Lab 06: Implementing Dynamic Programming Algorithms

ICT 2202 :-ALGORITHM ANALYSIS AND DESIGN LAB

Submitted By

Group: 01

Name:	Class Roll:
Tahshin Islam Tuhin	1957
Nobin Chandra	1958
Md. Shaon Khan	1984

Submitted To

Professor Fahima Tabassum, PhD

Institute of Information Technology

Jahangirnagar University

Submission Date: 02/12/2025

Overview

Dynamic Programming (DP) is a technique for solving problems with overlapping subproblems and optimal substructure by storing intermediate results and reusing them to avoid redundant calculations. It is widely used in optimization and sequence-related problems.

1. **0/1 Knapsack:** Determines the maximum total value of items that can fit in a knapsack without exceeding its weight limit. DP stores solutions for smaller capacities and builds up to the full capacity. Time complexity is O(n*W), where *n* is the number of items and *W* is the knapsack capacity. Advantage: Finds exact optimal solution.

   Disadvantage: Requires extra memory and can be slow for very large capacities.

2. **Rod Cutting:** Maximizes profit from cutting a rod of length *n* into pieces with given prices. DP evaluates all possible cuts for smaller lengths and builds the solution for the full rod. Complexity is $O(n^2)$.

   Advantage: Efficiently handles all cut combinations.

   Disadvantage: Quadratic time can be costly for large *n*.

3. **Longest Common Subsequence (LCS):** Finds the longest sequence appearing in the same relative order in two strings. DP avoids recomputing sub-sequences by storing results for smaller prefixes. Complexity is O(m*n), where *m* and *n* are string lengths.

   Advantage: Handles partially matching sequences efficiently.

   Disadvantage: Requires a 2D table, which uses O(m*n) memory.

4. **Floyd-Warshall Algorithm:** Computes shortest distances between all pairs of vertices in a weighted graph. DP iteratively updates distances using intermediate vertices. Time complexity is **$O(V^3)$**, where *V* is the number of vertices.

   Advantage: Simple and computes all-pairs shortest paths.

   Disadvantage: Inefficient for very large graphs due to cubic complexity.

Problem 01: The Mars Rover Sample Protocol (0/1 Knapsack)

You are programming the AI for the latest Mars Rover, Curiosity II. The rover has drilled several rock samples, but a sudden dust storm is approaching. The drone needs to fly back to the base station immediately. The drone has a strict weight capacity limit, and each rock sample has a specific scientific value and weight. You cannot break the rocks (you must take the whole rock or leave it).

Maximize the total scientific value of the samples carried back without exceeding the drone's weight capacity.

Input:

 W: The maximum weight capacity of the drone (integer).

 n: The number of available rock samples.

 Two arrays:

o weights[]: An array where weights[i] is the weight of the i-th rock.

o values[]: An array where values[i] is the scientific value of the i-th rock.

Output:

 The maximum scientific value achievable.

Core Logic: This is a classic 0/1 Knapsack problem because you have a binary choice (take or leave) for each item, and you are constrained by a single resource (weight).

Algorithm: 0/1 Knapsack Using 1-D Dynamic Programming

Input:

- W: Maximum capacity of the knapsack
- n: Number of items
- weights[1…n]: Weight of each item
- values[1…n]: Value of each item

Output:

- Maximum value achievable without exceeding capacity W

Step-by-Step Algorithm

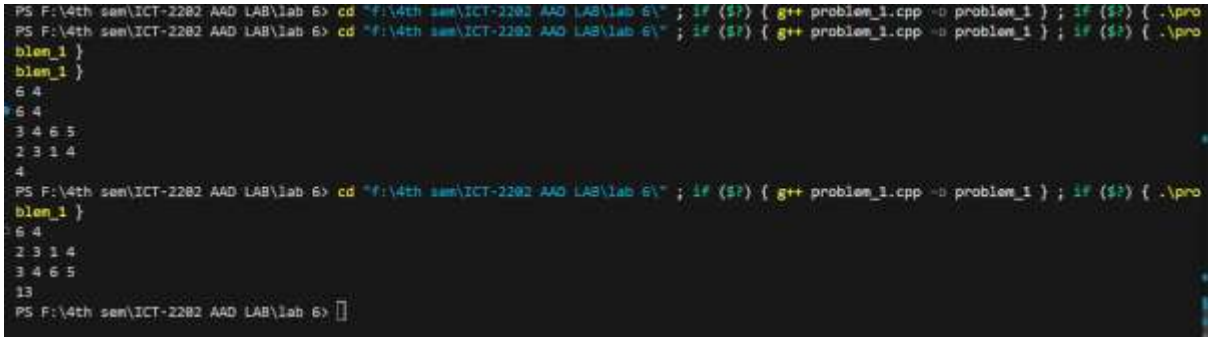
1. **Read input values**
    - Read W and n.
    - Read the weights[] array.
    - Read the values[] array.
2. **Initialize DP array**
    - Create a 1-dimensional array dp[0…W].
    - Set all values to 0.
    - dp[w] will store the maximum value achievable with capacity w.
3. **Process each item**
   For each item i = 1 to n:
    - Traverse capacities backward from W down to weights[i]:
        - Update:
        - dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
4. **Final answer**
    - The value dp[W] contains the maximum value that fits into the knapsack.
5. **Print result**
    - Output dp[W].

Code/ Solution:

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
void solve()
{
   int W, n;
   cin >> W >> n;
   vector<ll> weights(n), values(n);
   for (int i = 0; i < n; i++)
      cin >> weights[i];
   for (int i = 0; i < n; i++)
      cin >> values[i];
   vector<ll> dp(W + 1, 0);
   for (int i = 0; i < n; i++){
      for (int w = W; w >= weights[i]; w--)      {
         dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
```

```
        }   }
    cout << dp[W] << "\n";
}
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);
    solve();
    return 0;
}
```

OUTPUT:



Explanation

The 0/1 Knapsack problem finds the maximum value that can fit in a knapsack of limited capacity when each item can be chosen only once. The program uses a 1D dynamic programming array where dp[w] represents the maximum value achievable for weight w. Iterating over items, the array is updated in reverse to ensure an item is counted only once. This method efficiently computes the optimal solution while saving memory compared to a 2D array. The final DP value gives the maximum value that fits in the knapsack.
Time Complexity: O(n × W), where n is the number of items and W is the knapsack capacity
Space Complexity: O(W)

Problem 02: The Rare Timber Merchant (Rod Cutting)

You manage a high-end lumber yard that sells rare Ebony wood. You receive a long log of raw timber of length n inches. However, the market price for wood is not linear; a 4-inch piece might be worth more than twice the value of a 2-inch piece due to its utility in furniture making, while a 10-inch piece might be difficult to sell and worth less per inch.
Determine the best way to cut the log of length n into smaller pieces to maximize the total selling price.
Input:
 n: The total length of the raw timber log.
 prices[]: An array where prices[i] represents the market price of a piece of wood of length i.
Output:
 The maximum profit obtainable.
Core Logic: This is the Rod Cutting problem. The optimal solution involves checking maximum profit for every possible cut point and combining the results of the sub-problems (the left side of the cut and the optimized right side).

Algorithm: Rod Cutting (Dynamic Programming)

Input:

- n: Length of the rod
- prices[1…n]: Price of a rod piece of each length

Output:

- Maximum obtainable revenue from a rod of length n

Step-by-Step Algorithm

1. **Read input**
   - Read integer n.
   - Read the array prices[1…n], where prices[i] represents the price of a rod of length i.
2. **Initialize DP table**
   - Create an array dp[0…n].
   - Set dp[0] = 0 (value of rod of length 0 is 0).
3. **Compute the best revenue for each rod length**
   For each rod length len from 1 to n:
   - Set best = prices[len] (selling the rod without cutting).
   - For every possible first cut position cut from 1 to len - 1:
     - Compute revenue if the rod is cut into two parts:
     - prices[cut] + dp[len - cut]
     - Update:
     - best = max(best, prices[cut] + dp[len - cut])
   - Store the best revenue:
   - dp[len] = best
4. **Final answer**
   - The maximum revenue for a rod of length n is stored in dp[n].
5. **Print result**
   - Output dp[n].

Code/Solution:

```
#include <bits/stdc++.h>

using namespace std;

#define ll long long

void solve(){

   ll n;

   cin >> n;

   vector<ll> prices(n + 1);

   for (ll i = 1; i <= n; i++)

      cin >> prices[i];

   vector<ll> dp(n + 1, 0);

   for (ll len = 1; len <= n; len++)  {

      ll best = prices[len];

      for (int cut = 1; cut < len; cut++)   {
```

```
            best = max(best, prices[cut] + dp[len - cut]);
        }
        dp[len] = best;
    }
    cout << dp[n] << endl;
}
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);
    solve();
    return 0;
}
```

OUTPUT:

```
PS F:\4th sem\ICT-2202 AAD LAB\lab 6> cd "f:\4th sem\ICT-2202 AAD LAB\lab 6\" ; if ($?)
 { g++ problem_2.cpp -o problem_2 } ; if ($?) { .\problem_2 }
8
1 5 8 9 10 17 17 20
22
PS F:\4th sem\ICT-2202 AAD LAB\lab 6> cd "f:\4th sem\ICT-2202 AAD LAB\lab 6\" ; if ($?)
 { g++ problem_2.cpp -o problem_2 } ; if ($?) { .\problem_2 }
12
4 5 8 8 9 12 13 13 14 15 15 20
48
PS F:\4th sem\ICT-2202 AAD LAB\lab 6>
```

Explanation:

The rod cutting problem determines the maximum revenue obtainable by cutting a rod of length n into pieces, each with a specific price. The program uses dynamic programming: dp[len] stores the maximum revenue for a rod of length len. For each length, it evaluates all possible cuts and chooses the combination yielding the highest total price. This approach avoids recomputation of subproblems and guarantees the optimal solution. The result dp[n] represents the maximum revenue for the full rod.
Time Complexity: $O(n^2)$, as all possible cut positions are considered for each rod length
Space Complexity: O(n)Runtime varies based on pivot position and input order, even though theoretical complexity is O(n log n) for most cases.

Problem 03: The Plagiarism Detector (Longest Common Subsequence)

You are building a tool for a university to detect academic dishonesty. You have two text documents (represented as sequences of characters). A student might have copied a paragraph but deleted a few words or sentences in between to hide the plagiarism. The order of the remaining words, however, remains the same.
Find the length of the longest sequence of characters that appear in both documents in the same relative order, essentially measuring how much " content DNA & quot; the two documents share.

Input:
 String X of length m.
 String Y of length n.
Output:
 The length of the Longest Common Subsequence.
 (Optional challenge: Print the actual string).
Core Logic: This is LCS. Unlike "Longest Common Substring," the characters in a subsequence do not
need to be contiguous, they just need to maintain their relative index order (e.g., "ABC" is a subsequence of "A-1-B-2-C").

Algorithm: Longest Common Subsequence (LCS)

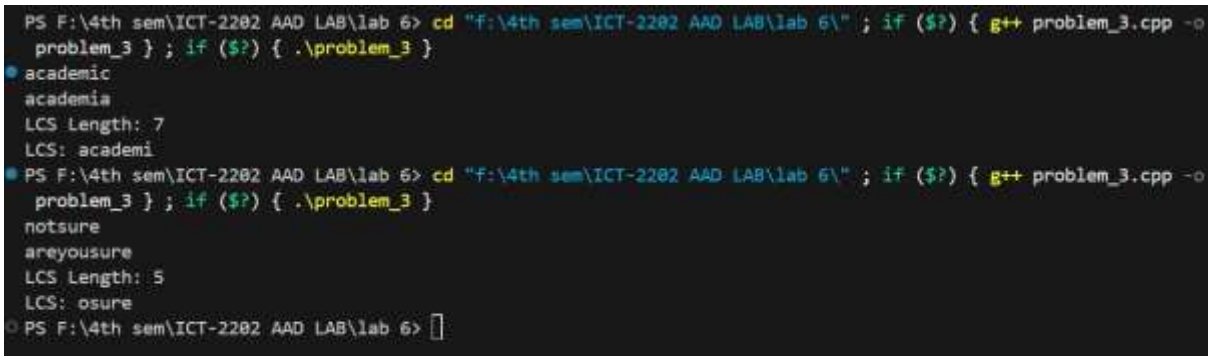
1. Create a DP table dp[m+1][n+1]
    - dp[i][j] = LCS length of X[0…i-1] & Y[0…j-1]
2. Initialize the first row and column with **0** (because LCS with an empty string = 0)
3. Fill the table:
    - If characters match: dp[i][j] = 1 + dp[i-1][j-1]
    - If they don't match: dp[i][j] = max(dp[i-1][j], dp[i][j-1])
4. The final answer = dp[m][n] (bottom-right cell)

Coed/Solution:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
   string X, Y;
   cin >> X >> Y;
   int m = X.size();
   int n = Y.size();
   vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
   for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
         if (X[i - 1] == Y[j - 1]) {
            dp[i][j] = 1 + dp[i - 1][j - 1];
         } else {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
         }    }    }
   cout << "LCS Length: " << dp[m][n] << "\n";
   string lcs = "";
   int i = m, j = n;
   while (i > 0 && j > 0) {
      if (X[i - 1] == Y[j - 1]) {
         lcs += X[i - 1];
         i--; j--;
```

```
    } else if (dp[i - 1][j] > dp[i][j - 1]) {
        i--;
    } else {
        j--;
    }   }
  reverse(lcs.begin(), lcs.end());
  cout << "LCS: " << lcs << "\n";
  return 0;
}
```

