#06-1. 동적 계획법

나정휘
https://justiceHui.github.io/

동적 계획법

동적 계획법

- 복잡한 문제를 간단한 문제들로 나누고
- 간단한 문제들을 해결한 뒤
- 간단한 문제들의 답을 이용해 복잡한 문제의 답을 구함

떠오르는 질문 리스트

- 문제의 복잡성이 뭐지?
  - 큰 문제 / 작은 문제로 생각하면 편함
- 항상 가능하지는 않을 텐데…
  - 최적 부분 구조 (Optimal Substructure)
- 이게 효율적일까?
  - 중복되는 부분 문제 (Overlapping Subproblem)

동적 계획법

최적 부분 구조 (Optimal Substructure)

- 큰 문제의 최적해가 작은 문제의 최적해를 포함한다.
- 최적 부분 구조가 아니라면 작은 문제의 답을 이용해서 큰 문제의 답을 구할 수 없음

- ex) 피보나치 수열
  - 큰 문제: fibonacci(N)
  - 작은 문제: fibonacci(0), fibonacci(1)
  - fibonacci(N)은 a*fibonacci(0) + b*fibonacci(1)로 나타낼 수 있음
- ex) 거스름돈 문제
  - 큰 문제: 12560원 만들기
  - 작은 문제: 2560원 만들기
  - 12560원을 만드는 것은 2560원을 만들고 10000원권 지폐 한 장을 추가하면 됨

동적 계획법

중복되는 부분 문제 (Overlapping Subproblem)

- 작은 문제의 답을 여러 번 참조한다.
- 한 번 계산한 답을 저장하면, 다시 참조할 때 저장된 값을 사용하면 되므로 연산량 감소

- ex) 피보나치 수열
  - 큰 문제: fibonacci(N), N > 5
  - 작은 문제: fibonacci(5), fibonacci(4), fibonacci(3), ...
  - fibonacci(N)은 a*fibonacci(5) + b*fibonacci(4)로 나타낼 수 있음
    - 이때 fibonacci(5)와 fibonacci(4)를 각각 a, b번 호출할 텐데
    - 한 번 계산한 다음 결과를 저장하면 실행 시간을 많이 줄일 수 있음
- ex) 거스름돈 문제
  - 큰 문제: 7560원 만들기, 12560원 만들기, 62560원 만들기
  - 작은 문제: 2560원 만들기
  - 2560원을 만드는 방법을 여러 번 참조함

질문

주어진 문제를 DP로 풀 수 있다는 것을 어떻게 알 수 있나요?

동적 계획법

동적 계획법 문제의 특징

- 큰 문제를 한 개 이상의 작은 문제로 분할할 수 있어야 함
- 큰 문제와 작은 문제를 동일한 방법으로 해결할 수 있어야 함
- 큰 문제의 최적해가 작은 문제들의 최적해들로 구성되어야 함

- 결론: Optimal Substructure, Overlapping Subproblem을 알아야 함

동적 계획법

동적 계획법 문제인지 판단하는 방법

- Optimal Substructure 성질을 만족함을 증명한다.
- 지금까지 문제를 풀어본 경험을 토대로 추측한다.
  - 최소/최대 비용, 트리, 냅색, 경우의 수, 기댓값, 확률, …
- 왠지 DP일 것 같으니까 일단 믿어본다.

- 증명할 자신이 없으면 그냥 문제를 많이 풀어서 유형을 외우는 것이 편함
- solved.ac 기준 골드까지는 물량으로 밀 수 있음

동적 계획법

동적 계획법 문제인 건 알겠는데...

- 큰 문제와 작은 문제 간의 상관 관계(점화식)을 어떻게 찾지?

