DP 알고리즘 - 중급

더 복잡한 **DP 문제**와 **기법의 확장**에 초점을 맞추며, **재귀 + 메모이제이션**, **bottom-up 타뷸레이션**, **최적 하위 구조 활용**, **서브문제의 조합** 등에 대한 깊이 있는 설명을 포함합니다.

📚 DP 알고리즘 - 중급 (C++ 기반)

> ✅ 이 정리는 **기초 문제를 넘어, 중급 수준의 DP 문제**를 다루며, **최적 하위 구조**, **서브문제의 조합**, **메모이제이션/타뷸레이션의 응용**, **상황에 맞는 접근 방식 선택**에 초점을 맞춥니다.

🔍 핵심 개념 확장

항목	설명
최적 하위 구조	전체 문제의 최적 해가 부분 문제의 최적 해의 조합으로 구성됨 (예: 최단 경로, 최대 합)
중복 서브문제	동일한 서브문제가 여러 번 재발생 (메모이제이션 필요)
DP 상태 정의	`dp[i][j]` 또는 `dp[i][j][k]` 형태로 문제를 정의하고, **상태 전이 방정식**을 도출
상태 압축 (Space Optimization)	2D DP를 1D로 줄이기 (예: 1D 배열로 최신 상태만 유지)

📌 1. 0/1 배낭 문제 (2D → 1D 최적화)

> ✅ **중급 확장**: 무게 제한에서 **물품 수 제한 추가** 또는 **무게 제한을 넘지 않도록 조건 추가**

🎯 문제 정의

- `n` 개의 물건, 각각 무게 `w[i]`, 가치 `v[i]`
- 최대 무게 `W` 제한
- 각 물건은 **0개 또는 1개만 선택 가능**
- 최대 가치를 구하라

✅ Bottom-up (2D → 1D 최적화)

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

```cpp
using namespace std;

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

    for (int i = 0; i < n; i++) {
        for (int w = W; w >= weights[i]; w--) {
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }
    return dp[W];
}

int main() {
    vector<int> weights = {2, 1, 3, 2};
    vector<int> values = {12, 10, 20, 15};
    int W = 5;
    cout << "Max value (optimized): " << knapsackOptimized(weights,
values, W) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n×W), 공간 복잡도: O(W)
> ✅ **2D → 1D 압축**으로 메모리 절반 줄임
> 💡 **역순 반복**은 `dp[w]`가 `w`가 커질수록 기존 값을 보존하기 위해 필요

📌 2. 부분 합 (Subset Sum) 문제

> ✅ **중급 문제**: 주어진 수열에서 **합이 특정 값**이 되는 부분 집합 존재 여부 확인

🎯 문제 정의

- 배열 `nums`, 목표 합 `target`
- 부분 집합의 합이 `target`이 되는지 여부를 판단

✅ Bottom-up (DP 테이블 활용)

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

bool subsetSum(vector<int>& nums, int target) {
    vector<bool> dp(target + 1, false);
    dp[0] = true;  // 0을 만들 수 있음
```

```
    for (int num : nums) {
        for (int j = target; j >= num; j--) {
            dp[j] = dp[j] || dp[j - num];
        }
    }
    return dp[target];
}

int main() {
    vector<int> nums = {3, 34, 4, 12, 2};
    int target = 4;
    if (subsetSum(nums, target)) {
        cout << "Subset sum exists!" << endl;
    } else {
        cout << "No subset sum found." << endl;
    }
    return 0;
}
```

> ✅ 시간 복잡도: O(n×target), 공간 복잡도: O(target)
> ✅ `dp[j]` = 합이 j가 되는 부분 집합 존재 여부

📌 3. 최대 부분 합 (Maximum Subarray Sum)

> ✅ **중급 문제**: 배열에서 **연속된 최대 합**을 찾는 문제 (Kadane 알고리즘)

🎯 문제 정의

- 배열 `nums`에서 연속된 부분 배열의 합 중 최댓값 찾기

✅ DP 기반 풀이 (DP 상태: `dp[i]` = `nums[0..i]`에서 시작한 최대 부분 합)

