Problem 1.1:

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.

ans:

```
fib(7)
└─ fib(6)
   └─ fib(5)
      └─ fib(4)
         └─ fib(3)
            └─ fib(2)
               └─ fib(1) → 1
               └─ fib(0) → 0
            → memo[2] = 1
            └─ fib(1) → 1
         → memo[3] = 2
         └─ fib(2) → 1 (from memo)
      → memo[4] = 3
      └─ fib(3) → 2 (from memo)
   → memo[5] = 5
   └─ fib(4) → 3 (from memo)
→ memo[6] = 8
└─ fib(5) → 5 (from memo)
→ memo[7] = 13
```

Call stage	New entry	Memo content
After fib(2)	memo[2] = 1	{2: 1}
After fib(3)	memo[3] = 2	{2: 1, 3: 2}
After fib(4)	memo[4] = 3	{2: 1, 3: 2, 4: 3}
After fib(5)	memo[5] = 5	{2: 1, 3: 2, 4: 3, 5: 5}
After fib(6)	memo[6] = 8	{2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
After fib(7)	memo[7] = 13	{2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}

Method	Time complexity	Space complexity
Naive Recursive	O(2^n)	O(n)
Tabulation DP	O(n)	O(n)
Memoization DP	O(n)	O(n)

Problem 1.2:

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].

ans:

```
#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) {
  if (n < 0)
  return 0;
  long long ways = 0;
  std::vector<long long> dp(n + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    if (i >= 1)
      dp[i] += dp[i - 1];
    if (i >= 2)
      dp[i] += dp[i - 2];
    if (i >= 3)
      dp[i] += dp[i - 3];
  }
  return ways = dp[n];
}
// ##### STUDENT'S IMPLEMENTATION AREA END #####
// ##### 學生的實作區域 END #####
```

Problem 2.1:

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 2.1 Solution

Step-by-step trace of `length_of_lis_tabulation({3, 10, 2, 1, 20})`

We define `dp[i]` as the length of the longest increasing subsequence (LIS) ending at index `i`.

Initial array:

```
nums = {3, 10, 2, 1, 20}
```

Initial dp:

```
dp = {1, 1, 1, 1, 1} // every element is a subsequence of length 1 by itself
```

Step i = 1 (nums[1] = 10):
Check all `j < 1`

- `nums[0] = 3 < 10` → `dp[1] = max(1, dp[0] + 1) = 2`

```
dp = {1, 2, 1, 1, 1}
```

Step i = 2 (nums[2] = 2):
Check all `j < 2`

- `nums[0] = 3 > 2` → skip
- `nums[1] = 10 > 2` → skip

No update

```
dp = {1, 2, 1, 1, 1}
```

Step i = 3 (nums[3] = 1):
Check all `j < 3`

- All previous numbers are greater → no update

```
dp = {1, 2, 1, 1, 1}
```

Step i = 4 (nums[4] = 20):
Check all `j < 4`

- `nums[0] = 3 < 20` → `dp[4] = max(1, dp[0] + 1) = 2`
- `nums[1] = 10 < 20` → `dp[4] = max(2, dp[1] + 1) = 3`
- `nums[2] = 2 < 20` → `dp[4] = max(3, dp[2] + 1) = 3`
- `nums[3] = 1 < 20` → `dp[4] = max(3, dp[3] + 1) = 3`

```
dp = {1, 2, 1, 1, 3}
```

Final LIS Length:

Answer: **`max(dp) = 3`**

LIS examples: `{3, 10, 20}` or `{2, 10, 20}` or `{1, 10, 20}`

How to reconstruct the LIS sequence

To reconstruct the actual LIS:

1. **Track predecessors:**
   Create an array `prev[i]` where each element stores the index of the previous number in the LIS ending at `i`.
2. **Update during DP loop:**
   Whenever `dp[i]` is updated from `dp[j] + 1`, also set `prev[i] = j`.
3. **Reconstruct path:**
   After finding the index `maxIndex` of the largest `dp[i]`, follow the `prev` array backward to collect the LIS.
4. **Reverse it:**
   Since the path was followed backward, reverse it at the end to get the correct order.

Problem 2.2:

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.

Problem 2.2 Solution

```
int longest_increasing_subsequence(const std::vector<int> &nums) {
  if (nums.empty())
    return 0;

  int n = nums.size();
  std::vector<int> dp(n, 1);

  for (int i = 1; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
```

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

  return *std::max_element(dp.begin(), dp.end());
}
```