Table of Contents

1. Introduction

Dynamic programming is a problem-solving technique used to solve complex problems by breaking them down into smaller overlapping subproblems and solving each subproblem only once. It is particularly useful when the problem exhibits optimal substructure, meaning that the optimal solution of the problem can be constructed from the optimal solutions of its subproblems.

Dynamic programming can be classified into two main categories: 1D dynamic programming and 2D dynamic programming.

1.1 Dynamic Programming - 1D:

1D dynamic programming is used when the problem can be solved by iterating over a 1-dimensional array or sequence. It involves defining a 1D array to store the solutions to subproblems and iteratively filling it up to compute the solution to the overall problem.

Variations of 1D Dynamic Programming:

- Bottom-up: The bottom-up approach starts solving the problem from the smallest subproblem and builds up to the overall problem. It typically involves using an iterative loop to fill up the memoization array.
- Top-down: The top-down approach, also known as memoization, starts solving the problem from the overall problem and recursively solves smaller subproblems. It uses memoization techniques to store and retrieve the solutions to subproblems, avoiding redundant calculations.
- Space optimization: In some cases, it is possible to optimize the space usage of the memoization array by only storing the necessary values instead of storing the entire array. This can be achieved by keeping track of only the required previous states or using rolling arrays.

1D dynamic programming is a powerful technique for solving problems with optimal substructure that can be represented through a 1-dimensional array. It provides efficient solutions to complex problems by breaking them down into smaller subproblems and memoizing their solutions.

1.2 Dynamic Programming - 2D:

2D dynamic programming is an extension of the dynamic programming technique used when the problem can be solved by iterating over a 2-dimensional grid or matrix. It involves defining a 2D array to store the solutions to subproblems and iteratively filling it up to compute the solution to the overall problem.

Variations of 2D Dynamic Programming:

- Space optimization: Similar to 1D dynamic programming, it is often possible to optimize the space usage of the memoization grid by only storing the necessary values instead of the entire grid. This can be achieved by keeping track of only the required previous states or using rolling grids.
- Path reconstruction: In some problems, it is not only necessary to find the optimal solution but also determine the path or sequence of choices that leads to that solution. Path reconstruction techniques can be applied alongside 2D dynamic programming to track the decisions made at each step.

Overall, 2D dynamic programming is a powerful technique for solving problems with two-dimensional structures. It allows for efficient computation of optimal solutions by breaking down the problem into smaller subproblems and memoizing their solutions in a 2D grid.

2. Algorithm

2.1 DP - 1D

The general steps involved in solving a problem using 1D dynamic programming are as follows:

- Define the problem: Clearly define the problem and identify the objective and constraints.
- Identify the recurrence relation: Determine the relationship between the optimal solution of the overall problem and the optimal solutions of its subproblems. This recurrence relation is often expressed as a recursive formula.
- Create a memoization array: Initialize a 1D array to store the solutions to subproblems. This array, often called a memoization or DP array, will be used to store and retrieve the computed solutions to avoid redundant calculations.
- Build up the DP array: Iterate over the elements of the memoization array, filling it up based on the recurrence relation. The order of iteration depends on the nature of the problem and the dependencies between subproblems.

- Return the final solution: Once the memoization array is filled up, the solution to the overall problem can be obtained from a specific position in the array.

2.2 DP - 2D

The general steps involved in solving a problem using 2D dynamic programming are similar to 1D dynamic programming:

- Define the problem: Clearly define the problem and identify the objective and constraints.
- Identify the recurrence relation: Determine the relationship between the optimal solution of the overall problem and the optimal solutions of its subproblems. This recurrence relation is often expressed as a recursive formula involving the elements of the 2D grid or matrix.
- Create a memoization grid: Initialize a 2D array to store the solutions to subproblems. This grid, often called a memoization or DP grid, will be used to store and retrieve the computed solutions to avoid redundant calculations.
- Build up the DP grid: Iterate over the elements of the memoization grid, filling it up based on the recurrence relation. The order of iteration depends on the nature of the problem and the dependencies between subproblems.
- Return the final solution: Once the memoization grid is filled up, the solution to the overall problem can be obtained from a specific position in the grid.

3. Example Usage

3.1 DP - 1D

Let's see an example to illustrate 1D dynamic programming using the Fibonacci sequence:

```
def fibonacci(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[0] = 0
    dp[1] = 1

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

In this example, the Fibonacci sequence is computed using 1D dynamic programming. The dp array is used to store the solutions to subproblems, and the loop iterates over the array to fill it up based on the recurrence relation dp[i] = dp[i - 1] + dp[i - 2]. Finally, the solution to the Fibonacci number at position n is returned.

3.2 DP - 2D

Let's see an example to illustrate 2D dynamic programming using the Longest Common Subsequence (LCS) problem:

```
def longestCommonSubsequence(text1, text2):
    m = len(text1)
    n = len(text2)

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

In this example, the LCS problem is solved using 2D dynamic programming. The dp grid is used to store the solutions to subproblems, and the nested loops iterate over the grid to fill it up based on the recurrence relation. The LCS length is obtained from the position dp[m][n], where m and n are the lengths of the input texts text1 and text2, respectively.

