06. Dynamic Programming

이화여자대학교 황채원

강의 목차

1. 동적 계획법이란?
2. 메모이제이션
3. 타뷸레이션
4. 분할 정복 vs DP
5. 백트래킹 vs DP
6. 응용문제 풀이

동적 계획법이란?

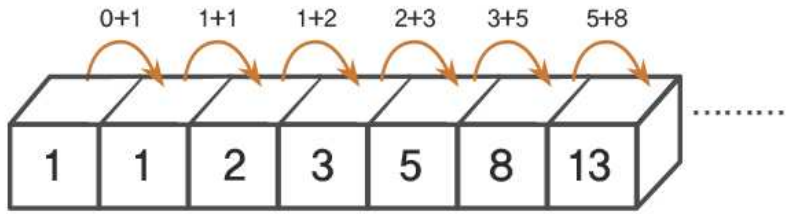
복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법

동적 계획법의 두 가지 조건

1. 중복되는 부분 문제(Overlapping Subproblem)
2. 최적 부분 구조(Optimal Substructure)

동적 계획법이란?

피보나치 수열을 구해봅시다.

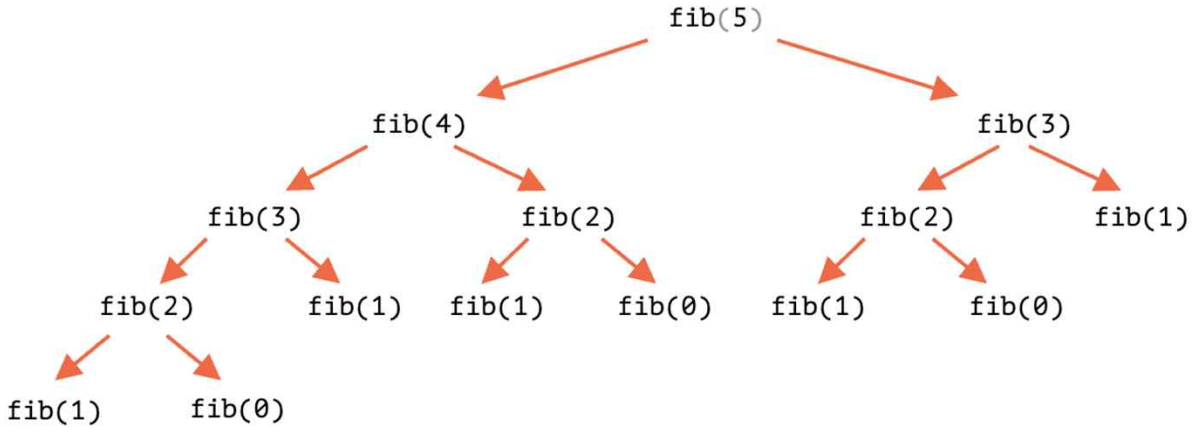


https://www.google.com/url?sa=i&url=https%3A%2F%2Fnews.samsungdisplay.com%2F23402&psig=AOvVaw02mpZ8Ym49m1yitfL8J01F&ust=1707337016045000&source=images&cd=vfe&opi=89978449&ved=0CBIQjRxqFwoTCNiepZTEl4QDFQAAAAAdAAAAABAE

중복되는 부분 문제

수열을 관찰하다보면 최종 해답까지 도달하기 위한 부분 문제들이 존재한다는 것을 발견할 수 있습니다.

전체 문제가 독립적인 부분 문제로 나뉘는 것이 아니라, 반복적으로 나타납니다.



https://velog.io/@sud665/%ED%94%BC%EB%B3%B4%EB%82%98%EC%B9%98%EC%88%98%EC%97%B4%EC%9D%98-%EA%B3%A0%EC%B0%B0

최적 부분 구조

전체 문제에 대한 최적해가 부분 문제의 최적해의 조합으로 표현될 수 있습니다.

fib(n)이 최적의 답이 되려면 부분 문제인 fib(n-1)과 fib(n-2)이 최적의 답이어야 합니다.

```
// 피보나치 수를 재귀적으로 계산하는 함수
int fibo(int n) {
    if (n >= 2)
        return fibo(n-1) + fibo(n-2);
    else
        return n;
}
```

동적 프로그래밍의 구현

탑다운 (Top-down)

큰 문제를 해결하기 위해 먼저 작은 하위 문제로 나누고, 이러한 하위 문제들을 재귀적으로 해결합니다.

