✅ Dynamic Programming

✅ Dynamic Programming – Coin Change Problem

📌 Problem Statement:

You are given an array of coins `coins[]` representing different denominations and an integer `amount` representing a total amount of money. Return the minimum number of coins required to make up that amount. If it’s not possible, return `-1`.

📥 Example:

`Input: coins = [1, 2, 5], amount = 11 Output: 3 Explanation: 5 + 5 + 1 = 11`

🔁 Brute Force Approach (Recursive)

- **Time Complexity**: Exponential — O(S^n), where S is the amount and n is the number of coins
- **Space Complexity**: O(S) due to recursive stack

```java
public class CoinChangeBrute {
    public static int coinChange(int[] coins, int amount) {
        if (amount == 0) return 0;
        if (amount < 0) return -1;

        int minCoins = Integer.MAX_VALUE;
        for (int coin : coins) {
            int res = coinChange(coins, amount - coin);
            if (res >= 0 && res < minCoins) {
                minCoins = res + 1;
            }
        }
        return (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
    }
}
```

💡 Optimal Approach (Top-Down with Memoization)

- **Time Complexity**: O(amount * number of coins)
- **Space Complexity**: O(amount) due to memoization array

```java
import java.util.HashMap;

public class CoinChangeMemo {
    public static int coinChange(int[] coins, int amount) {
        return helper(coins, amount, new HashMap<>());
    }

    private static int helper(int[] coins, int amount, HashMap<Integer, Integer>
memo) {
        if (amount == 0) return 0;
        if (amount < 0) return -1;
        if (memo.containsKey(amount)) return memo.get(amount);

        int minCoins = Integer.MAX_VALUE;
        for (int coin : coins) {
            int res = helper(coins, amount - coin, memo);
            if (res >= 0 && res < minCoins) {
                minCoins = res + 1;
            }
        }

        int result = (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
        memo.put(amount, result);
        return result;
    }
}
```

🚀 Most Optimal Approach (Bottom-Up Dynamic Programming)

- **Time Complexity**: O(amount × number of coins)
- **Space Complexity**: O(amount)

```java
import java.util.Arrays;

public class CoinChangeDP {
    public static int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1); // use amount + 1 as infinity
        dp[0] = 0;

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

✅ Dynamic Programming – Candies

📌 Problem Statement:

There are **N** children standing in a line. Each child is assigned a **rating** value. You are to distribute **candies** to these children subject to the following conditions:

1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.

What is the **minimum** number of candies you must give?

📥 Example:

java

CopyEdit

`Input: ratings = [1, 0, 2] Output: 5 Explanation: [2,1,2] candies => total = 5`

🔁 Brute Force Approach

- **Time Complexity**: O(N^2)
- **Space Complexity**: O(N)

This approach keeps updating the array until all constraints are satisfied.

```
import java.util.Arrays;

public class CandiesBrute {
    public static int minCandies(int[] ratings) {
        int n = ratings.length;
        int[] candies = new int[n];
        Arrays.fill(candies, 1);

        boolean changed = true;
        while (changed) {
            changed = false;
            for (int i = 0; i < n; i++) {
                if (i > 0 && ratings[i] > ratings[i - 1] && candies[i] <= candies[i
- 1]) {
```

```java
                    candies[i] = candies[i - 1] + 1;
                    changed = true;
                }
                if (i < n - 1 && ratings[i] > ratings[i + 1] && candies[i] <=
candies[i + 1]) {
                    candies[i] = candies[i + 1] + 1;
                    changed = true;
                }
            }
        }

        int total = 0;
        for (int c : candies) total += c;
        return total;
    }
}
```

🚀 Optimal Approach (Two-Pass Dynamic Programming)

- **Time Complexity**: O(N)
- **Space Complexity**: O(N)

```java
import java.util.Arrays;

public class CandiesDP {
    public static int minCandies(int[] ratings) {
        int n = ratings.length;
        int[] candies = new int[n];
        Arrays.fill(candies, 1);

        // Left to Right Pass
        for (int i = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1]) {
                candies[i] = candies[i - 1] + 1;
            }
        }

