Lab 08: Dynamic Programming 3

Introduction to Dynamic Programming (動態規劃簡介)

Dynamic Programming (DP) is a powerful algorithmic technique used for solving complex problems by breaking them down into simpler, overlapping subproblems. The results of these subproblems are stored (usually in an array or a table) to avoid redundant computations, leading to efficient solutions. This approach is particularly effective when a problem exhibits two key characteristics:

1. **Optimal Substructure (最佳子結構):** An optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
   (問題的最佳解可由其子問題的最佳解有效地建構而成。)
2. **Overlapping Subproblems (重疊子問題):** The same subproblems are encountered and solved multiple times during the recursive computation. DP stores the solutions to these subproblems (memoization or tabulation) to reuse them.
   (相同的子問題在遞迴計算過程中會被多次遇到並求解。動態規劃會儲存這些子問題的解 (使用記憶化或列表法) 以便重複使用。)

Unlike greedy algorithms, which make a locally optimal choice at each step hoping to find a global optimum, dynamic programming typically explores all possibilities by considering different choices for subproblems and then combining their results to find the best overall solution.

There are two main approaches to implementing DP:

- **Memoization (Top-Down with Memoization / 記憶化遞迴 - 由上而下):** Solve the problem recursively, but store the result of each computed subproblem in a lookup table. If a subproblem is encountered again, return the stored result instead of recomputing.
  (透過遞迴方式解決問題，但將每個已計算子問題的結果儲存在查詢表中。如果再次遇到相同的子問題，則直接回傳儲存的結果，而非重新計算。)
- **Tabulation (Bottom-Up / 列表法 - 由下而上):** Solve subproblems in a specific order, typically starting from the smallest base cases and iteratively building up solutions to larger subproblems. Results are stored in a table.
  (依照特定順序解決子問題，通常從最小的基礎案例開始，逐步建立較大子問題的解，並將結果儲存在表格中。)

This lab will explore a few classic problems that can be solved efficiently using dynamic programming.
(本實驗將探討一些可以使用動態規劃有效解決的經典問題。)

---

1. Fibonacci Sequence / Climbing Stairs (費氏數列 / 爬樓梯問題)

Problem Description (問題描述):

The Fibonacci sequence is defined as follows: $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. The task is to compute the n-th Fibonacci number.
(費氏數列定義如下：$F_0 = 0, F_1 = 1$，且對於 $n > 1$，$F_n = F_{n-1} + F_{n-2}$。任務是計算第 n 個費氏數。)

This problem is structurally identical to the "climbing stairs" problem where you can climb 1 or 2 stairs at a time. F_n would represent the number of distinct ways to reach the n-th stair (if we adjust base cases, e.g., ways to reach stair 1 is 1, stair 2 is 2).
(這個問題在結構上與「爬樓梯」問題相同，每次可以爬 1 階或 2 階。F_n 可以代表到達第 n 階樓梯的不同方法數量 (如果我們調整基礎案例，例如，到達第 1 階的方法有 1 種，第 2 階有 2 種)。)

DP Approach (動態規劃方法):

A naive recursive solution $F(n) = F(n-1) + F(n-2)$ has exponential time complexity due to recomputing the same Fibonacci numbers multiple times. DP can optimize this.
(一個樸素的遞迴解法 $F(n) = F(n-1) + F(n-2)$ 具有指數級的時間複雜度，因為它會多次重複計算相同的費氏數。動態規劃可以優化這個問題。)

- **State Definition (狀態定義):**
  `dp[i]` will store the i-th Fibonacci number, F_i.
  (`dp[i]` 將儲存第 i 個費氏數 F_i。)
- **Base Cases (基礎案例):**
  `dp[0] = 0`
  `dp[1] = 1`
- **Recurrence Relation (遞迴關係式):**
  `dp[i] = dp[i-1] + dp[i-2]` for $i > 1$.
- **Final Answer (最終答案):**
  The n-th Fibonacci number is `dp[n]`.
  (第 n 個費氏數是 `dp[n]`。)