➔ 메모이제이션

바텀업 (Bottom-up)

가장 작은 하위 문제부터 시작하여, 순차적으로 더 큰 문제를 해결해 나갑니다.

➔ 타뷸레이션

동적 프로그래밍의 구현

탑다운 방식은

문제를 이해하고 구현하기 쉬울 때 선호됩니다.

하지만 재귀의 깊이가 너무 깊어지면 성능 문제나 스택 오버플로우가 발생할 수 있습니다.

바텀업 방식은

재귀 호출에 대한 오버헤드가 없으며, 모든 중간 결과를 체계적으로 계산합니다.

대규모 문제나 스택 오버플로우의 위험이 있는 경우에 적합합니다.

메모이제이션(Memoization)

컴퓨터 프로그램이 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써

동일한 계산의 반복 수행을 제거하여 프로그램 실행 속도를 빠르게 하는 기술입니다.

https://ko.wikipedia.org/wiki/%EB%A9%94%EB%AA%A8%EC%9D%B4%EC%A0%9C%EC%9D%B4%EC%85%98

메모이제이션(Memoization)

1. 결과 저장: 함수의 결과값을 저장할 자료 구조를 생성합니다.

2. 결과 재사용: 함수가 호출될 때, 먼저 해당 입력값에 대한 결과가 저장된 자료 구조를 확인합니다.

- 만약 결과가 이미 저장되어 있다면, 저장된 값을 반환하여 불필요한 계산을 줄입니다.
- 결과가 저장되어 있지 않다면, 계산을 수행하고 그 결과를 자료 구조에 저장한 후 결과를 반환합니다.

단, 재귀 함수 호출 횟수가 많아질 경우 스택 오버플로우가 발생할 수 있으니 주의해야 합니다.

https://ko.wikipedia.org/wiki/%EB%A9%94%EB%AA%A8%EC%9D%B4%EC%A0%9C%EC%9D%B4%EC%85%98

메모이제이션(Memoization)

피보나치 수열을 구하는 문제에 메모이제이션을 적용해보겠습니다.

```cpp
std::vector<long long> memo;

long long fibo(int n) {
    // 기저 사례: n이 0 또는 1일 때
    if (n <= 1) return n;

    // 이미 계산된 값이 있다면 반환
    if (memo[n] != -1) return memo[n];

    // 재귀 호출을 통해 fibo(n-1)과 fibo(n-2)의 값을 계산하고, 결과를 memo에 저장
    memo[n] = fibo(n-1) + fibo(n-2);

    // 계산된 값을 반환
    return memo[n];
}
```

타뷸레이션(Tabulation)

계산 결과를 단계별로 테이블에 저장함으로써

중복 계산을 방지하고 프로그램의 실행 속도를 빠르게 하는 기술입니다.

작은 문제부터 차례대로 해결해 나가며 최종 결과를 구합니다.

재귀 호출 없이 반복문을 사용하여 구현되며, 모든 필요한 계산을 명시적으로 수행합니다.

타뷸레이션(Tabulation)

1. 테이블 초기화: 문제의 해결을 위한 테이블(주로 배열)을 생성하고 초기화합니다.
2. 하위 문제 해결: 가장 작은 하위 문제부터 시작하여, 순차적으로 모든 하위 문제를 해결합니다.

   각 하위 문제의 결과는 테이블에 저장됩니다.
3. 결과 사용: 하위 문제의 결과를 테이블에서 직접 참조하여 상위 문제를 해결합니다.
4. 최종 결과 도출: 테이블에 저장된 결과를 활용하여 최종 문제의 해결책을 도출합니다.

타뷸레이션(Tabulation)

1. 재귀 호출 없음:

   타뷸레이션은 재귀 호출을 사용하지 않고 반복문을 통해 구현됩니다.

   이로 인해 스택 오버플로우의 위험이 없으며, 프로그램의 성능을 개선할 수 있습니다.

2. 계산 과정의 명시성:

   모든 계산 과정이 명시적으로 테이블에 기록되기 때문에 디버깅하기가 쉽습니다.

