UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Advanced algorithm II Dynamic Programming

Mai Dahshan

November 4, 2024

Learning Objectives

- Understand the concept of dynamic programming
- Be able to identify if a given problem can be solved by DP
- Be able to formulate a DP solution
- Be able to implement the DP solution of given problems.

Example: Finding the n-th Fibonacci number

- Let's say that we want to find the **5th** Fibonacci number
- Rules: $F_0 = 0$, $F_1 = 1$, $F(n) = F(n-1) + F(n-2)$

n	0	1	2	3	4	5
Fib_n:	0	1	1	2	3	5

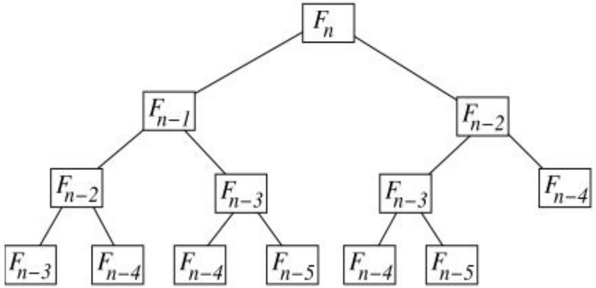
Example: Finding the n-th Fibonacci number

- *Basic rules for calculation:*

  $F_0 = 0,$

  $F_1 = 1,$

  $F_n = F_{n-1} + F_{n-2}$




```
FUNCTION Fibonacci(n)
    // Check for invalid input
    IF n < 0 THEN
        RETURN "Invalid input"

    // Base cases
    IF n = 0 THEN
        RETURN 0
    ELSE IF n = 1 THEN
        RETURN 1

    // Recursive case
    RETURN Fibonacci(n − 1) + Fibonacci(n − 2)
```

Example: Finding the n-th Fibonacci number

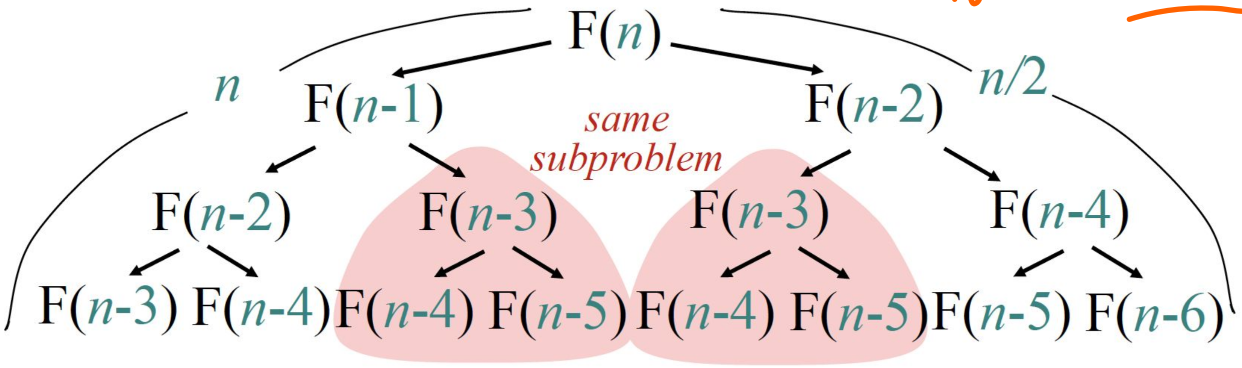
```python
def fib(n):
    # Define a function named 'fib' that calculates the nth Fibonacci number.
    # The function takes a single integer parameter 'n'.
    if n == 0: # Check if 'n' is 0. This is a base case for the Fibonacci sequence.Fibonacci(0) is defined as 0.
        return 0
    elif n == 1: # Check if 'n' is 1. This is another base case for the Fibonacci sequence.Fibonacci(1) is defined as 1
        return 1
    else:
        # If 'n' is greater than 1, the function computes the Fibonacci number
        # recursively by summing the results of fib(n - 1) and fib(n - 2).
        return fib(n - 1) + fib(n - 2)
```

Tracing Recursive Fibonacci(5)

