AlgoZenith | Vivek Gupta

AZ DP WORKSHOP

How to Structure up Dynamic Programming Code?

Problem:

People can often solve DP problems while they are practicing themselves, but fail to perform with the same speed and accuracy in coding contests and online assessment rounds. Both coding contests and online assessment rounds are timed contests and are designed to put pressure on you. Under that time constraint and pressure, it is not only difficult to code, but it becomes a nightmare to debug.

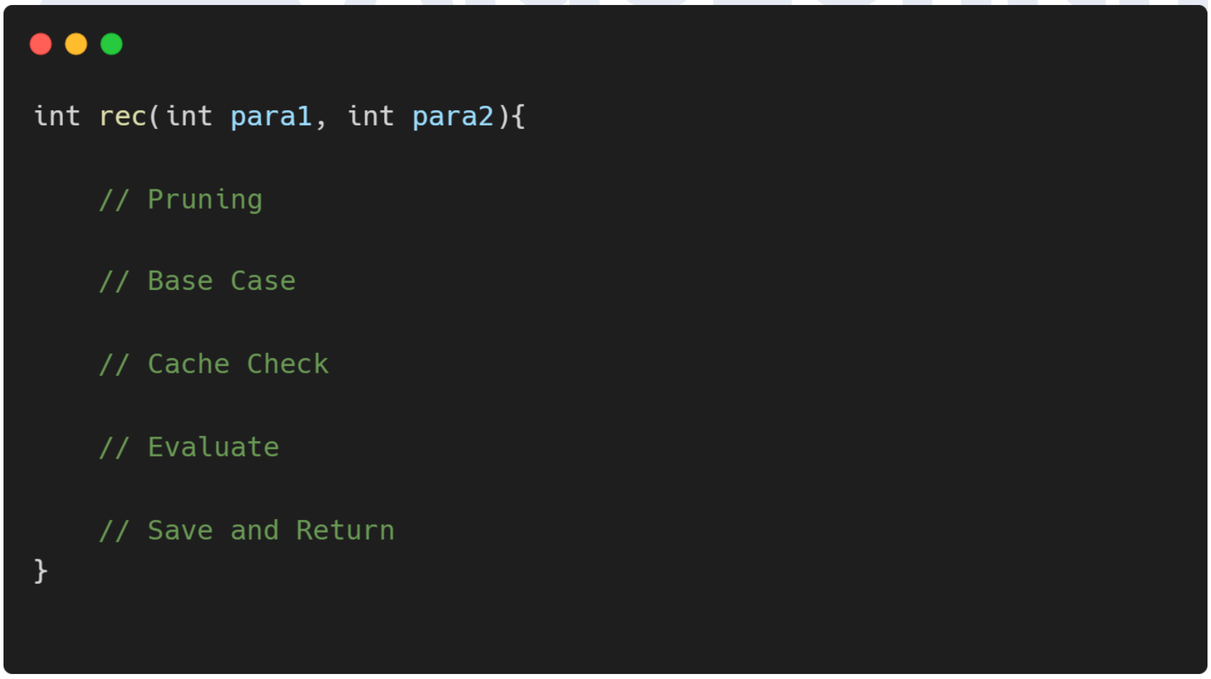
Solution:

A lot of pressure can be released off of you if you start thinking and coding structurally. Structure not only makes us think fast, but helps to debug code even faster. Let’s learn a way to structure-up our DP programs, so that we get AC every time.

Structure:

We’ll use a 5-step structure to write our DP programs. We’ll use the recursive approach to DP to solve our problem.

We’ll start with writing a recursive function, something like this:



```
int rec(int para1, int para2){

    // Pruning

    // Base Case

    // Cache Check

    // Evaluate

    // Save and Return
}
```

Before jumping into the 5 steps, let's look at some other things.

Return Value:

In 95% of the cases, the return value of the recursive function is going to be the answer that is asked in the problem statement. If the problem asks for a yes or no value, the return datatype would be boolean. If the problem asks for the count / number of ways, then the return type would be integer.

This value is going to be stored in our DP array, thus, our DP array would be of the same datatype.

