

- You are a professional robber planning to rob houses along a street.
- Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
- Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1

- Input:
  - nums = [1,2,3,1]
- Output:
  - 4
- Explanation:
  - Rob house 1 (money = 1) and then rob house 3 (money = 3).
  - Total amount you can rob = 1 + 3 = 4.

Example 2

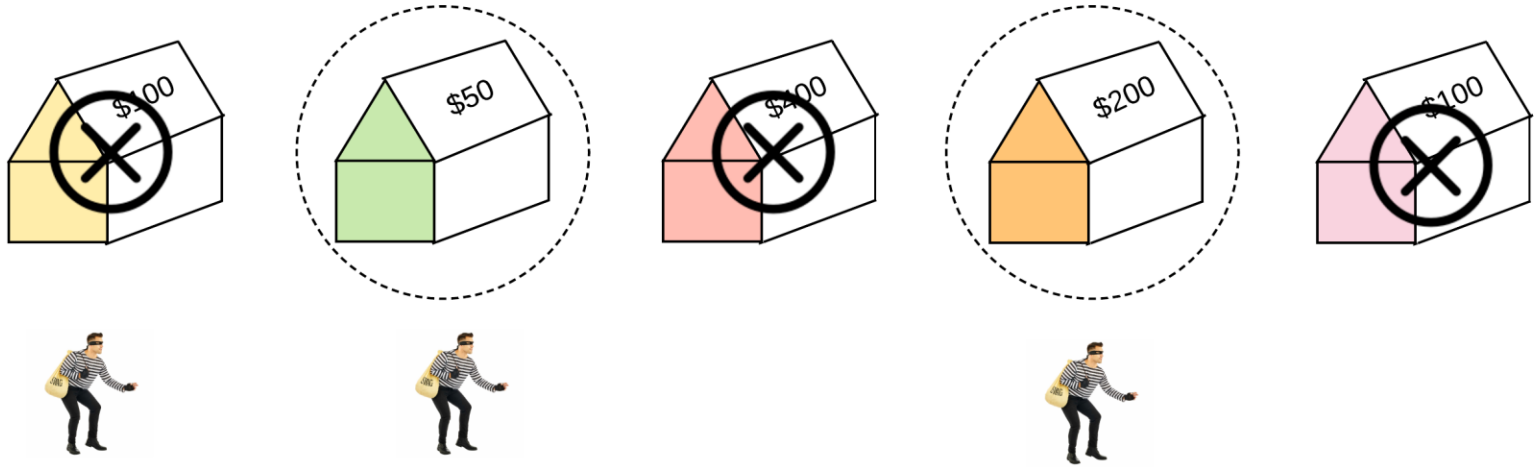
- Input:
  - nums = [2,7,9,3,1]
- Output:
  - 12
- Explanation:
  - Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
  - Total amount you can rob = 2 + 9 + 1 = 12.

Constraints

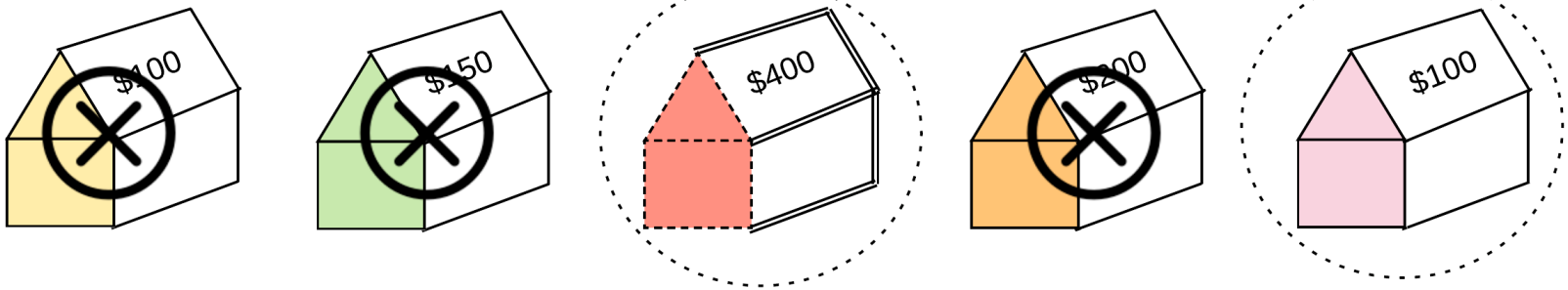
- 1 <= nums.length <= 100
- 0 <= nums[i] <= 400

Robber making the optimal choices

Robber making sub-optimal choices



Failure of a greedy strategy



Skip this house since the next house gives more money than this one.

Ummm, let's skip this one too since the neighbors are giving more than this house.

