Google Developer Student Clubs

Dynamic Programming

Memoization, Tabulation & Space Optimization

Hardik Pawar
5th Sem CSE, RVCE

Table of Contents

Introduction to DP

When can DP be applied?

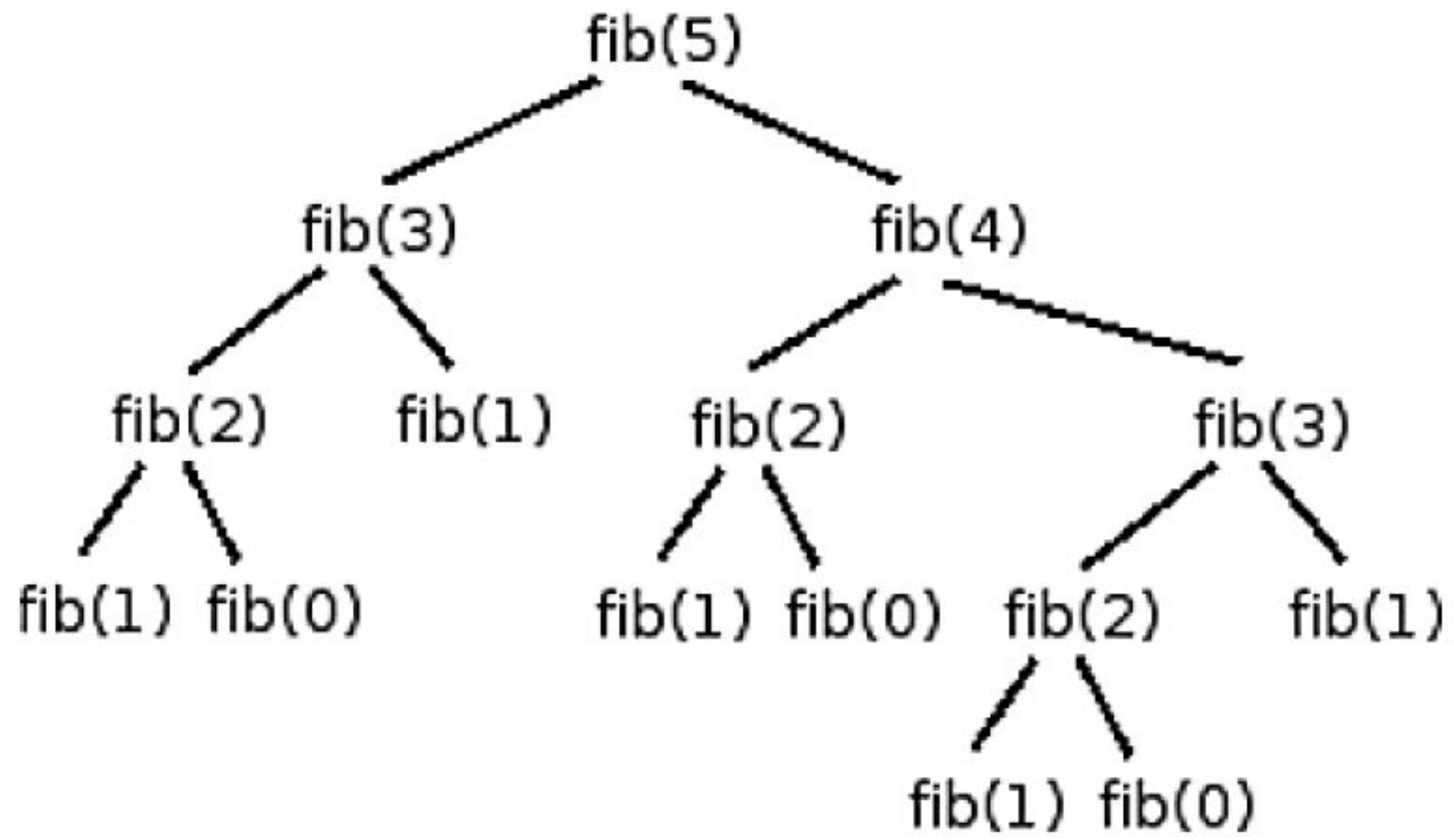
- **Optimal Substructure:** The solution to the main problem can be constructed from the optimal solutions of its subproblems
- **Overlapping Subproblems:** If there is redundancy in solving the same subproblems, DP can be applied to store & retrieve the intermediate results from an in-memory cache

Fibonacci Sequence

Using Divide & Conquer

We know that fib(n) = fib(n-1) + fib(n-2); fib(0) = 0, fib(1) = 1

Now, compute fib(5):



15 calls (nodes)
7 additions (parents)

Fibonacci Sequence

Using DP Bottom-Up Approach (Tabulation)

Compute fib(5):

$$
\begin{aligned}
fib(0) &= 0\\
fib(1) &= 1\\
fib(2) &= fib(1) + fib(0)\\
&= 0 + 1\\
&= 1\\
fib(3) &= fib(2) + fib(1)\\
&= 1 + 1\\
&= 2\\
fib(4) &= fib(3) + fib(2)\\
&= 2 + 1\\
&= 3\\
fib(5) &= fib(4) + fib(3)\\
&= 3 + 2\\
&= 5
\end{aligned}
$$

4 calls
4 additions

Observation

There is significant reduction in **CPU time** and **function call overhead** along with the decrease in the **no. of basic operations** performed.