Note: In some cases, you might need to store something else in the DP array, because of the constraints on one of the parameters. In that case, you might need to exchange one of the parameters with the solution asked. One such example is present in Atcoder Educational DP contest. Look for the difference between the two problems:

1. Normal Case: https://atcoder.jp/contests/dp/tasks/dp_d
2. Special Case (State rotation): https://atcoder.jp/contests/dp/tasks/dp_e

Parameters:

The parameters are almost always the dimensions of the DP array. All the things that are necessary to identify a unique state are kept in the parameters.
For example: In the famous Knapsack problem we keep two parameters:

1. The array index (array till we have traversed)
2. The weight taken (or the weight left, depending on how you implement it)

The Return Value and the Parameters together define how our DP array will look like.

For example:

```
// N --> Largest value of para1
// M --> Largest value of para2
int dp[N][M];
```

The 5-Step Structure:

Let us understand the structure by taking an example problem in mind.

Problem statement:

Given a grid of size n*m, you need to find the number of paths from (1, 1) to (n, m). You are allowed to move from (x, y) to (x+1, y) and (x, y) to (x, y+1). The grid may have some blocked cells, represented by 1, and it is not allowed to move to a blocked cell. An empty cell is represented by 0.

Solution:

Based on our current understanding, we'll declare some global variables and the DP and grid array. It'll look something like this:

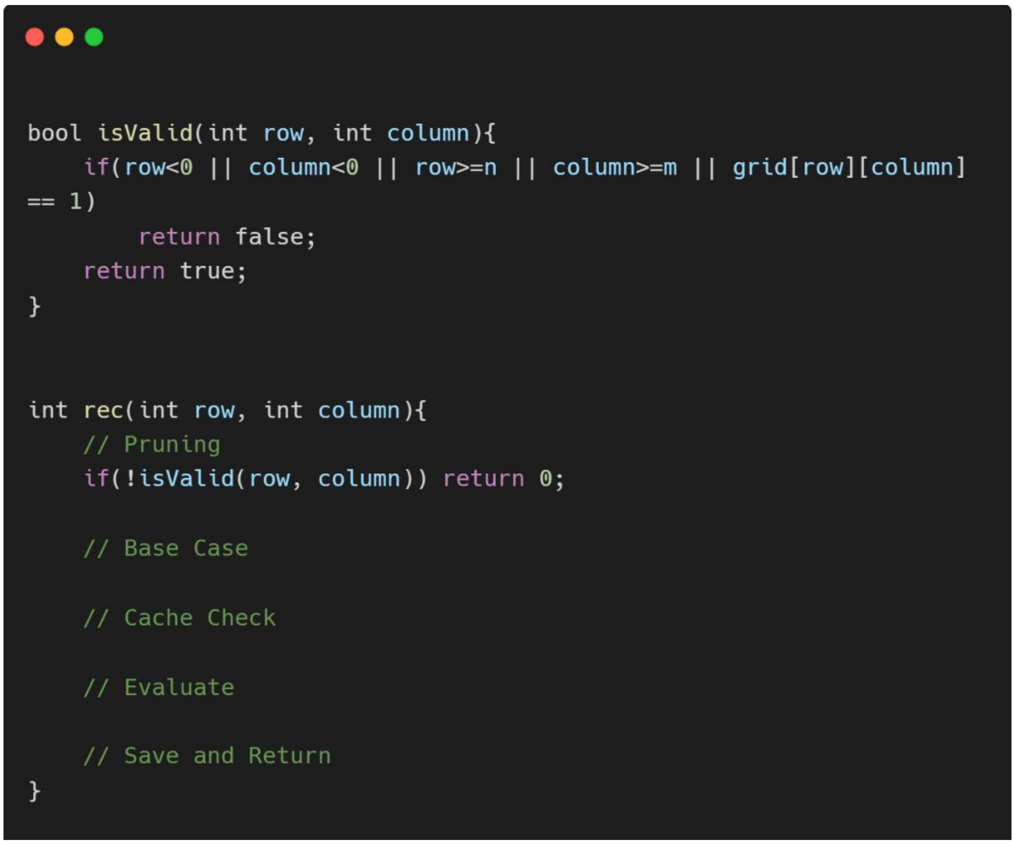
```
const int N = 1010;
int grid[N][N];
int dp[N][N];
int n, m;
const int mod = 1e9 + 7;

int rec(int para1, int para2){
    // Pruning

    // Base Case

    // Cache Check

    // Evaluate

    // Save and Return
}
```

Here, 'N' is the max size of rows and columns. 'n' and 'm' are the variables in which we'll take input of the size of the grid.