C++ Implementation (Tabulation - 列表法):

```
#include <vector>
#include <iostream>

long long fib_tabulation(int n) {
```

```
    if (n <= 0) return 0;
    if (n == 1) return 1;

    std::vector<long long> dp(n + 1);
    dp[0] = 0; // F_0
    dp[1] = 1; // F_1

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

C++ Implementation (Memoization - 記憶化遞迴):

C++

```
#include <vector>
#include <iostream>
#include <map> // Or use a vector if n is reasonably small

// Using a map for memoization table for potentially sparse calls or larger n flexibility
// 使用 map 作為記憶化表格，適用於可能的稀疏呼叫或較大 n 值的彈性
std::map<int, long long> memo;

long long fib_memoization(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;

    // Check if already computed
    // 檢查是否已經計算過
    if (memo.count(n)) {
        return memo[n];
    }

    // Compute and store
    // 計算並儲存結果
    memo[n] = fib_memoization(n - 1) + fib_memoization(n - 2);
    return memo[n];
}
```

Problem 1.1
題目描述:
背景： 你獲得了上述計算費氏數列的 fib_tabulation (列表法) 和 fib_memoization (記憶化遞迴) 兩種動態規劃實現。
任務：
對於 n = 7，請手動追蹤 fib_memoization(7) 的呼叫過程。哪些子問題的結果被儲存和重複使用？列出 memo 表在計算過程中的變化。 (例如，第一次呼叫 fib_memoization(5) 時，memo 中尚無 5，計算後 memo[5] 被賦值。)
比較樸素遞迴解法 (不使用DP)、列表法DP、記憶化DP 在計算 F_n 時的時間複雜度和空間複雜度。
回答格式：
文字描述追蹤過程，並展示 memo 表的狀態變化。
一個表格，列出三種方法的時間和空間複雜度。
Problem Description:
Background: You are provided with the fib_tabulation (tabulation method) and fib_memoization (memoization method) dynamic programming implementations for calculating Fibonacci numbers, as shown above.
Task:
For n = 7, manually trace the execution of fib_memoization(7). Which subproblem results are stored and reused? List the changes to the memo table during the computation. (For example, when fib_memoization(5) is called for the first time, 5 is not in memo; after computation, memo[5] is assigned.)
Compare the time complexity and space complexity for calculating F_n using: a. Naive recursive solution (without DP). b. Tabulation DP. c. Memoization DP.
Answer Format:
A textual description of the trace, showing the state changes of the memo table.
A table listing the time and space complexities for the three methods.
Problem 1.2
題目描述:
背景： 「爬樓梯」問題：你正在爬一個有 n 階的樓梯。每次你可以爬 1 階、2 階或 3 階。請問有多少種不同的方法可以爬到樓頂？
任務： 請實作一個函式 long long count_ways_to_climb(int n)，使用動態規劃 (可選擇列表法或記憶化遞迴) 來計算爬 n 階樓梯的不同方法總數。
輸入：
n: 一個 int，表示樓梯的階數 ($n \geq 0$)。 (An int representing the number of stairs ($n \geq 0$).)
輸出：

一個 long long，表示爬到第 n 階樓梯的不同方法總數。 (A long long representing the total number of distinct ways to climb to the n-th stair.)
範例 (Example)：
n = 0: count_ways_to_climb(0) 應回傳 1 (一種方法：不爬 / one way: don't climb at all, already at the top if conceptually starting before stair 1).
n = 1: count_ways_to_climb(1) 應回傳 1 (方法: {1} way: {1}).
n = 2: count_ways_to_climb(2) 應回傳 2 (方法: {1,1}, {2} ways: {1,1}, {2}).
n = 3: count_ways_to_climb(3) 應回傳 4 (方法: {1,1,1}, {1,2}, {2,1}, {3} ways: {1,1,1}, {1,2}, {2,1}, {3}).
提示：
定義 dp[i] 為到達第 i 階的方法數。 (Define dp[i] as the number of ways to reach the i-th stair.)
思考基礎案例：dp[0]，以及如何從 dp[i-1], dp[i-2], dp[i-3] 推導 dp[i]。 (Consider the base case dp[0], and how dp[i] can be derived from dp[i-1], dp[i-2], and dp[i-3].)
Problem Description:
Background: "Climbing Stairs" problem: You are climbing a staircase that has n stairs. You can climb 1, 2, or 3 stairs at a time. In how many distinct ways can you climb to the top?
Task: Implement a function long long count_ways_to_climb(int n) using dynamic programming (you can choose either tabulation or memoization) to calculate the total number of distinct ways to climb n stairs.
Input:
n: An int representing the number of stairs ($n \geq 0$).
Output:
A long long representing the total number of distinct ways to climb to the n-th stair.
Example:
n = 0: count_ways_to_climb(0) should return 1 (one way: don't climb at all, conceptually already at the "top" if considering reaching the level of stair 0).
n = 1: count_ways_to_climb(1) should return 1 (way: {1}).
n = 2: count_ways_to_climb(2) should return 2 (ways: {1,1}, {2}).
n = 3: count_ways_to_climb(3) should return 4 (ways: {1,1,1}, {1,2}, {2,1}, {3}).
Hint:
Define dp[i] as the number of ways to reach the i-th stair.
Consider the base case dp[0], and how dp[i] can be derived from dp[i-1], dp[i-2], and dp[i-3].

C++