- 점화식을 정의한다.
  - 현재 "상태"를 잘 표현할 방법을 생각한다.
    - 배열 인덱스, 선택한 원소의 개수/합/곱(을 M으로 나눈 나머지) 등
  - (최적화) 다른 방식으로 표현할 수 없는지 생각한다.
    - ex) knapsack, 뒤에서 다룸
  - (최적화) 상태의 차원을 줄여도 온전하게 표현할 수 있는지 생각한다.
    - ex) 현재 상태를 (A+B, A, B)로 표현하는 경우, (A+B, A)만 사용해도 온전히 표현할 수 있음

- 점화 관계를 찾는다.
  - 현재 "상태"로 가기 바로 직전 "상태"는 무엇이 있을까?
  - (최적화) 다양한 DP 최적화 기법들(CHT, DnC Opt, Kitamasa, etc.)

질문?

동적 계획법 – 예시 1

BOJ 2748 피보나치 수 2

- n ≤ 90 번째 피보나치 수를 구하는 문제
- 재귀 함수의 계산 결과를 저장해서 중복 계산을 피함
  - 한 번도 계산하지 않은 값은 -1로 초기화
  - dp[n] ≠ -1 이면 f(n)을 계산한 적 있다는 뜻

- 시간 복잡도: O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll dp[91];
ll f(int n){
    if(n == 0 || n == 1) return n;
    ll &res = dp[n];
    if(res != -1) return res;
    return res = f(n-1) + f(n-2);
}

int main(){
    int n; cin >> n;
    for(int i=0; i<91; i++) dp[i] = -1;
    cout << f(n);
}
```

동적 계획법 – 예시 1

BOJ 2748 피보나치 수 2

- n ≤ 90 번째 피보나치 수를 구하는 문제
- 재귀 함수의 계산 결과를 저장해서 중복 계산을 피함
  - 한 번도 계산하지 않은 값은 -1로 초기화
  - dp[n] ≠ -1 이면 f(n)을 계산한 적 있다는 뜻
- 시간 복잡도: O(N)
- 반복문을 이용해서 점화식을 계산할 수도 있음
- 두 가지 방법 모두 알아야 함

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll dp[91];

int main(){
    int n; cin >> n;
    dp[0] = 0; dp[1] = 1;
    for(int i=2; i<=n; i++) dp[i] = dp[i-1] + dp[i-2];
    cout << dp[n];
}
```

동적 계획법 – 예시 2

BOJ 1463 1로 만들기

- 아래 세 가지 연산을 적절히 사용해서 X를 1로 만드는 최소 연산 횟수
  - X가 3의 배수라면 $X \leftarrow X/3$
  - X가 2의 배수라면 $X \leftarrow X/2$
  - $X \leftarrow X-1$

- $f(x) \leftarrow f(x-1) + 1$
- $f(x) \leftarrow f(x/2) + 1 \quad \text{if } x \equiv 0 \pmod 2$
- $f(x) \leftarrow f(x/3) + 1 \quad \text{if } x \equiv 0 \pmod 3$

- 시간 복잡도: O(N)

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

int D[1010101];
int f(int n){
    if(n == 1) return 0;
    int &res = D[n];
    if(res != -1) return res;
    res = f(n - 1) + 1;
    if(n % 2 == 0) res = min(res, f(n / 2) + 1);
    if(n % 3 == 0) res = min(res, f(n / 3) + 1);
    return res;
}

int main(){
    int n; cin >> n;
    for(int i=0; i<=n; i++) D[i] = -1;
    cout << f(n);
}
```

동적 계획법 – 예시 2

BOJ 1463 1로 만들기

- 아래 세 가지 연산을 적절히 사용해서 X를 1로 만드는 최소 연산 횟수
  - X가 3의 배수라면 X ← X/3
  - X가 2의 배수라면 X ← X/2
  - X ← X-1

- $f(x) \leftarrow f(x-1) + 1$
- $f(x) \leftarrow f(x/2) + 1 \qquad \text{if } x \equiv 0 \pmod 2$
- $f(x) \leftarrow f(x/3) + 1 \qquad \text{if } x \equiv 0 \pmod 3$

- 시간 복잡도: O(N)

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

int D[1010101];

int main(){
    int n; cin >> n;
    D[1] = 0;
    for(int i=2; i<=n; i++){
        D[i] = D[i-1] + 1;
        if(i % 2 == 0) D[i] = min(D[i], D[i/2] + 1);
        if(i % 3 == 0) D[i] = min(D[i], D[i/3] + 1);
    }
    cout << D[n];
}
```

질문?