Let’s start with our structure:

Step 1: Pruning

We prune off the edge / boundary cases here. Anything that might be an exception, or might create a problem with our code can be handled here.
For eg: In our example problem, the parameters that are passed, that is, the row and the column can be neither negative, nor greater than the size of the grid. Therefore, we can fill up the code like this:



```
bool isValid(int row, int column){
    if(row<0 || column<0 || row>=n || column>=m || grid[row][column]
== 1)
        return false;
    return true;
}


int rec(int row, int column){
    // Pruning
    if(!isValid(row, column)) return 0;

    // Base Case

    // Cache Check

    // Evaluate

    // Save and Return
}
```

Step 2: Base Case:

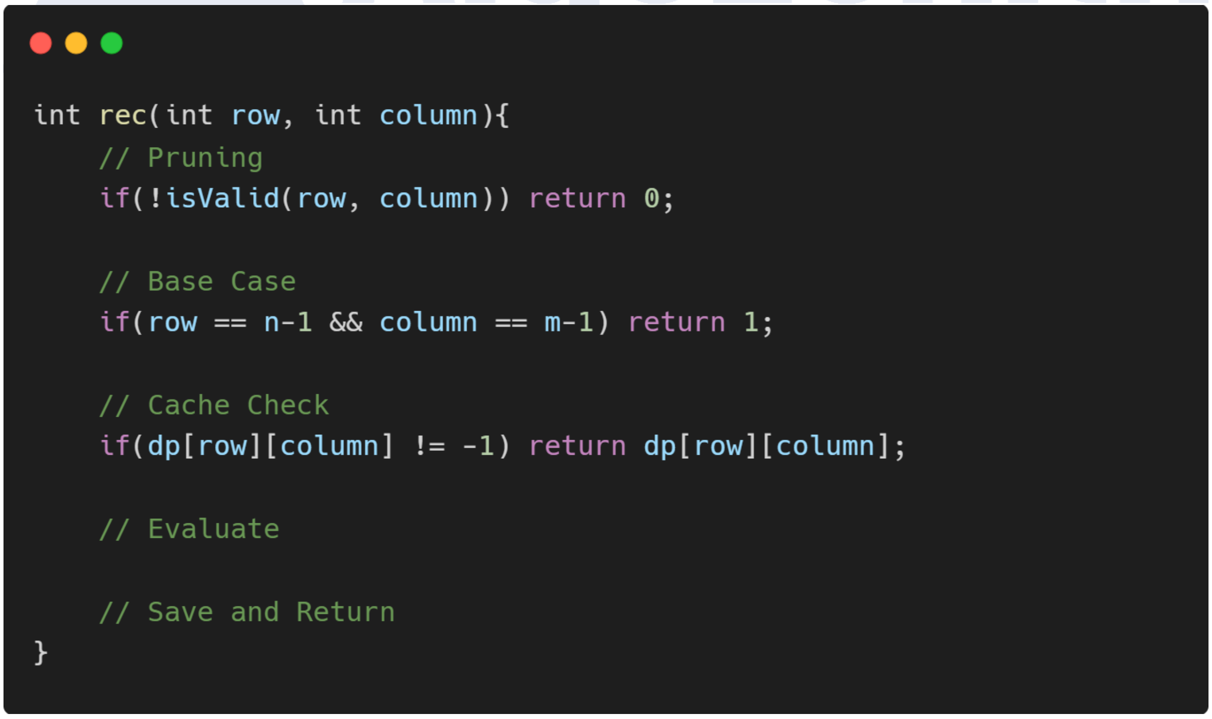
This is where we handle the base case of the recursion.

