Advanced_Linear_DP.cpp

```
/**
 *    Author: devesh95
 *
 *    Topic: Advanced Linear Dynamic Programming (1D DP) Examples
 *
 *    Description:
 *    This file provides a large collection of advanced 1D DP problems.
 *    The examples included here are different from the basic set and
 *    cover a variety of real-world and competitive programming problems.
 *
 *    Problems Covered:
 *      1. House Robber Problem:
 *           - Given an array representing money in houses, choose houses
 *             to rob such that adjacent houses are not robbed.
 *
 *      2. Delete and Earn:
 *           - A variant of the House Robber problem where earning a
 *             particular value removes its neighbors.
 *
 *      3. Minimum Jumps to Reach End:
 *           - Given an array where each element represents the maximum
 *             jump length from that position, find the minimum number of
 *             jumps to reach the end.
 *
 *      4. Decode Ways:
 *           - Given a digit-only string where 'A' = 1, 'B' = 2, ..., 'Z' = 26,
 *             determine the number of ways to decode it.
 *
 *      5. Dice Throw Problem:
 *           - Count the number of ways to get a given sum by throwing dice
 *             a certain number of times.
 *
 *      6. Frog Jump (Minimum Cost Path):
 *           - A frog jumps from stone 0 to stone n-1 with a cost to jump.
 *             Calculate the minimum total cost to reach the last stone.
 *
 *      7. Longest Wiggle Subsequence:
 *           - Determine the length of the longest subsequence where the
 *             differences between successive elements strictly alternate.
 *
 *      8. Maximum Product Subarray:
 *           - Find the contiguous subarray within a one-dimensional array
 *             (containing at least one number) which has the largest product.
 *
 *    Each function is self-contained with input, processing, and output.
 *    Comments within each function explain the approach and state transitions.
 *
 *    Compile with:
 *         g++ -std=c++17 -O2 -Wall Linear_DP_Advanced.cpp -o advanced_dp
 *
 *    Run with:
```

```
 *         ./advanced_dp
 */

#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pb push_back
#define F first
#define S second

////////////////////////////////////////////////////////////////
// 1. House Robber Problem
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of non-negative integers representing the amount of money of each house,
     determine the maximum amount you can rob tonight without alerting the police (i.e., you cannot
     rob adjacent houses).

   DP Approach:
     Let dp[i] be the maximum amount that can be robbed from houses 0 to i.
     - Base cases: dp[0] = nums[0]
                   dp[1] = max(nums[0], nums[1])
     - Transition: dp[i] = max(dp[i-1], dp[i-2] + nums[i])
*/
void solve_house_robber() {
    cout << "\n----- House Robber Problem -----\n";
    int n;
    cout << "Enter number of houses: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the amount of money in each house:\n";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }
    if (n == 0) {
        cout << "No houses to rob.\n";
        return;
    }
    if (n == 1) {
        cout << "Maximum amount robbed: " << nums[0] << "\n";
        return;
    }
    vector<int> dp(n, 0);
    dp[0] = nums[0];
    dp[1] = max(nums[0], nums[1]);
    for (int i = 2; i < n; i++) {
        dp[i] = max(dp[i-1], dp[i-2] + nums[i]);
    }
    cout << "Maximum amount robbed: " << dp[n-1] << "\n";
}

```

```cpp
////////////////////////////////////////////////////////////
// 2. Delete and Earn Problem
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of integers, you can earn points by deleting a number. However, when you delete a number,
     all elements equal to number-1 and number+1 are also removed. Find the maximum points you can earn.

   DP Approach:
     This problem can be transformed into a variation of the House Robber problem.
     - First, aggregate total points for each unique value.
     - Then, use DP where dp[i] = max(dp[i-1], dp[i-2] + points[i]).
*/
void solve_delete_and_earn() {
    cout << "\n----- Delete and Earn Problem -----\n";
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }
    int maxVal = *max_element(nums.begin(), nums.end());
    vector<int> points(maxVal + 1, 0);
    for (int num : nums) {
        points[num] += num;
    }
    vector<int> dp(maxVal + 1, 0);
    dp[0] = points[0];
    dp[1] = max(points[0], points[1]);
    for (int i = 2; i <= maxVal; i++) {
        dp[i] = max(dp[i-1], dp[i-2] + points[i]);
    }
    cout << "Maximum points earned: " << dp[maxVal] << "\n";
}

////////////////////////////////////////////////////////////
// 3. Minimum Jumps to Reach End
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array where each element represents the maximum jump length from that position,
     determine the minimum number of jumps required to reach the last index. If it is not possible, return -1.

   DP Approach:
     Let dp[i] be the minimum jumps needed to reach index i.
     Initialize dp[0] = 0, and for other indices dp[i] = INF.
