Dynamic Programming
Day 1

Why Dynamic Programming?

"THOSE WHO CANNOT REMEMBER THE PAST ARE CONDEMNED TO REPEAT IT"

-George Santayana-

Why Dynamic Programming?

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

Need of DP

- Let's understand this from a problem
    - Find n^{th} fibonacci number
    - $F(n) = F(n - 1) + F(n - 2)$
    - $F(1) = F(2) = 1$

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

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

Some ways to solve the problem

Having only 1 parameter to represent the state

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]

Let's solve another problem!

Given a 2D grid (N X M) with numbers written in each cell, find the path from top left (0, 0) to bottom right (n - 1, m - 1) with minimum sum of values on the path

1	5	8
6	2	7
9	3	4

Naive Way

Explore all paths. Standing at (i, j) try both possibilities (i + 1, j), (i, j + 1)

Every cell has two choices

Time complexity: $O(2^{m*n})$?

Actual Time complexity: $O(C(n + m - 2, m - 1))$

Efficient Way

Overlapping subproblems

Memoization

Time complexity: O(n * m)

Space complexity: O(n * m)

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.

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

Problem : Your task is to count the number of ways to construct sum n by throwing a dice one or more times. Each throw produces an outcome between 1 and 6.

- 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: You are given an integer n. On each step, you may subtract one of the digits from the number.

How many steps are required to make the number equal to 0?

- State:
  - dp[x] = min steps to convert x to 0
- Transition:
  - dp[x] = min(dp[x - some digit of x]) + 1
- Base Case:
  - dp[0] = 0
- Final Subproblem:
  - dp[n]

Problem: Consider a money system consisting of n coins. Each coin has a positive integer value. Your task is to produce a sum of money x using the available coins in such a way that the number of coins is minimal.

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

Problem: Consider a money system consisting of n coins. Each coin has a positive integer value. Your task is to calculate the number of distinct ways you can produce a money sum x using the available coins.

- 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]

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?

Problem: Consider an n * n grid whose squares may have traps. It is not allowed to move to a square with a trap.

Your task is to calculate the number of paths from the upper-left square to the lower-right square. You can only move right or down.

- State:
  - dp[i][j] = number of ways to go from (i, j) to (n - 1, n - 1)
- Transition:
  - dp[i][j] = dp[i + 1][j] + dp[i][j + 1]
- Base Case:
  - dp[n - 1][n - 1] = 1, dp[i][j] = 0, when (i, j) is a trap
- Final Subproblem:
  - dp[0][0]