For Eg: In our example problem, reaching the target cell, that is (n, m), would be the base case.

```
int rec(int row, int column){
    // Pruning
    if(!isValid(row, column)) return 0;

    // Base Case
    if(row == n-1 && column == m-1) return 1;

    // Cache Check

    // Evaluate

    // Save and Return
}
```

Step 3: Cache / Memory Check:

This is where we check if we have previously computed and stored the answer for this state or not. If yes, we return the stored value. If not, we continue to our next step.

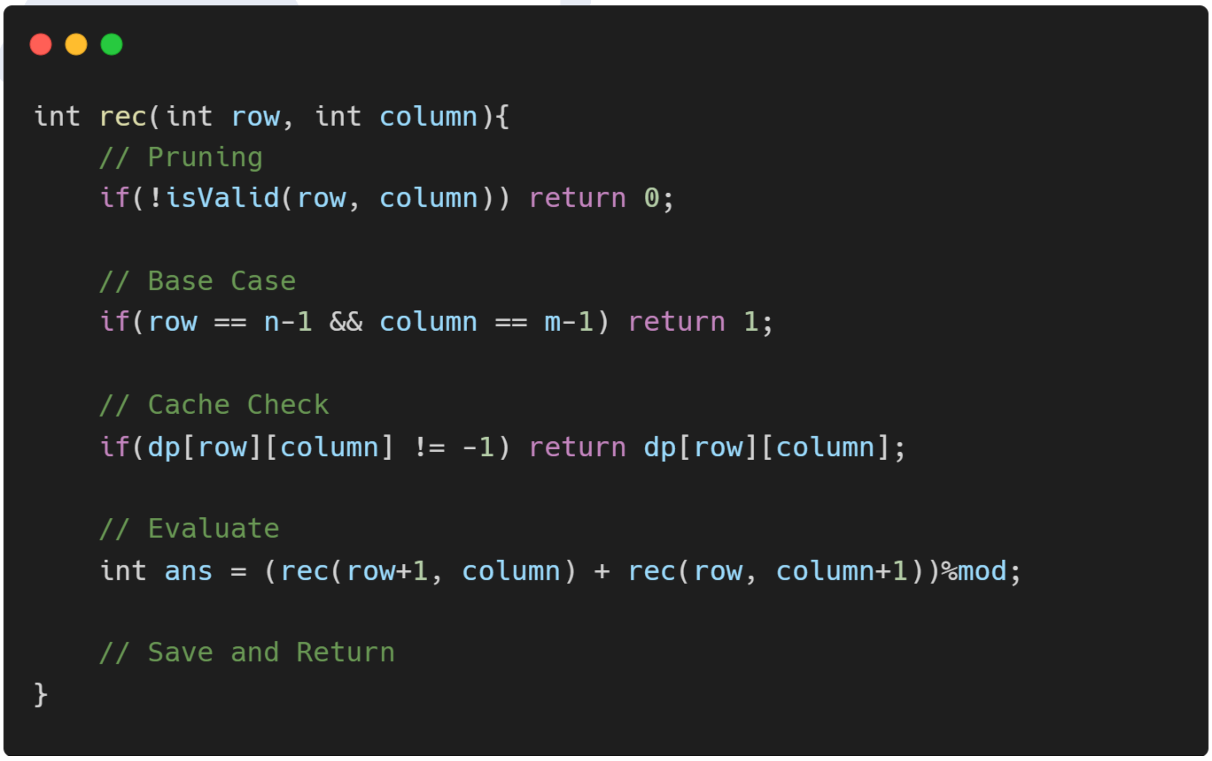


```cpp
int rec(int row, int column){
    // Pruning
    if(!isValid(row, column)) return 0;

    // Base Case
    if(row == n-1 && column == m-1) return 1;

    // Cache Check
    if(dp[row][column] != -1) return dp[row][column];

    // Evaluate

    // Save and Return
}
```

Step 4: Evaluate:

We reach this step if we haven't previously stored the answer for this state. We create an ‘answer’ variable, compute the answer