OUTPUT :



Explanation:

The LCS algorithm identifies the longest sequence of characters that appears in both input strings while preserving order, even if some characters are missing. This is useful for plagiarism detection or comparing DNA-like sequences in text. The program uses dynamic programming by creating a 2D table where dp[i][j] represents the LCS length of the first i characters of string X and the first j characters of string Y. If characters match, the value is incremented from the diagonal; otherwise, it takes the maximum of the left and top cells. The final value gives the LCS length.

Time Complexity: O(m × n), where m and n are string lengths

Space Complexity: O(m × n)

Problem 04: The Inter-City Logistics Network (Floyd-Warshall)

You are the CTO of a logistics company operating in a country with N cities. Due to road construction and traffic patterns, the travel cost (or time) between directly connected cities changes frequently. Your delivery drivers need to know the shortest path from any city to any other city at a moment's notice. Because queries happen frequently between random city pairs, you need a pre-computed table of all shortest paths.

Given a graph of cities and the weighted roads connecting them (directed or undirected), compute the shortest distance between every pair of vertices (u, v).

Input:

 V: The number of cities.

 A weighted adjacency matrix Graph[V][V], where Graph[i][j] is the weight of the edge from city i to city j. If no edge exists, the value is ∞ (infinity)

Output:

 A V × V matrix where Matrix[i][j] contains the shortest distance from city i to city j.

Algorithm: Floyd-Warshall

1. Input:

- Number of vertices V
- A V × V adjacency matrix dist, where
  - dist[i][j] = weight of edge from i to j
  - dist[i][j] = ∞ if no direct edge exists
  - dist[i][i] = 0 for all i

2. Process:
For each vertex k from 0 to V-1:

- Treat k as an intermediate point.
- For every pair (i, j):
  - If going from i → k → j is shorter than the current i → j, update it:
  - if dist[i][j] > dist[i][k] + dist[k][j]:
  - dist[i][j] = dist[i][k] + dist[k][j]

3.After all updates:

- dist[i][j] now contains the shortest distance from i to j.

