

2022-1학기 스터디 #4

나정휘

<https://justicehui.github.io/>

목차

- solved.ac 소개
- 동적 계획법
- 예시 (6문제)

solved.ac

- BOJ 문제에 알고리즘 태그와 난이도를 붙이는 커뮤니티 프로젝트
 - 문제를 해결한 유저가 직접 난이도와 사용한 알고리즘을 투표하는 시스템
- 문제 난이도
 - Unrated / Bronze V, IV, III, II, I / Silver V, IV, III, II, I / Gold V, IV, III, II, I
 - Platinum V, IV, III, II, I / Diamond V, IV, III, II, I / Ruby V, IV, III, II, I
- 코딩 테스트 합격을 가르는 문제는 대부분 S1~G3
 - 임의의 G3 문제를 30분 안에 해결할 수 있으면 전세계 상위 20% 정도 (Codeforces 1600+)
- G1~P5 이하의 문제를 모두 해결하면 높은 확률로 ICPC Seoul Regional 진출
 - 임의의 G1 문제를 30분 안에 해결할 수 있으면 전세계 상위 3% 정도 (Codeforces 2100+)
- P1 이하의 문제를 모두 해결하고 D5 이상 하나 풀면 ICPC Seoul Regional 수상 가능

solved.ac

- 어떤 문제를 풀어야 할까?
 - 클래스 문제집 (solved.ac/class)
 - 고인물들이 직접 엄선한 좋은 문제 (shiftpsht solvedac 제작자, jh05013 B0J 9등, koosaga 1등, jhnah917 5등)
 - 새로운 개념, 구현 연습, 풀이가 어려운 문제
 - 어떤 걸 공부해야 할지 모를 때 하나씩 풀어보면서 모르는 거 공부하면 좋음
- 문제 검색
 - ex. 난이도 S2~G3인 동적 계획법 문제 중 내가 아직 안 푼 문제
 - *s2..g3 #dp -s@\$me
 - ex. 난이도가 S5~S1인 정수론 문제 중 jhnah917이 풀고 내가 안 푼 문제
 - *s #number_theory s@jhnah917 -s@\$me
 - ex. 난이도가 G5..G3인 문제 중 수학이 들어가지 않은 문제
 - *g5..g3 -#math

solved.ac

- 어떤 문제를 풀어야 할까?
 - 올림피아드 문제
 - 주로 여러 단계의 사고 과정을 거쳐야 하는 문제
 - USACO Bronze / Silver 추천
 - ICPC 기출 문제
 - 올림피아드보다 더 넓은 출제 범위인류가 아는 모든 지식
 - 코딩테스트 기출 문제
 - (삼성) 구현해야 할 것이 많은 문제
 - BOJ 문제집에 정리되어 있음
 - (카카오) 기초적인 개념을 응용하는 문제
 - Programmers에 정리되어 있음
 - 대부분 봄 스터디에서 다루는 범위 안에서 출제

질문?

동적 계획법

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 않을텐데...
 - 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 $a \cdot \text{fibonacci}(0) + b \cdot \text{fibonacci}(1)$ 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 $a \cdot \text{fibonacci}(5) + b \cdot \text{fibonacci}(4)$ 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, 탐색, 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 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?

예시

동적 계획법 - 예시 1

- BOJ 2748 피보나치 수 2
 - $n \leq 90$ 번째 피보나치 수를 구하는 문제
 - 재귀, 반복문 두 가지 방법 모두 알아야 함
 - 시간 복잡도: $O(N)$

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

long long fibo[91];

long long f(int n){
    if(n == 0 || n == 1) return n;
    long long &res = fibo[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++) fibo[i] = -1;
    cout << f(n);
}

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

long long fibo[91];

int main(){
    int n; cin >> n;
    fibo[0] = 0;
    fibo[1] = 1;
    for(int i=2; i<=n; i++) fibo[i] = fibo[i-1] + fibo[i-2];
    cout << fibo[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 $X \leftarrow X/2$
 - $X \leftarrow X-1$
 - $f(x) \leftarrow f(x-1) + 1$
 - $f(x) \leftarrow f(x/2) + 1$ if $x \equiv 0 \pmod{2}$
 - $f(x) \leftarrow f(x/3) + 1$ if $x \equiv 0 \pmod{3}$
- 시간 복잡도: $O(N)$

```
#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);
}

////////////////////////////////////

#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 12852 1로 만들기 2
 - 최소 횟수로 1을 만드는 과정을 출력하는 문제
 - $f(x)$ 가 최소가 되는 바로 직전 상태 $P[x]$ 를 저장
 - ex) $P[2] = 1, P[4] = 2, P[5] = 4, P[8] = 4$
 - x 가 1이 될 때까지 $P[x]$ 를 따라가면 됨
 - $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
- 시간 복잡도: $O(N)$

