Dynamic Programming

A top-down approach

Part I

Objectives

- Introduce the basic concept of dynamic programming
- Explain Memoization and it’s implementation
- Introduce common variants of dynamic programming and identify patterns of problems that can be solved using DP
- Explain how to optimize common problem patterns using Dynamic Programming

Lecture Flow

- Prerequisites
- The Fibonacci Sequence
- Memoization
- Optimal Substructure and Overlapping subproblems
- Common Variants
- Applications
- Common Pitfalls
- Checkpoint
- References and Resources
- Practice Questions
- Quote of the day

Prerequisites

- Recursion I

The Fibonacci Sequence

The Fibonacci Sequence

- 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, . . .
```

The Fibonacci Sequence

- What is the `n`-th Fibonacci number (Zero based counting) ?

```
Input: n = 6

Output: ?
```

The Fibonacci Sequence

- What is the `n`-th Fibonacci number?

```
Input: n = 6

Output: 8
```

How do we solve such problems?

- We have the following recursive definition

```
fib(0) = 0

fib(1) = 1

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

Implementation

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

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

Time Complexity = ?
Space Complexity = ?

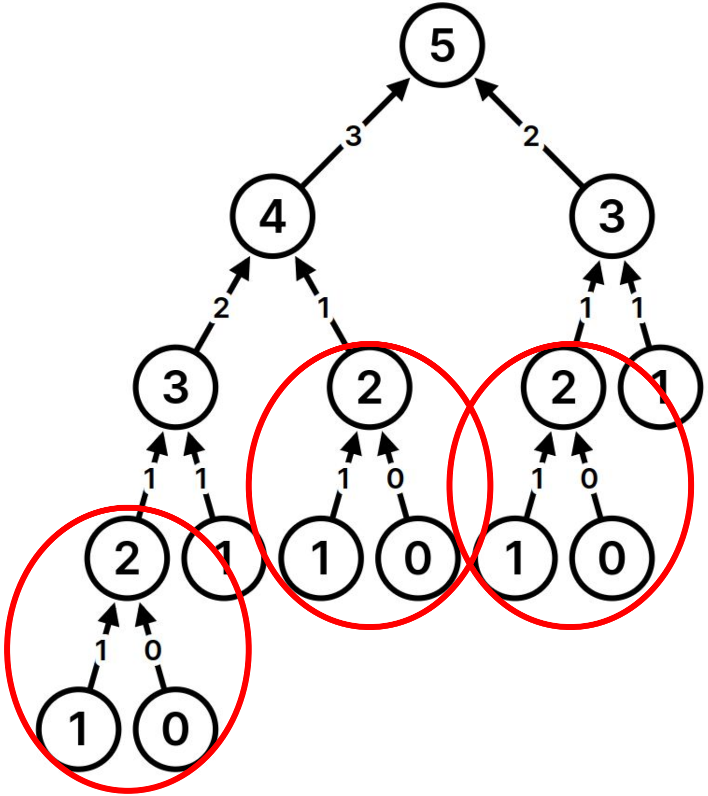
The Fibonacci Sequence

[Visualization link]

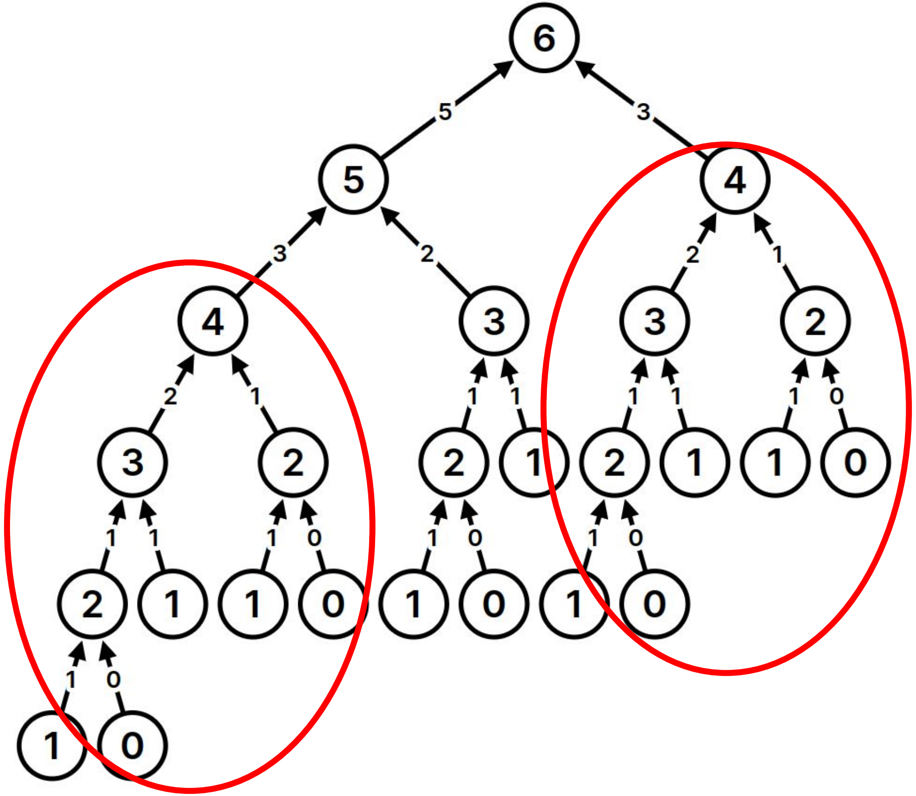
The Fibonacci Sequence

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

The Execution Tree



The Execution Tree



The Fibonacci Sequence

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

Memoization - Pseudocode

```
memo = Hashmap/Array

function fib(integer i):

    if i is 0 or 1:

        return i

    if i doesn't exist in memo:

        memo[i] = fib(i - 1) + fib(i - 2)

    return memo[i]
```

Time Complexity = ?
Space Complexity = ?