- For Fibonacci(5), we call **Fibonacci(4)** and **Fibonacci(3)**
    - For **Fibonacci(4)**, we call Fibonacci(3) and Fibonacci(2)
        - For Fibonacci(3), we call Fibonacci(2) and Fibonacci(1)
            - For Fibonacci(2), we call Fibonacci(1) and Fibonacci(0). **Base cases!**
            - Fibonacci(1). ***Base case!***
        - For Fibonacci(2), we call Fibonacci(1) and Fibonacci(0). **Base cases!**
    - For **Fibonacci(3)**, we call Fibonacci(2) and Fibonacci(1)
        - For Fibonacci (2), we call Fibonacci(1) and Fibonacci(0) **Base cases!**
        - Fibonacci(1) ***Base case!***

Finding the n-th Fibonacci number

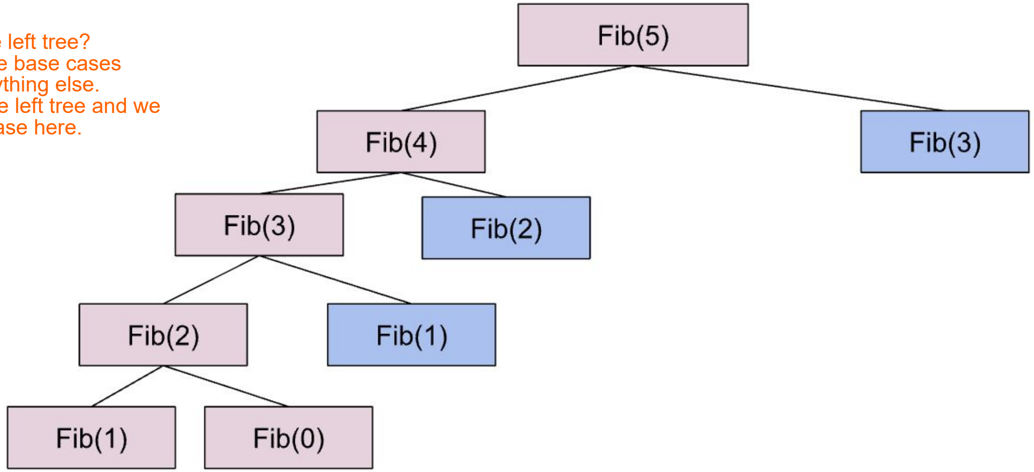
- Below, we can observe the overlapping to compute Fib(n-2) and Fib(n-3),...etc.
- This effort can be saved by computing only once and re-using the results. This technique is called memoization. → notice this is not memoization



Finding the n-th Fibonacci number

- The tree after memoization for `Fibonacci(5)` will look something like this:

Why do we execute the left tree?
we want to calculate the base cases first prior to doing everything else.
Here, the Fib(4) is in the left tree and we want to find the base case here.

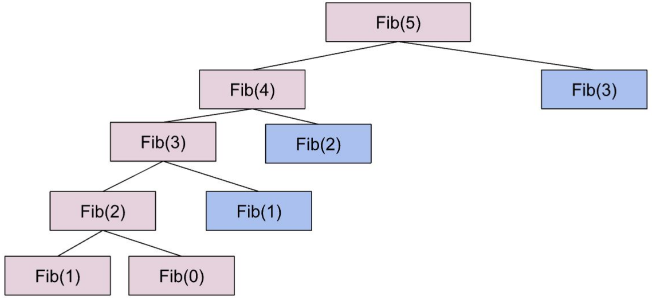


Here **blue** boxes represent results being used from memory and **pink** boxes represent results being computed

