다이나믹 프로그래밍

경북대학교 전현승
dogdriip@gmail.com

Table of contents

- 0x00 다이나믹 프로그래밍?
- 0x01 예시 – 피보나치 수 구하기
- 0x02 문제를 풀어봅시다
- 0x03 가장 긴 증가하는 부분 수열

▬ 0x00

다이나믹 프로그래밍?

다이나믹 프로그래밍

- Dynamic Programming (동적 계획법)

- 일반적으로 주어진 문제를 풀기 위해서,
  문제를 **여러 개의 하위 문제(Subproblem, 부분 문제)**로 나누어 푼 다음,
  그것을 결합하여 최종적인 목적에 도달하는 것이다.

다이나믹 프로그래밍

- **각 하위 문제의 해결을 계산한 뒤,**
  그 해결책을 저장하여 후에 같은 하위 문제가 나왔을 경우 그것을 간단하게 해결할 수 있다.
- 이러한 방법으로 동적 계획법은 **계산 횟수를 줄일 수 있다**.
  특히 이 방법은 하위 문제의 수가 기하급수적으로 증가할 때 유용하다. (Wikipedia)

다이나믹 프로그래밍

- "문제를 여러 하위 문제로 나눠서 풀되,
  각 하위 문제의 답을 계산 후 따로 저장해 둬서 계산 횟수를 줄이자!"

다이나믹 프로그래밍의 조건

- DP를 적용해서 문제를 풀기 위해서는 문제가 두 가지 속성을 가져야 한다
- Overlapping Subproblem : 부분문제가 겹친다
- Optimal Substructure : 문제의 정답을 부분문제의 정답에서 구할 수 있다

- 답을 구하기 위해서 했던 계산을 또 하고 또 하고 계속해야 하는 류의 문제
- 큰 문제에서 작은 부분문제의 계산 결과를 계속해서 이용해야 하는 류의 문제

근데 왜 "다이나믹"?

"The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive."

Eddy, S. R. (2004). "What is Dynamic Programming?". Nature Biotechnology. 22 (7): 909–910.

- 멋있어 보여서, 연구소 펀딩 받으려고
- "이전 답을 기억하며 풀기", "답 재활용하기" 등이 더 와닿을지도

Top-down, Bottom-up

- DP를 구현하는 방식을 크게 두 개로 나눌 수 있다
- Top-down : 일단 큰 문제를 마주치면 문제를 부분문제로 나누고, 부분문제를 풀고, 다시 돌아와서 큰 문제를 푼다
  - **큰 케이스부터, 위에서부터 아래로** 풀어나가는 방식. 보통 재귀함수로 구현하는 게 일반적
- Bottom-up : 작은 부분문제부터 풀어나가다 보면 문제를 풀 수 있다
  - **작은 케이스부터, 아래에서부터 위로** 풀어나가는 방식. 간단한 for loop 정도?
- 문제마다 구현하기 편한 스타일이 있지만, 처음에는 한 방식만 연습해보는 것을 추천

▬ 0x01

예시 – 피보나치 수 구하기

N번째 피보나치 수를 구해보자

- $F_n = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ F_{n-1} + F_{n-2} & (n > 1) \end{cases}$
- F_n을 구하기 위해 F_{n-1}와 F_{n-2}를 계산해야 하고,
- F_{n-1}를 구하기 위해서는 F_{n-2}와 F_{n-3}를 계산해야 하고,
- $\cdots$

- 일단은 그냥 해 보자!

