Dynamic Programming

Aws Ben Gandouz
ENSI, 2023

Objectif of this course

Solve intermediate and hard dp questions related to contests and especially technical interviews

Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

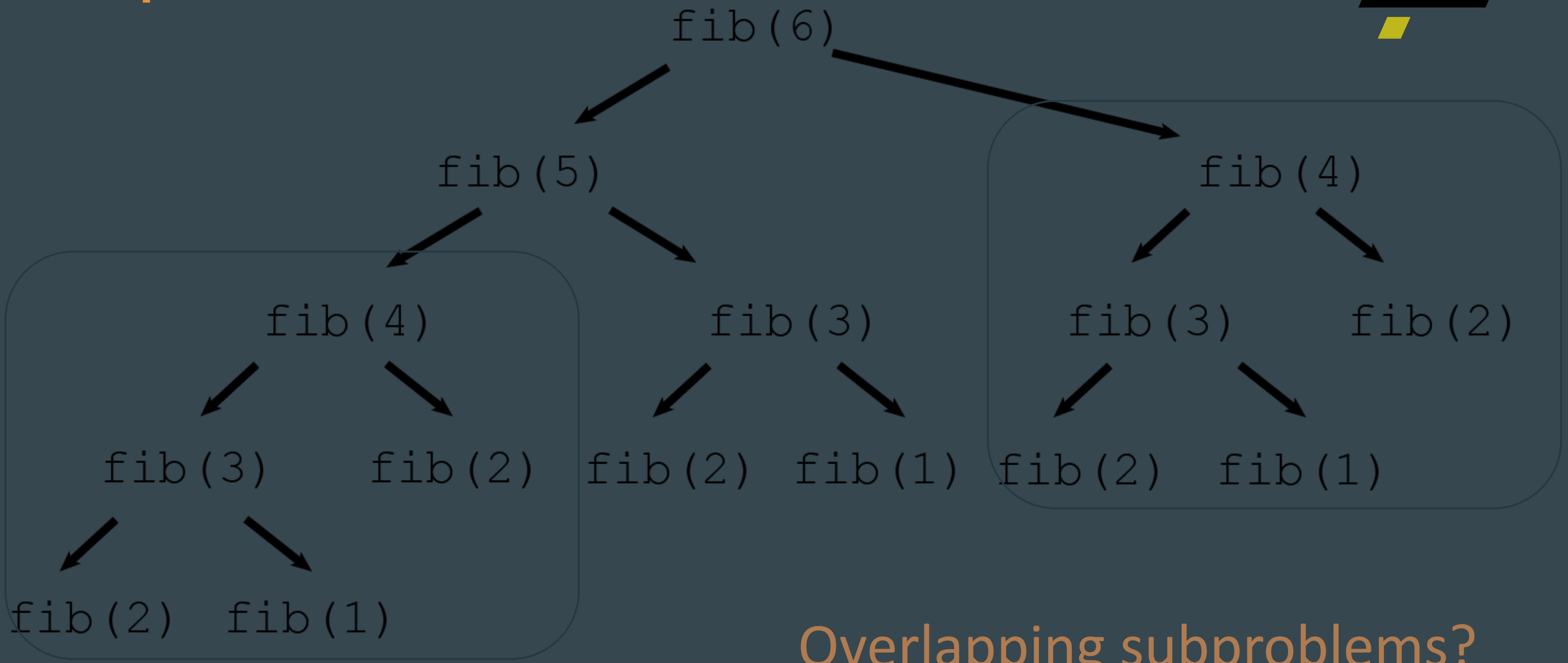
Why Dynamic Programming?

- Overlapping subproblems
- Maximize/Minimize some value
- Finding number of ways
- Covering all cases (DP vs Greedy)

Need of DP

- Let's understand this from a problem
    - Find n^{th} fibonacci number
    - $F(n) = F(n - 1) + F(n - 2)$ [recursive formula]
    - $F(1) = F(2) = 1$
- Drawing the code paths as a tree (recursion tree) is useful in many recursive problems.

state space tree



Overlapping subproblems?