```
#include <iostream>
#include <vector>
#include <map> // For memoization approach, if chosen

// ##### STUDENT'S IMPLEMENTATION AREA START #####
// ##### 學生的實作區域 START #####
long long count_ways_to_climb(int n) {
    // TODO: Implement this function using Dynamic Programming (Tabulation or Memoization).
    // TODO: 使用動態規劃 (列表法或記憶化遞迴) 實作此函式。
    //
    // If using Tabulation:
    // 1. Handle base cases for n=0. If n < 0, perhaps return 0 or handle as an error.
    // 2. Create a dp array/vector of size n+1.
    // 3. Initialize dp[0] (e.g., dp[0]=1 way to be at stair 0).
    // 4. Iterate from i = 1 to n:
    //    dp[i] = 0;
    //    if (i >= 1) dp[i] += dp[i-1];
    //    if (i >= 2) dp[i] += dp[i-2];
    //    if (i >= 3) dp[i] += dp[i-3];
    // 5. Return dp[n].
    //
    // 若使用列表法 (Tabulation):
    // 1. 處理 n=0 的基礎案例。若 n < 0，可能回傳 0 或作為錯誤處理。
    // 2. 建立一個大小為 n+1 的 dp 陣列/向量。
    // 3. 初始化 dp[0] (例如，dp[0]=1 種方法到達第 0 階)。
    // 4. 從 i = 1 迭代到 n：
    //    dp[i] = 0;
    //    若 (i >= 1) dp[i] += dp[i-1];
    //    若 (i >= 2) dp[i] += dp[i-2];
    //    若 (i >= 3) dp[i] += dp[i-3];
    // 5. 回傳 dp[n]。
    //
    // If using Memoization:
    // 1. Handle base cases: n=0 return 1; n < 0 return 0.
    // 2. Check memo table for n. If present, return stored value.
    // 3. Recursively compute: ways = climb(n-1) + climb(n-2) + climb(n-3).
    // 4. Store result in memo table and return it.
    //
    // 若使用記憶化遞迴 (Memoization):
```

