Linear_DP.cpp

```cpp
/**
 *    Author: devesh95
 *
 *    Topic: Linear Dynamic Programming (1D DP)
 *
 *    Description:
 *    This file contains a collection of example problems solved using
 *    linear dynamic programming. The idea behind 1D DP is to build an
 *    array (or vector) 'dp' where each entry dp[i] represents the solution
 *    for a subproblem involving the first i elements or a state 'i'.
 *
 *    Use Cases Covered:
 *      1. Fibonacci Sequence
 *      2. Climbing Stairs Problem
 *      3. Minimum Coin Change (Unbounded Knapsack)
 *      4. Maximum Subarray Sum (Kadane's Algorithm as DP)
 *      5. Longest Increasing Subsequence (LIS) - O(n^2) approach
 *      6. Rod Cutting Problem
 *
 *    Each function is self-contained with input, processing, and output.
 *    The comments within each function provide step-by-step explanations
 *    of the DP approach used.
 *
 *    Note: All problems use a 1D DP formulation where each state depends
 *          on one or more previous states.
 *
 *    Compile with:
 *         g++ -std=c++17 -O2 -Wall Linear_Dynamic_Programming.cpp -o ldp
 *
 *    Run with:
 *         ./ldp
 */

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

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

////////////////////////////////////////////////////////////
// 1. Fibonacci Sequence using DP
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given n, compute the nth Fibonacci number.
     Fibonacci numbers are defined as:
         F(0) = 0, F(1) = 1
         F(n) = F(n-1) + F(n-2) for n >= 2

```

```cpp
   DP Approach:
     We use a dp array where dp[i] stores F(i).
     Base cases: dp[0] = 0, dp[1] = 1.
     For each i from 2 to n, we compute dp[i] = dp[i-1] + dp[i-2].
*/
void solve_fibonacci() {
    cout << "\n----- Fibonacci Sequence using DP -----\n";
    int n;
    cout << "Enter n (0-indexed): ";
    cin >> n;

    if(n < 0) {
        cout << "Invalid input. n must be non-negative.\n";
        return;
    }
    // Create a DP array of size n+1
    vector<int> dp(n+1, 0);
    dp[0] = 0;
    if(n > 0)
        dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    cout << "Fibonacci number F(" << n << ") = " << dp[n] << "\n";
}

////////////////////////////////////////////////////////////
// 2. Climbing Stairs Problem using DP
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     You are climbing a staircase. It takes n steps to reach the top.
     Each time you can climb either 1 or 2 steps.
     In how many distinct ways can you climb to the top?

   DP Approach:
     Let dp[i] be the number of ways to reach step i.
     Base cases: dp[0] = 1 (one way to stand at the base), dp[1] = 1.
     For i >= 2: dp[i] = dp[i-1] + dp[i-2] because you can come
     to i from i-1 (1 step) or i-2 (2 steps).
*/
void solve_climbing_stairs() {
    cout << "\n----- Climbing Stairs using DP -----\n";
    int n;
    cout << "Enter the number of stairs: ";
    cin >> n;

    vector<int> dp(n+1, 0);
    dp[0] = 1; // 1 way to be at the ground level
    if(n >= 1)
        dp[1] = 1; // Only one step possible

```

```cpp
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    cout << "Total distinct ways to climb " << n << " stairs: " << dp[n] << "\n";
}

////////////////////////////////////////////////////////////
// 3. Minimum Coin Change (Unbounded Knapsack) using DP
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given coins of different denominations and a total amount,
     find the minimum number of coins that you need to make up that amount.
     If that amount cannot be made up by any combination of the coins, return -1.

   DP Approach:
     Let dp[i] be the minimum number of coins needed for amount i.
     Base case: dp[0] = 0.
     For each amount i from 1 to total amount:
         For each coin value c:
             if i-c >= 0, dp[i] = min(dp[i], dp[i-c] + 1)
*/
void solve_coin_change() {
    cout << "\n----- Minimum Coin Change using DP -----\n";
    int n;
    cout << "Enter the number of coin denominations: ";
    cin >> n;
    vector<int> coins(n);
    cout << "Enter the coin denominations: ";
    for (int i = 0; i < n; i++) {
        cin >> coins[i];
    }
    int amount;
    cout << "Enter the total amount: ";
    cin >> amount;

    const int INF = 1e9;
    vector<int> dp(amount + 1, INF);
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int coin : coins) {
            if (i - coin >= 0) {
                dp[i] = min(dp[i], dp[i-coin] + 1);
            }
        }
    }
    if(dp[amount] == INF)
        cout << "It is not possible to form the amount with given coins.\n";
    else
        cout << "Minimum coins required: " << dp[amount] << "\n";
}