Finding the n-th Fibonacci number

```python
def fib(n, memo={}):
    # Define a function named 'fib' that calculates the nth Fibonacci number.
    # The function takes an integer 'n' and an optional dictionary 'memo' to store computed Fibonacci values.
    if n in memo:
        # Check if the Fibonacci number for 'n' is already computed and stored in 'memo'.
        # If it is, return the cached value to avoid redundant calculations.
        return memo[n]
    if n <= 1: # Base case: if 'n' is 0 or 1, return 'n' itself. Fibonacci(0) = 0 and Fibonacci(1) = 1.
        return n
    # Recursive case: calculate Fibonacci(n) as the sum of Fibonacci(n-1) and Fibonacci(n-2).
    # Store the computed result in the 'memo' dictionary to avoid recalculatir
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    # Return the computed Fibonacci number for 'n'.
    return memo[n]
```

memoization
stores vals for reuse



Finding the n-th Fibonacci number

- In the previous code, instead of calling Fib(n−1) and Fib(n−2), we are **first checking if they are already present in memory or not**.
- Each computation stores the result in memory for future references. If a computation has already been done before, the **program fetches the stored results and uses them**
- Such an approach is called **Dynamic Programming**

Dynamic Programming

- Dynamic programming is an algorithm design technique (like divide and conquer) that solves complex problems by
  - dividing them into simpler sub problems, then
  - combining the solutions of sub problems to achieve an overall optimal solution
- Applicable when subproblems are **not** independent main difference between this one and the other one

can we combine dynamic programming & merge sort? Yes, technically, but the overall complexity will be terrible (O(nlogn)). It's better to just divide and conquer

Dynamic Programming

- The properties needed for a dynamic programming solution to be applicable are:
  - Overlapping subproblems: It must be possible to break the original problem down into subproblems, with some overlapping (some subproblems occur more than once)
    - Each of these subproblems will be solved only once, and the results saved and reused if necessary
  - Optimal substructure: It must be possible to calculate the optimal solution to a subproblem
    - Solving the smaller problems leads to the final solution

Dynamic Programming

condition #1

- A problem is said to have overlapping subproblems if
  - The problem can be broken down into subproblems which are reused several times, or ….
  - A recursive algorithm for the problem solves the same subproblem over and over
- This issue of unnecessary repetition is handled well by dynamic programming

Dynamic Programming

Condition #2

- A given problem has an Optimal Substructure property if:
  - An optimal solution of the given problem can be obtained by using optimal solutions of its subproblems

Dynamic Programming

a problem that you want to optimize with a constraint,
normally min or max constraint. Minimization or maximalization problem

- Dynamic Programming is used for **optimization problems**
    - A set of choices must be made to get an optimal solution
    - Find a solution with the optimal value (minimum or maximum)
    - There may be many solutions that lead to an optimal value

Dynamic Programming

- There are two variants of dynamic programming:
  - Tabulation (i.e., Bottom-up dynamic programming) (often referred to as “dynamic programming”)
  - Memoization

Dynamic Programming Improve the efficiency

- Dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations

- By eliminating unnecessary repetitions, Dynamic programming can bring down the order of time complexity to **polynomial** (O(n^c)) instead of being **exponential**

Dynamic Programming Steps

- Characterize the structure of an optimal solution
- Define the optimal solution
- Compute the value of an optimal solution in a bottom-up fashion by solving the subproblems (some overlaping)
  → maximization
  ↳ top-down : minimalization
- Construct an optimal solution from computed information (not always necessary)

Example: Finding the 5th Fibonacci number

- Let us look, once again, at the properties needed for a dynamic programming solution to be applicable are:
- **Overlapping subproblems:** to find Fib(5), we must find Fib(4) and Fib (3), to find Fib(4) we must find Fib(3) and Fib(2) and so on…
- **Optimal substructure:** if our solutions to subproblems (say Fib(3) and Fib(4)) are **optimal**, the **final solution will also be optimal**

Example: Finding the 5th Fibonacci number

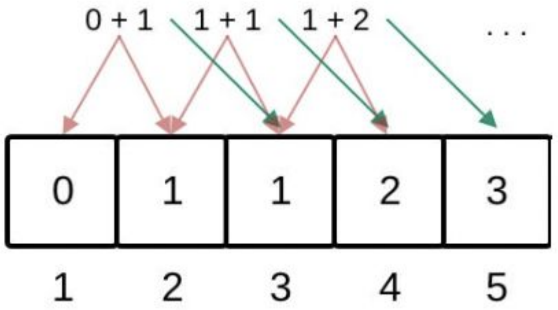
- Dynamic Programming Bottom up Approach
- Build “from the bottom up”
- Running time: O(n) $o(2^n) = O(n)$
- Very fast in practice – just need an array (of linear size) to store the F(i) values

When working with bottom down, 1) you need to find the base case and 2) then the recursive relationship

You could use hash tables to implement this...