Memoization

- Why calculate F(x) again and again when we can calculate it once and use it every time it is required?
  - Check if F(x) has been calculated
    - If No, calculate it and store it somewhere
    - If Yes, return the value without calculating again

Without DP

```cpp
int ans = 0;
int dp[40];

int fib(int n){
    ans++;
    if(n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}

void solve(){
    int n;
    cin >> n;
    cout << fib(n) << endl;
    cout << ans << endl;
}
```

```
aws@aws-laptop: ~/Desktop/C++
aws@aws-laptop:~/Desktop/C++$ g++ SOL.cpp -o test
aws@aws-laptop:~/Desktop/C++$ ./test
30
832040
2692537
aws@aws-laptop:~/Desktop/C++$
```

With DP

```cpp
int ans = 0;
int dp[40];

int fib(int n){
    ans++;
    // base case
    if(n <= 1) return n;
    //transition
    if(dp[n] != -1) return dp[n];
    return dp[n] = fib(n - 1) + fib(n - 2);
}

void solve(){
    int n;
    cin >> n;
    memset(dp , -1 , sizeof dp);
    cout << fib(n) << endl;
    cout << ans << endl;
}
```

aws@aws-laptop: ~/Desktop/C++

```
aws@aws-laptop:~/Desktop/C++$ g++ SOL.cpp -o test
aws@aws-laptop:~/Desktop/C++$ ./test
30
832040
59
aws@aws-laptop:~/Desktop/C++$
```

General Technique to solve any DP problem

1. State

   Clearly define the subproblem. Clearly understand when you are saying dp[i][j][k], what does it represent exactly

2. Transition:

   Define a relation b/w states. Assume that states on the right side of the equation have been calculated. Don't worry about them.

3. Base Case

   When does your transition fail? Call them base cases answer before hand. Basically handle them separately.

4. Final Subproblem

   What is the problem demanding you to find?

Let's solve another problem!

64. Minimum Path Sum

Medium 9.4K 122

Companies

Given a `m x n` `grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:

1	3	1
1	5	1
4	2	1

Solution

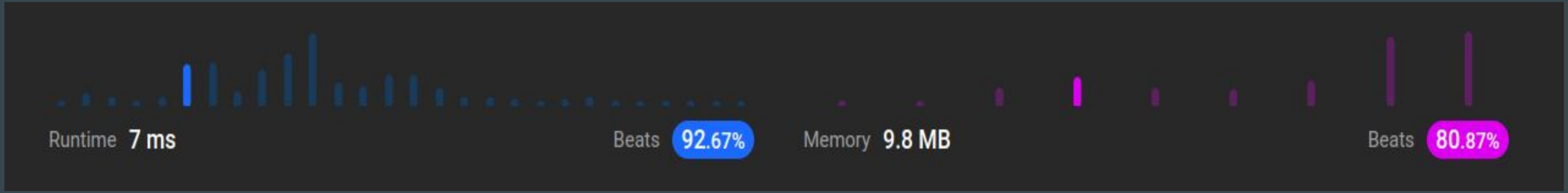
```
class Solution {
public:
    int helper(vector<vector<int>>& arr , int i , int j , int n , int m , vector<vector<int>>& dp){
        // Base case
        if(i == n-1 && j == m-1) return arr[i][j] ;

        // check if dp[i][j] is calculated , if yes return it
        if(dp[i][j] != -1) return dp[i][j] ;

        // transition
        int right = 1e9 , down = 1e9 ;
        if(j+1 < m) right = arr[i][j] + helper(arr , i , j+1 , n , m, dp) ;
        if(i+1 < n) down = arr[i][j] + helper(arr , i+1 , j , n , m, dp) ;

        return dp[i][j] = min(right , down) ;
    }

    int minPathSum(vector<vector<int>>& arr) {
        int n = arr.size() ;
        int m = arr[0].size() ;
        vector<vector<int>>dp(n , vector<int>(m , -1)) ;
        return helper(arr , 0 , 0 , n , m , dp) ;
    }
};
```