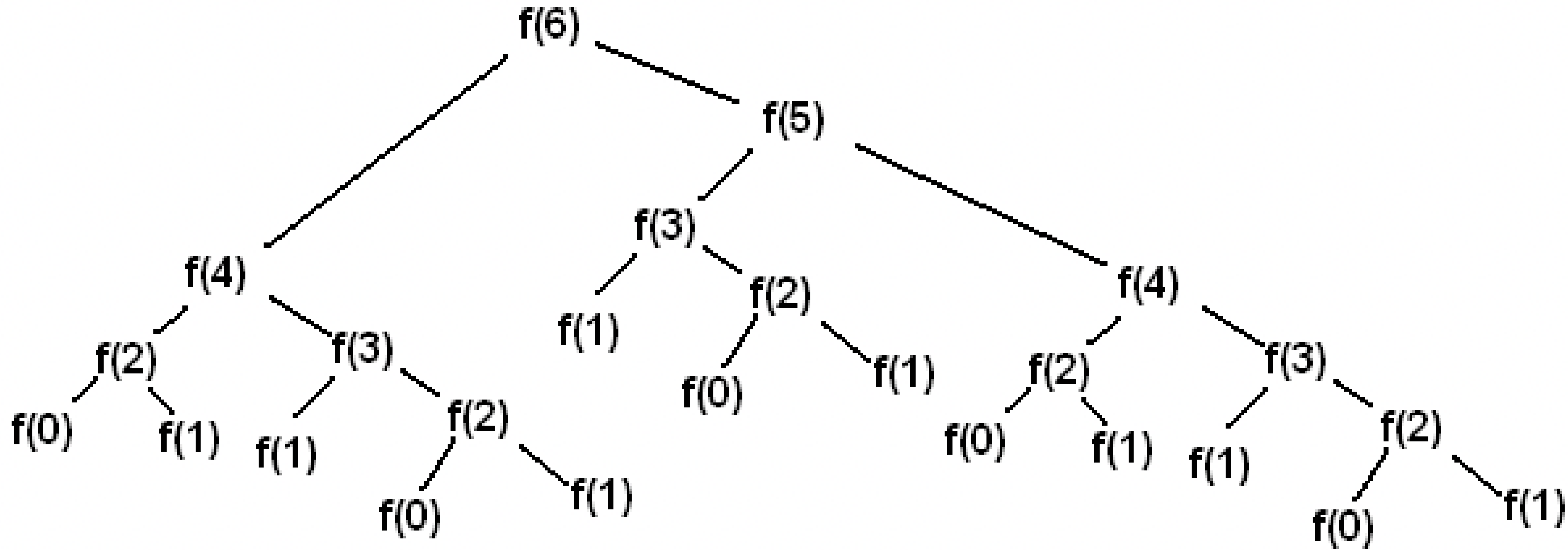
그냥 구하면 어떻게 되는데요?

- DP가 적용되지 않은 일반적인 재귀를 이용한 피보나치 수 구하기

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

ll f(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;
    return f(x - 2) + f(x - 1);
}