```
    // 1. 處理基礎案例：n=0 回傳 1；n < 0 回傳 0。
    // 2. 檢查 n 是否在記憶化表格中。若存在，回傳儲存的值。
    // 3. 遞迴計算：ways = climb(n-1) + climb(n-2) + climb(n-3)。
    // 4. 將結果儲存於記憶化表格中並回傳。
    long long ways = 0;
    // Your code here // 你的程式碼寫在這裡
    return ways;
}
// ##### STUDENT'S IMPLEMENTATION AREA END #####
// ##### 學生的實作區域 END #####

int main_fib_climb() { // Renamed to avoid conflict if other mains are present
    using namespace std;
    cout << "--- Fibonacci / Climbing Stairs ---" << endl;
    // Test Problem 1.1 examples here (manual trace for memoization)
    // You can add calls to fib_tabulation and fib_memoization for testing.
    // 例如：
    // cout << "fib_tabulation(7): " << fib_tabulation(7) << endl;
    // memo.clear(); // Clear memo before each new independent test of fib_memoization
    // cout << "fib_memoization(7): " << fib_memoization(7) << endl;


    cout << "\n--- Climbing Stairs (1, 2, or 3 steps) ---" << endl;
    cout << "Ways to climb 0 stairs: " << count_ways_to_climb(0) << endl; // Expected: 1
    cout << "Ways to climb 1 stair: " << count_ways_to_climb(1) << endl;  // Expected: 1
    cout << "Ways to climb 2 stairs: " << count_ways_to_climb(2) << endl; // Expected: 2
    cout << "Ways to climb 3 stairs: " << count_ways_to_climb(3) << endl; // Expected: 4
    cout << "Ways to climb 4 stairs: " << count_ways_to_climb(4) << endl; // Expected: 1+2+4 = 7
    cout << "Ways to climb 5 stairs: " << count_ways_to_climb(5) << endl; // Expected: 2+4+7 = 13
    return 0;
}
```

2. Longest Increasing Subsequence (LIS) (最長遞增子序列)
   Problem Description (問題描述):
   Given an array of integers nums, find the length of the longest strictly increasing subsequence. A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements.
   (給定一個整數陣列 nums，找出最長嚴格遞增子序列的長度。子序列是可從陣列中刪除一些或不刪除元素，而不改變其餘元素順序所得到的序列。)
   DP Approach (動態規劃方法):
   State Definition (狀態定義): dp[i] represents the length of the longest increasing subsequence that ends with the element nums[i]. (dp[i] 代表以元素 nums[i] 結尾 的最長遞增子序列的長度。)
   Base Cases (基礎案例): dp[i] = 1 for all i (each element itself is an increasing subsequence of length 1). (對於所有 i，$dp[i] = 1$ (每個元素本身都是長度為 1 的遞增子序列)。)
   Recurrence Relation (遞迴關係式): To calculate dp[i], we look at all previous elements nums[j] where $j < i$. If nums[j] < nums[i], it means nums[i] can extend an increasing subsequence ending at nums[j]. So, dp[i] can be dp[j] + 1. We take the maximum among all such possibilities. dp[i] = 1 + max(dp[j]) for all $0 \le j < i$ such that nums[j] < nums[i]. If no such j exists, dp[i] remains 1. (要計算 dp[i]，我們查看所有 $j < i$ 的先前元素 nums[j]。如果 nums[j] < nums[i]，代表 nums[i] 可以擴展一個以 nums[j] 結尾的遞增子序列。因此，dp[i] 可以是 dp[j] + 1。我們在所有這些可能性中取最大值。) (dp[i] = 1 + max(dp[j])，其中 $0 \le j < i$ 且 nums[j] < nums[i]。如果不存在這樣的 j，$dp[i]$保持為1\。)$FinalAnswer(最終答案)：The length of the LIS of the entire array is the maximum value in the dp array：max(dp[i])$ƒ $0 \le i <$ length of nums. (整個陣列的 LIS 長度是 dp 陣列中的最大值：max(dp[i])，其中 $0 \le i <$ nums 的長度。)
   C++ Implementation (Tabulation - 列表法):

C++

```
#include <vector>
#include <algorithm> // For std::max_element and std::max

int length_of_lis_tabulation(const std::vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }
    int n = nums.size();
    std::vector<int> dp(n, 1); // Initialize dp[i] = 1 for all i // 初始化所有 dp[i] = 1

