Dynamic Programming

Tawhid Bin Omar

What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving optimization problems by breaking them down into simpler subproblems. It stores solutions to subproblems to avoid redundant computations.

Key Insight

When to use DP?

- Problem has **optimal substructure**: optimal solution contains optimal solutions to subproblems
- Problem has **overlapping subproblems**: same subproblems are solved multiple times
- You need to count/optimize/decide something across choices

1 DP Approaches

1. Top-Down (Memoization)

Start with original problem, recursively break down, cache results.

```cpp
// Top-down template
vector<int> memo;

int solve(int state) {
    // Base case
    if (base_condition) return base_value;

    // Check memo
    if (memo[state] != -1)
        return memo[state];

    // Compute and store
    int result = /* recurrence */;
    return memo[state] = result;
}
```

2. Bottom-Up (Tabulation)

Build solution from smallest subproblems up to final answer.

```cpp
// Bottom-up template
int solve(int n) {
    vector<int> dp(n + 1);

    // Base cases
    dp[0] = base_value;

    // Fill table
    for (int i = 1; i <= n; i++) {
        dp[i] = /* recurrence using dp[j], j < i */;
    }

    return dp[n];
}
```

2 1D DP Problems

Fibonacci Sequence

Problem

Find the n-th Fibonacci number where $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$.

Recurrence Relation

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Solution

```cpp
// Space-optimized O(1) space
long long fibonacci(int n) {
    if (n <= 1) return n;
    long long a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        long long c = a + b;
        a = b;
        b = c;
    }
    return b;
}

// With memoization
vector<long long> memo(100, -1);
long long fib(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fib(n-1) + fib(n-2);
}
```

Climbing Stairs

Problem

You can climb 1 or 2 steps at a time. How many distinct ways to climb n steps?

Recurrence Relation

$$dp[i] = dp[i-1] + dp[i-2]$$

Number of ways to reach step i is sum of ways to reach previous two steps.

Solution

```
int climbStairs(int n) {
    if (n <= 2) return n;
    int prev2 = 1, prev1 = 2;
    for (int i = 3; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

House Robber

Problem

Rob houses in a line. Cannot rob adjacent houses. Maximize money robbed.

Recurrence Relation

$$dp[i] = \max(dp[i-1],\ dp[i-2] + \text{arr}[i])$$

Either skip current house or rob it (can't rob previous).

Solution

```
int rob(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;
    if (n == 1) return nums[0];

    int prev2 = nums[0];
    int prev1 = max(nums[0], nums[1]);

    for (int i = 2; i < n; i++) {
        int curr = max(prev1, prev2 + nums[i]);
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Maximum Subarray Sum (Kadane)

Problem

Find contiguous subarray with maximum sum.

Recurrence Relation

$$dp[i] = \max(\text{arr}[i],\ dp[i-1] + \text{arr}[i])$$

Either start fresh or extend previous subarray.

Solution

```
int maxSubArray(vector<int>& nums) {
    int maxSum = nums[0];
    int currentSum = nums[0];

    for (int i = 1; i < nums.size(); i++) {
        currentSum = max(nums[i],
            currentSum + nums[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

Longest Increasing Subsequence

Problem: LIS

Find length of longest strictly increasing subsequence. arr $= [10, 9, 2, 5, 3, 7, 101, 18]$ gives LIS length 4: $[2, 3, 7, 101]$.

Recurrence Relation

$dp[i] = \max(dp[j]+1)$ for all $j < i$ where $\text{arr}[j] < \text{arr}[i]$

Solution

```cpp
// O(n^2) solution
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1);
    int maxLen = 1;

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        maxLen = max(maxLen, dp[i]);
    }
    return maxLen;
}

// O(n log n) using binary search
int lengthOfLIS_Optimized(vector<int>& nums)
    {
    vector<int> tails;

    for (int num : nums) {
        auto it = lower_bound(tails.begin(),
            tails.end(), num);
        if (it == tails.end()) {
            tails.push_back(num);
        } else {
            *it = num;
        }
    }
    return tails.size();
}
```

Coin Change

Problem

Given coin denominations, find minimum coins to make amount n. Return -1 if impossible.

Recurrence Relation

$$dp[i] = \min(dp[i - \text{coin}] + 1) \text{ for all coins } \leq i$$

Solution

