DP 알고리즘 - 고급

📚 DP 알고리즘 - 고급 (LCS, LIS 기반)

> ✅ 이 정리는 **LCS**(Longest Common Subsequence)과 **LIS**(Longest Increasing Subsequence)에 대한 **DP 풀이법**, **핵심 키 패턴**(Key Patterns), **상태 정의**, **전이 방정식**, **효율성 최적화**를 중심으로 정리했습니다.

🔍 핵심 개념 확장

항목	설명
LCS (Longest Common Subsequence)	두 문자열의 공통 부분 문자열 중 가장 긴 것 (순서 유지)
LIS (Longest Increasing Subsequence)	배열에서 증가하는 부분 수열 중 가장 긴 것
DP 키 패턴	문제를 `dp[i][j]` 또는 `dp[i]` 형태로 정의하고, **상태 전이 방정식**을 도출
공통 구조	두 문제 모두 **서브문제를 기반으로 최적 해를 조합**

📌 1. LCS (Longest Common Subsequence)

🎯 문제 정의

- 두 문자열 `s1`, `s2`에서 공통으로 나타나는 **순서를 유지한 부분 문자열** 중 가장 긴 것을 찾기

✅ Bottom-up (2D DP) - 기반 풀이

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

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++) {
```

```cpp
        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];
}

// 출력: 공통 부분 문자열 (추가로 추출 가능)
void printLCS(const string& s1, const string& s2) {
    int m = s1.size(), 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]);

    // LCS 문자열 재구성
    string lcs = "";
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (s1[i-1] == s2[j-1]) {
            lcs = s1[i-1] + lcs;
            i--; j--;
        } else if (dp[i-1][j] >= dp[i][j-1]) {
            i--;
        } else {
            j--;
        }
    }
    cout << "LCS: " << lcs << endl;
    cout << "Length: " << lcs.size() << endl;
}

int main() {
    string s1 = "ABCDGH";
    string s2 = "AEDFHR";
    cout << "LCS length: " << lcsBottomUp(s1, s2) << endl;
    printLCS(s1, s2);
    return 0;
}
```

> ✅ 시간 복잡도: O(m×n), 공간 복잡도: O(m×n)
> ✅ **2D DP 테이블**으로 공통 부분 문자열 길이 구함
> 💡 `dp[i][j]` = `s1[0..i-1]`과 `s2[0..j-1]`의 LCS 길이

✅ Top-down (재귀 + 메모이제이션)

```cpp
#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 깊이 제한 주의 (매우 큰 문자열에서는 불가)

📌 2. LIS (Longest Increasing Subsequence)

🎯 문제 정의

- 배열 `nums`에서 **증가하는 순서를 유지한 부분 수열** 중 가장 긴 것을 찾기

✅ O(n²) - DP 풀이 (2D DP)

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

int lisBottomUp(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    vector<int> dp(n, 1);  // dp[i] = nums[0..i]에서 끝나는 LIS 길이

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }
    return *max_element(dp.begin(), dp.end());
}

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << "LIS length: " << lisBottomUp(nums) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n²), 공간 복잡도: O(n)
> ✅ `dp[i]` = `nums[0..i]`에서 끝나는 LIS 길이

✅ O(n log n) - 최적화 버전 (Binary Indexed Tree / Patience Sorting)

> 💡 **LIS 최적화**: `O(n log n)`으로 풀 수 있음 (추가 알고리즘 필요)

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

int lisOptimized(vector<int>& nums) {
    vector<int> tails;  // tails[i] = LIS의 마지막 원소 중, 길이 i에 해당하는 최소
```

값

```
    for (int num : nums) {
        auto it = lower_bound(tails.begin(), tails.end(), num);
        if (it == tails.end()) {
            tails.push_back(num);
        } else {
            *it = num;
        }
    }
    return tails.size();
}

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << "LIS length (optimal): " << lisOptimized(nums) << endl;
    return 0;
}
```

> ✅ 시간 복잡도: O(n log n), 공간 복잡도: O(n)
> ✅ **이진 탐색 기반**으로 최적화 가능

📌 3. LCS와 LIS의 공통 키 패턴 (Key Patterns)

키 패턴	설명	적용 예
상태 정의	`dp[i][j]` = `i` 번째 원소까지, `j` 번째 원소까지의 최적 해	LCS, 2D 행렬 문제
전이 방정식	`dp[i][j] = max(dp[i-1][j], dp[i][j-1])` (공통)	LCS, 행렬 경로
조건 기반 선택	`if (s1[i-1] == s2[j-1])` → `+1`, 아니면 `max(상향, 횡행)`	LCS
서브문제 조합	`dp[i] = max(dp[j] + 1)` (조건: `nums[j] < nums[i]`)	LIS
역추적 (Backtracking)	`dp` 테이블을 사용해 실제 문자열/수열 재구성	LCS, LIS

📊 비교 요약 (LCS vs LIS)

항목	LCS	LIS
문제 유형	문자열	배열
순서 유지	순서 유지 (subsequence)	순서 유지 (subsequence)
시간 복잡도	O(m×n)	O(n²) 또는 O(n log n)
공간 복잡도	O(m×n)	O(n)
응용	문자 비교, 버전 비교	정렬, 수열 분석

📌 핵심 팁 (고급 수준)

1. **문제를 "서브문제"로 분해**
   → "LCS는 두 문자열의 공통 부분을 찾는다" → "각 위치에서 같은 문자가 있으면 +1, 아니면 최대값 선택"
2. **전이 방정식은 항상 '최대' 또는 '합' 형태**
   - `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`
   - `dp[i] = max(dp[j] + 1)` (조건 만족 시)
3. **LIS는 O(n²)이지만, O(n log n)으로 최적화 가능**
   → `tails` 배열을 사용해 **증가하는 최소 끝 원소**를 추적
4. **LCS는 실제 문자열을 재구성 가능**
   → `dp` 테이블을 역추적하여 **정확한 문자열**을 출력 가능
5. **문제 유형에 따라 접근 방식 선택**
   - **문자열 문제**: LCS → 2D DP
   - **수열 문제**: LIS → 1D DP 또는 O(n log n) 최적화

📚 학습 순서 제안 (고급 수준)

1. **LCS** (기본 2D DP, 문자열 문제)
2. **LIS** (1D DP, 수열 문제)
3. **LIS 최적화 (O(n log n))**
4. **LCS 문자열 재구성**
5. **문제 유형에 따른 DP 키 패턴 정리**

✅ 추가 팁

- **문제를 "최적 하위 구조"로 분해**하라.
  → "LCS는 두 문자열의 공통 부분을 찾는다" → "각 위치에서 같은 문자가 있으면 +1, 아니면 이전

최대값 선택"

- **테스트 케이스를 먼저 작성**하고, 그에 맞춰 DP 테이블을 채운다.
- **메모이제이션 키를 고유하게** (예: `i,j` 또는 `i,j,w`) 정의하여 중복 방지
- **LIS는 `tails` 배열을 통해 O(n log n)으로 최적화 가능**