This not only enhances **computational efficiency** but also contributes to a more **streamlined** and **optimized execution** of the algorithm.

DP Techniques

- **Memoization:** Top-Down, Recursive formulation, typically uses a helper function
- **Tabulation:** Bottom-Up, Iterative approach, reducing the usage of stack space
- **Space-Optimized Tabulation:** Similar to Tabulation, with more efficient space usage

Fibonacci Sequence

Memoization

Direction: main problem to subproblem

Time: O(n)

Space: O(n) (dp array) + O(n) (recursion stack)

```cpp
int fun(int n, vector<int> &dp)
{
    // Base case
    if (n <= 1)
        return n;

    // Lookup
    if (dp[n] != -1)
        return dp[n];

    // Compute, store and return
    return dp[n] = fun(n - 1, dp) + fun(n - 2, dp);
}

int fib(int n)
{
    vector<int> dp(n + 1, -1); // dp[i] = fib(i)
    return fun(n, dp);
}
```

Fibonacci Sequence

Tabulation

Direction: subproblem to main problem

Time: O(n)

Space: O(n) (for dp array)

```
int fib2(int n)
{
    vector<int> dp(n + 1, -1);

    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i - 1] + dp[i - 2];

    return dp[n];
}
```

Fibonacci Sequence

Space-Optimized Tabulation

Direction: subproblem to main problem

Time: O(n)

Space: O(1) (no dp array)

```
int fib3(int n)
{
    // Base case
    if (n <= 1)
        return n;

    int first = 0;
    int second = 1;
    int third;
    for (int i = 2; i <= n; i++)
    {
        third = first + second;
        first = second;
        second = third;
    }

    return second; // OR return third;
}
```

Applications of DP

- **Reduce TLE** (Time Limit Exceeded) in Competitive Programming
- Path Finding & Shortest Paths questions. Eg: Shortest Sum in grid
- Majority of the **Optimization problems**. Eg: 0/1 Knapsack

Exercise 1: Frog Jump

1-dimensional DP Question

Given the no. of stairs and a frog.
The frog wants to climb from the **0^{th} stair** to the **$(N-1)^{th}$ stair**.
At a time the frog can climb either **one or two steps**.
A **height[N] array** is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is **abs(height[i] - height[j])**, where abs() means the absolute difference.

Return the **min energy** that can be used by the frog to jump from stair 0 to stair N-1.

Exercise 1: Frog Jump

Intuition

- **Optimization Problem**: Find the min energy to go from 0th to $(N-1)^{th}$ stair
- **Optimal Substructure & Overlapping Subproblems:** The i^{th} stair, can be reached from the $(i-1)^{th}$ or $(i-2)^{th}$ stair.

Exercise 1: Frog Jump

Memoization

Main function

Initializing the DP array

```cpp
int minEnergy(int n, vector<int> &height)
{
    vector<int> dp(n, -1); // 0 to n-1 stair
    return fun(n - 1, height, dp);
}
```

Exercise 1: Frog Jump

Memoization

Helper function

Recursive formulation of logic

```
int fun(int ind, vector<int> &height, vector<int> &dp)
{
    // Base Case
    if (ind == 0) // 0th stair
        return 0;

    // Lookup
    if (dp[ind] != -1)
        return dp[ind];

    // Recursive Case

    // Energy to reach current stair from 1 stair before
    int jump1 = abs(height[ind] - height[ind - 1]) + fun(ind - 1, height, dp);

    // Energy to reach current stair from 2 stairs before
    int jump2 = INT_MAX;
    if (ind >= 2) // valid index
        jump2 = abs(height[ind] - height[ind - 2]) + fun(ind - 2, height, dp);

    return dp[ind] = min(jump1, jump2);
}
```

Exercise 1: Frog Jump

Tabulation

Iteratively filling the dp array in bottom-up fashion

```
int minEnergy2(int n, vector<int> &height)
{
    vector<int> dp(n, -1); // 0 to n-1 stair

    dp[0] = 0;                     // 0th stair
    for (int i = 1; i < n; i++) // 1 to n-1
    {
        int jump1 = abs(height[i] - height[i - 1]) + dp[i - 1];
        int jump2 = INT_MAX;
        if (i >= 2)
            jump2 = abs(height[i] - height[i - 2]) + dp[i - 2];

        dp[i] = min(jump1, jump2);
    }

    return dp[n - 1];
}
```