3. 효율적인 메모리 사용:

   필요한 모든 하위 문제의 결과를 효율적으로 저장하고 관리할 수 있어, 메모리 사용을 최적화할 수 있습니다.

타뷸레이션(Tabulation)

피보나치 수열을 구하는 문제에 타뷸레이션을 적용해보겠습니다.

```cpp
int fibo(int n) {
    if (n <= 1) return n;

    std::vector<int> table(n+1, 0); // n+1 크기의 벡터를 0으로 초기화
    table[1] = 1; // 초기 조건 설정

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i-1] + table[i-2]; // 피보나치 수열 계산
    }

    return table[n];
}
```

분할정복

상위 문제를 하위 문제로 나누어 해결하는 Top-down approach 입니다.

- 분할(Divide): 원래 문제를 작은 하위 문제로 분할합니다.
- 정복(Conquer): 분할된 작은 문제들을 재귀적으로 해결합니다.
- 결합(Combine): 정복된 문제들의 해를 결합하여 원래 문제의 해를 얻습니다.

분할정복 vs 동적 프로그래밍

분할 정복(Divide and Conquer)

중복 하위 문제: 하위 문제들은 서로 독립적입니다. 각 하위 문제는 한 번씩만 해결됩니다.

적용 예시: 병합 정렬, 퀵 정렬, 이진 검색

동적 프로그래밍(Dynamic Programming)

중복 하위 문제: 중복되는 하위 문제들의 결과를 저장하고 재사용함으로써 계산량을 크게 줄일 수 있습니다.

적용 예시: 피보나치 수열, 배낭 문제, 최장 공통 부분 수열(Longest Common Subsequence)

백트래킹

현재 상태에서 가능한 모든 후보군을 따라 들어가며 해결책에 대한 후보를 구축해 나아가다

가능성이 없다고 판단되면 즉시 후보를 포기하면서 정답을 찾아가는 알고리즘입니다.

유망하지 않은 노드에 가지 않는 것을 가지치기(pruning)라고 합니다.

백트래킹 vs 동적 계획법

백트래킹

- 가능한 모든 해를 탐색하되, 실패가 확실한 경로는 조기에 포기합니다.
- 하위 문제의 중복 없이 각 경우를 개별적으로 탐색합니다.
- 해의 조합을 찾아야 하는 복잡한 결정 문제나 최적화 문제에 주로 사용됩니다.

동적 프로그래밍

- 문제를 작은 부분으로 나누어 해결하고, 중복 계산을 방지하기 위해 결과를 저장합니다.
- 중복되는 하위 문제의 결과를 저장하여 재사용함으로써 효율성을 극대화합니다.
- 중복되는 하위 문제가 많고, 최적 부분 구조를 가진 문제에 적합합니다.

BOJ 12865 (평범한 배낭)

평범한 배낭 성공

☆

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
2 초	512 MB	124412	46347	29720	35.789%

문제

이 문제는 아주 평범한 배낭에 관한 문제이다.

한 달 후면 국가의 부름을 받게 되는 준서는 여행을 가려고 한다. 세상과의 단절을 슬퍼하며 최대한 즐기기 위한 여행이기 때문에, 가지고 다닐 배낭 또한 최대한 가치 있게 싸려고 한다.

준서가 여행에 필요하다고 생각하는 N개의 물건이 있다. 각 물건은 무게 W와 가치 V를 가지는데, 해당 물건을 배낭에 넣어서 가면 준서가 V만큼 즐길 수 있다. 아직 행군을 해본 적이 없는 준서는 최대 K만큼의 무게만을 넣을 수 있는 배낭만 들고 다닐 수 있다. 준서가 최대한 즐거운 여행을 하기 위해 배낭에 넣을 수 있는 물건들의 가치의 최댓값을 알려주자.

입력

첫 줄에 물품의 수 N($1 \le N \le 100$)과 준서가 버틸 수 있는 무게 K($1 \le K \le 100,000$)가 주어진다. 두 번째 줄부터 N개의 줄에 거쳐 각 물건의 무게 W($1 \le W \le 100,000$)와 해당 물건의 가치 V($0 \le V \le 1,000$)가 주어진다.

입력으로 주어지는 모든 수는 정수이다.

출력

한 줄에 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 출력한다.

