



University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial
Intelligence



Laboratory Exercise 5

Artificial Intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1	Goal of the Exercise	1
2	Theoretical Introduction to Dynamic Programming	1
2.1	Core Concepts	1
2.2	Approaches to Dynamic Programming	1
2.2.1	Top-Down with Memoization	1
2.2.2	Bottom-Up (Tabulation)	2
3	Example Task: The Fibonacci Sequence	2
3.1	Task 1: Naive Recursive Implementation	2
3.2	Task 2: Top-Down with Memoization	2
3.3	Task 3: Bottom-Up (Tabulation)	3
4	Main Task: The Rabbit Problem	3
4.1	Step 1: Define the State and Recurrence Relation	3
4.2	Step 2: Identify the Base Cases	3
4.3	Step 3: Implementation	4
5	Bonus Task: Visualizing the Rabbit's Optimal Path	4
5.1	Task 1: Path Reconstruction	4
5.2	Task 2: Visualization with <code>natId</code>	4
6	For Those Who Want to Learn More	5

1 Goal of the Exercise

The goal of this laboratory exercise is to provide a hands-on introduction to Dynamic Programming (DP), a powerful technique for solving optimization problems. Students will learn to identify the key characteristics of problems solvable with DP: **overlapping subproblems** and **optimal substructure**.

The exercise will begin with the classic Fibonacci sequence to illustrate the inefficiency of naive recursion and the dramatic performance gains achieved by two DP approaches: top-down with memoization and bottom-up (tabulation). The main task will involve applying these concepts to solve a more complex grid-based pathfinding problem. Finally, the bonus task will challenge students to use the `natId` C++ framework to visualize the grid and the optimal path found by their DP algorithm, bridging the gap between the abstract algorithm and a concrete visual result.

2 Theoretical Introduction to Dynamic Programming

Dynamic Programming is an algorithmic technique for solving complex problems by breaking them down into simpler, overlapping subproblems. The key principle is to **"never solve the same problem twice."** Instead of recomputing the solution to a subproblem every time it's needed, the solution is computed once and stored in a lookup table or memo.

2.1 Core Concepts

For a problem to be solvable with Dynamic Programming, it must exhibit two main properties:

- **Overlapping Subproblems:** A problem has overlapping subproblems if it can be broken down into subproblems that are reused several times. For example, in the naive recursive calculation of `Fibonacci(5)`, `Fibonacci(3)` is computed twice. As the input grows, the number of redundant computations explodes, leading to exponential time complexity. DP solves this by storing the result of `Fibonacci(3)` the first time it's calculated and simply looking it up the second time.
- **Optimal Substructure:** A problem has an optimal substructure if an optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems. For instance, the shortest path to a destination in a graph is composed of shortest paths to intermediate nodes.

2.2 Approaches to Dynamic Programming

There are two primary methods for implementing a DP solution:

2.2.1 Top-Down with Memoization

This approach follows the logic of a natural recursive solution but adds a caching layer.

1. A recursive function is written to solve the problem as usual.

2. A data structure (e.g., an array, map, or vector), often called a "memo," is used to store the results of subproblems. It's typically initialized with a sentinel value to indicate that a subproblem has not yet been solved.
3. At the beginning of the recursive function, it first checks the memo. If a result for the current state already exists, it is returned immediately without further computation.
4. If the result is not in the memo, the function computes it, **stores the result in the memo**, and then returns it.

This method is often intuitive to write as it mirrors the recursive definition of the problem.

2.2.2 Bottom-Up (Tabulation)

This approach is iterative and avoids recursion altogether.

1. The solutions to the smallest possible subproblems are calculated first.
2. A table (e.g., a 1D or 2D array) is created to store the results.
3. The table is filled iteratively, where each new entry is calculated based on previously computed values in the table.
4. This process continues until the solution to the original, largest problem is found.

Tabulation is generally faster in practice due to the absence of recursion overhead and can be more space-efficient in some cases.

3 Example Task: The Fibonacci Sequence

This introductory task will guide you through solving the Fibonacci sequence problem, clearly demonstrating the difference in performance between naive recursion and the two DP approaches.

3.1 Task 1: Naive Recursive Implementation

Write a C++ function `long long fib_naive(int n)` that calculates the n -th Fibonacci number using a purely recursive approach based on the definition: $F(n) = F(n - 1) + F(n - 2)$, with base cases $F(0) = 0$ and $F(1) = 1$.

To observe the inefficiency, add a global counter to track the number of function calls. Run your function for $n = 10, 20, 40$ and report the results and call counts. Notice how quickly the execution time and call count grow.

3.2 Task 2: Top-Down with Memoization

Create a new function, `long long fib_memo(int n, std::vector<long long>& memo)`. This function will implement the memoized top-down approach.

- The `memo` vector should be initialized with a value like -1.
- Inside the function, first check if `memo[n]` is not -1. If it contains a valid result, return it immediately.

- Otherwise, compute the result recursively as in the naive version.
- Before returning the computed result, store it in `memo[n]`.

Compare the performance and call counts with the naive version for $n = 40$.