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i]) {
                dp[i] = std::max(dp[i], dp[j] + 1);
            }
        }
    }
```

```
    // The LIS length is the maximum value in the dp array
    // LIS 長度是 dp 陣列中的最大值
    int max_len = 0;
    if (n > 0) { // Ensure dp is not empty before accessing
        max_len = dp[0];
        for (int i = 1; i < n; ++i) {
            if (dp[i] > max_len) {
                max_len = dp[i];
            }
        }
    }
    // Or simply: if (n > 0) max_len = *std::max_element(dp.begin(), dp.end()); else max_len = 0;
    return max_len;
}
```

Problem 2.1
題目描述:
背景： 你獲得了上述計算最長遞增子序列長度的 length_of_lis_tabulation 函式。
任務：
對於輸入陣列 nums = {3, 10, 2, 1, 20}，手動追蹤 length_of_lis_tabulation 的執行過程。列出 dp 陣列在每次外部迴圈 i 完成後的值。
最終的 LIS 長度是多少？
length_of_lis_tabulation 函式只回傳 LIS 的長度。請描述如何修改或擴展此方法 (不需寫完整程式碼，描述思路即可) 以便實際重建 LIS 本身 (即找出是哪些數字組成了最長遞增子序列)。
回答格式：
逐步展示 dp 陣列的變化。
寫出 LIS 長度。
文字描述重建 LIS 的思路。
Problem Description:
Background: You are provided with the length_of_lis_tabulation function above, which calculates the length of the Longest Increasing Subsequence.
Task:
For the input array nums = {3, 10, 2, 1, 20}, manually trace the execution of length_of_lis_tabulation. List the values of the dp array after each completion of the outer loop (loop for i).
What is the final LIS length?
The length_of_lis_tabulation function only returns the length of the LIS. Describe how you could modify or extend this method (no need to write full code, just describe the approach) to actually reconstruct the LIS itself (i.e., find which numbers form one such longest increasing subsequence).
Answer Format:
Step-by-step illustration of the dp array changes.
State the LIS length.
A textual description of the approach to reconstruct the LIS.
Problem 2.2
題目描述:
背景： 給定一個整數陣列 nums。
任務： 實作一個函式 int longest_increasing_subsequence(const std::vector& nums)，使用動態規劃計算並回傳 nums 中最長遞增子序列的長度。你應自行實現 DP 邏輯，類似於 length_of_lis_tabulation。
輸入：
nums: 一個 const std::vector。 (A const std::vector.)
輸出：
一個 int，表示最長遞增子序列的長度。 (An int representing the length of the longest increasing subsequence.)
範例 (Example)：
nums = {10, 9, 2, 5, 3, 7, 101, 18} -> LIS: {2, 3, 7, 18} 或 {2, 5, 7, 18} 或 {2, 3, 7, 101} 或 {2, 5, 7, 101}. 長度 (Length) = 4.
nums = {0, 1, 0, 3, 2, 3} -> LIS: {0, 1, 2, 3} 或 {0, 0, 2, 3} (不嚴格遞增時) {0,1,2,3} (嚴格遞增時)。長度 (Length) = 4.
nums = {7, 7, 7, 7, 7} -> LIS: {7}. 長度 (Length) = 1.
提示： 遵循文件範例中 LIS 的 DP 狀態定義和遞迴關係。 (Follow the DP state definition and recurrence relation for LIS as described in the document's example.)
Problem Description:
Background: You are given an array of integers nums.
Task: Implement a function int longest_increasing_subsequence(const std::vector& nums) that uses dynamic programming to calculate and return the length of the longest strictly increasing subsequence in nums. You should implement the DP logic yourself, similar to length_of_lis_tabulation.
Input:
nums: A const std::vector.
Output:
An int representing the length of the longest increasing subsequence.
Example:
nums = {10, 9, 2, 5, 3, 7, 101, 18} -> LIS: {2, 3, 7, 18} or {2, 5, 7, 18} etc. Length = 4.