4. Advantages

4.1 DP - 1D

Advantages of 1D Dynamic Programming:

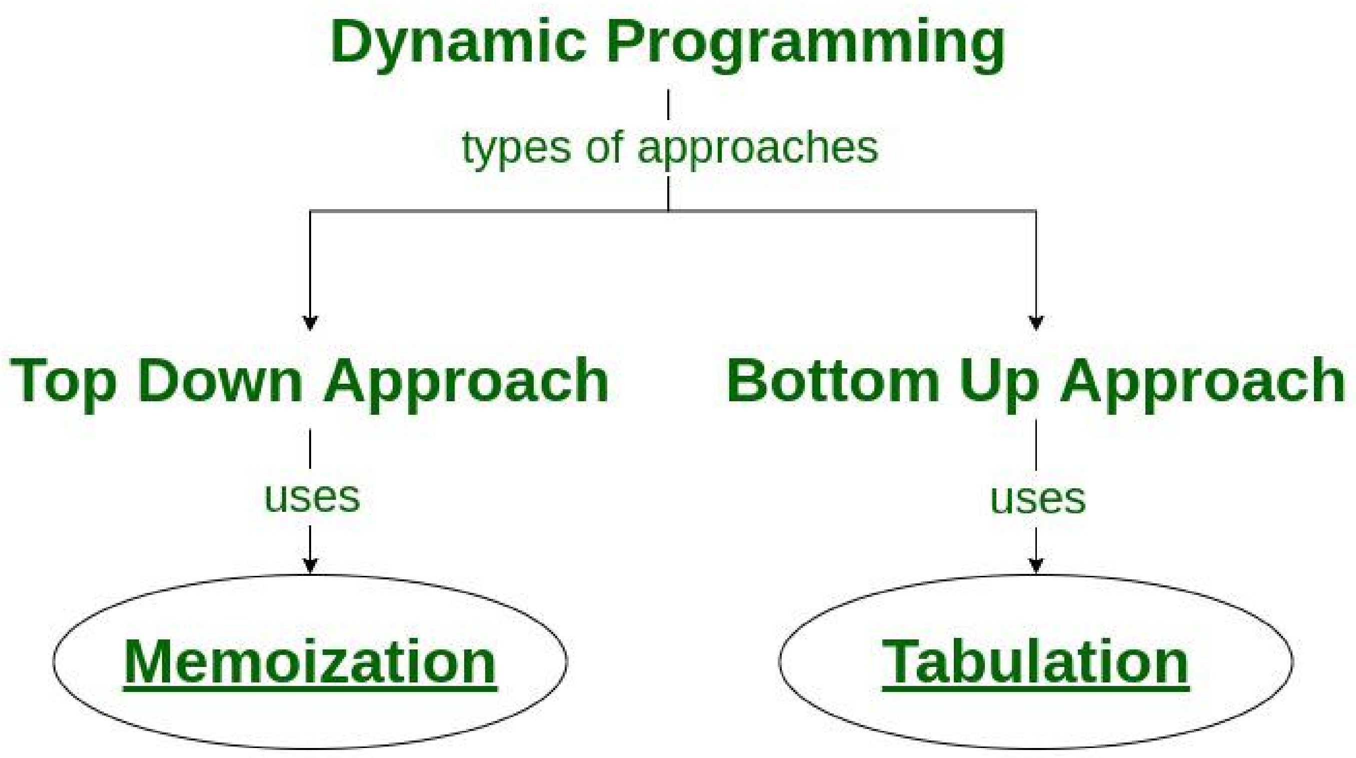
- **Reduces redundant calculations**: By storing the solutions to subproblems in the memoization array, 1D dynamic programming avoids redundant calculations and improves efficiency.

- **Time complexity optimization**: 1D dynamic programming allows for efficient computation of solutions that would otherwise require exponential time complexity.
- **Simplifies problem-solving**: Dynamic programming breaks down complex problems into smaller subproblems, making them easier to understand and solve.

4.2 DP - 2D

Advantages of 2D Dynamic Programming:

- **Handles problems with two-dimensional structures**: 2D dynamic programming is well-suited for problems that involve grids, matrices, or two-dimensional sequences.
- **Efficiently solves complex problems**: By breaking down the problem into smaller overlapping subproblems and storing their solutions, 2D dynamic programming allows for efficient computation of the optimal solution to the overall problem.
- **Provides flexibility in problem representation**: 2D dynamic programming can handle problems where the optimal solution depends on both the current position and the previously visited positions in a 2D grid.


Dynamic Programming
types of approaches
Top Down Approach
Bottom Up Approach
uses
uses
Memoization
Tabulation