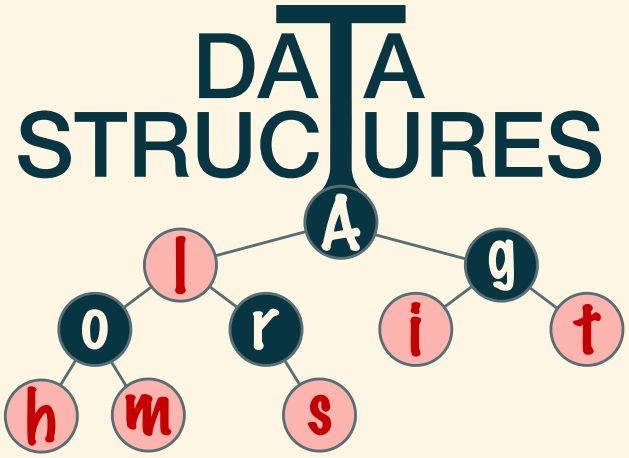
Home



WEEK 11: DYNAMIC PROGRAMMING (LAB)

Stefan-Cristian Roata

2026-03-19

Home

LEARNING OBJECTIVES

By the end of this lab session, you should be able to:

- Implement recursive and dynamic programming solutions to problems.
- Understand the time complexity differences between naive recursion and optimized DP approaches.
- Solve algorithmic problems such as Mobile Game Level Progression, DNA Sequence Analysis, and Vending Machine Programming.

Home

OVERVIEW OF LAB ACTIVITIES

Here are the activities for which you will upload solutions to xSITe or Gradescope:

1. **Mobile Game Level Progression - Recursive**: Submit using Gradescope. Total 25 marks.
2. **Mobile Game Level Progression - Iterative (DP)**: Submit using Gradescope. Total 25 marks.
3. **DNA Sequence Analysis - Longest Common DNA Subsequence**: Submit using Gradescope. Total 25 marks.
4. **Vending Machine Programming - Optimal Coin Change**: Submit using Gradescope. Total 25 marks.

Home

EXERCISE 1A: MOBILE GAME LEVEL PROGRESSION - RECURSIVE

TASK

You are developing a progression system for a popular mobile RPG game called **"Level Nexus"**. In this game, players can advance through levels by completing quests or challenges. The game mechanics allow a player to advance either `1` or `2` levels at a time (depending on quest difficulty they choose).

Given that a player wants to reach level `n`, your task is to calculate the number of distinct progression paths they can take to reach that level from level 0. **Code a recursive solution to this problem. Avoid using loops or arrays.**

EXAMPLES

```
Input: n = 2
Output: 2
Explanation: (1 level + 1 level) OR (2 levels at once)

Input: n = 3
Output: 3
Explanation: (1 level + 1 level + 1 level) OR (1 level + 2 levels) OR (2 levels + 1 level)
```

Home

EXERCISE 1A: MOBILE GAME LEVEL PROGRESSION - RECURSIVE

CONSTRAINTS

The parameter `n` can take values between 1 and 45 (inclusive), representing the target level.

TEMPLATE

You can find the template files on xSITe. Use the provided `makefile` to compile and run your project.

SUBMISSION

Implement the `calculateProgressionPaths` function using recursion. Submit your completed `mobile_game_progression_recursive.cpp` file to the Gradescope assignment titled **Week 11 Lab Exercise 1A: Mobile Game Level Progression - Recursive**.

Home

EXERCISE 1B: MOBILE GAME LEVEL PROGRESSION - ITERATIVE (DP)

TASK

Continuing with the **"Level Nexus"** mobile game, you now need to optimize the progression path calculation system. The game's backend needs to handle thousands of concurrent players checking their possible progression routes efficiently.

Given that a player wants to reach level n, calculate the number of distinct progression paths they can take (advancing 1 or 2 levels at a time).

REQUIREMENTS

Give a bottom-up dynamic programming solution to this problem (tabulation). You are encouraged to use arrays. Do NOT use recursion for this problem.

EXAMPLE

```
Input: n = 4
Output: 5
Explanation: (1+1+1+1) OR (2+1+1) OR (1+2+1) OR (1+1+2) OR (2+2)
```

Home

EXERCISE 1B: MOBILE GAME LEVEL PROGRESSION - ITERATIVE (DP)

CONSTRAINTS

The parameter `n` can take values between 1 and 45 (inclusive), representing the target level.

TEMPLATE

You can find the template files on xSITe.

Implement the `calculateProgressionPathsDP` function using **dynamic programming (bottom-up)**. Use either an array-based approach (`O(n)` space) or an optimized two-variable approach (`O(1)` space).

SUBMISSION

Submit your completed `mobile_game_progression_dp.cpp` file to the Gradescope assignment titled **Week 11 Lab Exercise 1B: Mobile Game Level Progression - Iterative (DP)**.

Home

EXERCISE 2: LONGEST COMMON DNA SUBSEQUENCE

TASK

You are working as a bioinformatics researcher studying **evolutionary relationships** between different organisms. Your task is to analyze DNA sequences from two different species to find their **longest common DNA subsequence**.

This helps scientists understand how closely related these organisms are - the longer the common DNA sequence, the more recent their evolutionary split.

Given two DNA strings `dna1` and `dna2` (containing only characters A, C, G, T), your task is to return **the length of their longest common subsequence (LCS).**

Home

EXERCISE 2: LONGEST COMMON DNA SUBSEQUENCE

DEFINITIONS

A **DNA subsequence** is a sequence that can be derived from the original DNA sequence by deleting some nucleotides without changing the order of the remaining nucleotides.

For example, from the DNA sequence **ACGTACGT**, valid subsequences include **ACG**, **AGCT**, **ACGT**. Sequences like *TGCA* or *CGTA* are **NOT** valid subsequences (order changed).