int main() {
    int n; cin >> n;
    cout << f(n) << '\n';
}
```

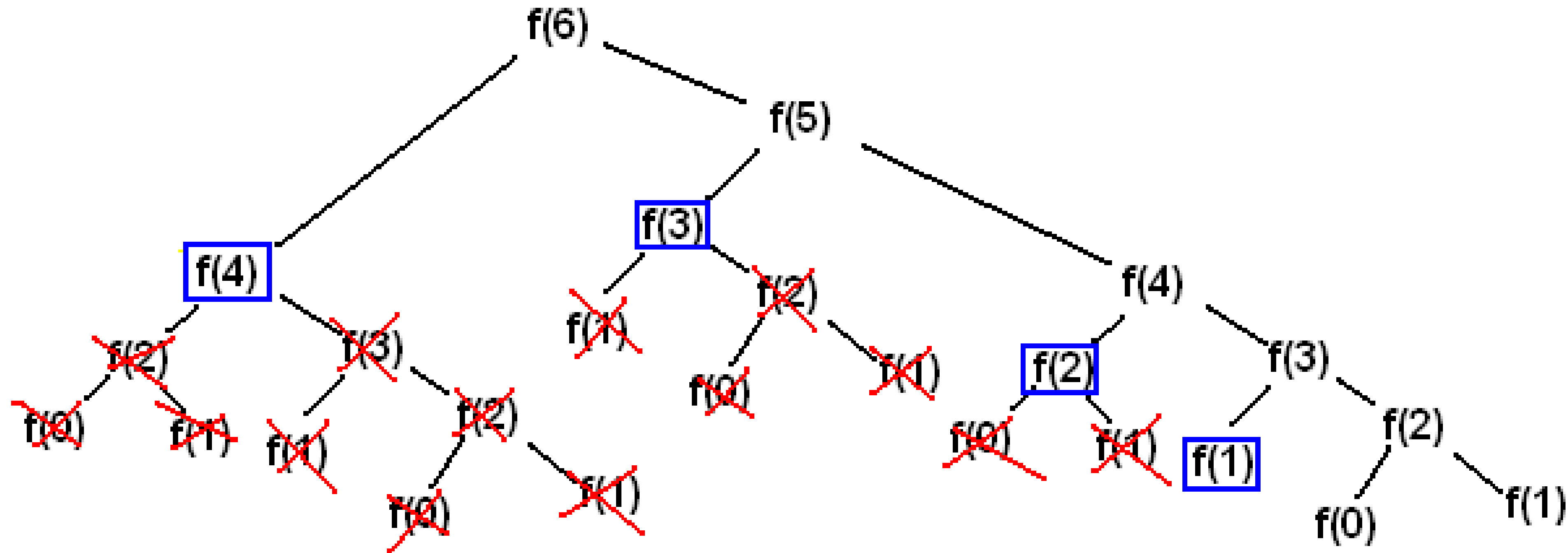
그냥 구하면 어떻게 되는데요?

- N=6일 때, f(6)은 f(4)와 f(5)를 호출 → f(5)는 f(3)과 f(4)를 호출, …



그냥 구하면 어떻게 되는데요?

- 이렇게 계산하면, 계산이 중복되는 부분이 많다



- f(4)를 구하기 위해 f(2) + f(3)을 두 번 똑같이 수행하고 있다
- f(3), f(2)도 마찬가지

그냥 구하면 어떻게 되는데요?

- Time Complexity $\approx O(2^N)$ (Exponential)
- N이 조금만 커져도 프로그램이 급격히 느려지는 것을 확인할 수 있다

다이나믹 프로그래밍의 적용

- 중복되는 계산을 줄일 수는 없을까?
- 이렇게 매번 `f(n)`을 **그때그때 계산하지 않고, 계산 결과를 따로 저장**해두면?
- `f(4)`, `f(3)` 등 **함수의 계산값을 저장해두고 풀면** 이 연산을 줄일 수 있을 것이다

- "값을 메모해둔다" → **메모이제이션 (Memoization)**

다이나믹 프로그래밍의 적용

- 피보나치 수를 구하는 문제에 DP를 적용할 수 있을까?

- " n번째 피보나치 수 구하는 문제 " 는
  " n-2번째 피보나치 수 구하는 문제 " 와
  " n-1번째 피보나치 수 구하는 문제 " 로 나눌 수 있다

- `f(n)`이라는 큰 문제를 → `f(n-2)`와 `f(n-1)`이라는 두 부분문제로 나눌 수 있는 것이다

다이나믹 프로그래밍의 적용

- 아까 봤던 그거

```
ll f(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;
    return f(x - 2) + f(x - 1);
}

int main() {
    int n; cin >> n;
    cout << f(n) << '\n';
}
```

다이나믹 프로그래밍의 적용

#2748 피보나치 수 2

- 값을 저장해두기 위한 배열을 마련하고, 함수에서 이를 이용하도록 함수를 약간 수정

```
ll dp[91];  // 메모이제이션을 위한 배열

ll f(int x) {
    // 메모값이 0이 아니라면 메모가 되어 있다는 것
    if (dp[x] != 0) return dp[x];

    if (x == 0) return 0;
    if (x == 1) return 1;
    return dp[x] = f(x - 2) + f(x - 1);
}

int main() {
    int n; cin >> n;
    cout << f(n) << '\n';
}
```

Remind: Top-down

- 방금 해 봤던 방식이 Top-down
  - Top-down : 일단 큰 문제를 마주치면 문제를 부분문제로 나누고, 부분문제를 풀고, 다시 돌아와서 큰 문제를 푼다

- 보통 일반적으로 짠 재귀함수에 현재 함수값을 저장해두는 코드와,
  저장해둔 값을 이용하는 코드를 추가하는 방식

How about Bottom-up?

- 작은 값부터 계산해나가는 방식이 Bottom-up
  - Bottom-up : 작은 부분문제부터 풀어나가다 보면 문제를 풀 수 있다

- `dp[0]`과 `dp[1]`값을 먼저 채워넣고,
  `dp[2]`부터 `dp[n]`까지 이전 값을 이용해 채워넣는 방식

How about Bottom-up?

#2748 피보나치 수 2