BOJ 12865 (평범한 배낭)

이 문제는 전형적인 "배낭 문제(Knapsack Problem)"로, 동적 프로그래밍을 사용하여 해결할 수 있습니다.

N: 사용 가능한 물건의 수 (1 ≤ N ≤ 100)

K: 배낭이 견딜 수 있는 최대 무게 (1 ≤ K ≤ 100,000)

W: 각 물건의 무게 (1 ≤ W ≤ 100,000)

V: 각 물건의 가치 (0 ≤ V ≤ 1,000)

목표: 주어진 무게 제한(K) 아래에서 물건들을 배낭에 넣어 얻을 수 있는 최대 가치의 합을 찾습니다.

BOJ 12865 (평범한 배낭)

브루트 포스를 사용해볼까요?

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

int N, K;
vector<int> W, V;

int bruteForce(int i, int weight) {
    if (i == N) return 0;
    int notTaken = bruteForce(i + 1, weight);
    int taken = 0;
    if (weight + W[i] <= K) taken = V[i] + bruteForce(i + 1, weight + W[i]);
    return max(notTaken, taken);
}

int main() {
    cin >> N >> K;
    W.resize(N); V.resize(N);
    for (int i = 0; i < N; ++i) cin >> W[i] >> V[i];
    cout << bruteForce(0, 0) << endl;
    return 0;
}
```

배낭 문제에서는 각 물건을 배낭에 넣거나 넣지 않는 두 가지 선택이 있으므로, N개의 물건에 대해 2^N개의 조합이 존재합니다. 따라서, 브루트 포스 방식의 시간 복잡도는 O(2^N)이 되어, 물건의 수가 증가함에 따라 계산 시간이 기하급수적으로 증가합니다.

BOJ 12865 (평범한 배낭)

백트래킹을 사용한다면?

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

int N, K;
vector<int> W, V;
int maxValue = 0;

void backtrack(int i, int currentWeight, int currentValue) {
    if (currentWeight > K) return; // 무게가 K를 초과하는 경우 탐색 중단
    if (i == N) {
        maxValue = max(maxValue, currentValue); // 최대 가치 갱신
        return;
    }
    // 현재 물건을 넣는 경우
    backtrack(i + 1, currentWeight + W[i], currentValue + V[i]);
    // 현재 물건을 넣지 않는 경우
    backtrack(i + 1, currentWeight, currentValue);
}

int main() {
    cin >> N >> K;
    W.resize(N); V.resize(N);
    for (int i = 0; i < N; ++i) {
        cin >> W[i] >> V[i];
    }
    backtrack(0, 0, 0);
    cout << maxValue << endl;
    return 0;
}
```

백트래킹은 모든 가능한 조합을 탐색하지만, 불가능한 조합(예: 배낭의 무게 한도를 초과하는 경우)을 조기에 배제합니다. 이는 브루트 포스에 비해 계산량을 줄일 수 있지만, 여전히 많은 수의 조합을 검토해야 합니다. 비록 백트래킹이 불필요한 조합을 조기에 배제하여 시간을 절약할 수 있지만, 최악의 경우에는 여전히 O(2^N)의 시간 복잡도를 가집니다.

BOJ 12865 (평범한 배낭)

왜 DP를 사용해야할까? 간단한 예시를 통해 알아봅시다.

배낭의 최대 무게(K)는 5kg이다.

사용 가능한 물건(N)은 3개이며, 각각의 무게(W)와 가치(V)는 다음과 같다:

물건	무게(W)	가치(V)
1	1kg	60
2	2kg	100
3	3kg	120

BOJ 12865 (평범한 배낭)

반복되는 하위 구조

물건	무게(W)	가치(V)
1	1kg	60
2	2kg	100
3	3kg	120

- 물건 1을 넣지 않을 경우, 남은 무게는 5kg이며,

  이후 물건 2와 3을 고려해야 합니다.

- 물건 1을 넣을 경우, 남은 무게는 4kg이며, 이후 물건 2와 3을 고려해야 합니다.

이렇게 각 단계에서 물건을 넣거나 넣지 않는 선택에 따라 생성되는 하위 문제들은 서로 겹치는 경우가 발생합니다. 예를 들어, 물건 2를 고려하는 시점에서 남은 무게가 3kg인 상황은 물건 1을 넣었을 때와 넣지 않았을 때 모두 발생할 수 있습니다.