동적 계획법 – 예시 3

BOJ 1912 연속합

- 수열이 주어지면 구간 합의 최댓값을 구하는 문제

- D[i] = i번째 수를 마지막 원소로 하는 구간 합의 최댓값
  - i번째 수 하나만 사용하면 D[i] = A[i]
  - i번째 수보다 앞에 있는 수도 사용할 때의 최댓값은 D[i-1] + A[i]
    - D[i-1]은 i-1번째 수로 끝나는 구간 중 합이 가장 크기 때문
  - 따라서 D[i] = max{ A[i], D[i-1] + A[i] }
  - D[i] = max{ 0, D[i-1] } + A[i]

- 시간 복잡도: O(N)

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

int N, A[101010], D[101010];

int main(){
    ios_base::sync_with_stdio(false);cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++) D[i] = max(0,D[i-1]) + A[i];
    int R = D[1];
    for(int i=1; i<=N; i++) R = max(R, D[i]);
    cout << R;
}
```

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i] = i번째 수를 마지막 원소로 하는 LIS의 길이
  - i번째 수만 사용하면 D[i] = 1
  - 앞에 있는 수도 사용하면…
  - A[j] < A[i]인 모든 j에 대해 D[i] = max D[j] + 1
    - A[i]보다 작은 수 뒤에 A[i]를 붙이는 방식

- 시간 복잡도: $O(N^2)$

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

int N, A[1010], D[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        D[i] = 1;
        for(int j=1; j<i; j++){
            if(A[j] < A[i]) D[i] = max(D[i], D[j] + 1);
        }
    }
    int R = 0;
    for(int i=1; i<=N; i++) R = max(R, D[i]);
    cout << R;
}
```

질문?

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i][j] = 1..i번째 수를 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이
  - j ≠ A[i] 이면 D[i][j] = D[i-1][j]
  - j = A[i] 이면 k < j에 대해 D[i][j] = max D[i-1][k] + 1
    - 1..i-1번째 수를 적당히 사용해서 j보다 작은 수로 끝나는 증가 부분 수열을 만든 다음
    - 그 뒤에 A[i] = j를 붙이는 방식

- 시간 복잡도: $O(NX^2)$
- 공간 복잡도: O(NX)

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

- 가장 긴 증가하는 부분 수열(LIS)의 길이를 구하는 문제
  - 부분 수열: 수열의 원소 몇 개를 선택해서 순서를 유지한 채로 만든 새로운 수열
  - 증가하는 수열: A[i-1] < A[i]를 만족하는 수열

- D[i][j] = 1..i번째 수를 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이

- i마다 값이 바뀌는 위치는 1개(D[i][A[i]])밖에 없음
  - 굳이 이차원 배열을 모두 들고 다닐 필요 없음

- D[j] = 지금까지 본 수들을 적당히 사용했을 때, 마지막 원소의 값이 j인 LIS의 길이
  - 각 i마다 D[A[i]]를 적당히 갱신하면 됨

- 시간 복잡도: O(NX)
- 공간 복잡도: O(X)

