DP 알고리즘 - 초급

DP 알고리즘의 기초를 C++으로 구현한 전체 정리로, top-down (재귀 + 메모이제이션) 과 bottom-up (반복 + 타뷸레이션) 을 각각의 문제에 대해 명확히 구분하여 제공합니다.

📚 DP 알고리즘 - 기초 정리 (C++ 기반)

DP는 **최적 하위 구조**와 **중복 서브문제**를 가지는 문제를 효율적으로 해결하기 위한 기법입니다.
이 정리는 **피보나치, 0/1 배낭, LCS** 문제를 기반으로, **top-down**과 **bottom-up** 방식을 각각 구현하여 비교하고 설명합니다.

🔍 핵심 개념 정리

항목	설명
Top-down	재귀적으로 문제를 나누고, 이미 계산한 서브문제를 `unordered_map`에 저장 (메모이제이션)
Bottom-up	작은 문제부터 점진적으로 풀며, DP 테이블을 채움 (타뷸레이션)
메모이제이션	중복 계산 방지 → `O(n)` 대신 `O(n)`으로 최적화
타뷸레이션	반복 구조로 풀기 → 스택 오버플로우 없음, 메모리 안정

📌 1. 피보나치 (Fibonacci)

✅ Top-down (재귀 + 메모이제이션)

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

int fibTopDown(int n, unordered_map<int, int>& memo) {
    if (n <= 1) return n;
    if (memo.find(n) != memo.end()) {
        return memo[n];
    }
    return memo[n] = fibTopDown(n-1, memo) + fibTopDown(n-2, memo);
}
```

```cpp
int main() {
    unordered_map<int, int> memo;
    int n = 10;
    cout << "Fib(" << n << ") = " << fibTopDown(n, memo) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n), 공간 복잡도: O(n)
> ⚠️ 재귀 깊이가 크면 스택 오버플로우 가능성 있음

✅ Bottom-up (반복)

```cpp
int fibBottomUp(int n) {
    if (n <= 1) return n;
    int prev2 = 0, prev1 = 1;
    for (int i = 2; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

> ✅ 시간 복잡도: O(n), 공간 복잡도: O(1)
> ✅ 더 안정적, 메모리 효율적

📌 2. 0/1 배낭 문제 (0/1 Knapsack)

✅ Top-down (재귀 + 메모이제이션)

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

int knapsackTopDown(vector<int>& weights, vector<int>& values, int W, int
idx, unordered_map<string, int>& memo) {
    if (idx == -1 || W == 0) return 0;
    string key = to_string(idx) + "," + to_string(W);
    if (memo.find(key) != memo.end()) {
        return memo[key];
    }
```

```
    int currWeight = weights[idx];
    int currValue = values[idx];

    int notTake = knapsackTopDown(weights, values, W, idx - 1, memo);
    int take = 0;
    if (currWeight <= W) {
        take = currValue + knapsackTopDown(weights, values, W -
currWeight, idx - 1, memo);
    } else {
        take = 0;
    }

    return memo[key] = max(notTake, take);
}

int main() {
    vector<int> weights = {2, 1, 3, 2};
    vector<int> values = {12, 10, 20, 15};
    int W = 5;

    unordered_map<string, int> memo;
    cout << "Max value (top-down): " << knapsackTopDown(weights, values,
W, (int)weights.size()-1, memo) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n×W), 공간 복잡도: O(n×W)
> ⚠️ `string key` 를 사용해 중복 방지 (서브문제를 고유하게 식별)

✅ Bottom-up (반복)

```
int knapsackBottomUp(vector<int>& weights, vector<int>& values, int W) {
    int n = weights.size();
    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (weights[i-1] <= w) {
                dp[i][w] = max(
                    dp[i-1][w],  // 물건 안 넣기
                    dp[i-1][w - weights[i-1]] + values[i-1]  // 물건 넣기
                );
            } else {
                dp[i][w] = dp[i-1][w];
            }
```

```
        }
    }
    return dp[n][W];
}
```

> ✅ 시간 복잡도: O(n×W), 공간 복잡도: O(n×W)
> ✅ 더 안정적, 스택 오버플로우 없음

📌 3. 최장 공통 부분 문자열 (LCS)

✅ Top-down (재귀 + 메모이제이션)

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

int lcsTopDown(const string& s1, const string& s2, int i, int j,
unordered_map<string, int>& memo) {
    if (i == 0 || j == 0) return 0;
    string key = to_string(i) + "," + to_string(j);
    if (memo.find(key) != memo.end()) {
        return memo[key];
    }

    if (s1[i-1] == s2[j-1]) {
        memo[key] = 1 + lcsTopDown(s1, s2, i-1, j-1, memo);
    } else {
        memo[key] = max(
            lcsTopDown(s1, s2, i-1, j, memo),
            lcsTopDown(s1, s2, i, j-1, memo)
        );
    }
    return memo[key];
}

int main() {
    string s1 = "ABCDGH";
    string s2 = "AEDFHR";
    unordered_map<string, int> memo;
    cout << "LCS length (top-down): " << lcsTopDown(s1, s2, s1.size(),
s2.size(), memo) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(m×n), 공간 복잡도: O(m×n)
> ✅ 재귀 구조가 자연스럽지만, `m×n` 메모리 사용

✅ Bottom-up (반복)

```cpp
int lcsBottomUp(const string& s1, const string& s2) {
    int m = s1.size();
    int n = s2.size();
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n];
}
```

> ✅ 시간 복잡도: O(m×n), 공간 복잡도: O(m×n)
> ✅ 가장 일반적으로 사용되는 방식

📊 비교 요약 (C++ 기반)

문제	Top-down	Bottom-up
피보나치	재귀 + `unordered_map`	반복 + O(1) 공간
0/1 배낭	재귀 + `memo` 키	반복 + 2D 배열
LCS	재귀 + `i,j` 키	반복 + 2D DP 테이블

특성	Top-down	Bottom-up
읽기 쉬움	자연스럽게 문제 구조 반영	반복 구조가 명확함
메모리	O(n×W) 또는 O(m×n)	O(n×W) 또는 O(m×n)
스택 오버플로우	가능 (재귀 깊이 제한)	없음
성능	일반적으로 동일	더 안정적, 더 효율적

✅ 핵심 팁

- **Top-down**은 문제의 자연스러운 구조를 반영하기 위해 유용하지만, **재귀 깊이 제한**과 **스택 오버플로우**에 주의.
- **Bottom-up**은 **더 안정**, **더 효율**, **더 확장 가능**하며, 대부분의 경우 **기준으로 사용**.
- `unordered_map`을 사용해 `key = "i,j"` 형태로 서브문제를 식별하면 재귀 메모이제이션 구현 가능.

📚 학습 순서 제안 (C++ 기반)

1. 피보나치 (기초)
2. 0/1 배낭 (물체 선택 문제)
3. LCS (문자열 문제)
4. 최단 경로 (다익스트라, 플로이드-워셜)
