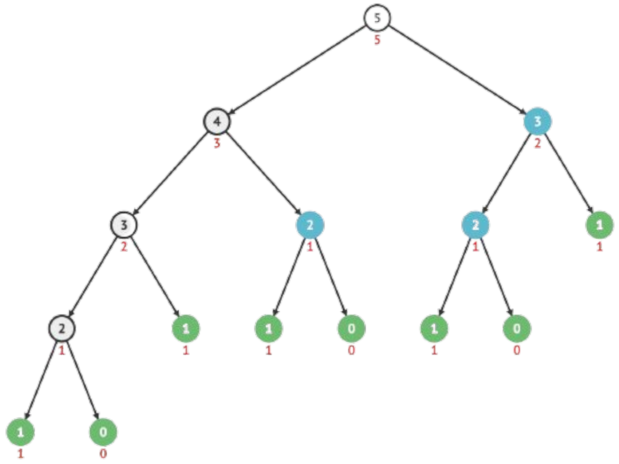
Dynamic Programming

Top-down Approach



Africa To Silicon Valley

Lecture Flow

- Real Life problem
- Memoization
- Optimal Substructure and overlapping subproblems
- Common Variants
- Common Pitfalls
- Recognizing in questions
- Checkpoint
- References and Resources
- Practice Questions
- Quote of the day

Problem: The Fibonacci Series

Problem description

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones. It starts with 0 and 1, and the subsequent numbers are obtained by adding the two previous numbers. The sequence begins as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Problem description

Now, let's consider a problem that requires computing the nth Fibonacci number.

Input: n = 6

Output: ?

A2SV
Africa To Silicon Valley

Problem description

Now, let's consider a problem that requires computing the nth Fibonacci number.

Input: n = 6

Output: 8

A2SV
Africa To Silicon Valley

How did we solve this problem?

Recursive Definition

- Fib(0) = 0
- Fib(1) = 1
- Fib(n) = Fib(n-1) + Fib(n-2)   (for n > 1)