```
int main() {
    ll dp[91];
    int n; cin >> n;

    dp[0] = 0; dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 2] + dp[i - 1];
    }
    cout << dp[n] << '\n';

    return 0;
}
```

0x02

문제를 풀어봅시다

1로 만들기

#1463 1로 만들기

- 정수 N $(1 \leq N \leq 10^6)$이 주어진다.
- 다음 연산 세 개를 적절히 사용해서 1을 만들려고 한다.
  1. X가 3으로 나누어 떨어지면, 3으로 나눈다.
  2. X가 2로 나누어 떨어지면, 2로 나눈다.
  3. 1을 뺀다.
- 최소 횟수로 연산을 사용해서 1을 만들 때, 연산을 사용하는 횟수의 최솟값

1로 만들기

#1463 1로 만들기

- N=10인 경우, 10 → 9 → 3 → 1
- 총 3번만에 1로 만들 수 있고, 이 때가 최소 횟수이다
- 점화식은?

1로 만들기 – 점화식

#1463 1로 만들기

- x를 1로 만드는 최소 횟수를 f(x)라 하면
- $f(x) = \min\begin{cases} f(x/3)+1, & x\text{는 3의 배수} \\ f(x/2)+1, & x\text{는 2의 배수} \\ f(x-1)+1 & \end{cases}$

1로 만들기 – Top-down

- Top-down 구현 – main() 부분

```
int main() {
    int n; cin >> n;
    memset(dp, -1, sizeof(dp));
    cout << solution(n) << '\n';

    return 0;
}
```

1로 만들기 – Top-down

#1463 1로 만들기

- Tip : `memset(배열, -1 또는 0, sizeof(배열))`

```
int main() {
    int n; cin >> n;
    memset(dp, -1, sizeof(dp));
    cout << solution(n) << '\n';

    return 0;
}
```

- 배열을 -1이나 0으로 초기화할 때 유용하게 사용할 수 있음
- 저렇게 하면 dp 배열의 값이 모두 -1로 초기화됨
- 왜 -1이나 0만 가능한지, 원리는 무엇인지 자세한 내용은 찾아보시길

1로 만들기 – Top-down

#1463 1로 만들기

- Top-down 구현 – solution() 부분

```
int dp[1000001];

int solution(int x) {  // solution(x) : x를 1로 만들 수 있는 최소 횟수
    if (x == 1) return 0;

    int& ret = dp[x];
    if (ret != -1) return ret;

    int res = (1 << 31) - 1;
    if (x % 3 == 0) res = min(res, solution(x / 3) + 1);
    if (x % 2 == 0) res = min(res, solution(x / 2) + 1);
    res = min(res, solution(x - 1) + 1);

    return ret = res;
}
```

1로 만들기 – Top-down

#1463 1로 만들기

- **기저 케이스 (Base case)** : 처리할 수 있는 최소 케이스. 재귀함수의 종료조건.

```
int dp[1000001];

int solution(int x) {  // x를 1로 만들 수 있는 최소 횟수
    if (x == 1) return 0;

    int& ret = dp[x];
    if (ret != -1) return ret;

    int res = (1 << 31) - 1;
    if (x % 3 == 0) res = min(res, solution(x / 3) + 1);
    if (x % 2 == 0) res = min(res, solution(x / 2) + 1);
    res = min(res, solution(x - 1) + 1);

    return ret = res;
}
```

1로 만들기 – Top-down

#1463 1로 만들기

- 메모값이 -1이 아니면 메모값이 들어있다는 것이므로 그대로 사용

```
int dp[1000001];

int solution(int x) {  // x를 1로 만들 수 있는 최소 횟수
    if (x == 1) return 0;

    int& ret = dp[x];
    if (ret != -1) return ret;

    int res = (1 << 31) - 1;
    if (x % 3 == 0) res = min(res, solution(x / 3) + 1);
    if (x % 2 == 0) res = min(res, solution(x / 2) + 1);
    res = min(res, solution(x - 1) + 1);

    return ret = res;
}
```

1로 만들기 – Top-down

#1463 1로 만들기

- res에 계속 최솟값을 갱신시켜서, 최종적으로는 return과 동시에 메모까지