```
#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 4

- BOJ 2293 동전 1
 - n 가지 종류의 동전을 적당히 사용해서 k 원을 만드는 경우의 수
 - k 원을 만드는 방법
 - 1.. i -1번째 동전으로 $k - cA_i$ 원을 만든 다음
 - A_i 원 동전을 c 개 사용
 - $D[i][j]$: 1.. i 번째 동전으로 j 원을 만드는 경우의 수
 - $D[0][0] = 1$
 - $D[i][j] = \sum D[i-1][j - cA_i]$
 - $0 \leq j - cA_i \leq k, c \geq 0$
 - $0 \leq c \leq j/A_i$
 - 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 4

- BOJ 2293 동전 1
 - $D[i][*]$ 를 계산할 때 $D[i-1][*]$ 만 사용함
 - $D[i][j] = \sum D[i-1][j - cA_i] \ (0 \leq c \leq j/A_i)$
 - $D[n][k]$ 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: $O(N+K)$

```
#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 4

- BOJ 2293 동전 1
 - 매번 c 를 전부 탐색하는 것은 비효율적
 - 각 상태 전이에서 동전을 1개만 사용하는 방법
 - 1.. i 번째 동전으로 $j - A_i$ 원을 만든 다음 A_i 원 동전 사용
 - $D[i][j] = D[i-1][j] + D[i][j-A_i]$
 - i 번째 동전을 사용하는 경우 / 사용하지 않는 경우
 - $j - A_i$ 음수 조심
- 시간 복잡도: $O(NK)$

```
#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 4

- BOJ 2293 동전 1
 - $D[i][j] = D[i-1][j] + D[i][j-A_i]$
 - $D[i-1][j]$ 를 $D[i][j]$ 로 복사한 다음, $D[i][j]$ 에 $D[i][j-A_i]$ 를 더함
 - 굳이 복사할 필요가 있을까?
 - 그냥 덮어쓰면 됨
 - $D[j]$ 를 계산하는 시점에
 - $D[j-A_i]$ 에 이미 i 번째 동전을 사용한 결과가 반영되어 있음
- 시간 복잡도: $O(NK)$
- 공간 복잡도: $O(N+K)$

```
#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 5

- BOJ 9251 LCS
 - 두 수열 A, B의 가장 공통 부분 수열의 길이를 구하는 문제
 - ex. ACAYKP, CAPCAK
 - $D[i][j]$ = A[1..i]와 B[1..j]의 가장 공통 부분 수열
 - A[i]와 B[j]가 같은 경우
 - A[1..i-1]과 B[1..j-1]의 가장 공통 부분 수열의 맨 뒤에 A[i]를 추가
 - $D[i][j] \leftarrow D[i-1][j-1] + 1$
 - A[i]와 B[j]가 다른 경우
 - A[1..i]와 B[1..j-1]의 가장 공통 부분 수열
 - $D[i][j] \leftarrow D[i][j-1]$
 - A[1..i-1]과 B[1..j]의 가장 공통 부분 수열
 - $D[i][j] \leftarrow D[i-1][j]$
 - 시간 복잡도: $O(|A||B|)$

```
#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];
}
```

동적 계획법 - 예시 5

- BOJ 9251 LCS
 - 최장 공통 부분 수열의 길이를 구할 때 메모리를 $O(N+M)$ 만 사용할 수 있을까?
 - 메모리를 $O(NM)$ 만큼 사용해서 최장 공통 부분 수열을 구할 수 있을까?

질문?

동적 계획법 - 예시 6

- BOJ 12865 평범한 배낭
 - 무게가 W_i , 가격이 V_i 인 물건 N 개
 - 최대 K 만큼의 무게를 넣을 수 있는 배낭
 - 배낭에 넣을 수 있는 물건들의 최대 가치
- $D[i][j]$
 - 1.. i 번째 물건을 적당히 선택해서 무게의 합이 j 일 때 가능한 가치의 최댓값
 - $D[i][j] \leftarrow D[i-1][j]$ (i 번째 물건을 선택하지 않는 경우)
 - $D[i][j] \leftarrow D[i-1][j-W_i] + V_i$ (i 번째 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 6

- 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 i 번째 물건을 반영하지 않은 결과가 있어야 함
 - j 를 K 부터 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 6

- BOJ 12865 평범한 배낭
 - 만약 V_i 가 작고 K 가 크다면? (ex. $V_i \leq 10, K \leq 10^9$)
 - $D[i][j] = 1..i$ 번째 물건을 적당히 선택해서 가격의 합을 j 로 만들 수 있는 무게의 최솟값
 - $D[i][j] \leq K$ 를 만족하는 가장 큰 j 가 정답

질문?

과제

- 필수
 - 12852 1로 만들기 2
 - 2293 동전 1 / 2294 동전 2
 - 10164 격자상의 경로
 - 9252 LCS 2
 - 12865 평범한 배낭
- 심화
 - 5557 1학년
 - 2616 소형기관차
 - 12869 뮤탈리스크
 - 1344 축구