Algorithm DYN-FIB(n)

$F[0] = 0$ O(1)
$F[1] = 1$ O(1)
for $i \leftarrow 2$ **to** n **do**   O(n)
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$ O(n)



Comparison of Common Advanced Algorithms

- Divide and Conquer
  - Divide into smaller sub-instances of the same problem, solve these recursively, and then put solutions together
  - Examples: Mergesort, Quicksort, and FFT
- Greedy Algorithms
  - Make a choice that looks optimal at the moment —don't look ahead, never go back
  - Examples: A* algorithm and Prim's algorithm
- Dynamic Programming
  - Bottom up: find optimal solutions to subproblems ("turns recursion upside down")
  - Example: Floyd-Warshall algorithm for the all pairs shortest path problem

Comparison with Divide & Conquer

- Divide and Conquer algorithms partition the problem into **independent** subproblems
- In Dynamic Programming the subproblems are **not independent**
- Dynamic Programming algorithm solves every subproblem just once and then saves its answer in a table
- If a subproblem pops up again, it is NOT processed, instead the saved results are used
  - *Question: Is this the case for Divide and Conquer?*

all subproblems will be executed

Comparison with Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment
- Often, greedy algorithms tend to be easier to code
- Greedy and Dynamic Programming, both are methods for solving optimization problems
- Greedy algorithms are usually **more *efficient*** than Dynamic Programming solutions
- However, often you need to use dynamic programming since the optimal solution **cannot** be guaranteed by a greedy algorithm

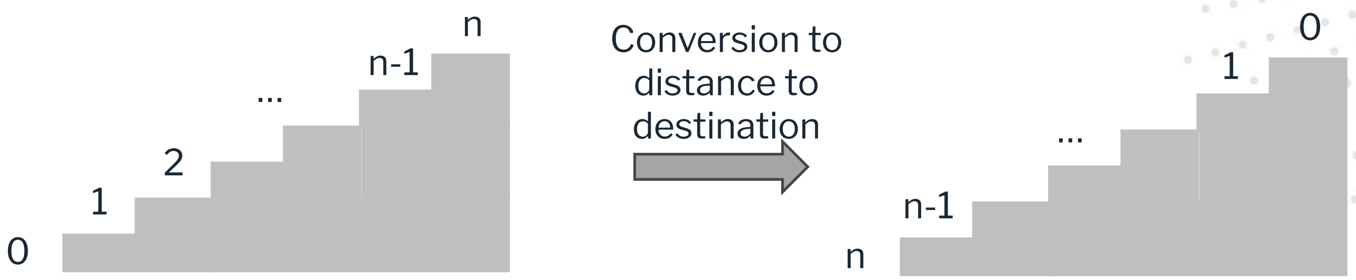
Dynamic Programming Examples

base case
↳ $f(1)=1$
↳ $f(0)=1$

recursive case
↳ $f(n)=f(n-1)+f(n-2)$

- **Climbing stairs problem**
  - Given a staircase with n steps
  - One can take 1 step or 2 steps
  - Goal: Count the total number of unique ways to reach to n

either you take 1 step or 2 steps



Dynamic Programming Examples

- Climbing Stairs Problem Base Case
  - **If there are 0 steps**, there is exactly **1 way** to reach the top (by doing nothing as you reach destination) -> $f(0)=1$
  - **If there is 1 step**, there is exactly **1 way** to reach the top (by taking a single step) -> $f(1)=1$
- Climbing Stairs Problem Recursive Case
  - $f(n)=f(n-1)+f(n-2)$
  - This formula is based on the idea that to reach step n:
    - We could have taken a 1-step from $n-1$, and there are $f(n-1)$ ways to get to $n-1$
    - Or we could have taken a 2-step from $n-2$, and there are $f(n-2)$ ways to get to $n-2$