BOJ 12865 (평범한 배낭)

최적 부분 구조

물건	무게(W)	가치(V)
1	1kg	60
2	2kg	100
3	3kg	120

배낭 문제의 해결책은 하위 문제의 최적 해결책에서 유도될 수 있습니다.

예를 들어, 물건 2와 3만을 고려하는 하위 문제에서 최대 가치가 계산되었다면, 이 값은 물건 1을 고려할 때의 최적 해결책을 찾는 데 사용될 수 있습니다. 즉, 각 단계에서의 최적 해결책은 이전 단계의 최적 해결책을 기반으로 합니다.

BOJ 12865 (평범한 배낭)

동적 프로그래밍의 접근법은? (타뷸레이션)

- 각 물품에 대해, 이 물품을 현재 배낭의 무게 한도 내에 넣을 수 있는지, 그리고 넣는다면 가치를 최대화할 수 있는지 판단합니다.
- 각 무게에 대해 배낭에 담을 수 있는 물품들의 최대 가치합을 저장합니다.
- 배낭의 무게를 1부터 최대 무게(K)까지 증가시키면서, 각 무게에서 가능한 최대 가치합을 계산합니다.

  각 단계에서, 모든 물품을 고려하여 해당 물품을 배낭에 추가할지 말지 결정합니다.
- 모든 물품과 모든 무게에 대해 최대 가치합을 계산한 후, 최대 무게(K)에서의 최대 가치합이 배낭에 담을 수 있는 물품들의 최대 가치합이 됩니다.

BOJ 12865 (평범한 배낭)

점화식을 유도해봅시다.

- dp[w]를 무게 w까지의 배낭에 담을 수 있는 물품들의 최대 가치합으로 정의합니다.
- 각 물품 i에 대해서, 두 가지 경우를 고려합니다:
  1. 물품 i를 배낭에 넣는 경우: 최대 가치= 물품 i의 가치 + (현재 무게 - 물품 i의 무게)까지의 최대 가치
  2. 물품 i를 배낭에 넣지 않는 경우: 이 경우의 최대 가치는 현재 무게까지의 최대 가치와 동일합니다.

dp[w] = max(dp[w], dp[w - 물품 i의 무게] + 물품 i의 가치) (물품 i를 배낭에 넣을 수 있는 경우)

- 이 점화식은 각 물품을 고려할 때마다 배낭의 각 가능한 무게에 대해 최대 가치합을 갱신합니다.

BOJ 12865 (평범한 배낭)


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

int knapsack(int N, int K, vector<int>& W, vector<int>& V) {
    vector<vector<int>> dp(N+1, vector<int>(K+1, 0));

    // 모든 물품에 대해 반복
    for (int i = 1; i <= N; ++i) {
        // 현재 물품을 고려하여 배낭의 무게를 1부터 K까지 증가시키며 최대 가치합을 계산
        for (int w = 1; w <= K; ++w) {
            // 현재 물품의 무게가 w보다 크면, 이 물품을 배낭에 넣을 수 없으므로 이전 단계의 값을 그대로 사용
            if (W[i-1] > w) {
                dp[i][w] = dp[i-1][w];
            } else {
                // 현재 물품을 배낭에 넣는 경우와 넣지 않는 경우 중 최대 가치를 선택
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-W[i-1]] + V[i-1]);
            }
        }
    }

    // 최대 무게에서의 최대 가치합 반환
    return dp[N][K];
}
```

```cpp
int main() {
    int N, K;
    cin >> N >> K;
    vector<int> W(N), V(N);

    // 물품의 무게와 가치 입력 받기
    for (int i = 0; i < N; ++i) {
        cin >> W[i] >> V[i];
    }

    cout << knapsack(N, K, W, V) << "\n";