```
int dp[1000001];

int solution(int x) {  // x를 1로 만들 수 있는 최소 횟수
    if (x == 1) return 0;

    int& ret = dp[x];
    if (ret != -1) return ret;

    int res = (1 << 31) - 1;
    if (x % 3 == 0) res = min(res, solution(x / 3) + 1);
    if (x % 2 == 0) res = min(res, solution(x / 2) + 1);
    res = min(res, solution(x - 1) + 1);

    return ret = res;
}
```

1로 만들기 – Bottom-up

#1463 1로 만들기

- Bottom-up 구현

```
int dp[1000001];
int n; cin >> n;

dp[1] = 0; dp[2] = 1; dp[3] = 1;
for (int i = 4; i <= n; i++) {
    dp[i] = (1 << 31) - 1;
    if (i % 3 == 0) dp[i] = min(dp[i], dp[i / 3] + 1);
    if (i % 2 == 0) dp[i] = min(dp[i], dp[i / 2] + 1);
    dp[i] = min(dp[i], dp[i - 1] + 1);
}

cout << dp[n] << '\n';
```

1로 만들기 – Bottom-up

#1463 1로 만들기

- 초기값 몇 개만 적절히 세팅해주고

```
int dp[1000001];
int n; cin >> n;

dp[1] = 0; dp[2] = 1; dp[3] = 1;
for (int i = 4; i <= n; i++) {
    dp[i] = (1 << 31) - 1;
    if (i % 3 == 0) dp[i] = min(dp[i], dp[i / 3] + 1);
    if (i % 2 == 0) dp[i] = min(dp[i], dp[i / 2] + 1);
    dp[i] = min(dp[i], dp[i - 1] + 1);
}

cout << dp[n] << '\n';
```

1로 만들기 – Bottom-up

#1463 1로 만들기

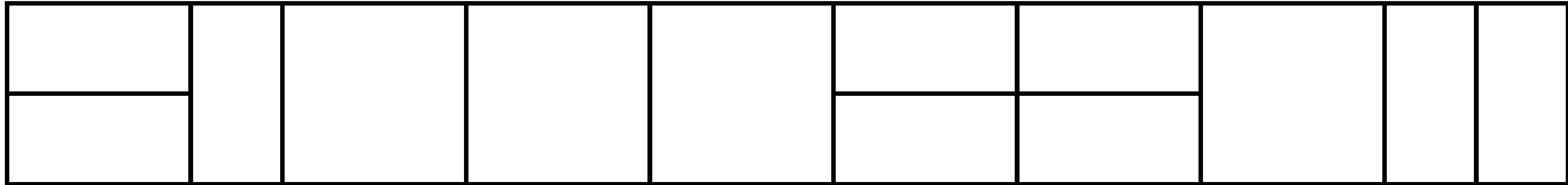
- dp[n] 값을 이전 값을 이용해서 구해나감

```
int dp[1000001];
int n; cin >> n;

dp[1] = 0; dp[2] = 1; dp[3] = 1;
for (int i = 4; i <= n; i++) {
    dp[i] = (1 << 31) - 1;
    if (i % 3 == 0) dp[i] = min(dp[i], dp[i / 3] + 1);
    if (i % 2 == 0) dp[i] = min(dp[i], dp[i / 2] + 1);
    dp[i] = min(dp[i], dp[i - 1] + 1);
}

cout << dp[n] << '\n';
```

2×n 타일링 2

#11727 2×n 타일링 2

- 2×n 직사각형을 2×1과 2×2 타일로 채우는 방법의 수를 10,007로 나눈 나머지를 출력하는 프로그램을 작성하시오.
- 아래 그림은 2×17 직사각형을 채운 한가지 예이다.



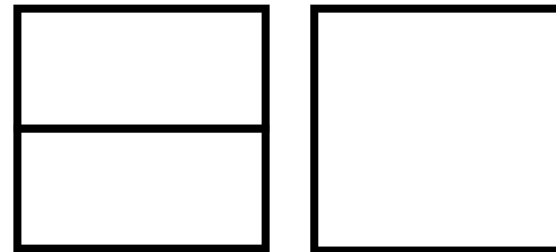
2×n 타일링 2

#11727 2×n 타일링 2

- 2×n 번째 칸까지 채우는 경우의 수는
  2×(n-1) 번째 칸까지 다 채우고, 맨 오른쪽 한 칸을 채우는 경우의 수와
  2×(n-2) 번째 칸까지 다 채우고, 맨 오른쪽 두 칸을 채우는 경우의 수의 합

- 맨 오른쪽 한 칸을 채우는 경우의 수는 하나 (세로로 한 개)

- 맨 오른쪽 두 칸을 채우는 경우의 수는 둘



2×n 타일링 2 – 점화식

#11727 2×n 타일링 2

- 2×x 번째 칸까지 채우는 경우의 수를 f(x)라 하면
- `f(x) = f(x-1) + 2*f(x-2)`

2×n 타일링 2 – Bottom-up

#11727 2×n 타일링 2

- Bottom-up 코드

