#3 다이나믹 프로그래밍

2019 SCSC Summer Coding Workshop

서강대학교 컴퓨터공학과 박수현

me@shiftpsh.com

피보나치 수열

$1, 1, 2, 3, 5, 8, 13, \cdots$

$\rightarrow F_n = F_{n-2} + F_{n-1}$

어떻게 계산하는 것이 효율적일까?

피보나치 수열

```cpp
#include <iostream>
using namespace std;
using ll = long long;

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

int main() {
    int n;
    cin >> n;
    cout << "F_" << n << " = " << f(n);
    return 0;
}
```

재귀 함수로 짠다면?

피보나치 수열

```cpp
#include <iostream>
using namespace std;
using ll = long long;

ll ops = 0;

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

int main() {
    int n;
    cin >> n;
    cout << "F_" << n << " = " << f(n) << "; " << ops << " calculations";
    return 0;
}
```

계산 횟수

피보나치 수열

```
F_0 = 0; 1 calculations
F_1 = 1; 1 calculations
F_2 = 1; 3 calculations
F_3 = 2; 5 calculations
F_4 = 3; 9 calculations
F_5 = 5; 15 calculations
F_6 = 8; 25 calculations
F_7 = 13; 41 calculations
F_8 = 21; 67 calculations
F_9 = 34; 109 calculations
F_10 = 55; 177 calculations
F_11 = 89; 287 calculations
F_12 = 144; 465 calculations
F_13 = 233; 753 calculations
F_14 = 377; 1219 calculations
F_15 = 610; 1973 calculations
F_16 = 987; 3193 calculations
F_17 = 1597; 5167 calculations
F_18 = 2584; 8361 calculations
F_19 = 4181; 13529 calculations
F_20 = 6765; 21891 calculations
```

?! (참고: $F_{40} \approx 10^8$)

피보나치 수열

$$\begin{aligned}
F_5 &= F_4 + F_3 \\
&= F_3 + F_2 + F_2 + F_1 \\
&= F_2 + F_1 + F_1 + F_0 + F_1 + F_0 + F_1 \\
&= F_1 + F_0 + F_1 + F_1 + F_0 + F_1 + F_0 + F_1 \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5
\end{aligned}$$

피보나치 수열

이미 구한 해는 다시 계산할 필요가 없지 않을까?

$\longrightarrow$ **메모이제이션**memoization

피보나치 수열

```cpp
#include <iostream>
using namespace std;
using ll = long long;

ll dp[300];
ll ops = 0;

ll f(int n) {
    ops++;
    if (dp[n]) return dp[n];
    if (n == 0) return 0;
    if (n == 1) return 1;
    return dp[n] = f(n - 2) + f(n - 1);
}

int main() {
    int n;
    cin >> n;
    cout << "F_" << n << " = " << f(n) << "; " << ops << " calculations";
    return 0;
}
```

계산된 값을 배열에 저장하고 이를 활용

피보나치 수열

```
F_0 = 0; 1 calculations
F_1 = 1; 2 calculations
F_2 = 1; 5 calculations
F_3 = 2; 8 calculations
F_4 = 3; 11 calculations
F_5 = 5; 14 calculations
F_6 = 8; 17 calculations
F_7 = 13; 20 calculations
F_8 = 21; 23 calculations
F_9 = 34; 26 calculations
F_10 = 55; 29 calculations
F_11 = 89; 32 calculations
F_12 = 144; 35 calculations
F_13 = 233; 38 calculations
F_14 = 377; 41 calculations
F_15 = 610; 44 calculations
F_16 = 987; 47 calculations
F_17 = 1597; 50 calculations
F_18 = 2584; 53 calculations
F_19 = 4181; 56 calculations
F_20 = 6765; 59 calculations
```

$\mathcal{O}(n)$!

다이나믹 프로그래밍

- 어떤 문제를 그보다 작은 문제의 연장선으로 생각하고
- 작은 문제의 답을 활용해 큰 문제를 계산하는 기법

다이나믹도 아니고 프로그래밍도 아닌 거 같은데 이름의 유래는 그냥 연구소에서 펀딩 받기 좋은 이름이었기 때문

다이나믹 프로그래밍

크게 두 가지가 있는데···

- 위에서 아래로 가는 방법 (아까 본 코드!)
- 아래에서 위로 가는 방법

다이나믹 프로그래밍

```cpp
#include <iostream>
using namespace std;
using ll = long long;

ll f[300];

int main() {
    f[1] = 1;
    for (int i = 2; f[i - 1] < (ll) 1e18; i++) {
        f[i] = f[i - 2] + f[i - 1];
    }
    int n;
    cin >> n;
    cout << "F_" << n << " = " << f[n];
    return 0;
}
```

아래에서 위로 가는 방법

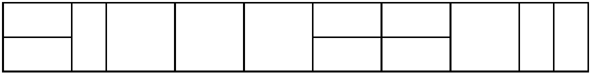
다이나믹 프로그래밍

접근하는 방법

- 주어진 문제를 더 작은 문제로 생각했을 때, 작은 문제의 답을 큰 문제에다 갖다 쓸 수 있는지 생각해 본다
- 가능하다면 점화식을 생각해 본다
- 다 했다면 코딩!