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

int maxSubarraySum(vector<int>& nums) {
    int maxSum = nums[0];
    int currentSum = nums[0];

    for (int i = 1; i < nums.size(); i++) {
        currentSum = max(nums[i], currentSum + nums[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

```
int main() {
    vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "Max subarray sum: " << maxSubarraySum(nums) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n), 공간 복잡도: O(1)
> ✅ **DP의 최소 형태**로, `currentSum`은 `dp[i]`의 최신 상태

📌 4. 행복한 수열 (Fibonacci + 조건 제약)

> ✅ **중급 확장**: 피보나치에 **조건 제약 추가** (예: 짝수만 선택, 특정 위치에서 제한)

🎯 문제 정의

- 피보나치 수열에서 **홀수 인덱스만 포함**하는 최대 합 구하기

✅ Top-down + 메모이제이션 (조건 추가)

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

int fibWithCondition(int n, unordered_map<int, int>& memo) {
    if (n <= 1) return n;
    if (memo.find(n) != memo.end()) return memo[n];

    // 조건: n이 홀수이면 1/2 더 추가
    if (n % 2 == 1) {
        return memo[n] = fibWithCondition(n-1, memo) + fibWithCondition(n-
2, memo) + 1;
    } else {
        return memo[n] = fibWithCondition(n-1, memo) + fibWithCondition(n-
2, memo);
    }
}

int main() {
    unordered_map<int, int> memo;
    cout << "Fib with condition (n=6): " << fibWithCondition(6, memo) <<
endl;
    return 0;
}
```

> ✅ 이는 **상황에 따라 DP 상태를 조건에 따라 변경**할 수 있음을 보여줌
> 💡 **조건 DP**는 **문제의 구조를 더 정확히 반영** 가능

📌 5. 행렬의 최대 경로 (Matrix DP)

> ✅ 중급 문제: 2D 행렬에서 **상하좌우 이동**하며 최대 합 경로 찾기

🎯 문제 정의

- 2D 행렬 `mat`에서 `(0,0)`에서 `(n-1, m-1)`까지 이동 (4방향), 경로의 합 최대

✅ Bottom-up (2D DP)

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

int maxPathSum(vector<vector<int>>& mat) {
    int n = mat.size();
    int m = mat[0].size();
    vector<vector<int>> dp(n, vector<int>(m, 0));

    dp[0][0] = mat[0][0];

    // 첫째 행
    for (int j = 1; j < m; j++) {
        dp[0][j] = dp[0][j-1] + mat[0][j];
    }

    // 첫째 열
    for (int i = 1; i < n; i++) {
        dp[i][0] = dp[i-1][0] + mat[i][0];
    }

    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {
            dp[i][j] = mat[i][j] + max(dp[i-1][j], dp[i][j-1]);
        }
    }

    return dp[n-1][m-1];
}

int main() {
    vector<vector<int>> mat = {
        {1, 3, 1},
        {2, 2, 3},
```

```
        {3, 4, 1}
    };
    cout << "Max path sum: " << maxPathSum(mat) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n×m), 공간 복잡도: O(n×m)
> ✅ **2D DP의 대표적 응용** (경로 문제)

📊 중급 DP 문제 요약

문제	핵심 기법	특징
0/1 배낭 (최적화)	1D 압축	메모리 최적화
부분 합	집합 생성	조합 문제 해결
최대 부분 합	Kadane	최단 경로 유사
행복한 수열	조건 DP	상태에 따라 분기
행렬 경로	2D DP	2D 상태 전이

📌 핵심 팁 (중급 수준)

1. **상태 정의는 핵심**
   - `dp[i][j]` 는 `i` 번째 원소에서 `j` 번째 상태까지의 최적 해를 의미
2. **상태 전이 방정식**
   - `dp[i][j] = max(dp[i-1][j], dp[i][j-1])` 처럼 **이전 상태와 비교**하여 최적 선택
3. **메모이제이션 vs 타뷸레이션**
   - **Top-down**: 문제 구조가 복잡할 때 유리 (재귀 구조)
   - **Bottom-up**: 메모리/시간 효율성 우선 (반복 구조)
4. **상태 압축**
   - 2D DP → 1D로 줄이기 (예: 0/1 배낭)

📚 학습 순서 제안 (중급 수준)

1. 0/1 배낭 (1D 최적화)
2. 부분 합 (Subset Sum)
3. 최대 부분 합 (Kadane)
4. 행렬 경로 (2D DP)

5. 조건 DP (예: 피보나치 조건 추가)

✅ 추가 팁

- **문제를 "최적 하위 구조"로 분해**하라.
  → "이 문제는 부분 문제의 최적 해를 조합하면 전체 최적 해가 된다"고 생각
- **테스트 케이스를 먼저 작성**하고, 그에 맞춰 DP 테이블을 채운다.
- **메모이제이션 키를 고유하게** (예: `i,j` 또는 `i,j,w`) 정의하여 중복 방지
