Code & Examples 1

DP for Fibonacci Problem

To illustrate Dynamic Programming, let's look at the classic **Fibonacci Sequence**. The rule is simple: each number is the sum of the two preceding ones ($F(n) = F(n-1) + F(n-2)$), starting with 0 and 1.

1. The Original Solution (Naive Recursion)

This is the most direct translation of the mathematical formula into code using recursion.

```
#include <iostream>
using namespace std;

long long fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cout << "Masukkan nilai n: ";
    cin >> n;
    cout << "Fibonacci(" << n << ") = " << fibonacci(n) << endl;
    return 0;
}
```

The Problem with Naive Recursion

While the code above looks clean, it is incredibly inefficient for larger numbers (Exponential Time Complexity: O(2n)).

picture 0

The problem is Overlapping Subproblems. The algorithm calculates the same values over and over again.

```
To calculate fib(5), it calculates fib(4) and fib(3).

To calculate fib(4), it calculates fib(3) and fib(2).
```

Here, fib(3) is computed twice from scratch. As n grows, the number of redundant calculations explodes, causing the program to freeze or crash.

Solution A. Top-Down Approach (Memoization)

In this approach, we still use recursion, but we create a "memo" (a dictionary or array) to store results we've already found. Before calculating fib(n), we check if we already have the answer.

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

long long fibonacci(int n, std::vector<long long> &memo) {
    if (n <= 1)
        return n;

    if (memo[n] != -1)
        return memo[n];

    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
    return memo[n];
}

int main() {
    int n;
    std::cout << "Masukkan nilai n: ";
    std::cin >> n;

    std::vector<long long> memo(n + 1, -1);

    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n, memo) << std::endl;
    return 0;
}
```

Result: Time complexity drops to Linear Time O(n).

Solution B. Bottom-Up Approach (Tabulation)

In this approach, we abandon recursion. We start from the base cases (0 and 1) and iteratively fill a table (list) until we reach n. This avoids the overhead of function call stacks.

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

#define MAX 100 // Maximum n value

long long fibonacci(int n) {
    std::vector<long long> dp(n + 1);

    dp[0] = 0;
    dp[1] = 1;

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

    return dp[n];
}

int main() {
    int n;
    std::cout << "Masukkan nilai n: ";
    std::cin >> n;

    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) << std::endl;
    return 0;
}
```

Result: Linear Time O(n) and often faster in practice due to no recursion overhead.

Revision #2
Created 2025-11-27 04:23:39 UTC by CH
Updated 2025-12-07 09:21:27 UTC by BH