Exercise 1: Frog Jump

Space-Optimized Tabulation

Reducing DP array
to two variables
(constant space)

```cpp
int minEnergy3(int n, vector<int> &height)
{
    int first = 0;
    int second = 0;
    for (int i = 1; i < n; i++) // 1 to n-1
    {
        int jump1 = abs(height[i] - height[i - 1]) + second;
        int jump2 = INT_MAX;
        if (i >= 2)
            jump2 = abs(height[i] - height[i - 2]) + first;

        first = second;
        second = min(jump1, jump2);
    }

    return second;
}
```

Exercise 2: House Robber

1-dimensional DP Question

A thief needs to rob money in a street.
The houses in the street are arranged in a **circular manner**.
Therefore the first and the last house are adjacent to each other.
The security system in the street is such that if **adjacent houses** are robbed, the **police** will get **notified**.

Given an array of integers "arr" which represents money at each house.
Return the **max amount of money** that the thief can rob without alerting the police.

Exercise 2: House Robber

Approach

- Make **2 reduced arrays**: arr1 (w/o last element) and arr2 (w/o first element).
- Find the **max sum of non-adjacent elements** arr1 and arr2 separately.
- Return **max(ans1, ans2)**

Exercise 2: House Robber

Tabulation

Main function

```cpp
int robStreet(int n, vector<int> &arr)
{
    if (n == 1) // Only 1 house
        return arr[0];

    // Reduced Arrays
    vector<int> arr1(arr.begin(), arr.end() - 1); // w/o last element
    vector<int> arr2(arr.begin() + 1, arr.end()); // w/o first element

    // Find the max sum of non-adjacent elements arr1 and arr2 separately
    int ans1 = maxSumNonAdj(n - 1, arr1);
    int ans2 = maxSumNonAdj(n - 1, arr2);
    return max(ans1, ans2);
}
```

Exercise 2: House Robber

Tabulation

Helper function to find the **max sum of non-adjacent elements** in an array

```
int maxSumNonAdj(int n, vector<int> &arr)
{
    vector<int> dp(n, -1); // 0 to n-1

    dp[0] = arr[0];              // Base Case (Only 1 element)
    for (int i = 1; i < n; i++) // 1 to n-1
    {
        int notPick = dp[i - 1];

        int pick = arr[i];
        if (i >= 2)
            pick += dp[i - 2];

        dp[i] = max(pick, notPick);
    }

    return dp[n - 1];
}
```

Exercise 3: Unique Paths

2-dimensional DP Question

Given two values M and N, which represent a matrix[M][N].
Find the **total unique paths**
from the **top-left cell (matrix[0][0])**
to the **rightmost cell (matrix[M-1][N-1])**.

At any cell we are allowed to move in only **two directions**:- **bottom** and **right**.

Exercise 3: Unique Paths

Approach

- Make a **2D dp array** of size MxN.
- **dp[i][j]** = total unique paths from the top-left cell to the cell (i, j).
- **Dependency Direction**: up & left

Exercise 3: Unique Paths

Memoization

Main function

```cpp
int uniquePaths(int m, int n)
{
    // rows: index (0 to m-1), cols: index (0 to n-1)
    vector<vector<int>> dp(m, vector<int>(n, -1));
    return fun(m - 1, n - 1, dp);
}
```

Exercise 3: Unique Paths

Memoization

Helper function to
calculate dp[i][j]

```cpp
int fun(int i, int j, vector<vector<int>> &dp)
{
    // Base Cases
    if (i == 0 && j == 0) // top-left cell
        return 1;
    if (i < 0 || j < 0) // out of bounds
        return 0;

    // Lookup
    if (dp[i][j] != -1)
        return dp[i][j];

    // Recursive Case

    // Move up
    int up = fun(i - 1, j, dp);

    // Move left
    int left = fun(i, j - 1, dp);

    return dp[i][j] = up + left;
}
```

Exercise 3: Unique Paths

Tabulation

Iteratively filling the dp matrix in bottom-up fashion

```cpp
int uniquePaths2(int m, int n)
{
    // rows: index (0 to m-1), cols: index (0 to n-1)
    vector<vector<int>> dp(m, vector<int>(n, 0));

    for (int i = 0; i < m; i++) // rows
    {
        for (int j = 0; j < n; j++) // cols
        {
            if (i == 0 && j == 0) // top-left cell
            {
                dp[i][j] = 1;
                continue;
            }

            // Move up
            int up = 0;
            if (i > 0)
                up = dp[i - 1][j];

            // Move left
            int left = 0;
            if (j > 0)
                left = dp[i][j - 1];

            dp[i][j] = up + left;
        }
    }

    return dp[m - 1][n - 1];
}
```

Thank You!