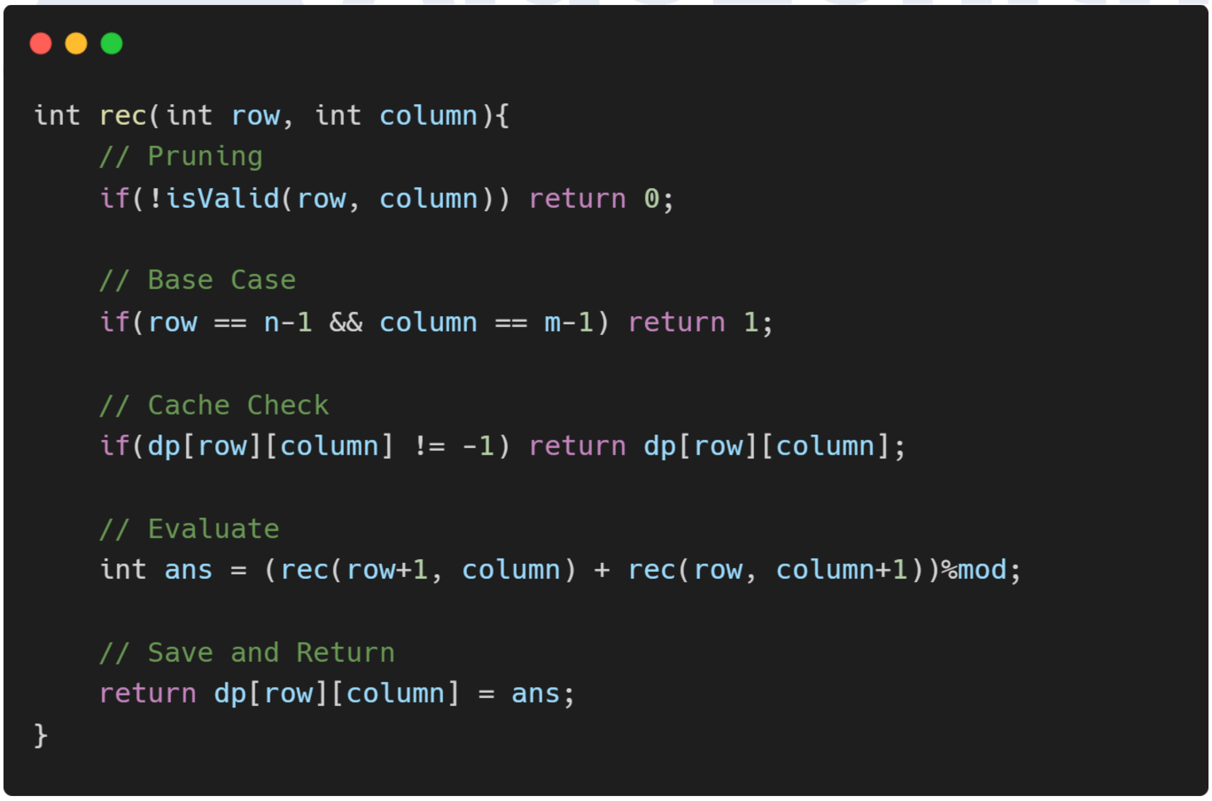
and store it in the ‘answer’ variable. Something like this:



```
int rec(int row, int column){
    // Pruning
    if(!isValid(row, column)) return 0;

    // Base Case
    if(row == n-1 && column == m-1) return 1;

    // Cache Check
    if(dp[row][column] != -1) return dp[row][column];

    // Evaluate
    int ans = (rec(row+1, column) + rec(row, column+1))%mod;

    // Save and Return
}
```

Step 5: Save and Return:

Finally, saving and returning is our final step. We save the answer in our DP array and return the computed value.



```cpp
int rec(int row, int column){
    // Pruning
    if(!isValid(row, column)) return 0;

    // Base Case
    if(row == n-1 && column == m-1) return 1;

    // Cache Check
    if(dp[row][column] != -1) return dp[row][column];

    // Evaluate
    int ans = (rec(row+1, column) + rec(row, column+1))%mod;

    // Save and Return
    return dp[row][column] = ans;
}
```

And we're done. Just like that we created a structure to write our DP codes in a very clean manner. Once you practice with this structure, you'll see the difference in your speed and accuracy. Not only that, it becomes much easier to debug as each component is clearly visible with clear meanings associated with them.

SEE YOU IN 7-DAYS OF DP WORKSHOP

Link to Register:

www.algozenith.com/dpworkshop