Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique that solves complex problems by breaking them down into overlapping subproblems and efficiently solving each subproblem only once. It stores the solutions to the subproblems in a table or array, allowing for efficient retrieval and re-use of these solutions.

Two Properties to Look For in Dynamic Programming

Optimal Substructure

One key property of problems suitable for dynamic programming is optimal substructure. This means that the optimal solution to the problem can be constructed from the optimal solutions of its subproblems. In other words, the problem exhibits a recursive structure, and the optimal solution can be expressed as a combination of optimal solutions to smaller subproblems.

Overlapping Subproblems

Another property required for dynamic programming is overlapping subproblems. This means that the problem can be divided into subproblems, and the same subproblems are encountered multiple times during the computation. Therefore, the solutions to these subproblems can be cached and reused rather than recomputed each time, leading to a significant improvement in efficiency.

If a problem exhibits both optimal substructure and overlapping subproblems, it indicates that dynamic programming can be applied to solve the problem efficiently.

Example: Fibonacci Sequence

Let’s consider the Fibonacci sequence as an example to illustrate the concepts of optimal substructure and overlapping subproblems.

The Fibonacci sequence is defined as follows:

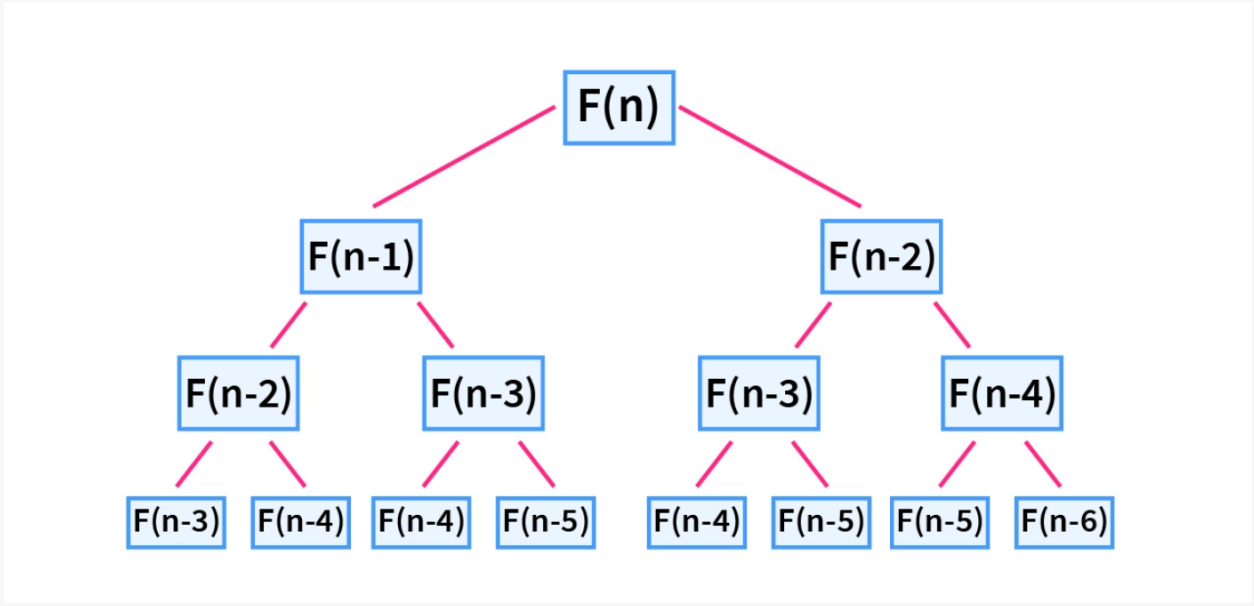
- F(0) = 0
- F(1) = 1

- F(n) = F(n-1) + F(n-2) for n > 1
- 0 1 1 2 3 5 8 ...