```cpp
int coinChange(vector<int>& coins, int
    amount) {
    vector<int> dp(amount + 1, INT_MAX);
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int coin : coins) {
            if (i >= coin && dp[i - coin] !=
                INT_MAX) {
                dp[i] = min(dp[i], dp[i - coin]
                    + 1);
            }
        }
    }
    return dp[amount] == INT_MAX ? -1 :
        dp[amount];
}
```

Coin Change II (Count Ways)

Problem

Count number of ways to make amount using given coins. Each coin can be used unlimited times.

Solution

```cpp
int change(int amount, vector<int>& coins) {
    vector<int> dp(amount + 1, 0);
    dp[0] = 1;

    // Important: iterate coins first
    for (int coin : coins) {
        for (int i = coin; i <= amount; i++) {
            dp[i] += dp[i - coin];
        }
    }
    return dp[amount];
}
```

Key Insight

Coin Change vs Coin Change II:

- Coin Change: minimize coins (optimization)
- Coin Change II: count ways (counting)
- Order matters: iterate coins outside for combinations, inside for permutations

3 2D DP Problems

0/1 Knapsack

Problem

Given items with weights and values, and knapsack capacity W. Each item can be taken at most once. Maximize total value.

Recurrence Relation

$$dp[i][w] = \max \begin{cases} dp[i-1][w] & \text{(skip item)} \\ dp[i-1][w - wt[i]] + val[i] & \text{(take item)} \end{cases}$$

Solution

```cpp
// 2D DP
int knapsack(vector<int>& wt, vector<int>&
    val,
   int W) {
   int n = wt.size();
   vector<vector<int>> dp(n + 1,
      vector<int>(W + 1, 0));

   for (int i = 1; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         dp[i][w] = dp[i-1][w]; // skip
         if (w >= wt[i-1]) {
            dp[i][w] = max(dp[i][w],
               dp[i-1][w - wt[i-1]] +
                    val[i-1]);
         }
      }
   }
   return dp[n][W];
}

// Space optimized O(W)
int knapsack_optimized(vector<int>& wt,
   vector<int>& val, int W) {
   int n = wt.size();
   vector<int> dp(W + 1, 0);

   for (int i = 0; i < n; i++) {
      // Traverse backwards to use old values
      for (int w = W; w >= wt[i]; w--) {
         dp[w] = max(dp[w],
            dp[w - wt[i]] + val[i]);
      }
   }
   return dp[W];
}
```

Unbounded Knapsack

Problem

Same as 0/1 knapsack but each item can be taken unlimited times.

Solution

```cpp
int unboundedKnapsack(vector<int>& wt,
   vector<int>& val, int W) {
   vector<int> dp(W + 1, 0);

   for (int w = 1; w <= W; w++) {
      for (int i = 0; i < wt.size(); i++) {
         if (w >= wt[i]) {
            dp[w] = max(dp[w],
               dp[w - wt[i]] + val[i]);
         }
      }
   }
   return dp[W];
}
```

Longest Common Subsequence

Problem: LCS

Find length of longest subsequence common to both strings. For `"abcde"` and `"ace"`, LCS is `"ace"` with length 3.

Recurrence Relation

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } s1[i] = s2[j] \\ \max(dp[i-1][j], dp[i][j-1]) & \text{otherwise} \end{cases}$$

Solution

```cpp
int longestCommonSubsequence(string s1,
   string s2) {
   int n = s1.length(), m = s2.length();
   vector<vector<int>> dp(n + 1,
      vector<int>(m + 1, 0));

   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++) {
         if (s1[i-1] == s2[j-1]) {
            dp[i][j] = dp[i-1][j-1] + 1;
         } else {
            dp[i][j] = max(dp[i-1][j],
               dp[i][j-1]);
         }
      }
   }
   return dp[n][m];
}

// Space optimized O(min(n,m))
int lcs_optimized(string s1, string s2) {
   int n = s1.length(), m = s2.length();
   vector<int> prev(m + 1, 0), curr(m + 1,
       0);

   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++) {
         if (s1[i-1] == s2[j-1]) {
            curr[j] = prev[j-1] + 1;
         } else {
            curr[j] = max(prev[j],
                curr[j-1]);
         }
      }
      swap(prev, curr);
   }
   return prev[m];
}
```

Edit Distance

Problem

Minimum operations (insert, delete, replace) to convert string $s1$ to $s2$.

Recurrence Relation

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & \text{if } s1[i \\ 1 + \min \begin{cases} dp[i-1][j] & \text{(delete)} \\ dp[i][j-1] & \text{(insert)} \\ dp[i-1][j-1] & \text{(replace)} \end{cases} \end{cases}$$

Solution