Dynamic Programming Examples

- **Optimal Substructure**: the number of ways to reach step nn can be expressed in terms of the number of ways to reach the previous two steps:$f(n)=f(n-1)+f(n-2)$
  - where $f(n)$ represents the number of ways to reach step n
  - This relationship allows us to build solutions incrementally and is a hallmark of dynamic programming optimization
- **Overlapping Subproblems**: The climbing stairs problem has overlapping subproblems, meaning that the same subproblems (e.g., calculating the number of ways to reach step $n-1$ and $n-2$) are solved multiple times if approached with a naive recursive method
- Using dynamic programming (either memoization or tabulation) optimally resolves these subproblems by storing previously computed results and reusing them

Dynamic Programming Examples

- Climbing stairs problem
  - Top-down approach memoization

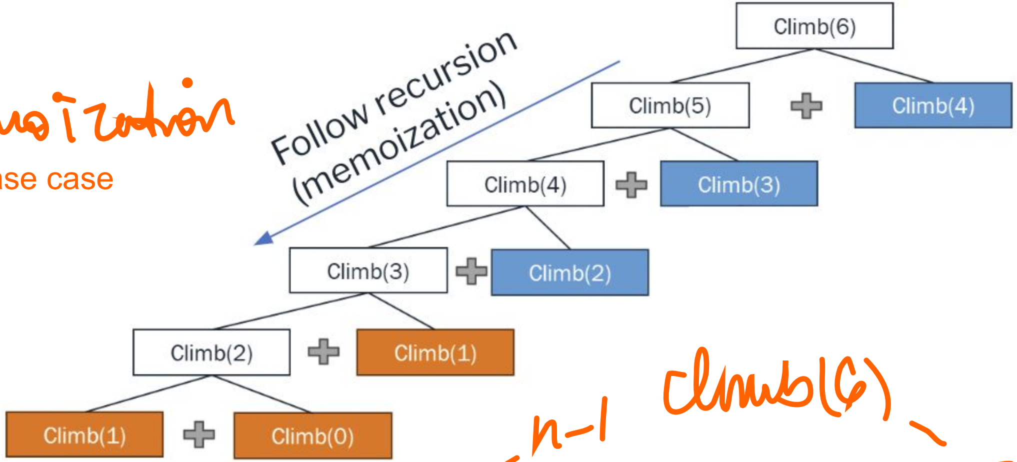
Start from top and move down to base case

```python
def countWays(n, dp):

    if (n <= 1):
        return 1

    if(dp[n] != -1):
        return dp[n]

    dp[n] = countWays(n - 1, dp) + countWays(n - 2, dp)
    return dp[n]
```



climb(6)

n-1 5 n-2

n-1 n-2 n-1 n-2

4 3 3 2

Dynamic Programming Examples

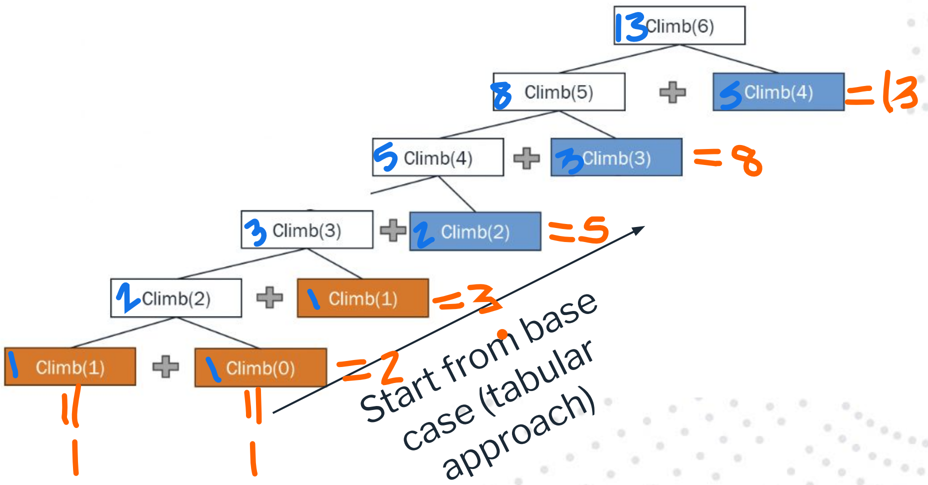
- Climbing stairs problem
  - done w/ tables
  - Bottom-up approach