$2 \times n$ 타일링 2 BOJ #11727

$2 \times n$ 직사각형을 2×1과 2×2 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.
아래 그림은 2×17 직사각형을 채운 한 가지 예이다.



- $n \leq 1000$

$2 \times n$ 타일링 2 BOJ #11727

주어진 문제를 더 작은 문제로 생각했을 때, 작은 문제의 답을 큰 문제에다 갖다 쓸 수 있는지 생각해 본다

$2 \times n$ 타일링은

- $2 \times (n-1)$ 타일링의 오른쪽에 ▯ 타일을 붙이거나
- $2 \times (n-2)$ 타일링의 오른쪽에 □ 타일을 붙이거나
- $2 \times (n-2)$ 타일링의 오른쪽에 ⊟ 타일을 붙이는

경우로 나눠진다

$2 \times n$ 타일링 2 BOJ #11727

가능하다면 점화식을 생각해 본다

따라서 $2 \times n$ 타일링의 개수를 T_n 이라고 하면

$$T_n = T_{n-1} + 2T_{n-2}$$

가 된다 (단 $T_1 = 1, T_2 = 3$)

$2 \times n$ 타일링 2 BOJ #11727

다 했다면 코딩!

```cpp
#include <iostream>
using namespace std;
using ll = long long;

ll dp[1001];

ll t(int n) {
    if (dp[n]) return dp[n];
    if (n == 1) return 1;
    if (n == 2) return 3;
    return dp[n] = (t(n - 1) + 2 * t(n - 2)) % 10007;
}

int main() {
    int n;
    cin >> n;
    cout << t(n);
    return 0;
}
```

정수 삼각형 BOJ #1932

$$
\begin{array}{ccccccccc}
 & & & & 7 & & & & \\
 & & & 3 & & 8 & & & \\
 & & 8 & & 1 & & 0 & & \\
 & 2 & & 7 & & 4 & & 4 & \\
4 & & 5 & & 2 & & 6 & & 5
\end{array}
$$

맨 위층부터 시작해서 아래 왼쪽이나 오른쪽에 있는 수 중 하나를 선택하여 아래층으로 내려올 때, **선택된 수들의 합이 최대가 되는 경로**에 있는 수의 합?

- 삼각형의 크기 $n \leq 500$
- 경로의 수는 2^n 개$\cdots$

정수 삼각형 BOJ #1932

$$\begin{array}{ccccccccc} & & & & T_{00} & & & & \\ & & & T_{10} & & T_{11} & & & \\ & & T_{20} & & T_{21} & & T_{22} & & \\ & T_{30} & & T_{31} & & T_{32} & & T_{33} & \\ T_{40} & & T_{41} & & T_{42} & & T_{43} & & T_{44} \end{array}$$

$$\begin{array}{ccccccccc} & & & & M_{00} & & & & \\ & & & M_{10} & & M_{11} & & & \\ & & M_{20} & & M_{21} & & M_{22} & & \\ & M_{30} & & M_{31} & & M_{32} & & M_{33} & \\ M_{40} & & M_{41} & & M_{42} & & M_{43} & & M_{44} \end{array}$$

M_{ij}를 맨 꼭대기에서 i열 왼쪽에서 j번째 수까지 도달했을 때의 최대 합이라고 하자

정수 삼각형 BOJ #1932

$$
\begin{array}{ccccccccc}
 & & & & T_{00} & & & & \\
 & & & T_{10} & & T_{11} & & & \\
 & & T_{20} & & T_{21} & & T_{22} & & \\
 & T_{30} & & T_{31} & & T_{32} & & T_{33} & \\
T_{40} & & T_{41} & & T_{42} & & T_{43} & & T_{44}
\end{array}
\qquad
\begin{array}{ccccccccc}
 & & & & M_{00} & & & & \\
 & & & M_{10} & & M_{11} & & & \\
 & & M_{20} & & M_{21} & & M_{22} & & \\
 & M_{30} & & M_{31} & & M_{32} & & M_{33} & \\
M_{40} & & M_{41} & & M_{42} & & M_{43} & & M_{44}
\end{array}
$$

$$M_{ij} = \max\left(M_{(i-1)(j-1)}, M_{(i-1)j}\right) + T_{ij}$$

예외로 맨 왼쪽에서는 $M_{i0} = M_{(i-1)0} + T_{i0}$, 맨 오른쪽에서는 $M_{ii} = M_{(i-1)(i-1)} + T_{ii}$