    return 0;
}
```

BOJ 12865 (평범한 배낭)

메모이제이션 방식으로도 해결해봅시다.

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

vector<int> W, V;
vector<vector<int>> memo;

int knapsack(int i, int w, int N) {
    if (i == N || w == 0) return 0; // 물품이 더 이상 없거나, 배낭의 무게가 0일 때
    if (memo[i][w] != -1) return memo[i][w]; // 이미 계산된 하위 문제의 경우

    // 현재 물품을 넣지 않는 경우
    int notTaken = knapsack(i + 1, w, N);

    // 현재 물품을 넣을 수 있는 경우
    int taken = 0;
    if (W[i] <= w) {
        taken = V[i] + knapsack(i + 1, w - W[i], N);
    }

    // 현재 물품을 넣는 경우와 넣지 않는 경우 중 최댓값을 선택
    return memo[i][w] = max(notTaken, taken);
}
```

```cpp
int main() {
    int N, K;
    cin >> N >> K;
    W.resize(N);
    V.resize(N);
    memo.assign(N, vector<int>(K+1, -1)); // 배열 초기화

    for (int i = 0; i < N; ++i) {
        cin >> W[i] >> V[i];
    }

    cout << knapsack(0, K, N) << "\n";

    return 0;
}
```

BOJ 11660 (구간 합 구하기 5)

구간 합 구하기 5 성공

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	64909	29756	22208	44.163%

문제

N×N개의 수가 N×N 크기의 표에 채워져 있다. (x1, y1)부터 (x2, y2)까지 합을 구하는 프로그램을 작성하시오. (x, y)는 x행 y열을 의미한다.

예를 들어, N = 4이고, 표가 아래와 같이 채워져 있는 경우를 살펴보자.

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

여기서 (2, 2)부터 (3, 4)까지 합을 구하면 3+4+5+4+5+6 = 27이고, (4, 4)부터 (4, 4)까지 합을 구하면 7이다.

표에 채워져 있는 수와 합을 구하는 연산이 주어졌을 때, 이를 처리하는 프로그램을 작성하시오.

입력

첫째 줄에 표의 크기 N과 합을 구해야 하는 횟수 M이 주어진다. (1 ≤ N ≤ 1024, 1 ≤ M ≤ 100,000) 둘째 줄부터 N개의 줄에는 표에 채워져 있는 수가 1행부터 차례대로 주어진다. 다음 M개의 줄에는 네 개의 정수 x1, y1, x2, y2 가 주어지며, (x1, y1)부터 (x2, y2)의 합을 구해 출력해야 한다. 표에 채워져 있는 수는 1,000보다 작거나 같은 자연수이다. (x1 ≤ x2, y1 ≤ y2)

출력

총 M줄에 걸쳐 (x1, y1)부터 (x2, y2)까지 합을 구해 출력한다.

BOJ 11660 (구간 합 구하기 5)

주어진 2차원 배열에서 구간 합을 구하는 문제입니다.

- N: 2차원 배열의 크기 (N×N).
- M: 합을 구해야 하는 횟수.

합을 구해야 하는 범위: M개의 쿼리로 주어지며, 각 쿼리는 (x1, y1, x2, y2) 형태로 주어진다.

목표: 각 쿼리에 대한 주어진 범위의 합을 M줄에 걸쳐 출력.

BOJ 11660 (구간 합 구하기 5)

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

예제 입력 1 복사

```
4 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
2 2 3 4
3 4 3 4
1 1 4 4
```

예제 출력 1 복사

```
27
6
64
```

BOJ 11660 (구간 합 구하기 5)

예제 입력 1 복사

```
4 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
2 2 3 4
3 4 3 4
1 1 4 4
```

예제 출력 1 복사

```
27
6
64
```

BOJ 11660 (구간 합 구하기 5)

예제 입력 1 복사

```
4 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
2 2 3 4
3 4 3 4
1 1 4 4
```

예제 출력 1 복사

```
27
6
64
```

BOJ 11660 (구간 합 구하기 5)

브루트포스?