    start from base case and move upwards until n

```
def countWays(n):
    # Creates list res with all elements 0
    dp = [0 for x in range(n+1)]
    dp[0], dp[1] = 1, 1

    for i in range(2, n+1):
        dp[i] = dp[i-1]+dp[i-2]
    return dp[n]


# Driver Program
n = 4
print "Number of ways =", countWays(n)
```



steps	# ways	cases
0	1	base case
1	1	base case
2	2	1 + 1
⋮		

Dynamic Programming Examples

- Climbing stairs problem
  - Bottom-up approach

Step(i)	Unique ways to reach step(i)	Calculations
0	1	Base case
1	1	Base case
2	2	f(2) = f(1) + f(0) = 1 + 1 = 2
3	3	f(3) = f(2) + f(1) = 2 + 1 = 3
4	5	f(4) = f(3) + f(2) = 3 + 2 = 5
5	8	f(5) = f(4) + f(3) = 5 + 3 = 8
6	13	f(6) = f(5) + f(4) = 5 + 8 = 13

Dynamic Programming Examples

- Coin change
  - Given number N, e.g., N = 10
  - Given a set of denominator c = [1, 2, 4, …]
  - Find the minimum set of denominator to make a sum equal to N

3 coins

4 4 2

base case

n = 0 (no coins)

recursive case

n

min

n-1 n-2 n-4

Dynamic Programming Examples

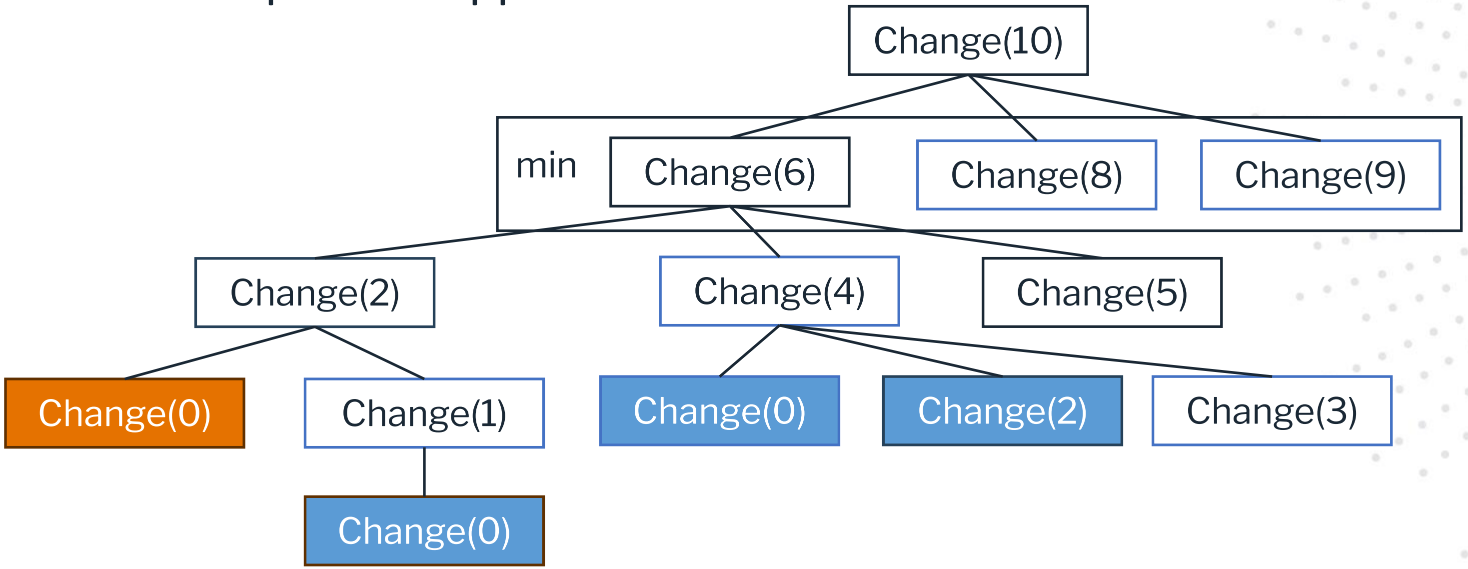
- The **base case** for this problem is when the number is 0.
  - If number = 0, we need 0 coins to make this amount.
- The **recursive case** is used to build up the solution for each amount greater than 0.
  - For each number i:
  - We check each coin denomination
    - If we use a coin of denomination coin, then the remaining number to be solved is i - coin.
    - The relation can be expressed as: min
      - dp[i]=min(dp[i],dp[i−coin]+1) (dp[10], dp[10-4]+1)
    - This means that for each coin, we look up dp[i - coin] (the minimum coins needed for the remaining amount) and add 1 (representing the use of the current coin)