Implementation

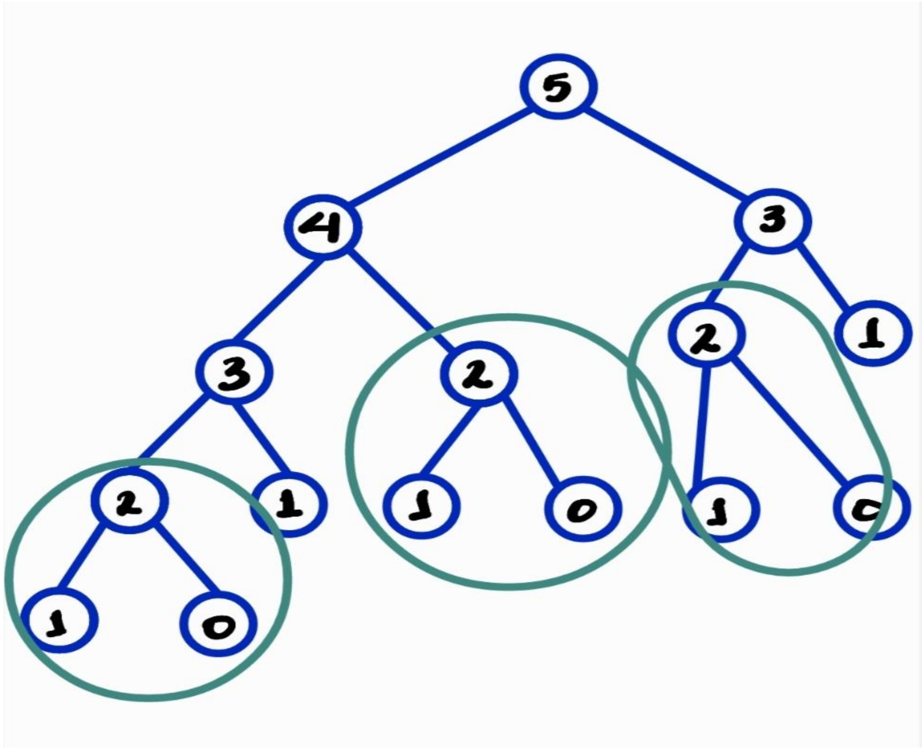
```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

Visualization link

How many times did we compute the subproblem fib(2) when computing for fib(5)?

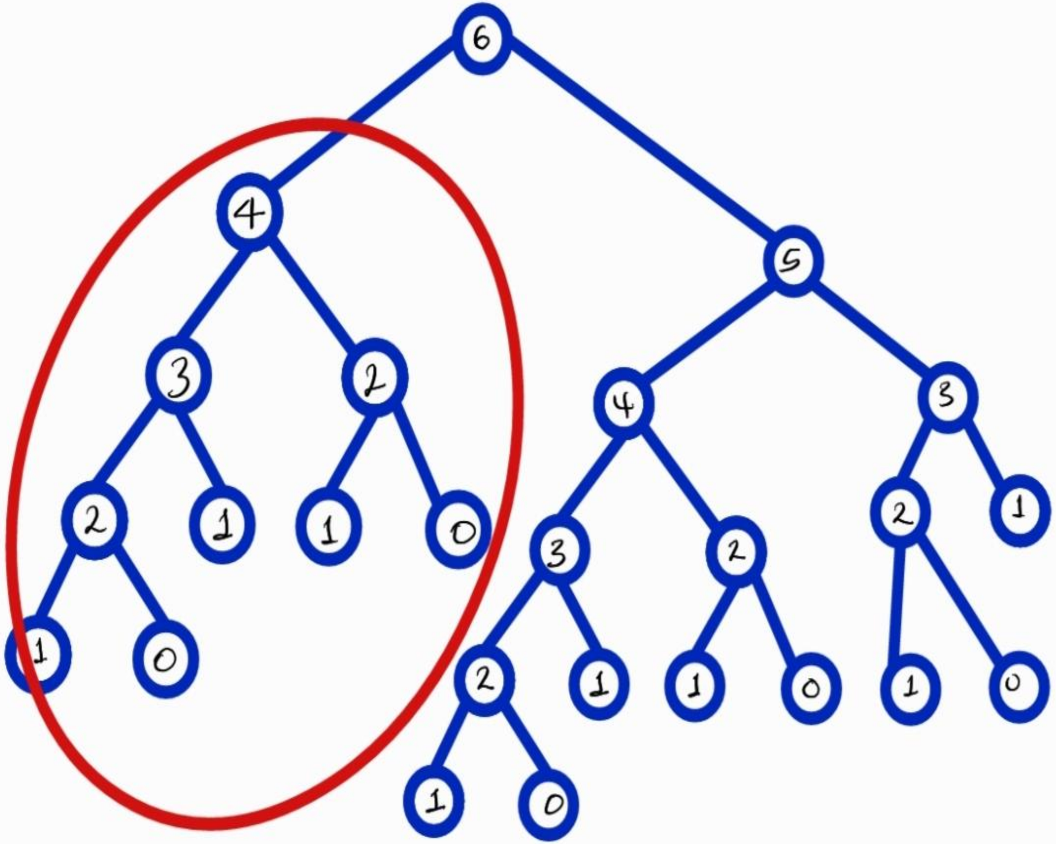
Decision Tree



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

This is called **Memoization**.

To see how the repetition happens exponentially, let's compute Fib(6)

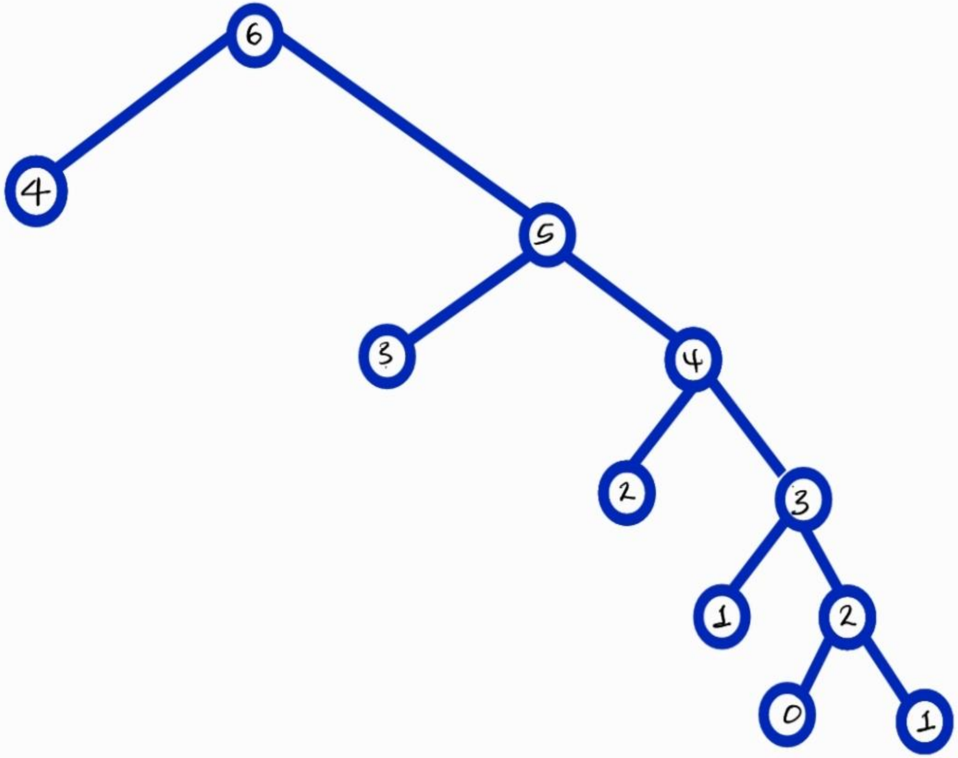


Notice how an entire subtree is repeated

What is the **time complexity** of computing Nth fibonacci number **using recursion** ?

What would the tree look like if we add **Memoization** ?

Visualization link



Memoization effectively removes repeated (Wasted) Computations.

What is the **time complexity** of computing Nth fibonacci number using **recursion with 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]
```