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

int main() {
    int N, M;
    cin >> N >> M;
    vector<vector<int>> A(N, vector<int>(N));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> A[i][j];
        }
    }
    while (M--) {
        int x1, y1, x2, y2, sum = 0;
        cin >> x1 >> y1 >> x2 >> y2;
        for (int i = x1 - 1; i < x2; ++i) {
            for (int j = y1 - 1; j < y2; ++j) {
                sum += A[i][j];
            }
        }
        cout << sum << endl;
    }
    return 0;
}
```

브루트 포스 방법은 각 쿼리마다 지정된 범위 내의 모든 수를 직접 더하는 방식으로 합을 계산합니다.

쿼리 하나를 처리하는 데 O(N^2)의 시간이 소요되며, 쿼리가 M개 주어졌을 경우 **전체 시간 복잡도는 O(MN^2)**이 됩니다. N과 M이 큰 경우, 이 방법은 매우 비효율적이며, 실행 시간이 지나치게 길어집니다.

BOJ 11660 (구간 합 구하기 5)

누적합

누적합은 일련의 데이터가 있을 때, 각 위치에서 시작점부터 해당 위치까지의 합을 미리 계산해 두는 방법입니다.

2차원 배열에서의 누적합을 구하는 문제에서는, 각 위치 (i, j)에서 왼쪽 상단 (1, 1)까지의 누적합을 저장하는

2차원 배열을 만듭니다.

이를 DP[i][j]라고 할 때, DP[i][j]는 다음과 같이 계산됩니다:

```
DP[i][j] = DP[i-1][j] + DP[i][j-1] - DP[i-1][j-1] + A[i][j]
```

BOJ 11660 (구간 합 구하기 5)

중복되는 부분 문제

중복의 발생: 특정 구간의 합은 여러 쿼리에 걸쳐 반복적으로 계산될 수 있습니다.

예를 들어, 어떤 쿼리에서 (1, 1)부터 (3, 3)까지의 합을 계산하고, 다른 쿼리에서는 (2, 2)부터 (4, 4)까지의 합을 계산할 때, (2, 2)부터 (3, 3)까지의 구간은 두 쿼리 모두에서 계산됩니다.

동적 프로그래밍의 적용:

누적 합 배열을 미리 계산하고 저장함으로써, 각 쿼리에서 구간 합을 계산할 때 중복되는 계산을 피할 수 있습니다.

BOJ 11660 (구간 합 구하기 5)

최적 부분 구조

최적 부분 구조의 활용: 누적 합 배열 구하기

각 위치에서 시작점 (1, 1)부터 해당 위치까지의 모든 값의 합을 저장합니다. 각 쿼리의 구간 합을 누적 합 배열을 활용하여 계산할 수 있습니다.

쿼리 (x1, y1)부터 (x2, y2)까지의 합은

누적합[x2][y2] - 누적합[x1-1][y2] - 누적합[x2][y1-1] + 누적합[x1-1][y1-1]로 계산

BOJ 11660 (구간 합 구하기 5)


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

using namespace std;

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  int N, M;
  cin >> N >> M;

  vector<vector<int>> chart(N + 1, vector<int>(N + 1, 0));
  vector<vector<int>> sumchart(N + 1, vector<int>(N + 1, 0));

  for (int i = 1; i <= N; i++) {
    int sum = 0;

    for (int j = 1; j <= N; j++) {
      cin >> chart[i][j];
      sum += chart[i][j];
      sumchart[i][j] = sum + sumchart[i - 1][j];
    }
  }

  int x1, x2, y1, y2;

  for (int i = 0; i < M; i++) {
    cin >> x1 >> y1 >> x2 >> y2;
    cout << sumchart[x2][y2] + sumchart[x1 - 1][y1 - 1] - sumchart[x2][y1 - 1] - sumchart[x1 - 1][y2] << "\n";
  }
  return 0;
}
```

BOJ 11053 (가장 긴 증가하는 부분 수열)

가장 긴 증가하는 부분 수열 성공

☆

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	158133	63177	41906	37.891%

문제

수열 A가 주어졌을 때, 가장 긴 증가하는 부분 수열을 구하는 프로그램을 작성하시오.

예를 들어, 수열 A = {10, 20, 10, 30, 20, 50} 인 경우에 가장 긴 증가하는 부분 수열은 A = {**10**, **20**, 10, **30**, 20, **50**} 이고, 길이는 4이다.

입력

첫째 줄에 수열 A의 크기 N (1 ≤ N ≤ 1,000)이 주어진다.

둘째 줄에는 수열 A를 이루고 있는 A_i가 주어진다. (1 ≤ A_i ≤ 1,000)

출력

첫째 줄에 수열 A의 가장 긴 증가하는 부분 수열의 길이를 출력한다.