동적 계획법 – 예시 4

BOJ 11053 가장 긴 증가하는 부분 수열

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

int N, A[1010], D[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        for(int j=0; j<A[i]; j++){
            D[A[i]] = max(D[A[i]], D[j] + 1);
        }
    }
    int R = 0;
    for(int i=1; i<=1000; i++) R = max(R, D[i]);
    cout << R;
}
```

질문?

동적 계획법 – 예시 5

BOJ 2293 동전 1

- n가지 종류의 동전을 적당히 사용해서 k원을 만드는 경우의 수

- k원을 만드는 방법
  - 1..i-1번째 동전으로 k-cA_i원을 만든 다음
  - A_i원 동전을 c개 사용
- D[i][j] : 1..i번째 동전으로 j원을 만드는 경우의 수
  - D[0][0] = 1
  - D[i][j] = sum D[i-1][j-cA_i]
    - $0 \le j - cA_i \le k, c \ge 0$
    - $0 \le c \le j/A_i$
- 시간 복잡도: O(NK log K)
- 공간 복잡도: O(NK), 메모리 초과

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

int N, K, A[111], D[111][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            for(int c=0; j-c*A[i]>=0; c++){
                D[i][j] += D[i-1][j-c*A[i]];
            }
        }
    }
    cout << D[N][K];
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- D[i][*]를 계산할 때 D[i-1][*]만 사용함
  - $D[i][j] = \text{sum } D[i-1][j-cA_i]$ $(0 \le c \le j/A_i)$
  - D[n][k] 크기의 배열 대신 D[2][k] 만 사용해도 됨
  - D[i][*] 대신 D[i%2][*]를 사용

- 시간 복잡도: O(NK log K)
- 공간 복잡도: O(N+K)

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

int N, K, A[111], D[2][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++) D[i%2][j] = 0; // init
        for(int j=0; j<=K; j++){
            for(int c=0; c<=j/A[i]; c++){
                D[i%2][j] += D[(i-1)%2][j-c*A[i]];
            }
        }
    }
    cout << D[N%2][K];
}
```

질문?

동적 계획법 – 예시 5

BOJ 2293 동전 1

- 매번 c를 전부 탐색하는 것은 비효율적

- 각 상태 전이에서 동전을 1개만 사용하는 방법
  - 1..i번째 동전으로 j-A_i원을 만든 다음 A_i원 동전 사용
  - D[i][j] = D[i-1][j] + D[i][j-A_i]
    - i번째 동전을 사용하는 경우 / 사용하지 않는 경우
    - j-A_i 음수 조심

- 시간 복잡도: O(NK)

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

int N, K, A[111], D[2][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            D[i%2][j] = D[(i-1)%2][j];
            if(j - A[i] >= 0) D[i%2][j] += D[i%2][j-A[i]];
        }
    }
    cout << D[N%2][K];
}
```

동적 계획법 – 예시 5

BOJ 2293 동전 1

- D[i][j] = D[i-1][j] + D[i][j-Ai]
  - D[i-1][j]를 D[i][j]로 복사한 다음, D[i][j]에 D[i][j-Ai]를 더함

  - 굳이 복사할 필요가 있을까?
  - 그냥 덮어쓰면 됨
    - D[j]를 계산하는 시점에
    - D[j-A_i]에 이미 i번째 동전을 사용한 결과가 반영되어 있음

- 시간 복잡도: O(NK)
- 공간 복잡도: O(N+K)

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

int N, K, A[111], D[10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    D[0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            if(j - A[i] >= 0) D[j] += D[j-A[i]];
        }
    }
    cout << D[K];
}
```

질문?

동적 계획법 – 예시 6

BOJ 2294 동전 2

- n가지 종류의 동전을 적당히 사용해서 k원을 만들기 위해 필요한 동전의 최소 개수

- D[i][j] : 1..i번째 동전으로 j원을 만들기 위해 필요한 최소 개수
  - D[0][0] = 0
  - D[i][j] = min{ D[i-1][j], D[i][j-A_i] + 1 }

- 동전 1과 동일한 방법으로 해결 가능
  - 시간 복잡도 O(NK)
  - 공간 복잡도 O(N+K)