        // Right to Left Pass
        for (int i = n - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candies[i] = Math.max(candies[i], candies[i + 1] + 1);
            }
        }

        int total = 0;
        for (int c : candies) total += c;
        return total;
    }
}
```

✅ Output

java

CopyEdit

`Input: [1, 0, 2]Output: 5Input: [1, 2, 2]Output: 4`

✅ Dynamic Programming – Equal

📌 Problem Statement:

You have a list of integers representing the number of chocolates each colleague has. You can perform the following operation any number of times: Choose **one colleague** and **give 1, 2, or 5** chocolates to **every other colleague**.

What is the **minimum number of operations** required so that everyone has the **same number** of chocolates?

📥 Example:

java

CopyEdit

`Input: [2, 2, 3, 7] Output: 2 Explanation: Reduce to [2,2,2,2] in 2 operations.`

🔁 Brute Force Approach

- **Idea**: Try to equalize all elements to a baseline (min, min - 1, ..., min - 4) and check which gives the least operations.
- **Time Complexity**: O(N * 5)
- **Space Complexity**: O(1)

java

CopyEdit

```
import java.util.List;

public class EqualBrute {
    public static int equal(List<Integer> arr) {
        int min = Integer.MAX_VALUE;
        for (int a : arr) min = Math.min(min, a);

        int res = Integer.MAX_VALUE;
        for (int base = 0; base <= 4; base++) {
            int operations = 0;
            for (int a : arr) {
                int diff = a - (min - base);
                operations += diff / 5 + (diff % 5) / 2 + (diff % 5) % 2;
            }
            res = Math.min(res, operations);
        }

        return res;
    }
}
```

🚀 Most Optimal Approach (Math + Greedy)

- **Time Complexity**: O(N)
- **Space Complexity**: O(1)

This is the same approach as brute force above but focuses only on trying `min, min-1, ..., min-4` which guarantees optimal.

```
import java.util.List;

public class EqualOptimal {
    public static int equal(List<Integer> arr) {
        int min = Integer.MAX_VALUE;
        for (int a : arr) {
            min = Math.min(min, a);
        }

        int minOps = Integer.MAX_VALUE;
        for (int base = 0; base <= 4; base++) {
            int ops = 0;
            for (int a : arr) {
                int diff = a - (min - base);
                ops += diff / 5;
                diff %= 5;
                ops += diff / 2;
                diff %= 2;
                ops += diff;
            }
            minOps = Math.min(minOps, ops);
        }

        return minOps;
    }
}
```

✅ Output

java

CopyEdit

`Input: [2, 2, 3, 7]Output: 2Input: [10, 7, 12]Output: 3`

✅ Dynamic Programming – Kingdom Division

📌 Problem Statement:

A kingdom has **n** cities connected by **n-1** roads such that there's one path between any two cities (i.e., it's a tree). The king wants to divide the kingdom among **two sons**, such that:

1. Every city belongs to exactly one of the two kingdoms.
2. No two adjacent cities belong to the same kingdom.
3. Each kingdom must have **at least one city**.

Count the number of ways to divide the kingdom modulo 109+710^9 + 7109+7.

📥 Example:

java

CopyEdit

`Input: n = 3, edges = [[1, 2], [1, 3]]Output: 2`

🧠 Intuition:

We use **DP on Trees**. For each node, we define:

- `dp[u][0]`: Ways to divide subtree rooted at `u` where `u` and its parent are **not in the same group**.
- `dp[u][1]`: Ways to divide subtree rooted at `u` where `u` and its parent are **in the same group**.

🚀 Optimal Solution (DFS + DP on Tree)

- **Time Complexity**: O(N)
- **Space Complexity**: O(N)