A **common DNA subsequence** between `dna1` and `dna2` is a subsequence that appears in both DNA strings. For example, the DNA subsequence **ACTG** might appear in both **ACGTACGT** and **ACTGTCCG**.

The **longest common DNA subsequence** between `dna1` and `dna2` is the common subsequence that has the maximum length, indicating the highest degree of evolutionary similarity.

Home

EXERCISE 2: LONGEST COMMON DNA SUBSEQUENCE

EXAMPLE INPUT AND OUTPUT

```
Input: dna1 = "ACGTACGT", dna2 = "ACTTGCGT"
Output: 6  // longest common subsequence is "ACTCGT"

Input: dna1 = "ATCGATCG", dna2 = "TCGAATCG"
Output: 7  // longest common subsequence is "TCGATCG"

Input: dna1 = "AAAA", dna2 = "TTTT"
Output: 0  // No common subsequence between these DNA sequences
```

TEMPLATE

You can find the template files on xSITe. Implement the `longestCommonSubsequence` function using **dynamic programming** with a 2D table. This table should store the length of the LCS for different prefixes of the input DNA sequences.

SUBMISSION

Submit your completed `longest_common_subsequence.cpp` file to the Gradescope assignment titled **Week 11 Lab Exercise 2: DNA Sequence Analysis - Longest Common DNA Subsequence**.

Home

EXERCISE 2: LONGEST COMMON DNA SUBSEQUENCE

HINTS

- Consider the prefixes of the DNA sequences `dna1` and `dna2`. The longest common subsequence of the `dna1` and `dna2` depends on **the longest common subsequence of their prefixes**.
- Use a 2D table to store the solutions to the subproblems instead of recursion. A table called **LCS(i,j)** should store the longest common subsequence of the prefix of `dna1` with length `i` and the prefix of `dna2` with length `j`.
- **If the last nucleotides of the prefixes match** (same DNA base), include them in the LCS and move to the previous nucleotides: `LCS(i,j)=LCS(i−1,j−1)+1`.
- **If the last nucleotides don't match**, try removing one nucleotide at a time and take the maximum result: `LCS(i,j) = max(LCS(i−1,j),LCS(i,j−1))`.

Home

EXERCISE 2: LONGEST COMMON DNA SUBSEQUENCE

HINTS

$$LCS(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1, & \text{if } dna1[i-1] = dna2[j-1 \\ \max(LCS(i-1,j), LCS(i,j-1)), & \text{if } dna1[i-1] \neq dna2[j-1 \end{cases}$$

- The final solution is `LCS[m][n]`, where `m` is the length of `dna1` and `n` is the length of `dna2`.

Home

EXERCISE 3: VENDING MACHINE PROGRAMMING - OPTIMAL COIN CHANGE

TASK

You are programming the **change dispensing system** for a modern smart vending machine. The vending machine has a limited set of coin denominations available in its coin hopper, and your goal is to **minimize the number of coins dispensed** as change to customers.

Given an integer array `coins` representing the available coin denominations in the vending machine and an integer `amount` representing the change amount to be dispensed, return **the fewest number of coins needed to make up that amount**.

NOTE: If the exact change amount cannot be made using the available coin denominations, return -1.

Home

EXERCISE 3: VENDING MACHINE PROGRAMMING - OPTIMAL COIN CHANGE

EXAMPLES

```
Input: coins = {1,2,5}, change amount = 11
Output: 3  // Dispense (5¢ + 5¢ + 1¢)
Explanation: The vending machine can dispense 11¢ change using only 3 coins: 5¢ + 5¢ + 1¢. Other
combinations like (5¢ + 2¢ + 2¢ + 2¢) use more coins and are less efficient.

Input: coins = {2}, change amount = 3
Output: -1
Explanation: The vending machine only has 2¢ coins available, so it cannot make exactly 3¢ change. In
your program, you should return -1. In real life, the machine should display "exact change only".
```

TEMPLATE

You can find the template files on xSITe.

Implement the `coinChange` function inside the partial `coin_change.cpp` file. Use a **dynamic programming** approach to solve this problem efficiently.

SUBMISSION

Submit your completed `coin_change.cpp` file to the Gradescope assignment titled **Week 11 Lab Exercise 3: Vending Machine Programming - Optimal Coin Change**.

Home

EXERCISE 3: VENDING MACHINE PROGRAMMING - OPTIMAL COIN CHANGE

HINTS

- Think in terms of choices. For each coin denomination, the vending machine can choose to dispense that coin or skip it.
- For each subproblem (`amount = 1`, `amount = 2` ... `amount = target`), try all available coin choices and select the one that uses the fewest total coins.
- Let `dp(amount)` represent the minimum number of coins needed to make change for that amount. The recurrence relation is: `dp(amount) = 1 + min(dp(amount-c))` for all coin denominations `c` available in the vending machine.
- If no valid combination exists for the target amount, return `-1` (machine cannot provide exact change).

CONCLUSION

Home

WRAP-UP

By the end of this lab you should be able to:

1. **Understand dynamic programming fundamentals**: Distinguish between recursive and dynamic programming approaches, and recognize when DP optimization is beneficial
2. **Implement simple top-down solutions**: Write clean recursive algorithms and analyze their exponential time complexity
3. **Apply bottom-up techniques**: Implement tabulation (bottom-up) for problems solvable using dynamic programming
4. **Solve real-world optimization problems**: Apply DP to mobile game progression systems, bioinformatics DNA analysis, and vending machine programming

OUTLOOK

This lab covered **Dynamic Programming** as a powerful optimization technique for solving complex problems with overlapping subproblems. The remaining weeks will explore:

Greedy Algorithms

Local optimization strategies that build globally optimal solutions step-by-step