```
dp[0] = 1;
dp[1] = 3;
for (int i = 2; i <= n; i++) {
    dp[i] = 2 * dp[i - 2] + dp[i - 1];
    dp[i] %= 10007;
}
printf("%d", dp[n - 1]);
```

2×n 타일링 2 – 나머지 연산

#11727 2×n 타일링 2

- 아니 근데 이건 뭐야

```
dp[0] = 1;
dp[1] = 3;
for (int i = 2; i <= n; i++) {
    dp[i] = 2 * dp[i - 2] + dp[i - 1];
    dp[i] %= 10007;
}
printf("%d", dp[n - 1]);
```

2×n 타일링 2 – 나머지 연산

#11727 2×n 타일링 2

- 2×n 직사각형을 2×1과 2×2 타일로 채우는 방법의 수를 **10,007로 나눈 나머지를 출력**하는 프로그램을 작성하시오.

- 문제에서 구하라는 수가 매우 커질 수 있으므로, 결과값의 나머지를 출력하라는 문제들이 있음
- 결과값을 구하는 과정에서 이미 자료형 범위를 넘어가는 경우가 대다수
- 결과값을 다 구해놓고 나머지 연산을 한번 하는 것으로는 정확한 답이 나오지 않는다
- 따라서 **매번 결과값을 구할 때 나머지 연산**을 해 주면 된다

0x03

가장 긴 증가하는 부분 수열

가장 긴 증가하는 부분 수열

- LIS (Longest Increasing Sequence)
- DP로 풀 수 있는 유명한 문제

- 부분 수열 : 수열에서 몇 개의 수들을 골라서 만든 수열

- 수열 A = {10, 20, 10, 30, 20, 50} 인 경우, 수열 A의 LIS는
- A = {**10**, **20**, 10, **30**, 20, **50**}이므로, 길이는 4

- 임의의 수열이 주어지면, DP를 이용해서 LIS의 길이를 구할 수 있을까?

가장 긴 증가하는 부분 수열

- $D[i]$: A의 i번째 원소를 마지막 원소로 가지는 LIS의 길이

- 이렇게 정의하면, $D[i]$는 $D[k]$ $(1 \leq \mathrm{k} \leq \mathrm{N}, \mathrm{A[i]} \leq \mathrm{A[k]})$ 중 최댓값에 1을 더한 값이 된다.
- i번째 원소를 마지막 원소로 하는 LIS의 길이는,
  0번째 원소부터 (i-1)번째 원소들 중 i번째 원소보다 작은 원소들에 대해
  그들을 마지막 원소로 가지는 LIS의 길이에 1을 더한 것이다

- i번째 원소를 맨 뒤에 붙일 수 있는지 없는지를 따지면서
  최댓값을 갱신해가면 된다

가장 긴 증가하는 부분 수열

#11053 가장 긴 …

- 모든 i에 대해 0~(i-1)을 살펴보아야 하므로
- 시간 복잡도 : $O(N^2)$

```
for (int i = 0; i < n; i++) {
    D[i] = 1;
    for (int j = 0; j < i; j++) {
        if (A[j] < A[i]) {
            D[i] = max(D[i], D[j] + 1);
        }
    }
}
```

가장 긴 증가하는 부분 수열

- 답은 D[0~(N-1)] 중 최댓값이 된다

```
for (int i = 0; i < n; i++) {
    D[i] = 1;
    for (int j = 0; j < i; j++) {
        if (A[j] < A[i]) {
            D[i] = max(D[i], D[j] + 1);
        }
    }
}
```

Practice

- #1003 피보나치 함수
- #1904 01타일
- #11726 2×n 타일링
- #9095 1, 2, 3 더하기
- #2579 계단 오르기
- #9465 스티커
- #1149 RGB거리
- #10844 쉬운 계단 수
- #2156 포도주 시식
- #1965 상자넣기
- #2565 전깃줄
- #11054 가장 긴 바이토닉 부분 수열

Sources

- https://ko.wikipedia.org/wiki/%EB%8F%99%EC%A0%81_%EA%B3%84%ED%9A%8D%EB%B2%95