BOJ 11053 (가장 긴 증가하는 부분 수열)

LIS(Longest Increasing Subsequence)

가장 긴 증가하는 부분 수열(Longest Increasing Subsequence, LIS) 문제는

주어진 수열에서 어떤 원소들을 순서대로 선택했을 때,

(1) 그 선택된 수열이 증가하는 순서를 유지하고

(2) 그 길이가 최대가 되게 하는 문제입니다.

여기서 부분 수열이란 주어진 수열의 일부 원소를 원래의 순서대로 선택해서 만들어낸 수열을 의미합니다.

BOJ 11053 (가장 긴 증가하는 부분 수열)

최적 부분 구조 (Optimal Substructure)

LIS 문제에서는 어떤 수열의 가장 긴 증가하는 부분 수열을 찾는 문제를 고려할 때, 이 문제의 해결 방법이

수열의 각 부분에 대한 가장 긴 증가하는 부분 수열의 해결 방법을 포함하고 있음을 볼 수 있습니다.

10	20	10	30	20	50

가능한 부분 수열: {10}

BOJ 11053 (가장 긴 증가하는 부분 수열)

최적 부분 구조 (Optimal Substructure)

LIS 문제에서는 어떤 수열의 가장 긴 증가하는 부분 수열을 찾는 문제를 고려할 때, 이 문제의 해결 방법이

수열의 각 부분에 대한 가장 긴 증가하는 부분 수열의 해결 방법을 포함하고 있음을 볼 수 있습니다.

10	20	10	30	20	50

가능한 부분 수열: {10}, {10, 20}

BOJ 11053 (가장 긴 증가하는 부분 수열)

최적 부분 구조 (Optimal Substructure)

LIS 문제에서는 어떤 수열의 가장 긴 증가하는 부분 수열을 찾는 문제를 고려할 때, 이 문제의 해결 방법이

수열의 각 부분에 대한 가장 긴 증가하는 부분 수열의 해결 방법을 포함하고 있음을 볼 수 있습니다.

10	20	10	30	20	50

가능한 부분 수열: {10}, {10, 20}, {10}

BOJ 11053 (가장 긴 증가하는 부분 수열)

최적 부분 구조 (Optimal Substructure)

LIS 문제에서는 어떤 수열의 가장 긴 증가하는 부분 수열을 찾는 문제를 고려할 때, 이 문제의 해결 방법이

수열의 각 부분에 대한 가장 긴 증가하는 부분 수열의 해결 방법을 포함하고 있음을 볼 수 있습니다.

10	20	10	30	20	50

가능한 부분 수열: {10}, {10, 20}, {10, 20, 30}

BOJ 11053 (가장 긴 증가하는 부분 수열)

중복되는 부분 문제 (Overlapping Subproblems)

DP를 사용하여 LIS 문제를 해결할 때, 각 원소를 마지막으로 하는 가장 긴 증가하는 부분 수열의 길이를 저장하는 배열을 사용합니다. 이 배열을 통해 한 번 계산한 부분 문제의 결과를 저장하고, 이후에 동일한 부분 문제가 발생하면 저장된 결과를 재사용함으로써 중복 계산을 피할 수 있습니다.

10	20	10	30	20	50

가능한 부분 수열: {10}, {10, 20}, {10, 20, 30}

BOJ 11053 (가장 긴 증가하는 부분 수열)


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

using namespace std;

int main() {
    int N; // 수열 A의 크기
    cin >> N;

    vector<int> A(N); // 수열 A
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }
```

```cpp
vector<int> dp(N, 1); // dp[i]: A[i]를 마지막으로 하는 가장 긴 증가하는 부분 수열의 길이

for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
        if (A[j] < A[i]) {
            dp[i] = max(dp[i], dp[j] + 1); // A[i] 앞의 원소들 중 A[i]보다 작은 것들을 고려하여 dp[i] 갱신
        }
    }
}

cout << *max_element(dp.begin(), dp.end()) << endl; // dp 배열에서 최댓값 찾기

return 0;
```

문제

필수 문제

- BOJ 11722 (가장 긴 감소하는 부분 수열)
- BOJ 2565 (전깃줄)
- BOJ 1535 (안녕)
- BOJ 11726 (2xn 타일링)
- BOJ 10844 (쉬운 계단 수)

심화 문제

- BOJ 9084 (동전)
- BOJ 9251 (LCS)
- BOJ 11066 (파일 합치기)
- BOJ 11049 (행렬 곱셈 순서)