Practice Problem

Introduction

- **Dynamic programming (DP)** is a technique that solves problems by **breaking** them into **overlapping subproblems** and reusing their solutions.

TOP-DOWN DP = Recursion + Memoization

To solve a problem with Top-down DP, we need to combine 3 things:

1. A State that will compute/contain the answer to the problem for every given state.
2. A Recurrence relation to transition between states.
3. Base cases, so that our recurrence relation doesn't go on infinitely.

And MEMOIZATION.

Two conditions must be met for DP

1. Optimal substructure property

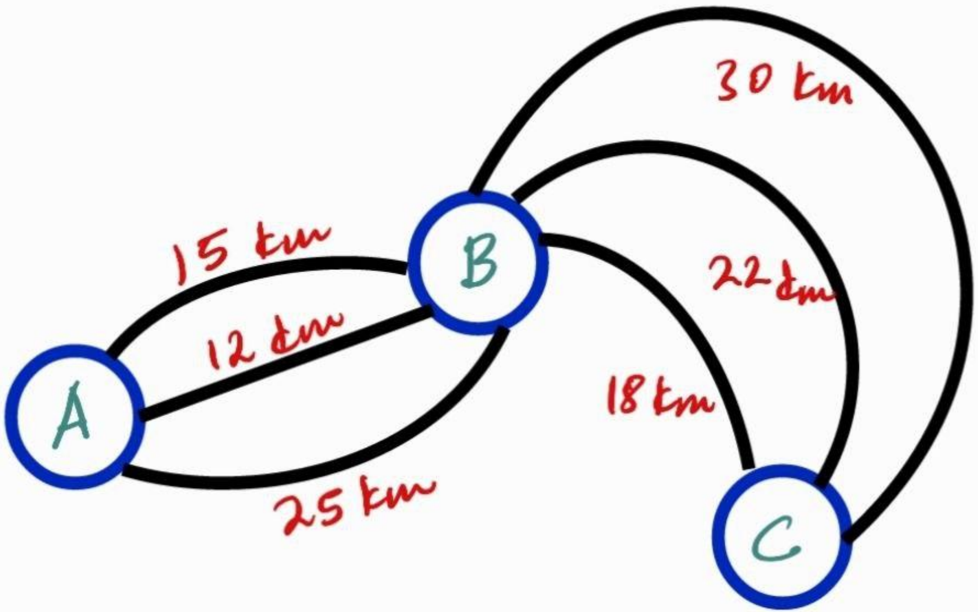
Am I just a recursion ?

1. Optimal substructure property

Optimal solution to a problem of size n (having n elements) is based on an optimal solution to the same problem of smaller size (less than n elements).

1. Optimal substructure property

Consider finding the shortest path for travelling between two cities by car. A person wants to drive from city A to city C, city B lies in between the two cities.



Shortest path between city A and city C ?

1. Optimal substructure property

The shortest path of going from A to C (30 km) will involve both, taking the shortest path from A to B and shortest path from B to C.

