Dynamic Programming
Class 2

- Priyansh Agarwal

Recursive vs Iterative DP (in Day 2)

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

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?

Solve Problems

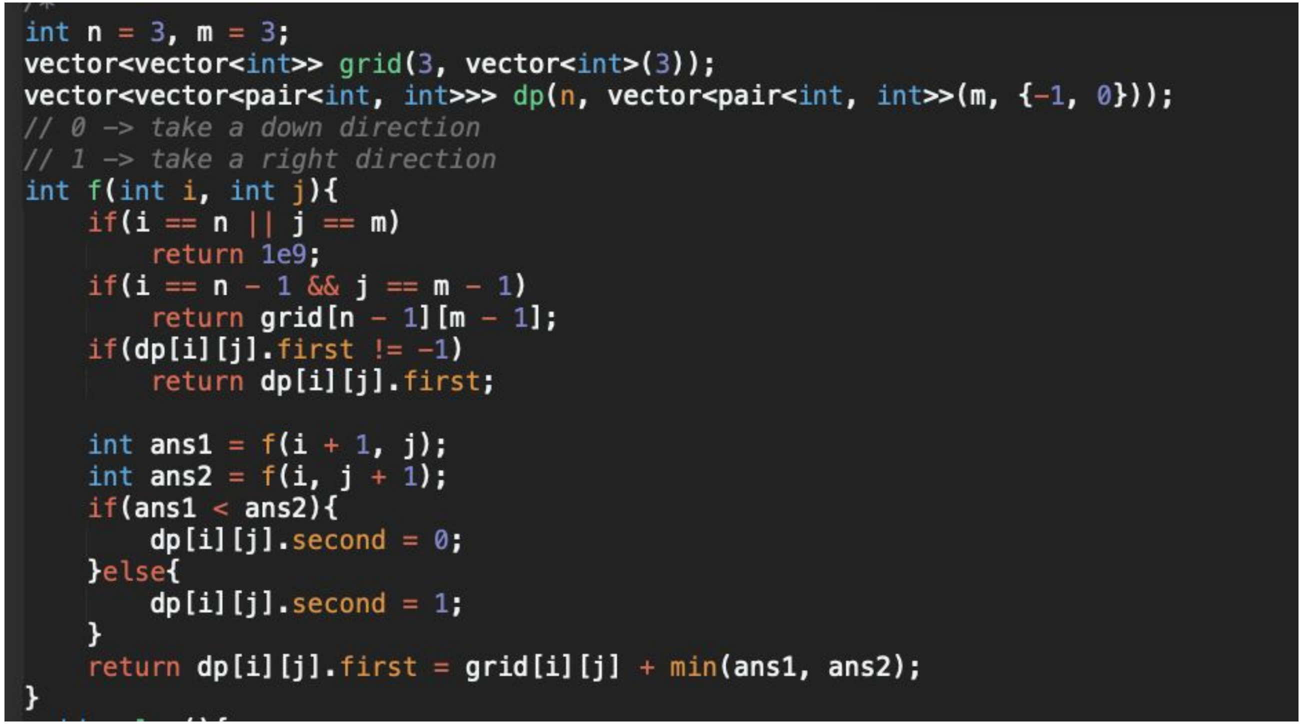
Problem 1: Link

Problem 2: Link

Answer Construction

- Grid problem: Find the actual path with the minimum sum.
- Minimizing coins problem: Find the actual choice of coins.
- At every state we are making some optimal choice.
    - If we store this choice, we can be sure that if we are at any state we know what is the best choice.
    - Start from the state that contains your final subproblem and keep making the best choice (which was already stored) until you reach the end.