```

```cpp
////////////////////////////////////////////////////////////
// 4. Maximum Subarray Sum (Kadane's Algorithm as DP)
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of integers, find the contiguous subarray (containing at least one number)
     which has the largest sum.

   DP Approach:
     Let dp[i] be the maximum subarray sum ending at index i.
     Then, dp[i] = max(a[i], dp[i-1] + a[i]).
     The answer is the maximum value in dp[].
*/
void solve_maximum_subarray() {
    cout << "\n----- Maximum Subarray Sum using DP (Kadane's Algorithm) -----\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++) {
        cin >> arr[i];
    }

    vector<int> dp(n);
    dp[0] = arr[0];
    int maxSum = dp[0];
    for (int i = 1; i < n; i++) {
        dp[i] = max(arr[i], dp[i-1] + arr[i]);
        maxSum = max(maxSum, dp[i]);
    }
    cout << "Maximum subarray sum is: " << maxSum << "\n";
}

////////////////////////////////////////////////////////////
// 5. Longest Increasing Subsequence (LIS) using DP (O(n^2))
////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of integers, find the length of the longest strictly increasing subsequence.

   DP Approach:
     Let dp[i] be the length of the longest increasing subsequence ending at i.
     For each i, iterate j from 0 to i-1. If arr[j] < arr[i], then update dp[i] = max(dp[i], dp[j] + 1).
     The answer is the maximum value in dp[].
*/
void solve_LIS() {
    cout << "\n----- Longest Increasing Subsequence (LIS) using DP -----\n";
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;
```

```cpp
    vector<int> arr(n);
    cout << "Enter the elements of the array:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    vector<int> dp(n, 1);
    int ans = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if(arr[j] < arr[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        ans = max(ans, dp[i]);
    }
    cout << "Length of Longest Increasing Subsequence is: " << ans << "\n";
}

////////////////////////////////////////////////////////////////
// 6. Rod Cutting Problem using DP
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a rod of length n inches and an array of prices that contains prices of all pieces of size 1 to n,
     determine the maximum revenue obtainable by cutting up the rod and selling the pieces.

   DP Approach:
     Let dp[i] be the maximum revenue obtainable for a rod of length i.
     For each length i from 1 to n:
         For each possible cut j from 1 to i:
             dp[i] = max(dp[i], price[j-1] + dp[i - j])
*/
void solve_rod_cutting() {
    cout << "\n----- Rod Cutting Problem using DP -----\n";
    int n;
    cout << "Enter the rod length: ";
    cin >> n;
    vector<int> price(n);
    cout << "Enter the prices for each rod length from 1 to " << n << ":\n";
    for (int i = 0; i < n; i++) {
        cin >> price[i];
    }

    vector<int> dp(n+1, 0);
    // dp[0] = 0 is already set by default.
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            dp[i] = max(dp[i], price[j-1] + dp[i-j]);
        }
    }
    cout << "Maximum revenue obtainable: " << dp[n] << "\n";
}
```

```cpp

////////////////////////////////////////////////////////////
// Main function with a menu to choose DP problems
////////////////////////////////////////////////////////////
int32_t main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    cout << "========================================\n";
    cout << "  Linear Dynamic Programming (1D DP) Notes\n";
    cout << "========================================\n";
    cout << "Select a problem to solve:\n";
    cout << "1. Fibonacci Sequence\n";
    cout << "2. Climbing Stairs\n";
    cout << "3. Minimum Coin Change\n";
    cout << "4. Maximum Subarray Sum\n";
    cout << "5. Longest Increasing Subsequence (LIS)\n";
    cout << "6. Rod Cutting Problem\n";
    cout << "7. Run All Examples\n";
    cout << "Enter your choice: ";

    int choice;
    cin >> choice;
    cout << "\n";

    switch(choice) {
        case 1:
            solve_fibonacci();
            break;
        case 2:
            solve_climbing_stairs();
            break;
        case 3:
            solve_coin_change();
            break;
        case 4:
            solve_maximum_subarray();
            break;
        case 5:
            solve_LIS();
            break;
        case 6:
            solve_rod_cutting();
            break;
        case 7:
            solve_fibonacci();
            solve_climbing_stairs();
            solve_coin_change();
            solve_maximum_subarray();
            solve_LIS();
            solve_rod_cutting();
            break;
        default:
```

```
318             cout << "Invalid choice. Exiting...\n";
319     }
320
321     return 0;
322 }
323
```