4. Output:

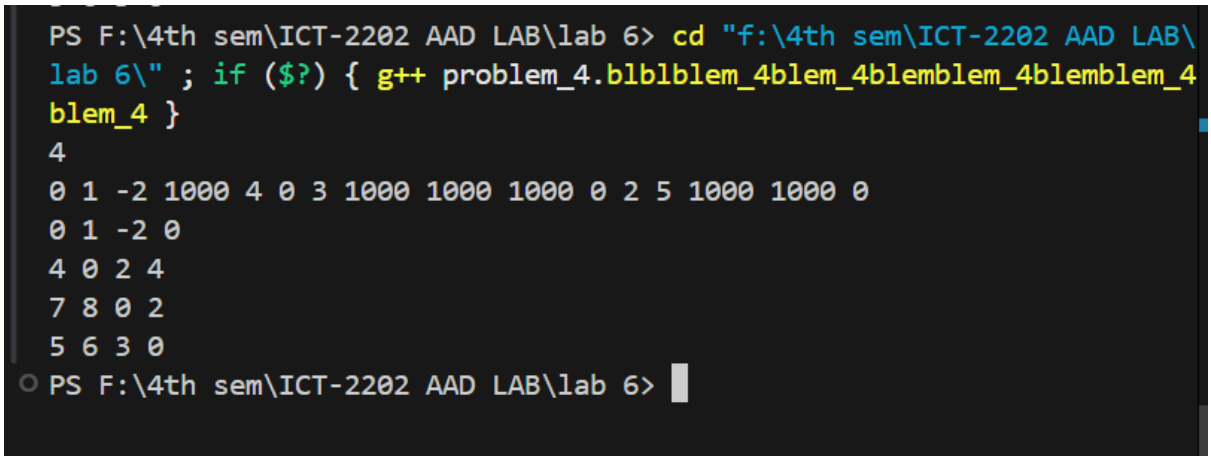
- The final dist matrix.

CODE/SOLUTION:-

```
#include <bits/stdc++.h>
using namespace std;
int main() {
   int V;
   cin >> V;
   vector<vector<long long>> dist(V, vector<long long>(V,LLONG_MAX));
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         int y;
         cin>>y;
         if(y==1000)
         dist[i][j]=LLONG_MAX;
         else dist[i][j]=y;
         }   }
   for (int k = 0; k < V; k++) {
      for (int i = 0; i < V; i++) {
         for (int j = 0; j < V; j++) {
            if (dist[i][k] != LLONG_MAX && dist[k][j] != LLONG_MAX) {
               dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }   }     }   }
```

```
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == LLONG_MAX) cout << "INF ";
            else cout << dist[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

OUTPUT:-



Explanation

This algorithm computes the shortest path between every pair of vertices in a weighted graph, useful for logistics and route optimization. The program starts with a matrix of direct distances between cities. It iteratively updates the distance between each pair (i, j) by checking if going through an intermediate vertex k reduces the distance. After all iterations, the matrix contains the shortest distance for every city pair. This approach is simple and guarantees correct results for positive or negative weights (no negative cycles).
Time Complexity: $O(V^3)$, where V is the number of vertices
Space Complexity: $O(V^2)$

Discussion

The four algorithms—Longest Common Subsequence (LCS), Floyd–Warshall, 0/1 Knapsack, and Rod Cutting—were chosen to illustrate practical applications of dynamic programming and graph optimization. LCS efficiently finds the longest ordered sequence common to two strings, useful in plagiarism detection or bioinformatics, but requires $O(m \times n)$ space and time. Floyd–Warshall precomputes all-pairs shortest paths in a graph, making it ideal for logistics and routing; it is simple and accurate but has $O(V^3)$ time complexity, which is costly for large networks. The 0/1 Knapsack problem models constrained resource allocation, guaranteeing optimal solutions via DP, though it can be slow for large capacities and item counts. Rod Cutting maximizes profit by evaluating all possible partitions, showing how DP avoids redundant calculations; however, its $O(n^2)$ complexity can be limiting for long rods. Overall, these algorithms demonstrate systematic problem-solving with guaranteed optimal solutions, but their main drawback is high computational or space requirements for large inputs. Choosing the right algorithm depends on problem type, input size, and available resources.