2. **Overlapping subproblems**

The book is delayed because of laziness, procrastination and lack of discipline. Oops! The problems overlap.

2. Overlapping subproblems

When we compute 20th term of Fibonacci without using memoization, then fib(3) is called 2584 times and fib(10) is called 89 times. It means that we are recomputing the 10th term of Fibonacci 89 times from scratch.

Commonly asked DP questions

A2SV
Africa To Silicon Valley

1.Count Number of Ways

A2SV
Africa To Silicon Valley

Given a target find a number of distinct ways to reach the target.

70. Climbing Stairs

Easy 18155 571 Add to List Share

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
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
```

State

What specific information or variable represents the current state in the problem of climbing a staircase?

State: Current position ‘i’ on the staircase.

Function: dp(i) returns the number of ways to climb to the i’th step.

Recurrence Relation

To determine the number of ways to climb to the 30th stair, let's consider our available options.

We can arrive at the 30th stair by taking either 1 or 2 steps at a time.

Recurrence Relation

How can we arrive at the 30th stair? Are there any possible combinations of steps that lead directly to the 30th stair?

Recurrence Relation

We can take one step from 29th state after arriving there.

Which means from dp(i - 1).

Recurrence Relation

We can take two steps from 28th state after arriving there.

Which means from dp(i - 2).

Recurrence Relation

Therefore, the number of ways we can climb to the 30th stair is equal to the number of ways we can climb to the 28th stair plus the number of ways we can climb to the 29th stair.

Recurrence Relation

$$dp(i) = dp(i - 1) + dp(i - 2)$$

Base Case

- Base case 1: dp(1) = 1 (one way to climb to the first stair).
- Base case 2: dp(2) = 2 (two ways to climb to the second stair).

Implementation

```python
def dp(n: int) -> int:
    if n < 3:
        return n
    return dp(n - 1) + dp(n - 2)
```

Do you **notice** something **missing** from the code?

We **haven't** added **memoization** yet.

```
memo = {}

def dp(n: int) -> int:
        if n < 3:
            return n
        if n not in memo:
            memo[n] = dp(n - 1) + dp(n - 2)
    return memo[n]
```

Practice Problem

2. Minimize or Maximize certain value (Optimizations)

Africa To Silicon Valley

Practice Problem

198. House Robber

Medium 17720 335 Add to List Share

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.
```

State

What specific information or variable represents the current state in the problem of robbing houses ?

Don’t start robbing houses guys :)

State

The **index of a house**.

Function dp(i) returns the **maximum** amount of money you can rob **up to and including** house i,

Recurrence Relation

If we are at some house, logically, we have **2 options**: we can choose to **rob** this house, or we can choose to **not rob** this house.

To be or not to be, Shakespeare in the park :)

Recurrence Relation

If we choose **not to rob** the current house, our available money remains the **same** as the previous house: **dp(i - 1)**.

Recurrence Relation

If we choose to **rob** the current house, the money gained is equal to **nums[i]**.

However, this is **only possible** if we **did not rob** the **previous house**. In that case, the total money we would have is **dp(i - 2) + nums[i]**.

A2SV
Africa To Silicon Valley

Recurrence Relation

$$dp(i) = \max(dp(i - 1), dp(i - 2) + nums[i])$$

Base Case

- If there is **only one** house, then the **most money** we can make is by **robbing the house**.
    - **dp(0) = nums[0]**

Base Case

- If there are **only two houses**, then the most money we can make is by robbing the house **with more money.**
  - **dp(1) = max(nums[0], nums[1])**

Implementation

```
def dp(i):
        if i == 0:
            return nums[i]
        if i == 1:
            return max(nums[0],nums[1])
        if i not in memo:
            memo[i] = max(dp(i - 1), dp(i - 2) + nums[i])
        return memo[i]
memo = {}
return dp(len(nums) - 1)
```

Implementation

```python
def dp(i):
    if i == 0:
        return nums[i]
    if i == 1:
        return max(nums[0],nums[1])
    if memo[i] == -1:
        memo[i] = max(dp(i - 1), dp(i - 2) + nums[i])
    return memo[i]

n = len(nums)
memo = [-1 for _ in range(n)]
return dp(n - 1)
```

A2SV
Africa To Silicon Valley

3. Yes/No Questions

Africa To Silicon Valley

416. Partition Equal Subset Sum

Medium 10428 186 Add to List Share

Given an integer array `nums`, return `true` *if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or* `false` *otherwise.*