     For each index i, for every index j > i where j <= i + arr[i], update dp[j] = min(dp[j], dp[i] + 1).
```

```
*/
void solve_minimum_jumps() {
    cout << "\n----- Minimum Jumps to Reach End -----\n";
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the jump lengths at each position:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    const int INF = 1e9;
    vector<int> dp(n, INF);
    dp[0] = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] == INF) continue;
        for (int j = i+1; j < n && j <= i + arr[i]; j++) {
            dp[j] = min(dp[j], dp[i] + 1);
        }
    }
    if(dp[n-1] == INF)
        cout << "It is not possible to reach the end.\n";
    else
        cout << "Minimum jumps required: " << dp[n-1] << "\n";
}

////////////////////////////////////////////////////////////////
// 4. Decode Ways
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     A message containing letters from A-Z can be encoded into numbers using 'A' -> 1, 'B' -> 2, ..., 'Z' -> 26.
     Given a non-empty string containing only digits, determine the total number of ways to decode it.

   DP Approach:
     Let dp[i] represent the number of ways to decode the substring of length i.
     - Base: dp[0] = 1 (empty string)
     - For each position i, check:
          a) if s[i-1] != '0', add dp[i-1] to dp[i]
          b) if the two-digit number formed by s[i-2] and s[i-1] is between 10 and 26, add dp[i-2]
*/
void solve_decode_ways() {
    cout << "\n----- Decode Ways -----\n";
    string s;
    cout << "Enter the digit string: ";
    cin >> s;
    int n = s.size();
    if(n == 0) {
        cout << "Empty string.\n";
        return;
    }
```

```cpp
    vector<int> dp(n+1, 0);
    dp[0] = 1;
    // If the first digit is '0', no valid decoding exists.
    dp[1] = (s[0] != '0') ? 1 : 0;
    for (int i = 2; i <= n; i++) {
        // Single digit decode (if not '0')
        if (s[i-1] != '0')
            dp[i] += dp[i-1];
        // Two digit decode
        int twoDigit = (s[i-2]-'0') * 10 + (s[i-1]-'0');
        if (twoDigit >= 10 && twoDigit <= 26)
            dp[i] += dp[i-2];
    }
    cout << "Total number of ways to decode: " << dp[n] << "\n";
}

////////////////////////////////////////////////////////////
// 5. Dice Throw Problem
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given N dice each with K faces numbered 1 to K, and a target sum S,
     determine the number of ways to achieve the sum S.

   DP Approach:
     Let dp[i] be the number of ways to obtain sum i.
     Base: dp[0] = 1 (one way to obtain sum 0, by not throwing any dice)
     For each dice throw, update dp for sums from S down to 0.
*/
void solve_dice_throw() {
    cout << "\n----- Dice Throw Problem -----\n";
    int N, K, S;
    cout << "Enter number of dice (N): ";
    cin >> N;
    cout << "Enter number of faces on each die (K): ";
    cin >> K;
    cout << "Enter target sum (S): ";
    cin >> S;
    vector<int> dp(S+1, 0);
    dp[0] = 1;
    for (int dice = 1; dice <= N; dice++) {
        vector<int> temp(S+1, 0);
        for (int s = 0; s <= S; s++) {
            if (dp[s] > 0) {
                for (int face = 1; face <= K; face++) {
                    if (s + face <= S) {
                        temp[s + face] += dp[s];
                    }
                }
            }
        }
        dp = temp;
    }
    cout << "Number of ways to achieve sum " << S << " is: " << dp[S] << "\n";
```

```cpp
}

////////////////////////////////////////////////////////////
// 6. Frog Jump (Minimum Cost Path)
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     A frog is trying to cross a river by jumping on stones.
     Given N stones with heights, the frog can jump from stone i to stone j with cost = |height[j] - height[i]|.
     Find the minimum cost to reach the last stone (stone N-1).

   DP Approach:
     Let dp[i] be the minimum cost to reach stone i.
     - Base: dp[0] = 0
     - Transition: for each stone i, dp[i] = min(dp[i], dp[j] + abs(height[i] - height[j]))
       where j ranges over all stones from which stone i can be reached (typically i-1 and/or i-2).
*/
void solve_frog_jump() {
    cout << "\n----- Frog Jump (Minimum Cost Path) -----\n";
    int n;
    cout << "Enter the number of stones: ";
    cin >> n;
    vector<int> height(n);
    cout << "Enter the heights of the stones:\n";
    for (int i = 0; i < n; i++) {
        cin >> height[i];
    }
    vector<int> dp(n, 1e9);
    dp[0] = 0;
    // Assuming the frog can only jump to the next stone or skip one stone.
    for (int i = 1; i < n; i++) {
        dp[i] = min(dp[i], dp[i-1] + abs(height[i] - height[i-1]));
        if (i > 1)
            dp[i] = min(dp[i], dp[i-2] + abs(height[i] - height[i-2]));
    }
    cout << "Minimum cost to reach the last stone: " << dp[n-1] << "\n";
}

////////////////////////////////////////////////////////////
// 7. Longest Wiggle Subsequence
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     A wiggle sequence is one where the differences between successive numbers strictly alternate
     between positive and negative. Given an array, find the length of the longest wiggle subsequence.

