

# “WEEK 3: The Curse of the Shadow Orb”

Code-

```
#include <iostream>
#include <queue>
#include <tuple>
#include <set>
using namespace std;

// Function to check if we can reduce power to 1 within given time and split limits
string canBreakCurse(int P, int T, int S) {
    queue<tuple<int, int, int>> q; // Stores (power, time left, splits left)
    set<tuple<int, int, int>> visited; // To avoid revisiting same state

    q.push({P, T, S});
    visited.insert({P, T, S});

    while (!q.empty()) {
        auto [power, time, splits] = q.front();
        q.pop();

        // Base condition: Curse broken
        if (power == 1) return "Yes";

        // No time left to perform any action
        if (time == 0) continue;

        // Option 1: Split magic (only if power is even and splits are left)
        if (power % 2 == 0 && splits > 0) {
            auto next = make_tuple(power / 2, time - 1, splits - 1);
            if (visited.find(next) == visited.end()) {
```

```

        visited.insert(next);
        q.push(next);
    }

}

// Option 2: Absorb shadows (reduce power by 1)

if (power > 1) {
    auto next = make_tuple(power - 1, time - 1, splits);
    if (visited.find(next) == visited.end()) {
        visited.insert(next);
        q.push(next);
    }
}

}

// If all options tried and power couldn't become 1

return "No";
}

int main() {
    int P, T, S; // Initial power, time units, and split chances
    cin >> P >> T >> S;
    cout << canBreakCurse(P, T, S) << endl;
    return 0;
}

```

## "Week 3: Mars Rover Energy Optimization"

Code-

```

#include <iostream>
#include <vector>
#include <algorithm>

```

```

#include <cmath>
#include <climits> // For INT_MIN

using namespace std;

// Function to find the closest sum of any three elements to the target
int closestThreeSum(int N, int target, vector<int>& arr) {
    sort(arr.begin(), arr.end()); // Sort the array for two-pointer approach
    int closestSum = INT_MIN; // Start with a very small value

    for (int i = 0; i < N - 2; i++) {
        int left = i + 1, right = N - 1;

        while (left < right) {
            int currentSum = arr[i] + arr[left] + arr[right];

            // If exact match, return it directly
            if (currentSum == target) {
                return currentSum;
            }

            // Update closest sum if this one is better
            if (abs(target - currentSum) < abs(target - closestSum)) {
                closestSum = currentSum;
            }

            // If both are equally close, pick the larger sum
            else if (abs(target - currentSum) == abs(target - closestSum)) {
                closestSum = max(closestSum, currentSum);
            }
        }
    }

    // Move pointers accordingly
}

```

```

        if (currentSum < target) {
            left++;
        } else {
            right--;
        }
    }

    return closestSum; // Return the best found sum
}

int main() {
    int N, target;
    cin >> N >> target;

    vector<int> arr(N);
    for (int i = 0; i < N; i++) {
        cin >> arr[i]; // Input the array
    }

    cout << closestThreeSum(N, target, arr) << endl;
    return 0;
}

```

## "Week 3: Magic Potion Maker"

Code-

```

#include <iostream>
#include <vector>
#include <algorithm>

```

```

using namespace std;

// Recursive backtracking function to find combinations
void backtrack(vector<int>& candidates, int target, int start, vector<int>& path, vector<vector<int>>& result) {
    if (target == 0) {
        result.push_back(path); // Found a valid combination
        return;
    }
    if (target < 0) return; // Invalid path

    for (int i = start; i < candidates.size(); ++i) {
        path.push_back(candidates[i]); // Choose current candidate
        backtrack(candidates, target - candidates[i], i, path, result); // Recurse with updated target
        path.pop_back(); // Backtrack to try next option
    }
}

// Main function to initiate combination sum logic
vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
    vector<vector<int>> result;
    vector<int> path;
    sort(candidates.begin(), candidates.end()); // Optional sort for optimization
    backtrack(candidates, target, 0, path, result);
    return result;
}

int main() {
    int n, target;
    cin >> n >> target; // Input size and target

    vector<int> candidates(n);
}

```

```

for (int i = 0; i < n; ++i) {
    cin >> candidates[i]; // Input array elements
}

vector<vector<int>> res = combinationSum(candidates, target);

if (res.empty()) {
    cout << "[]\n"; // No combinations found
} else {
    for (const auto& combo : res) {
        for (int num : combo) {
            cout << num << " "; // Print each valid combination
        }
        cout << "\n";
    }
}

return 0;
}

```

## “WEEK 3: Mantra math mission”

Code-

```

#include <iostream>
using namespace std;

const int MOD = 1e9 + 7; // Modulo value to avoid overflow

// Function to calculate (start * (start+1) * ... * (start+count-1)) % MOD
long long factorialMod(int start, int count) {
    long long result = 1;
    for (int i = start; i < start + count; i++) {
        result = (result * i) % MOD;
    }
}

```

```

    }

    return result;
}

// Function to calculate total energy as per the pattern

long long totalEnergy(int n) {

    long long total = 0;
    int start = 1;

    for (int i = 1; i <= n; i++) {
        long long term = factorialMod(start, i); // Multiply i consecutive numbers starting from 'start'
        total = (total + term) % MOD; // Add to total energy with modulo
        start += i; // Move start for next term
    }

    return total;
}

int main() {
    int n;
    cin >> n; // Input number of terms
    cout << totalEnergy(n) << endl; // Output the final energy
    return 0;
}

```

## “WEEK 3:Rahul and the Risky Rooftop”

Code-

```

#include <iostream>
#include <vector>

using namespace std;

```

```

// Function to count number of ways to reach step 'n'

// avoiding forbidden step 'w' and treating 's' specially

int countWays(int n, int w, int s, vector<int>& dp) {

    if (n == 0) return 1; // Base case: one way to stay at ground

    if (n < 0 || n == w) return 0; // Invalid or forbidden step

    if (dp[n] != -1) return dp[n]; // Return cached result if already computed

    int ways;

    if (n == s) {

        // Special step: can only come from (n - 1)

        ways = countWays(n - 1, w, s, dp);

    } else {

        // Normal step: can come from (n-1), (n-2), or (n-3)

        ways = countWays(n - 1, w, s, dp)

            + countWays(n - 2, w, s, dp)

            + countWays(n - 3, w, s, dp);

    }

    dp[n] = ways; // Store result in dp array

    return ways;

}

int main() {

    int N, W, S;

    cin >> N >> W >> S; // N = target step, W = forbidden, S = special step

    vector<int> dp(N + 1, -1); // Initialize dp with -1 (uncomputed)

    cout << countWays(N, W, S, dp) << endl; // Output the result

    return 0;
}

```