Recursive Approach:

A naive recursive approach to compute the nth Fibonacci number would be as follows:

```
function fibonacci(int n) {
    if (n less than 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```



The recursion tree looks something like this.

Optimal Substructure:

The optimal substructure property of the Fibonacci sequence can be observed from the recursive formula. To compute `F(n)`, we need to compute `F(n-1)` and `F(n-2)`, which are smaller subproblems. The optimal solution to the Fibonacci sequence relies on the optimal solutions to these smaller subproblems.

Overlapping Subproblems:
In the recursive approach, the computation of Fibonacci numbers involves redundant calculations. For example, when computing `F(n)`, both `F(n-1)` and `F(n-2)` are recomputed separately, even though they have already been computed during the computation of `f(n-1)` and `f(n-2)` respectively. This results in an exponential increase in the number of function calls and inefficient computation.

Dynamic Programming Approach

In dynamic programming, there are two common approaches to solving problems: the top-down approach (memoization) and the bottom-up approach. Let's explore both of these approaches:

Top-down Approach (Memoization)

In the top-down approach, also known as memoization, we break down the problem into smaller subproblems and solve them recursively. We store the results of these subproblems in a memoization table (usually an array or a hashmap) so that we can avoid redundant calculations. The memoization table serves as a cache, allowing us to retrieve precomputed results when needed.

Here's how the top-down approach works:

- Before computing the solution to a subproblem, we check if its result is already available in the memoization table.
- If the result is present, we return it.
- If the result is not available, we compute it recursively by solving the subproblems and store the result in the memoization table for future reference.
- We continue this process until we solve the original problem.

Pseudo Code for implementing Fibonacci Series using Top-Down Approach

```
memo[] is the memoization or DP array
function fibonacci(n):
    if memo[n] is defined:
        return memo[n]  // Return the precomputed value if available
    if n <= 1:
        memo[n] = n  // Store the computed value in the memoization table
    else:
        memo[n] = fibonacci(n - 1) + fibonacci(n - 2)  // Compute and store the
value
    return memo[n]  // Return the computed value
```

The top-down approach is intuitive and easier to implement when you have a clear recursive structure. It helps avoid redundant computations and can greatly improve the efficiency of the algorithm.

Bottom-Up Approach:

The bottom-up approach, also known as tabulation, involves solving the problem iteratively by starting from the smallest subproblems (Base Case) and gradually building up to the larger problem. Instead of relying on recursive calls, we use a loop to solve subproblems in a systematic manner.

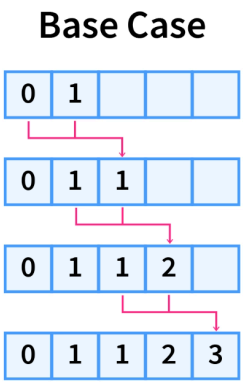
Here's how the bottom-up approach works:

- We start by solving the smallest subproblems and store their results in a table (usually an array or a matrix).
- We then use the results of the solved subproblems to compute the solutions to larger subproblems.
- We continue this process, solving subproblems in a bottom-up fashion, until we reach the original problem's solution.
- The bottom-up approach eliminates the overhead of function calls and recursion, making it more efficient in some cases. It ensures that every subproblem is solved only once, and we can retrieve the final solution from the table.

Pseudo Code for implementing Fibonacci Series using Bottom-up Approach

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int dp[n+1];  // Create a table to store the computed values
    // Initialize the base cases
    dp[0] = 0;
    dp[1] = 1;
    // Compute the Fibonacci numbers bottom-up
    for (i = 2 to n) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];  // Return the computed Fibonacci number
}
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(n) because of the array, no stack space is being used.



Both approaches, top-down (memoization) and bottom-up (tabulation), can be used in dynamic programming depending on the problem and the available resources. The choice between the two approaches often depends on the problem structure, complexity, and specific requirements.

What are States in DP?

In dynamic programming, states refer to what we are storing in the dp array or matrix.

`dp[i]` **=> What are we storing.**

Note: In Bottom up DP, we need to be aware of the order in which the states need to be computed.

Difference between Top-down and Bottom-up Approaches

Top down Approach	**Bottom Up Approach**
Easier to think	Little complicated to think at first
Extra Stack Space needed	No Extra Stack Space needed

No control over the order in which States are computed	More Control over the order in which states are computed, so more Optimization can be done!

Steps to solve a DP Problem

Problem

Compute the number of ways in which you can reach the N-th stair, provided you can make only two types of jumps.

- a jump of one step, i.e. from stair `i` to stair `i+1`.
- a jump of two steps, i.e. from stair `i` to stair `i+2`.

Element of choice

If there is an element of choice, we can be sure that there's a recursive solution. Here we have a choice, so this has a recursive solution.

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

As you might identify from the recurrence relation, this is the same as the fibonacci series.

How to represent a state (What do my states represent?)

Here, my states should represent the number of steps to reach the i-th stair = Steps(i).

Use the Step 1 to write the recurrence relation

All our choices should be mutually exclusive and mutually exhaustive, i.e. the choices should not overlap and they should cover all the cases.

Which State is the final answer for the solution.

We can very simply say, for the given problem it will be `steps(n)`. But it's not necessary that the last step will always be the solution. Depends on the problem.

Example - LIS (Longest Increasing Subsequence)

Problem Statement

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Example

Input: {10, 22, 9, 33, 21, 50, 41, 60, 80}
Output: 6

Explanation:

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

InterviewBit

Solution

Brute Force Approach:

- The brute force approach involves generating all possible subsequences of the given array.
- For each subsequence, check if it is strictly increasing.
- Keep track of the length of the longest increasing subsequence found so far.
- Return the length of the longest increasing subsequence.

While the brute force approach is straightforward, it has an exponential time complexity since it generates all subsequences. Hence, it is not an efficient solution.

Time Complexity: $O(n*2^n)$

Dynamic Programming Solution

- **Element of choice:** At each index `i` in the array, we have a choice to either include or exclude the current element `nums[i]` in the increasing subsequence.
- **State representation:** We can represent the state as `dp[i]`, which stores the length of the longest increasing subsequence ending at index i.
- **Recurrence relation:** To calculate `dp[i]`, we iterate over all previous elements `j` (from 0 to i-1), and if `nums[i] > nums[j]`, we update `dp[i]` as `max(dp[i], dp[j] + 1)`. This means the length of the LIS ending at index i is either the length already stored in `dp[i]` or the length of the LIS ending at index `j` plus one.
- **Final answer:** The final answer for the solution will be the maximum value in the dp array.