This problem can be solved iteratively

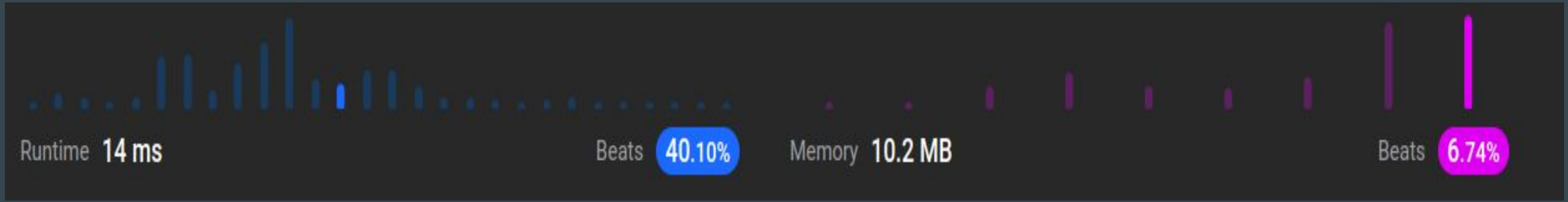
```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int n = grid.size() , m = grid[0].size();

        int dp[n][m];
        memset(dp , 0 , sizeof dp);


        dp[0][0] = grid[0][0];

        // base case i == 0 or j == 0
        for(int i = 1; i < n ; i++){
            dp[i][0] += dp[i-1][0] + grid[i][0];
        }
        for(int j = 1 ; j < m ; j++){
            dp[0][j] += dp[0][j-1] + grid[0][j];
        }
        // transition
        for(int i= 1 ; i < n ; i++){
            for(int j = 1 ; j < m ; j++){
                dp[i][j] = grid[i][j] + min(dp[i-1][j] , dp[i][j-1]);
            }
        }
        return dp[n-1][m-1];
    }
};
```

Difference between recursive / iterative solution

iterative solution



recursive solution



Recursive vs Iterative DP

Recursive	Iterative
Slower (runtime)	Faster (runtime)
No need to care about the flow	Important to calculate states in a way that current state can be derived from previously calculated states
Does not evaluate unnecessary states	All states are evaluated
Cannot apply many optimizations	Can apply optimizations

Important Terminology

State: A subproblem that we want to solve. The subproblem may be complex or easy to solve but the final aim is to solve the final problem which may be defined by a relation between the smaller subproblems. Represented with some parameters.

Transition: Calculating the answer for a state (subproblem) by using the answers of other smaller states (subproblems). Represented as a relation b/w states.

Exercise 1

Fibonacci Problem:

- State
  - dp[i] or f(i) meaning i^{th} fibonacci number
- Transition
  - dp[i] = dp[i - 1] + dp[i - 2]

Exercise 2

Matrix Problem:

- State
  - dp[i][j] = shortest sum path from (i, j) to (n - 1, m - 1)
- Transition
  - dp[i][j] = grid[i][j] + min(dp[i + 1][j], dp[i][j + 1])

Let’s solve another problem

Given an array of integers (both positive and negative). Pick a subsequence of elements from it such that no 2 adjacent elements are picked and the sum of picked elements is maximized.

1	**4**	2	-10	**10**	5

Sum = 14

1	4	**2**	-10	**10**	5

Sum = 13

Solution

Having only 1 parameter to represent the state (state variable)

State:

dp[i] = max sum in (0 to i) not caring if we picked i^{th} element or not

Transition: 2 cases

- pick i^{th} element: cannot pick the last element : arr[i] + dp[i - 2]
- leave i^{th} element: can pick the last element : dp[i - 1]

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

Final Answer:

dp[n - 1]

```cpp
int a[n]; // input array

int dp[n]; // filled with -INF to represent uncalculated state

// f(i) = max sum till index i
int f(int index){
    if(index < 0) // reached outside the array
        return 0;
    if(dp[index] != -INF) // state already calculated
        return dp[index];

    // try both cases and store the answer
    dp[index] = max(a[index] + f(index - 2), f(index - 1));
    return dp[index];
}

void solve(){
    cout << f(n - 1) << nline;
}
```

Problem 1: Link

- State:
  - dp[i] = number of ways to get sum == i
- Transition:
  - dp[i] = dp[i - 1] + dp[i - 2]...... + dp[i - 6]
- Final Subproblem:
  - dp[n]

Problem 2: Link

- State:
  - dp[k] = min coins required to make sum == k
- Transition:
  - $dp[k] = 1 + \min\{dp[k - coins_i]\}$ $(0 \le i \le n - 1)$
- Final Subproblem:
  - dp[x]

Problem 3: Link

- State:
  - dp[i] = number of ways to make sum == i
- Transition:
  - dp[i] = sum of dp[i - $coins_j$] (0 <= j <= n - 1)
- Final Subproblem:
  - dp[x]

Time and Space Complexity in DP

Time Complexity:

Estimate: Number of States * Transition time for each state

Exact: Total transition time for all states

Space Complexity:

Number of States * Space required for each state

Thank you for reading !