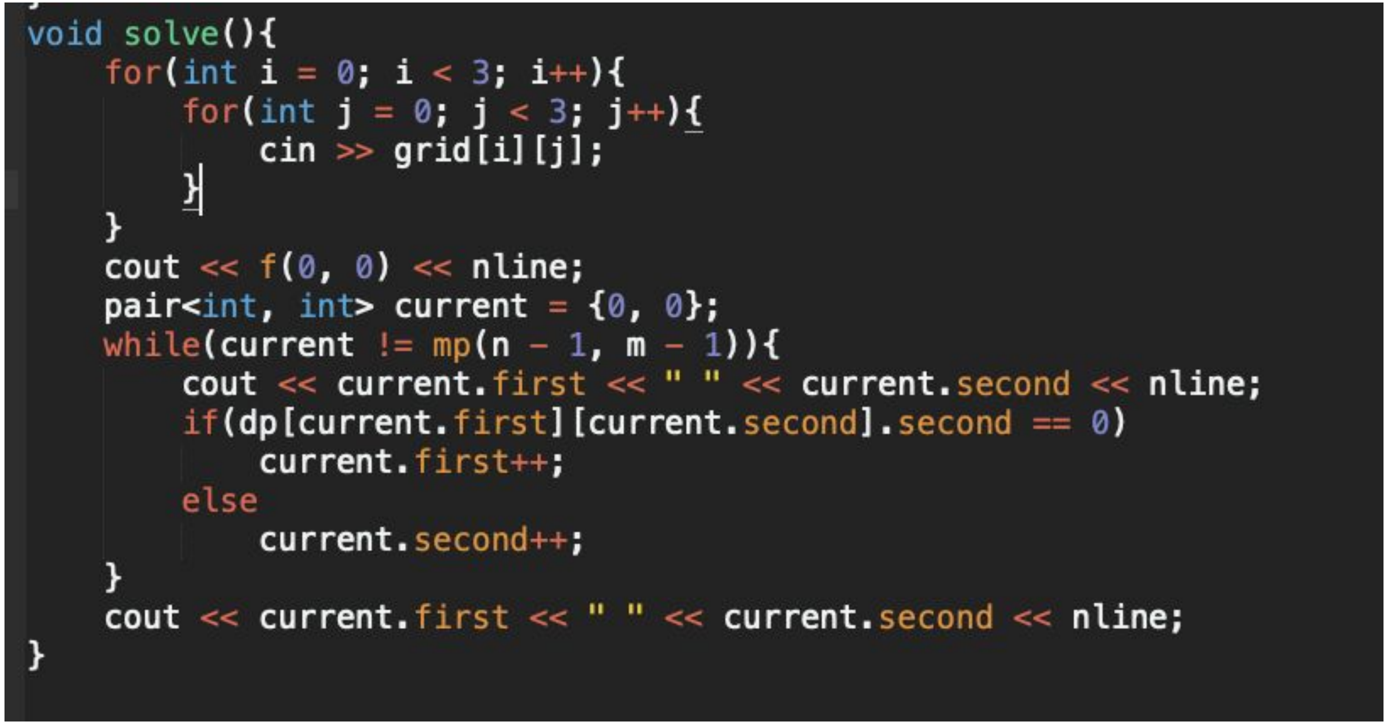
Answer Construction - Grid Problem



```
int n = 3, m = 3;
vector<vector<int>> grid(3, vector<int>(3));
vector<vector<pair<int, int>>> dp(n, vector<pair<int, int>>(m, {-1, 0}));
// 0 -> take a down direction
// 1 -> take a right direction
int f(int i, int j){
    if(i == n || j == m)
        return 1e9;
    if(i == n - 1 && j == m - 1)
        return grid[n - 1][m - 1];
    if(dp[i][j].first != -1)
        return dp[i][j].first;

    int ans1 = f(i + 1, j);
    int ans2 = f(i, j + 1);
    if(ans1 < ans2){
        dp[i][j].second = 0;
    }else{
        dp[i][j].second = 1;
    }
    return dp[i][j].first = grid[i][j] + min(ans1, ans2);
}
```

Answer Construction - Grid Problem



```
void solve(){
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            cin >> grid[i][j];
        }
    }
    cout << f(0, 0) << nline;
    pair<int, int> current = {0, 0};
    while(current != mp(n - 1, m - 1)){
        cout << current.first << " " << current.second << nline;
        if(dp[current.first][current.second].second == 0)
            current.first++;
        else
            current.second++;
    }
    cout << current.first << " " << current.second << nline;
}
```

State Optimization

- Ask yourself do you need all the parameters in the dp state?
- If you have dp[a][b][c], and a + b = c, do you need to store c as a parameter or can you just compute it on spot?
- If you can compute a parameter in dp state from other parameters, no need to store it.
- Which parameters should you remove? Highest

Transition Optimization

- Observe the transition equation.
- Can you do some pre-computation to evaluate the equation faster?
- Using clever observations.
- Using range query data structures

Space Optimization

- What other state does our current state depend on?
- Do we need answers to all subproblems at all times?
- Well, let’s store only relevant states then!
- But wait, does this always work?
    - What if the final subproblem requires all the states?
    - What if we need to backtrack? [more on this in later lectures]