3.3 Task 3: Bottom-Up (Tabulation)

Finally, write an iterative function `long long fib_bottom_up(int n)`.

- Create a DP table, e.g., `std::vector<long long> dp(n + 1)`.
- Initialize the base cases: `dp[0] = 0` and `dp[1] = 1`.
- Use a `for` loop to iterate from 2 up to n , filling the table with the rule `dp[i] = dp[i-1] + dp[i-2]`.
- The final answer is `dp[n]`.

This version should be the fastest, as it has no recursive overhead.

4 Main Task: The Rabbit Problem

A rabbit is at the top-left corner (cell (0,0)) of an $n \times n$ grid and wants to reach the bottom-right corner (cell (n-1, n-1)). The rabbit can only move **right** or **down**. The paths between cells contain carrots. You are given two matrices representing the number of carrots on each path:

- `carrots_right[i][j]` stores the number of carrots on the path from cell (i, j) to (i, j+1).
- `carrots_down[i][j]` stores the number of carrots on the path from cell (i, j) to (i+1, j).

Your task is to find the path from (0,0) to (n-1, n-1) that maximizes the total number of carrots collected.

4.1 Step 1: Define the State and Recurrence Relation

First, define your DP state. A natural choice is `DP[i][j]`: the maximum number of carrots the rabbit can collect to reach cell (i, j).

Next, formulate the recurrence relation. To arrive at cell (i, j), the rabbit must have come from either the cell above, (i-1, j), or the cell to the left, (i, j-1). Therefore, the optimal value for `DP[i][j]` is the maximum of the values from these two preceding cells, plus the carrots collected on the final move.

$$DP[i][j] = \max(DP[i-1][j] + \text{carrots_down}[i-1][j], DP[i][j-1] + \text{carrots_right}[i][j-1])$$

4.2 Step 2: Identify the Base Cases

The DP table cannot be filled without starting values (base cases).

- The starting cell has 0 carrots: $DP[0][0] = 0$.
- The first row ($i=0$) can only be reached by moving right. The value for $DP[0][j]$ depends only on the value of $DP[0][j-1]$.
- Similarly, the first column ($j=0$) can only be reached by moving down. The value for $DP[i][0]$ depends only on $DP[i-1][0]$.

4.3 Step 3: Implementation

Implement a function that takes the two carrot matrices and the grid size n as input and returns the maximum number of carrots.

- Create an $n \times n$ DP table.
- Initialize the base cases (the starting cell, the first row, and the first column).
- Use nested loops to fill the rest of the DP table according to your recurrence relation.
- The final answer will be the value stored in the bottom-right cell of the table, $DP[n-1][n-1]$.

5 Bonus Task: Visualizing the Rabbit's Optimal Path

The DP table gives you the maximum number of carrots, but not the path itself. This bonus task involves first reconstructing the optimal path and then visualizing it using the `natId` framework.

5.1 Task 1: Path Reconstruction

Write a function that takes your filled DP table, the carrot matrices, and the grid size as input, and returns the optimal path as a sequence of coordinates (e.g., `std::vector<td::Point>`).

- Start at the destination cell ($n-1, n-1$).
- Iteratively determine which cell the rabbit came from to achieve the optimal score at the current cell. You can do this by comparing the values in the DP table: if $DP[i][j]$ was calculated from $DP[i-1][j]$, then the previous cell was $(i-1, j)$. Otherwise, it was $(i, j-1)$.
- Add the current cell's coordinates to your path list.
- Move to the preceding cell you identified and repeat the process until you reach the starting cell $(0,0)$.
- Since you traced the path backward, you may need to reverse the list of coordinates to get the path from start to finish.

5.2 Task 2: Visualization with natId

Create a simple `natId` application to draw the rabbit's grid and its optimal path.

- **Create a RabbitCanvas Class:** This class will inherit from `gui::Canvas` and will be responsible for all the drawing logic inside its `onDraw` method.

- **Draw the Grid:** In `onDraw`, use nested loops to draw the grid points (as small circles using `gui::Shape`) and the paths between them (as lines).
- **Display Carrot Values:** Use `gui::DrawableString` to draw the number of carrots next to the midpoint of each path line.
- **Highlight the Optimal Path:** After drawing the full grid, iterate through the reconstructed path you generated in Task 1. For each segment in the optimal path, draw it again using a different color (e.g., green) and a thicker line to make it stand out.
- **Integrate into an Application:** Create a `MainView` that holds an instance of your `RabbitCanvas` and a "Solve" button. When the button is clicked, it should execute your DP algorithm, reconstruct the path, pass the path data to the canvas, and call `reDraw()` to update the display.

6 For Those Who Want to Learn More

Dynamic Programming is a vast and fundamental topic in computer science. The problems below, mentioned in the lecture, are classic examples and are excellent for further practice:

- Longest Increasing Subsequence (LIS)
- Rod Cutting Problem
- 0/1 Knapsack Problem
- Edit Distance (Levenshtein Distance)
- Longest Common Subsequence (used in DNA comparison)
- Matrix Chain Multiplication
- Coin Change Problem
- Job Scheduling
- Traveling Salesperson Problem (TSP)