Dynamic Programming Examples

- To determine the minimum number of coins needed to make a specific number N, we can build up this solution by solving smaller subproblems. Thus, the problem has an **optimal substructure** because the optimal solution for any number N can be derived from the optimal solutions of smaller subproblems
- The **overlapping subproblems** property occurs when a recursive solution revisits the same subproblems multiple times. When solving recursively, we may end up calculating the minimum coins required for the same number multiple times

Dynamic Programming Examples

- Coin change problem
    - Top-down approach



Dynamic Programming Examples

- Coin change problem
  - Bottom-up approach

[1, 2, 4]

Amount(i)	dp[i] = min(dp[i], dp[i - coin] + 1)	DP Array
0	Base case, dp[0] = 0 zero coins	[0, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞]
1	dp[1] = min(∞, dp[1 - 1] + 1) = 1 (size of array; the minimum coin you take out)	[0, 1, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞]
2	dp[2] = min(∞, dp[2 - 1] + 1, dp[2 - 2] + 1) = 1 (size of array; min coin; size of array; next minimum coin to take out)	[0, 1, 1, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞]
3	dp[3] = min(∞, dp[3 - 1] + 1, dp[3 - 2] + 1) = 2	[0, 1, 1, 2, ∞, ∞, ∞, ∞, ∞, ∞, ∞]
4	dp[4] = min(∞, dp[4 - 1] + 1, dp[4 - 2] + 1, dp[4 - 4] + 1) = 1	[0, 1, 1, 2, 1, ∞, ∞, ∞, ∞, ∞, ∞]

Dynamic Programming Examples

Amount(i)	dp[i] = min(dp[i], dp[i - coin] + 1)	DP Array
5	dp[5] = min(∞, dp[5 - 1] + 1, dp[5 - 2] + 1, dp[5 - 4] + 1) = 2	[0, 1, 1, 2, 1, 2, ∞, ∞, ∞, ∞, ∞]
6	dp[6] = min(∞, dp[6 - 1] + 1, dp[6 - 2] + 1, dp[6 - 4] + 1) = 2	[0, 1, 1, 2, 1, 2, 2, ∞, ∞, ∞, ∞]
7	dp[7] = min(∞, dp[7 - 1] + 1, dp[7 - 2] + 1, dp[7 - 4] + 1) = 3	[0, 1, 1, 2, 1, 2, 2, 3, ∞, ∞, ∞]
8	dp[8] = min(∞, dp[8 - 1] + 1, dp[8 - 2] + 1, dp[8 - 4] + 1) = 2	[0, 1, 1, 2, 1, 2, 2, 3, 2, ∞, ∞]
9	dp[9] = min(∞, dp[9 - 1] + 1, dp[9 - 2] + 1, dp[9 - 4] + 1) = 3	[0, 1, 1, 2, 1, 2, 2, 3, 2, 3, ∞]
10	dp[10] = min(∞, dp[10 - 1] + 1, dp[10 - 2] + 1, dp[10 - 4] + 1) = 3	[0, 1, 1, 2, 1, 2, 2, 3, 2, 3, 3]

Using backtracking, to get the number 10, we used [4, 4, 2], which gives a total of 3 coins

Dynamic Programming Activity

- You are given a target amount N = 7 and coin denominations [1, 2, 5]. Your task is to use dynamic programming bottom up and top down approaches to determine the minimum number of coins required to make the amount N using these denominations