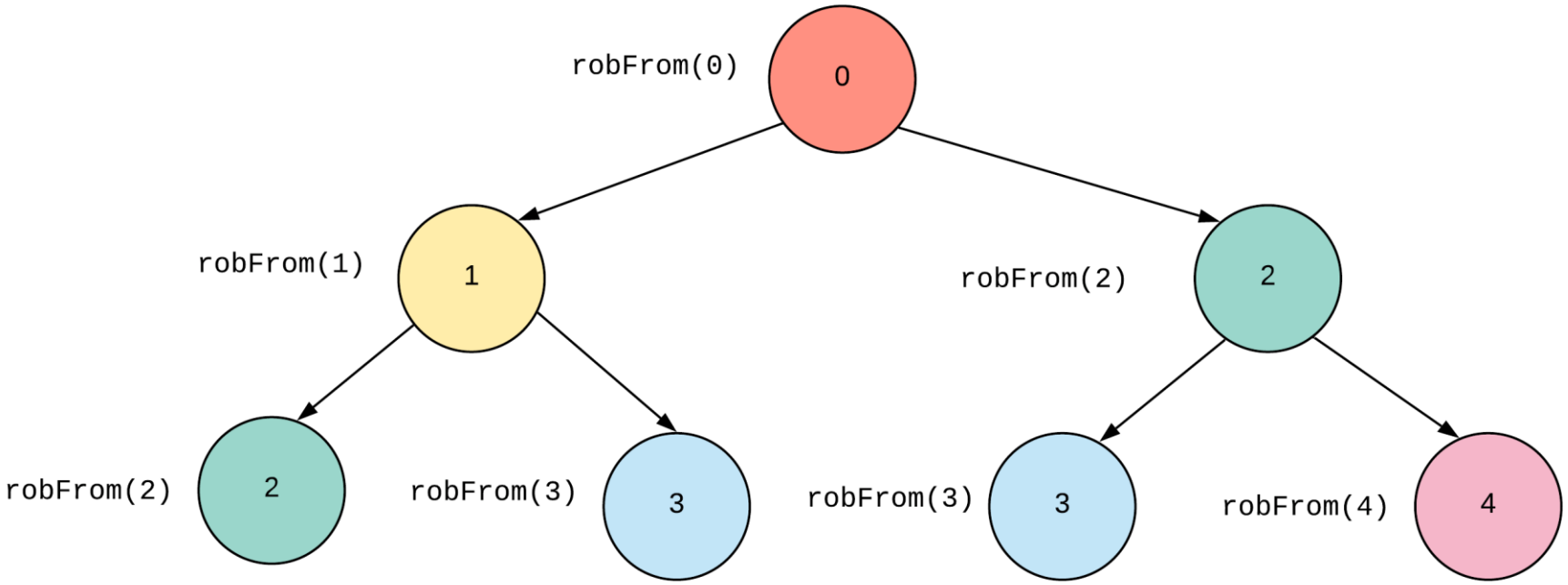
Yayy! I hit the jackpot with this house.

Total = $500

Recursion with Memoization

- To approach a problem recursively, we need to make sure that it can be broken down into sub-problems.
- Additionally, we need to ensure that the optimal solution to these sub-problems can be used to form the solution to the main problem.

Overlapping sub-problems



Recursion with Memoization

```
int robFrom(int i, int nums[], int n)
{
        int ans;
        // No more houses left to examine.
        if (i >= n)
                return 0;
        // Return cached value.
        if (memo[i] > -1)
                return memo[i];
        // Recursive relation evaluation to get the optimal answer.
        ans = MAX(robFrom(i + 1, nums), robFrom(i + 2, nums) + nums[i]);
        // Cache for future use.
        memo[i] = ans;
        return ans;
}
```

Complexity Analysis

- Time Complexity: O(N) since we process at most N recursive calls, thanks to caching, and during each of these calls, we make an O(1) computation which is simply making two other recursive calls, finding their maximum, and populating the cache based on that.
- Space Complexity: O(N) which is occupied by the cache and also by the recursion stack.

Dynamic Programming

```
int rob(int nums[], int n)
{
        int i, maxRobbedAmount[n + 1];
        // Special handling for empty array case.
        if (n == 0)
                return 0;
        // Base case initializations.
        maxRobbedAmount[n] = 0;
        maxRobbedAmount[n - 1] = nums[n - 1];
        // DP table calculations.
        for (i = n - 2; i >= 0; --i)
                maxRobbedAmount[i] = MAX(maxRobbedAmount[i + 1],
                                         maxRobbedAmount[i + 2] + nums[i]);
        return maxRobbedAmount[0];
}
```

Complexity Analysis

- Time Complexity: O(N) since we have a loop from N – 2 . . . 0 and we simply use the pre-calculated values of our dynamic programming table for calculating the current value in the table which is a constant time operation.
- Space Complexity: O(N) which is used by the table.
- So what is the real advantage of this solution over the previous solution?
  - In this case, we don't have a recursion stack. When the number of houses is large, a recursion stack can become a serious limitation, because the recursion stack size will be huge and the compiler will eventually run into stack-overflow problems (no pun intended!).

Optimized Dynamic Programming

```
int rob(int nums[], int n)
{
        int i, current;
        int robNext, robNextPlusOne;
        // Special handling for empty array case.
        if (n == 0)
                return 0;
        // Base case initializations.
        robNextPlusOne = 0;
        robNext = nums[n - 1];
        // DP table calculations. Note: we are not using any
        // table here for storing values. Just using two
        // variables will suffice.
        for (i = n - 2; i >= 0; --i)
        {
                current = MAX(robNext, robNextPlusOne + nums[i]);
                // Update the variables
                robNextPlusOne = robNext;
                robNext = current;
        }
        return robNext;
}
```

Complexity Analysis

- Time Complexity: O(N) since we have a loop from N – 2 . . . 0 and we use the precalculated values of our dynamic programming table to calculate the current value in the table which is a constant time operation.
- Space Complexity: O(1) since we are not using a table to store our values.
- Simply using two variables will suffice for our calculations.

Queries?

Thank You…!