Module 10 - Dynamic Programming

- 1. Introduction to DP
- Code & Examples 1
- Code & Examples 2

1. Introduction to DP

What is DP?

Dynamic programming (DP) is defined as a powerful design technique that successfully combines the correctness of brute force algorithms with the efficiency of greedy algorithms.

The fundamental idea of dynamic programming is to remember the solutions to subproblems and reuse them to solve larger problems. DP involves identifying and solving all relevant sub problems.

Common use cases for dynamic programming include:

1. Finding an optimal solution, such as determining a minimum or maximum answer to a question.
2. Counting the total number of solutions that a problem may have.

A primary challenge when designing a DP solution is figuring out what sub problems can help solve the main problem. For simpler problems, the subproblems often have the same form as the actual problem.

DP Methodologies

Dynamic programming problems can generally be solved using one of two approaches:

1. **Top-Down Approach (Memoization)**
   This approach begins with the final problem and recursively computes the sub problems when needed.
   - Memorization is the technique used here, which means "remembering". The idea is to remember the computation of each value (like a Fibonacci number) and compute it at most once.
   - This is typically achieved by storing the computed value in a dictionary, often called a memo.
   - When the function is called, it first checks if the value is in the dictionary; if so, the computation is replaced by a single Dictionary lookup, avoiding the need to run the entire recursive computation again. This significantly speeds up solutions, such as converting an exponential-time recursive Fibonacci solution into a linear-time memoized solution

2. **Bottom-Up Approach (Tabulation)**
   This approach involves computing the sub problems first, starting from the base case, and working up to the final solution.

- Most people prefer the bottom-up approach because it typically uses iteration (like a for loop) instead of recursion, which means it doesn't have function calls and is often more efficient in practice.
- It may also allow for savings in memory in some cases (e.g., when solving the Fibonacci problem, only the last two values need to be remembered).
- When using the bottom-up method, attention must be paid to the order in which sub problems are solved. The dependencies of a subproblem must be solved first. If each subproblem is considered a node with edges representing dependencies, the sub problems must be solved in the topological sort order.

While the bottom-up approach has a lower constant factor and is slightly shorter to implement, thinking about DP problems in terms of recursive functions is often easier for the conceptualization phase.

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.

Code & Examples 2

Knapsack Problem

Given n items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible. The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

maxresdefault.jpg

Solution:

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the maximum profit in a knapsack
int knapsack(int W, vector<int> &val, vector<int> &wt) {

    // Initialize dp array with initial value 0
    vector<int> dp(W + 1, 0);

    // Iterate through each item, starting from the 1st item to the nth item
    for (int i = 1; i <= wt.size(); i++) {

        // Start from the back, so we also have data from
        // previous calculations of i-1 items and avoid duplication
        for (int j = W; j >= wt[i - 1]; j--) {
            dp[j] = max(dp[j], dp[j - wt[i - 1]] + val[i - 1]);
        }
    }
    return dp[W];
}

int main() {
    vector<int> val = {1, 2, 3};
    vector<int> wt = {4, 5, 1};
```

```
    int W = 4;

    cout << knapsack(W, val, wt) << endl;
    return 0;
}
```

Explanation:
The code above is already optimized as it saves memory usage and execution time. The analogy for this program is as follows:

- We have a bag (knapsack) with a certain capacity (W).
- We have several items with certain values (val) and weights (wt).
- Our goal is to fill the bag with items so that the total value of items in the bag is maximized without exceeding the bag's capacity.
- We use a dynamic programming approach to solve this problem efficiently.
- First, we iterate through each item, and for each item, we iterate the bag capacity from back to front (e.g., from maximum weight to 0).
- This way, we will find the maximum value that can be put into the bag without exceeding its capacity.
- The code already covers the situation of comparing between using the item with the highest weight or adding an item with a lower weight with the remaining capacity.