Memoization - Pseudocode

```
memo = Hashmap/Array

function fib(integer i):

    if i is 0 or 1:

        return i

    if i doesn't exist in memo:

        memo[i] = fib(i - 1) + fib(i - 2)

    return memo[i]
```

Time Complexity = `O(n)`
Space Complexity = `O(n)`

Memoization - Practice

[Problem Link]

Dynamic Programming

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

Introduction

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

1. A state is the information required to determine the result(return) of a function
2. A recurrence relation to transition between states.
3. Base case(s), so that our recurrence relation doesn't go on infinitely.

and . . .

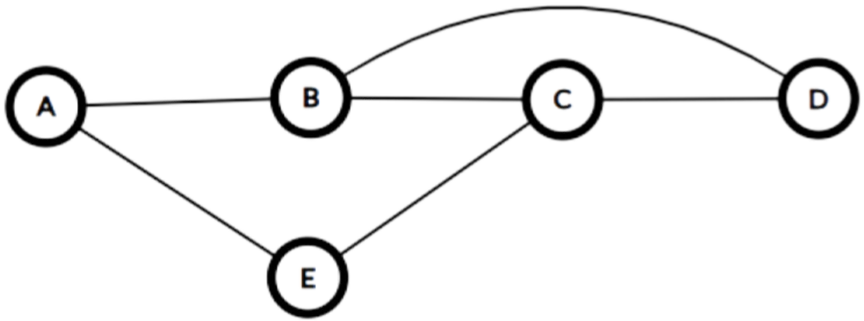
Memoization

Two Conditions

Optimal Substructure

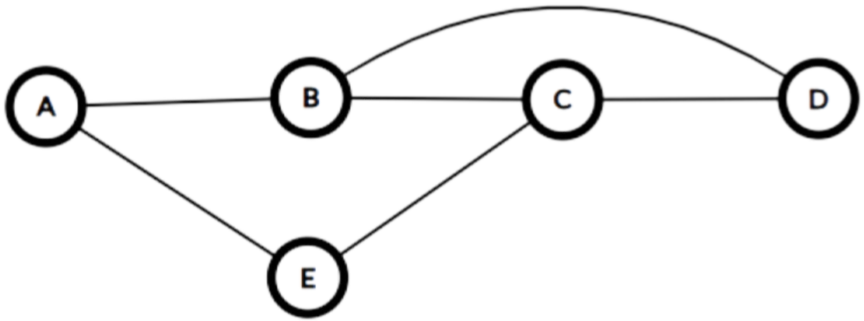
Optimal solution to a problem of size `n` (having `n` elements) is **based on an optimal solution to the same problem of smaller size** (less than `n` elements).

Optimal Substructure



Shortest path between city **A** and city **D** ?

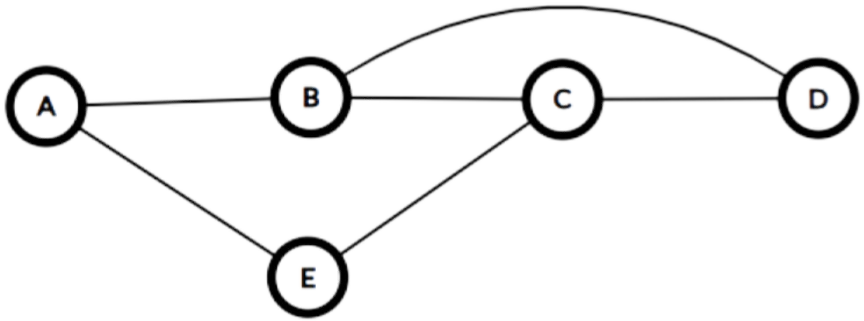
Optimal Substructure



Shortest path between city **A** and city **D** ?

The shortest path of going from A to D (2 edges) will involve both, taking the shortest path from A to B and shortest path from B to D.

Optimal Substructure



Longest path between city **A** and city **D** **with no repeated vertex**?

This has no "good" optimal substructure. Why not?

Overlapping Subproblems

Overlapping subproblems are when you **repeatedly encounter the same smaller problems** while solving a larger problem.

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.

Common DP Applications

Enumeration

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

Enumeration

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

Enumeration

State: Current position `‘i’` on the staircase.

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

Enumeration - 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.

Enumeration - Recurrence Relation

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

Enumeration- Recurrence Relation

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

Which means from dp(i - 1).

Enumeration - Recurrence Relation

We can take one step from 28th state after arriving there.

Which means from dp(i - 2).

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

Enumeration - 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).

Optimizations

Problem Link

Optimizations

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

Don’t start robbing houses guys :)

Optimizations - State

State: The **index** of a house.

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

Optimizations - Recurrence Relations

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 :)

Optimizations - Recurrence Relations

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

Optimizations - Recurrence Relations

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]`.

Optimizations - Recurrence Relations

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

Optimizations - Base Case

- If there is **only one** house, then the **most money** we can make is by **robbing the house.**

- If there are **only two houses**, then the most money we can make is by robbing the house with more money.

```
dp(0) = nums[0]

dp(1) = max(nums[0], nums[1])
```

Yes/No Questions

[Pair Programming]

Yes/No Questions - Implementation

What is the issue with the above implementation ?

Yes/No Questions - Implementation

And Memoize . . .

Yes/No Questions - Implementation

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

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

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.”

— George Santayana