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

int N, K, A[111], D[10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=0; i<10101; i++) D[i] = 1e9;
    D[0] = 0;
    for(int i=1; i<=N; i++){
        for(int j=0; j<=K; j++){
            if(j - A[i] >= 0) D[j] = min(D[j], D[j-A[i]]
+ 1);
        }
    }
    cout << (D[K] < 1e9 ? D[K] : -1);
}
```

동적 계획법 – 예시 7

BOJ 12865 평범한 배낭

- 무게가 W_i, 가격이 V_i인 물건 N개
- 최대 K만큼의 무게를 넣을 수 있는 배낭
- 배낭에 넣을 수 있는 물건들의 최대 가치
- 0/1 Knapsack Problem

- D[i][j]
  - 1..i번째 물건을 적당히 선택해서 무게의 합이 j일 때 가능한 가치의 최댓값
  - D[i][j] ← D[i-1][j] (i번째 물건을 선택하지 않는 경우)
  - D[i][j] ← D[i-1][j-W_i] + V_i (i번째 물건을 선택하는 경우)
  - D[i][*]를 계산할 때 D[i-1][*]만 사용하므로 D[N][K] 대신 D[2][K] 사용
- 시간 복잡도: O(NK)
- 공간 복잡도: O(N+K)

동적 계획법 – 예시 7

BOJ 12865 평범한 배낭

- $D[i][j] = \max(D[i-1][j], D[i-1][j-W_i] + V_i)$
  - D[i-1][j]를 가져오는 것은 D[i-1][*]를 D[i][*]로 복사하는 것
  - 동전 1 문제처럼 덮어쓰는 방식으로 할 수 있을까?

- $D[j] \leftarrow D[j-W_i] + V_i$
  - 동전 1은 각 원소를 중복해서 사용할 수 있었지만 이 문제는 불가능
  - D[j]를 계산하는 시점에 $D[j-W_i]$에 i번째 물건을 반영하지 않은 결과가 있어야 함
  - j를 K부터 0까지 역순으로 보면 됨

  - `for(int i=1; i<=N; i++) for(int j=K; j>=W[i]; j--) D[j] = max(D[j], D[j-W[i]] + V[i]);`

동적 계획법 - 예시 7

BOJ 12865 평범한 배낭

- 만약 V_i가 작고 K가 크다면? (ex. $V_i \le 10$, $K \le 10^9$)
- D[i][j] = 1..i번째 물건을 적당히 선택해서 가격의 합을 j로 만들 수 있는 무게의 최솟값
- D[i][j] ≤ K를 만족하는 가장 큰 j가 정답

질문?

동적 계획법 – 예시 8

BOJ 9251 LCS

- 두 수열 A, B의 최장 공통 부분 수열의 길이를 구하는 문제
  - ex. ACAYKP, CAPCAK

- D[i][j] = A[1..i]와 B[1..j]의 최장 공통 부분 수열
  - A[i]와 B[j]가 같은 경우
    - A[1..i-1]과 B[1..j-1]의 최장 공통 부분 수열의 맨 뒤에 A[i]를 추가
      - D[i][j] ← D[i-1][j-1] + 1
  - A[i]와 B[j]가 다른 경우
    - A[1..i]와 B[1..j-1]의 최장 공통 부분 수열
      - D[i][j] ← D[i][j-1]
    - A[1..i-1]과 B[1..j]의 최장 공통 부분 수열
      - D[i][j] ← D[i-1][j]

- 시간 복잡도, 공간 복잡도: O(|A||B|)

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

int N, M, D[1010][1010];
string A, B;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> A >> B;
    N = A.size(); M = B.size();
    A = "#" + A; B = "#" + B; // 0-based to 1-based

    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            if(A[i] == B[j]) D[i][j] = D[i-1][j-1] + 1;
            else D[i][j] = max(D[i-1][j], D[i][j-1]);
        }
    }
    cout << D[N][M];
}
```

질문?