   DP Approach:
     Maintain two arrays (or two variables) up and down:
       - up[i] is the length of the longest wiggle subsequence ending at i with a positive difference.
```

```
       - down[i] is the length ending at i with a negative difference.
     Update these as you iterate through the array.
*/
void solve_longest_wiggle() {
    cout << "\n----- Longest Wiggle Subsequence -----\n";
    int n;
    cout << "Enter the number of elements in the sequence: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the sequence elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }
    if (n == 0) {
        cout << "Sequence is empty.\n";
        return;
    }
    int up = 1, down = 1;
    for (int i = 1; i < n; i++) {
        if (nums[i] > nums[i-1]) {
            up = down + 1;
        } else if (nums[i] < nums[i-1]) {
            down = up + 1;
        }
    }
    cout << "Length of the longest wiggle subsequence: " << max(up, down) << "\n";
}

////////////////////////////////////////////////////////////
// 8. Maximum Product Subarray
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of integers, find the contiguous subarray within an array (containing at least one number)
     which has the largest product.

   DP Approach:
     The presence of negative numbers requires tracking both maximum and minimum products up to index i.
     Let maxProd[i] be the maximum product ending at i, and minProd[i] be the minimum product ending at i.
     Transition:
         maxProd[i] = max(arr[i], arr[i] * maxProd[i-1], arr[i] * minProd[i-1])
         minProd[i] = min(arr[i], arr[i] * maxProd[i-1], arr[i] * minProd[i-1])
*/
void solve_maximum_product_subarray() {
    cout << "\n----- Maximum Product Subarray -----\n";
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements of the array:\n";
    for (int i = 0; i < n; i++) {
```

```
359         cin >> arr[i];
360     }
361     if (n == 0) {
362         cout << "Array is empty.\n";
363         return;
364     }
365     int maxProd = arr[0], minProd = arr[0], ans = arr[0];
366     for (int i = 1; i < n; i++) {
367         if (arr[i] < 0) swap(maxProd, minProd);
368         maxProd = max(arr[i], arr[i] * maxProd);
369         minProd = min(arr[i], arr[i] * minProd);
370         ans = max(ans, maxProd);
371     }
372     cout << "Maximum product of a subarray is: " << ans << "\n";
373 }
374 
375 ////////////////////////////////////////////////////////////
376 // Main function with an interactive menu for advanced DP problems
377 ////////////////////////////////////////////////////////////
378 int32_t main() {
379     ios_base::sync_with_stdio(false);
380     cin.tie(nullptr);
381     cout.tie(nullptr);
382 
383     cout << "================================================\n";
384     cout << "   Advanced Linear Dynamic Programming (1D DP)\n";
385     cout << "================================================\n";
386     cout << "Select a problem to solve:\n";
387     cout << "1. House Robber Problem\n";
388     cout << "2. Delete and Earn Problem\n";
389     cout << "3. Minimum Jumps to Reach End\n";
390     cout << "4. Decode Ways\n";
391     cout << "5. Dice Throw Problem\n";
392     cout << "6. Frog Jump (Minimum Cost Path)\n";
393     cout << "7. Longest Wiggle Subsequence\n";
394     cout << "8. Maximum Product Subarray\n";
395     cout << "9. Run All Examples\n";
396     cout << "Enter your choice: ";
397 
398     int choice;
399     cin >> choice;
400     cout << "\n";
401 
402     switch(choice) {
403         case 1:
404             solve_house_robber();
405             break;
406         case 2:
407             solve_delete_and_earn();
408             break;
409         case 3:
410             solve_minimum_jumps();
411             break;
412         case 4:
```

```
            solve_decode_ways();
            break;
        case 5:
            solve_dice_throw();
            break;
        case 6:
            solve_frog_jump();
            break;
        case 7:
            solve_longest_wiggle();
            break;
        case 8:
            solve_maximum_product_subarray();
            break;
        case 9:
            solve_house_robber();
            solve_delete_and_earn();
            solve_minimum_jumps();
            solve_decode_ways();
            solve_dice_throw();
            solve_frog_jump();
            solve_longest_wiggle();
            solve_maximum_product_subarray();
            break;
        default:
            cout << "Invalid choice. Exiting...\n";
    }

    return 0;
}

```