Example 1:

```
Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

Example 2:

```
Input: nums = [1,2,3,5]
Output: false
Explanation: The array cannot be partitioned into equal sum
subsets.
```

Pair Programming

Implementation

```python
def canPartition(self, nums: List[int]) -> bool:
        n = len(nums)
        def dp(i, first_part, second_part):
            if i >= n:
                return sum(first_part) == sum(second_part)

            return dp(i + 1, first_part + [nums[i]], second_part) or\
                dp(i + 1, first_part, second_part + [nums[i]])

        return dp(0, [], [])
```

What is the **issue** with the above **implementation** ?

A better Implementation

```
def canPartition(self, nums: List[int]) -> bool:

        n = len(nums)
        def dp(i, target_sum):
            if i >= n or target_sum <= 0:
                return target_sum == 0

            return dp(i + 1, target_sum - nums[i]) or\
                dp(i + 1, target_sum)

        return sum(nums) % 2 == 0 and dp(0, sum(nums) // 2)
```

Don’t forget **Memoizing** :)

A better Implementation - Not

```
def canPartition(self, nums: List[int]) -> bool:

    n = len(nums)
    memo = [-1 for _ in range(n)]
    def dp(i, target_sum):
        if i >= n or target_sum <= 0:
            return target_sum == 0

        if memo[i] == -1:
            memo[i] = dp(i + 1, target_sum - nums[i]) or\
                dp(i + 1, target_sum)

        return memo[i]

    return sum(nums) % 2 == 0 and dp(0, sum(nums) // 2)
```

Don't forget **a problem is defined by its states. All of them.**

A better better Implementation

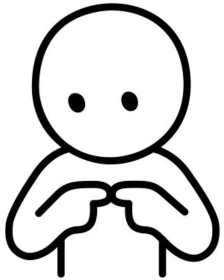
```python
def canPartition(self, nums: List[int]) -> bool:
    n = len(nums)
    memo = defaultdict(int)
    def dp(i, target_sum):
        if i >= n or target_sum <= 0:
            return target_sum == 0

        state = (i, target_sum)
        if state not in memo:
            memo[state] = dp(i + 1, target_sum - nums[i]) or\
                dp(i + 1, target_sum)

        return memo[state]

    return sum(nums) % 2 == 0 and dp(0, sum(nums) // 2)
```

Recognizing in questions



Common Pitfalls

Africa To Silicon Valley

1. Thinking of greedy approach

- Greedy algorithms aim for local optimality, but they can fall short in finding the global optimum, leading to suboptimal or incorrect outcomes.

continued…

E.g. Consider the "Minimum Coin Change" problem, where you are given a target amount and a set of coin denominations. The task is to find the minimum number of coins needed to make up the target amount.

Greedy approach would be to always select the largest coin denomination that is less than or equal to the remaining amount at each step.

consider the target amount 11 with denomination coins [1,2,5]

continued…

- However, this greedy approach can fail to find the minimum number of coins in some cases.

  consider the target amount 14 with denomination coins [5, 3, 2].

  If we go greedily we will choose 5, 5, 3 and end up without getting the answer.

  But the optimal solution is 5, 5, 2, 2.

2. Incorrect Recurrence Relation

- Formulating an incorrect recurrence relation, can lead to incorrect results or inefficiencies.

3. Lack of proper memoization

- Forgetting to apply memoization or implementing it incorrectly can lead to redundant computations.

4. Inefficient time complexity

DP is a smart bruteforce at the end of the day. Hence, it can be applied to most problems but it does not mean it is always efficient. Especially, if the subproblems have little to none overlappingness.

Don't fail to consider alternative data structures or optimizing the recurrence relation.

Checkpoint

[Link]

Africa To Silicon Valley

Practice Questions

N-th Tribonacci Number

Coin Change

Target Sum

Unique Paths

Minimum Path Sum

Best Time to Buy and Sell Stock with Transaction Fee

Best Time to Buy and Sell Stock with Cooldown

House Robber III

Delete and Earn

Resources

- Leetcode Explore Card
- Dynamic Programming lecture #1 - Fibonacci, iteration vs recursion
- Competitive Programmer's Handbook
- Dynamic programming for Coding Interviews: A bottom-up approach to problem solving

Quote of the day

"

Those who cannot remember the past are condemned to repeat it.

"

A2SV

Africa To Silicon Valley