```
int minDistance(string s1, string s2) {
    int n = s1.length(), m = s2.length();
    vector<vector<int>> dp(n + 1,
        vector<int>(m + 1));

    // Base cases
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = 1 + min({
                    dp[i-1][j], // delete
                    dp[i][j-1], // insert
                    dp[i-1][j-1] // replace
                });
            }
        }
    }
    return dp[n][m];
}
```

Unique Paths

Problem

Count paths from top-left to bottom-right in $m \times n$ grid. Can only move right or down.

Recurrence Relation

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

Paths to (i, j) = paths from above + paths from left.

Solution

```
int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n,
        1));

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    return dp[m-1][n-1];
}

// Space optimized O(n)
int uniquePaths_optimized(int m, int n) {
    vector<int> dp(n, 1);

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[j] += dp[j-1];
        }
    }
    return dp[n-1];
}
```

Minimum Path Sum

Problem

Find path from top-left to bottom-right in grid with minimum sum. Can only move right or down.

Solution

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

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

    // First row
    for (int j = 1; j < n; j++)
        dp[0][j] = dp[0][j-1] + grid[0][j];

    // First column
    for (int i = 1; i < m; i++)
        dp[i][0] = dp[i-1][0] + grid[i][0];

    // Rest of grid
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = grid[i][j] +
                min(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m-1][n-1];
}
```

4 Subsequence Problems

Longest Palindromic Subsequence

Problem

Find length of longest palindromic subsequence in string.

Key Insight

Key Insight: LPS of string s equals LCS of s and reverse of s.

Solution

```
int longestPalindromeSubseq(string s) {
    int n = s.length();
    vector<vector<int>> dp(n, vector<int>(n,
        0));

    // Every single char is palindrome
    for (int i = 0; i < n; i++)
        dp[i][i] = 1;

    // Fill by length
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j]) {
                dp[i][j] = 2 + (len > 2 ?
                    dp[i+1][j-1] : 0);
            } else {
                dp[i][j] = max(dp[i+1][j],
                    dp[i][j-1]);
            }
        }
    }
    return dp[0][n-1];
}
```

Longest Palindromic Substring

Problem

Find longest contiguous palindromic substring.

Solution

```
string longestPalindrome(string s) {
    int n = s.length();
    if (n == 0) return "";

    vector<vector<bool>> dp(n,
        vector<bool>(n, false));
    int start = 0, maxLen = 1;

    // Single characters
    for (int i = 0; i < n; i++)
        dp[i][i] = true;

    // Two characters
    for (int i = 0; i < n - 1; i++) {
        if (s[i] == s[i+1]) {
            dp[i][i+1] = true;
            start = i;
            maxLen = 2;
        }
    }

    // Longer substrings
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j] && dp[i+1][j-1]) {
                dp[i][j] = true;
                start = i;
                maxLen = len;
            }
        }
    }
    return s.substr(start, maxLen);
}
```

5 Partition Problems

Partition Equal Subset Sum

Problem

Can array be partitioned into two subsets with equal sum?

Key Insight

Reduction: If total sum is odd, return false. Otherwise, find if subset with sum = total/2 exists (subset sum problem).

Solution

```cpp
bool canPartition(vector<int>& nums) {
    int sum = accumulate(nums.begin(),
        nums.end(), 0);
    if (sum % 2 != 0) return false;

    int target = sum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        for (int j = target; j >= num; j--) {
            dp[j] = dp[j] || dp[j - num];
        }
    }
    return dp[target];
}
```

Target Sum

Problem

Assign $+$ or $-$ to each element to reach target sum. Count ways.

Key Insight

Let P = sum of positive numbers, N = sum of negatives.
$P - N$ = target and $P + N$ = sum
Therefore: $P = (\text{target} + \text{sum})/2$
Problem reduces to: count subsets with sum P.

Solution

```cpp
int findTargetSumWays(vector<int>& nums,
    int target) {
    int sum = accumulate(nums.begin(),
        nums.end(), 0);
    if ((target + sum) % 2 != 0 ||
        abs(target) > sum)
        return 0;

    int P = (target + sum) / 2;
    vector<int> dp(P + 1, 0);
    dp[0] = 1;

    for (int num : nums) {
        for (int j = P; j >= num; j--) {
            dp[j] += dp[j - num];
        }
    }
    return dp[P];
}
```