```
import java.util.*;

public class KingdomDivision {
    static final int MOD = 1_000_000_007;

    static long[][] dp;
    static List<List<Integer>> tree;

    public static int kingdomDivision(int n, int[][] edges) {
        dp = new long[n + 1][2];
        tree = new ArrayList<>();
        for (int i = 0; i <= n; i++) tree.add(new ArrayList<>());

        for (int[] edge : edges) {
            int u = edge[0], v = edge[1];
            tree.get(u).add(v);
            tree.get(v).add(u);
        }

        dfs(1, 0);
        return (int) dp[1][0];
    }

    private static void dfs(int u, int parent) {
        dp[u][0] = 1; // diff group from parent
        dp[u][1] = 1; // same group as parent

        for (int v : tree.get(u)) {
            if (v == parent) continue;
            dfs(v, u);

            long same = dp[v][1];
            long diff = dp[v][0];

            dp[u][1] = dp[u][1] * (same + diff) % MOD;
            dp[u][0] = dp[u][0] * diff % MOD;
        }
    }
}
```

✅ Output

java

CopyEdit

`Input: n = 3, edges = [[1,2],[1,3]]Output: 2`

✅ Dynamic Programming – Common Child

📌 Problem Statement:

Given two strings, find the length of their **longest common subsequence** (LCS). This LCS represents the "common child".

📥 Example:

java

CopyEdit

`Input: s1 = "HARRY", s2 = "SALLY" Output: 2 Explanation: The common child is "AY"`

🔁 Brute Force (Not practical for long strings)

- Try all subsequences of both strings and compare.
- **Time Complexity**: O(2^n * 2^m)
- **Not feasible** for strings > 10 characters.

🚀 Optimal Solution (DP Table)

- Use 2D DP table `dp[i][j]` = LCS length of first `i` chars of `s1` and first `j` chars of `s2`.
- **Time Complexity**: O(n * m)
- **Space Complexity**: O(n * m)

```java
public class CommonChild {
    public static int commonChild(String s1, String s2) {
        int n = s1.length(), m = s2.length();
        int[][] dp = new int[n + 1][m + 1];

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

✅ Output

java

CopyEdit

`Input: "ABCD", "ABDC"Output: 3Input: "HARRY", "SALLY"Output: 2`

✅ Dynamic Programming – Down to Zero II

📌 Problem Statement:

You are given an integer `n`. You can perform the following operations:

1. Replace `n` with any factor `f` such that `f < n` and `n % f == 0`.
2. Subtract 1 from `n`.

The goal is to reduce `n` to 0 using **minimum steps**.

📥 Example:

java

CopyEdit

`Input: 10 Output: 4 Explanation: 10 → 5 → 4 → 2 → 0`

🧠 Intuition:

- At each state `n`, we can:
  - Decrease `n` by 1.
  - Or jump to any factor `f` such that `f < n`.

We use **BFS** to find the minimum steps.

🚀 Optimal Solution (BFS)

- **Time Complexity**: O(n√n)
- **Space Complexity**: O(n)

```java
import java.util.*;

public class DownToZero {
    public static int downToZero(int n) {
        Queue<Integer> queue = new LinkedList<>();
        boolean[] visited = new boolean[n + 1];
        int[] dist = new int[n + 1];

        queue.offer(n);
        visited[n] = true;
        dist[n] = 0;

        while (!queue.isEmpty()) {
            int current = queue.poll();
            if (current == 0) return dist[0];

            // Option 1: subtract 1
            if (!visited[current - 1]) {
                queue.offer(current - 1);
                visited[current - 1] = true;
                dist[current - 1] = dist[current] + 1;
            }

            // Option 2: replace with a factor
            for (int i = 2; i * i <= current; i++) {
                if (current % i == 0) {
                    int f1 = i;
                    int f2 = current / i;
                    if (f1 < current && !visited[f1]) {
                        queue.offer(f1);
                        visited[f1] = true;
                        dist[f1] = dist[current] + 1;
                    }
                    if (f2 < current && !visited[f2]) {
                        queue.offer(f2);
                        visited[f2] = true;
                        dist[f2] = dist[current] + 1;
                    }
                }
            }
        }

        return -1; // Should never reach
    }
}
```

✅ Output

java

CopyEdit

`Input: 10Output: 4Input: 1Output: 1Input: 3Output: 3`