```
function lengthOfLIS(nums) {
    n = length(nums)
    if n == 0
        return 0
    dp = array of size n, initialized with all 1s
    for i = 1 to n-1 {
        for j = 0 to i-1 {
            if nums[i] > nums[j]
                dp[i] = max(dp[i], dp[j] + 1)
        }
    }
    max_length = maximum value in dp array

    return max_length
}
```

The above pseudo code outlines the steps for the DP solution to find the length of the longest increasing subsequence in an array nums. It initializes a DP array with 1s, and then iterates over the elements of nums, comparing each element with previous elements to determine the length of the LIS ending at that index. Finally, it returns the maximum value in the DP array, which represents the length of the longest increasing subsequence.

Dry Run

Input:

arr[] = **{3, 10, 2, 11}**
LIS[] = **{1, 1, 1, 1}** (initially)

Iteration-wise simulation:

- **Iteration 1:**
  - arr[2] = 2, arr[1] = 10: Since arr[2] < arr[1], there is no change in LIS[].
  - LIS[] = **{1, 1, 1, 1}**
- **Iteration 2:**
  - arr[3] = 11, arr[1] = 10: Since arr[3] > arr[1], update LIS[3] to max(LIS[3], LIS[1] + 1) = 2.
  - LIS[] = **{1, 1, 2, 1}**
- **Iteration 3:**
  - arr[3] = 11, arr[2] = 2: Since arr[3] > arr[2], update LIS[3] to max(LIS[3], LIS[2] + 1) = 2.
    = LIS[] = **{1, 1, 2, 1}**
- **Iteration 4:**
  - arr[4] = 11, arr[1] = 10: Since arr[4] > arr[1], update LIS[4] to max(LIS[4], LIS[1] + 1) = 2.
  - arr[4] = 11, arr[2] = 2: Since arr[4] > arr[2], update LIS[4] to max(LIS[4], LIS[2] + 1) = 3.
  - arr[4] = 11, arr[3] = 2: Since arr[4] > arr[3], update LIS[4] to max(LIS[4], LIS[3] + 1) = 3.
  - LIS[] = **{1, 1, 2, 3}**

The final LIS[] array represents the length of the longest increasing subsequence ending at each index. The maximum value in the LIS[] array is the length of the longest increasing subsequence in the original array.

In this example, the length of the longest increasing subsequence is **(3, 10, 11)**.

2D DP

2D Dynamic Programming is a technique used to solve problems that require optimizing over two dimensions or involve two nested loops. It is an extension of the traditional Dynamic Programming approach, which is focused on solving problems with a single dimension.

There's a list of common types of problems that we will see next.

- Rat in a Maze
- Dungeon Princess
- Knapsack

Example - Dungeon Princess

Problem Statement

The demons had captured the princess § and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Example

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Solution

Recursive Approach / Brute Force Approach

The recursive approach explores all possible paths from the top-left cell to the bottom-right cell. It calculates the minimum health required at each cell by considering the health required to reach the next cell and subtracting the dungeon value of the current cell.

Pseudo Code:

```
function calculateMinimumHP(dungeon, row, col):
    if row == length(dungeon) - 1 and col == length(dungeon[0]) - 1:
        // Base case: Reached the princess cell
        // Return the minimum health required to reach this cell
        return max(1, 1 - dungeon[row][col])

    if row == length(dungeon) or col == length(dungeon[0]):
        // Base case: Out of bounds
        // Return a large value to indicate invalid path
        return infinity

    // Calculate the minimum health required to reach the next cell by moving
right or down
    next_health = min(
        calculateMinimumHP(dungeon, row, col + 1),
        calculateMinimumHP(dungeon, row + 1, col)
    )

    // Calculate the minimum health required to reach the current cell
    current_health = max(1, next_health - dungeon[row][col])

    return current_health
```

Dynamic Programming

To optimize the recursive solution, we can use either top-down (memoization) or bottom-up (tabulation) dynamic programming. Let's see both approaches.

Top-Down Approach (Memoization):

```
memo [][]   // Declare the memoization array globally

function calculateMinimumHP(dungeon):
    m, n = length(dungeon), length(dungeon[0])
    memo = create a 2D memoization table of size (m x n) initialized with -1

    return dp(dungeon, 0, 0)
```

```
function dp(dungeon, row, col):
    if row == m - 1 and col == n - 1:
        return max(1, 1 - dungeon[row][col])

    if row == m or col == n:
        return infinity

    if memo[row][col] != -1:
        return memo[row][col]

    next_health = min(
        dp(dungeon, row, col + 1),
        dp(dungeon, row + 1, col)
    )
    current_health = max(1, next_health - dungeon[row][col])

    memo[row][col] = current_health

    return current_health
```

Bottom-Up Approach:

```
function calculateMinimumHP(dungeon):
    m, n = length(dungeon), length(dungeon[0])

    // Create a 2D DP table initialized with infinity
    dp = create a 2D table of size (m x n) initialized with infinity

    // Set the bottom-right cell value
    dp[m-1][n-1] = max(1, 1 - dungeon[m-1][n-1])

    // Fill the DP table from bottom-right to top-left
    for i from m-1 down to 0:
        for j from n-1 down to 0:
            if i == m-1 and j == n-1:
                continue   // Skip the bottom-right cell
            // Calculate the minimum health required to reach the current cell
            right_health = i == m-1 ? infinity : dp[i+1][j]// Out of bounds
check
            down_health = j == n-1 ? infinity : dp[i][j+1]   // Out of bounds
check
            next_health = min(right_health, down_health)
            current_health = max(1, next_health - dungeon[i][j])
            dp[i][j] = current_health
    // Return the minimum health required to reach the top-left cell
    return dp[0][0]
```

This pseudo code represents the bottom-up dynamic programming approach of the Dungeon Princess problem. It initializes a 2D DP table and fills it from bottom-right to top-left, calculating the minimum health required to reach each cell. The final answer is stored in dp[0][0], which represents the top-left cell.

Example - 0/1 Knapsack Problem

Problem Statement

Given N items each with a weight and a value. Find the maximum value that can be obtained by picking up items such that the total weight of all items picked is less than K (The weight capacity of knapsack).

Note:

- The “0/1” aspect of the problem implies that we can either include an item or exclude it, but we cannot include a fraction of an item.
- We can pick each item only once.

Example

Input
W[] = **{20, 10, 30, 40}**
V[] = **{100, 60, 120, 150}**
K = **50**

Output
220

Explanation:

We take Item 1 and Item 3 so the total weight is 20+30=50, and the value is **100+120=220**.

Solution

Brute Force Solution (Recursive)

One approach to solving the 0/1 Knapsack problem is by using a brute force recursive solution. Here's the recursive code:

```
function knapsackRecursive(items, capacity, currentIndex):

    // Base case: If either we have no items left or the knapsack is full
    if currentIndex < 0 or capacity <= 0:
        return 0
    // If the weight of the current item exceeds the remaining capacity, exclude it
    if items[currentIndex].weight > capacity:
        return knapsackRecursive(items, capacity, currentIndex - 1)

    // Find the maximum value by either including or excluding the current item
    includeCurrent = items[currentIndex].value + knapsackRecursive(items, capacity - items[currentIndex].weight, currentIndex - 1)
    excludeCurrent = knapsackRecursive(items, capacity, currentIndex - 1)

    return max(includeCurrent, excludeCurrent)  // Return the maximum value
```

This recursive solution explores all possible combinations of items by either including or excluding each item and returns the maximum value.

Dynamic Programming Approach

Top-Down (Memoization) Approach

In the top-down approach, we'll use memoization to store the computed values of subproblems and avoid redundant computations. Here's the pseudo code for the top-down approach:

```
memo = 2D array used for memoizing.

function knapsackTopDown(items, capacity, currentIndex, memo):

    // Base case: If either we have no items left or the knapsack is full
    if currentIndex < 0 or capacity <= 0:
        return 0

    // If the result is already memoized, return it
    if memo[currentIndex][capacity] is not null:
        return memo[currentIndex][capacity]

    // If the weight of the current item exceeds the remaining capacity, exclude it
    if items[currentIndex].weight > capacity:
```

```
        memo[currentIndex][capacity] = knapsackTopDown(items, capacity,
currentIndex - 1, memo)
        return memo[currentIndex][capacity]

    // Find the maximum value by either including or excluding the current item
    includeCurrent = items[currentIndex].value + knapsackTopDown(items,
capacity - items[currentIndex].weight, currentIndex - 1, memo)
    excludeCurrent = knapsackTopDown(items, capacity, currentIndex - 1, memo)

    // Store the result in the memoization table
    memo[currentIndex][capacity] = max(includeCurrent, excludeCurrent)

    // Return the maximum value
    return memo[currentIndex][capacity]
```

Bottom-Up (Tabulation) Approach

In the bottom-up approach, we'll use a table to store the computed values of subproblems iteratively, starting from smaller subproblems and building up to the larger ones.

- **Elements of Choice:** We have a set of items, each with its own weight and value. We can either include an item in the knapsack or exclude it.
- **States:** `dp[i][j]` represents the maximum value that can be obtained by considering the first `i` items and having a knapsack capacity of `j`.
- **Recurrence Relation:** The recurrence relation can be written as follows: `dp[i][j] = max(dp[i-1][j], dp[i-1][j - weight[i]] + value[i])`
- **Final Answer:** The final answer is stored in the state `dp[n][W]`, where `n` is the total number of items and `W` is the total weight capacity of the knapsack. This represents the maximum value that can be achieved by considering all the items and having the full weight capacity of the knapsack.

Pseudo Code:

```
function knapsackBottomUp(items, capacity):
    n = length(items)
    dp = create a 2D table of size (n+1) x (capacity+1) initialized with 0

    for i from 1 to n:
        for j from 1 to capacity:
            // If the weight of the current item exceeds the remaining
capacity, exclude it
            if items[i-1].weight > j:
                dp[i][j] = dp[i-1][j]
            else:
```

```
                // Find the maximum value by either including or excluding the
current item
                includeCurrent = items[i-1].value + dp[i-1][j -
items[i-1].weight]
                excludeCurrent = dp[i-1][j]
                dp[i][j] = max(includeCurrent, excludeCurrent)

    // Return the maximum value stored in the bottom-right cell of the table
    return dp[n][capacity]
```

Time Complexity: `O(n * W)`;
Space Complexity:`O(n * W)`;

Revision Video - https://drive.google.com/drive/folders/1Ad9FiDTi8XV8j5r8XFG0SgnlV3YfQZwg?usp=share_link