동적 계획법 – 예시 9

BOJ 12852 1로 만들기 2

- 최소 횟수로 1을 만드는 과정을 출력하는 문제

- f(x)가 최소가 되는 바로 직전 상태 P[x]를 저장
  - ex) P[2] = 1, P[4] = 2, P[5] = 4, P[8] = 4
- x가 1이 될 때까지 P[x]를 따라가면 됨
  - 8 → 4 → 2 → 1

- 시간 복잡도: O(N)

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

int D[1010101], P[1010101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int n; cin >> n;
    D[1] = 0; P[1] = -1;
    for(int i=2; i<=n; i++){
        D[i] = D[i-1] + 1;
        P[i] = i - 1;
        if(i % 2 == 0 && D[i] > D[i/2] + 1){
            D[i] = D[i/2] + 1;
            P[i] = i / 2;
        }
        if(i % 3 == 0 && D[i] > D[i/3] + 1){
            D[i] = D[i/3] + 1;
            P[i] = i / 3;
        }
    }
    cout << D[n] << "\n";
    for(int i=n; i!=-1; i=P[i]) cout << i << " ";
}
```

동적 계획법 – 예시 10

BOJ 14002 가장 긴 증가하는 부분 수열 4

- 가장 긴 증가하는 부분 수열(LIS)을 구하는 문제

- D[i] = i번째 수를 마지막 원소로 하는 LIS의 길이
- D[i] = max D[j] + 1
  - 편의상 A[0] = -inf, D[0] = 0이라고 정의하자.

- 각 i마다 D[j]가 최대인 j를 P[i]로 저장
- D[*]가 최대인 지점부터 시작해서 P를 따라가면 됨

- 시간 복잡도: $O(N^2)$

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

int N, A[1010], D[1010], P[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        for(int j=0; j<i; j++){
            if(A[j] < A[i] && D[i] < D[j] + 1){
                D[i] = D[j] + 1;
                P[i] = j;
            }
        }
    }

    int pos = 1;
    for(int i=1; i<=N; i++) if(D[i] > D[pos]) pos = i;
    cout << D[pos] << "\n";

    vector<int> track;
    for(int i=pos; i; i=P[i]) track.push_back(A[i]);
    reverse(track.begin(), track.end());
    for(auto i : track) cout << i << " ";
}
```

동적 계획법 – 예시 11

BOJ 9252 LCS 2

- 두 수열 A, B의 최장 공통 부분 수열을 구하는 문제

- D[i][j] = A[1..i]와 B[1..j]의 최장 공통 부분 수열
- D[i][j]를 구성하는 방법
  - A[i]와 B[j]가 같으면 D[i][j] = D[i-1][j-1] + 1
  - A[i]와 B[j]가 다르면 D[i][j] = max{ D[i][j-1], D[i-1][j] }

- 역추적
  - D[N][M]부터 시작해서, D[i][j]의 값이 왔던 곳으로 따라가면 됨

  - A[i] = B[j]이면 정답에 A[i]를 추가한 다음 D[i-1][j-1]로 이동
  - A[i] ≠ B[j]이면 D[i-1][j]과 D[i][j-1] 중 더 큰 곳으로 이동

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

int N, M, D[1010][1010];
string A, B;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> A >> B;
    N = A.size(); M = B.size();
    A = "#" + A; B = "#" + B; // 0-based to 1-based

    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            if(A[i] == B[j]) D[i][j] = D[i-1][j-1] + 1;
            else D[i][j] = max(D[i-1][j], D[i][j-1]);
        }
    }

    string R;
    for(int i=N, j=M; i>0 && j>0; ){
        if(A[i] == B[j]) R += A[i], i--, j--;
        else if(D[i-1][j] > D[i][j-1]) i--;
        else j--;
    }
    reverse(R.begin(), R.end());
    cout << R.size() << "\n" << R;
}
```

질문?