nums = {0, 1, 0, 3, 2, 3} -> LIS: {0, 1, 2, 3}. Length = 4.
nums = {7, 7, 7, 7, 7} -> LIS: {7}. Length = 1.
Hint: Follow the DP state definition and recurrence relation for LIS as described in the document's example.

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For std::max or std::max_element

// ##### STUDENT'S IMPLEMENTATION AREA START #####
// ##### 學生的實作區域 START #####
int longest_increasing_subsequence(const std::vector<int>& nums) {
    // TODO: Implement this function using Dynamic Programming.
    // TODO: 使用動態規劃實作此函式。
    //
    // 1. Handle empty input: if nums is empty, LIS length is 0.
    //    處理空輸入：如果 nums 為空，LIS 長度為 0。
    // 2. Get the size of nums, say `n`.
    //    取得 nums 的大小，例如 `n`。
    // 3. Create a dp vector of size `n`, and initialize all its elements to 1
    //    (since each element itself is an LIS of length 1).
    //    建立一個大小為 `n` 的 dp 向量，並將其所有元素初始化為 1
    //    (因為每個元素本身都是長度為 1 的 LIS)。
    // 4. Iterate from i = 1 to n-1 (or 0 to n-1, adjusting inner loop):
    //    For each `nums[i]`, iterate from j = 0 to i-1:
    //        If `nums[j] < nums[i]` (strictly increasing condition):
    //            It means `nums[i]` can potentially extend the LIS ending at `nums[j]`.
    //            So, `dp[i]` could be `dp[j] + 1`.
    //            Update `dp[i] = std::max(dp[i], dp[j] + 1)`.
    //    從 i = 1 到 n-1 (或從 0 到 n-1，並調整內部迴圈) 迭代：
    //    對於每個 `nums[i]`，從 j = 0 到 i-1 迭代：
    //        如果 `nums[j] < nums[i]` (嚴格遞增條件)：
    //            這意味著 `nums[i]` 可能可以擴展以 `nums[j]` 結尾的 LIS。
    //            因此，`dp[i]` 可以是 `dp[j] + 1`。
    //            更新 `dp[i] = std::max(dp[i], dp[j] + 1)`。
    // 5. After filling the dp table, the length of the LIS for the whole array
    //    is the maximum value in the dp vector. Find and return this maximum.
    //    (If n=0, handle this to return 0. If n > 0, find max in dp).
    //    填寫完 dp 表後，整個陣列的 LIS 長度是 dp 向量中的最大值。
    //    找到並回傳這個最大值。(若 n=0，處理回傳 0。若 n > 0，則在 dp 中找最大值)。
    int max_length = 0;
    // Your code here // 你的程式碼寫在這裡
    return max_length;
}
// ##### STUDENT'S IMPLEMENTATION AREA END #####
// ##### 學生的實作區域 END #####

int main_lis() { // Renamed to avoid conflict
    using namespace std;
    cout << "\n--- Longest Increasing Subsequence ---" << endl;

    vector<int> nums1 = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << "LIS length for {10, 9, 2, 5, 3, 7, 101, 18}: "
         << longest_increasing_subsequence(nums1) << endl; // Expected: 4

    vector<int> nums2 = {0, 1, 0, 3, 2, 3};
    cout << "LIS length for {0, 1, 0, 3, 2, 3}: "
         << longest_increasing_subsequence(nums2) << endl; // Expected: 4

    vector<int> nums3 = {7, 7, 7, 7, 7};
    cout << "LIS length for {7, 7, 7, 7, 7}: "
         << longest_increasing_subsequence(nums3) << endl; // Expected: 1

    vector<int> nums4 = {3, 10, 2, 1, 20};
    cout << "LIS length for {3, 10, 2, 1, 20}: "
         << longest_increasing_subsequence(nums4) << endl; // Expected: 3

    vector<int> nums5 = {};
    cout << "LIS length for {}: "
         << longest_increasing_subsequence(nums5) << endl; // Expected: 0
    return 0;
}
```