정수 삼각형 BOJ #1932

$$
\begin{array}{ccccccccc}
 & & & & M_{00} & & & & \\
 & & & M_{10} & & M_{11} & & & \\
 & & M_{20} & & M_{21} & & M_{22} & & \\
 & M_{30} & & M_{31} & & M_{32} & & M_{33} & \\
\color{red}{M_{40}} & & \color{red}{M_{41}} & & \color{red}{M_{42}} & & \color{red}{M_{43}} & & \color{red}{M_{44}}
\end{array}
$$

그럼 답은 마지막 줄에 저장된 수들의 최댓값!

정수 삼각형 BOJ #1932

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

int t[500][500], m[500][500];

int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            cin >> t[i][j];
        }
    }
```

입력을 2차원 배열로 잘 받는다

정수 삼각형 BOJ #1932

```
m[0][0] = t[0][0];
for (int i = 0; i < n; i++) {
    m[i][0] = m[i - 1][0] + t[i][0];
    for (int j = 1; j < i; j++) {
        m[i][j] = max(m[i - 1][j - 1], m[i - 1][j]) + t[i][j];
    }
    m[i][i] = m[i - 1][i - 1] + t[i][i];
}
```

초기값 설정: 맨 위의 수는 무조건 골라야 하니까

정수 삼각형 BOJ #1932

```
m[0][0] = t[0][0];
for (int i = 0; i < n; i++) {
    m[i][0] = m[i - 1][0] + t[i][0];
    for (int j = 1; j < i; j++) {
        m[i][j] = max(m[i - 1][j - 1], m[i - 1][j]) + t[i][j];
    }
    m[i][i] = m[i - 1][i - 1] + t[i][i];
}
```

각 줄마다 맨 왼쪽, 오른쪽 처리

정수 삼각형 BOJ #1932

```
m[0][0] = t[0][0];
for (int i = 0; i < n; i++) {
    m[i][0] = m[i - 1][0] + t[i][0];
    for (int j = 1; j < i; j++) {
        m[i][j] = max(m[i - 1][j - 1], m[i - 1][j]) + t[i][j];
    }
    m[i][i] = m[i - 1][i - 1] + t[i][i];
}
```

구해 둔 점화식으로 DP 배열 완성 – $\mathcal{O}\left(n^2\right)$

정수 삼각형 BOJ #1932

```cpp
    int mx = 0;
    for (int j = 0; j < n; j++) {
        mx = max(mx, m[n - 1][j]);
    }

    cout << mx;

    return 0;
}
```

마지막 줄의 최댓값을 출력하면 정답!

몇 가지 팁

아래에서 위로? 위에서 아래로?

- 문제를 읽어보고 구현하기 편하다고 생각되는 쪽으로

배열 이름 정하기가 곤란하다면?

- 종이 같은 곳에 적은 점화식에서 쓴 이름 그대로 따라가도 되고, 아니면 간단히 dp라고 해도 되고 (헷갈리지 않는 게 중요)

점화식 빨리 짤 수 있는 방법?

- 문제 많이 푸세요! 다른 방법 있다면 저도 좀 알려주세요

가장 긴 증가하는 부분 수열 BOJ #11053

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

예를 들어 $A = \{10, 20, 10, 30, 20, 50\}$인 경우에 가장 긴 증가하는 부분 수열은 $A = \{10, 20, 10, 30, 20, 50\}$이고, 길이는 4

- 수열의 길이 $N \leq 1000$

가장 긴 증가하는 부분 수열 BOJ #11053

D_i: 수열 A의 부분수열 $\{A_0, \cdots, A_i\}$에서의 '가장 긴 증가하는 부분 수열'의 길이라고 하면,

- i보다 작은 모든 j에 대해 $(0 \leq j < i)$
- A_j보다 A_i가 더 크다면 $(A_j < A_i)$
- $\{A_0, \cdots, A_j\}$에서의 '가장 긴 증가하는 부분 수열' 뒤에다 A_i를 붙이면 이것도 증가하는 부분 수열이 된다
- D_i는 그 중 가장 긴 것!

가장 긴 증가하는 부분 수열 BOJ #11053

식으로 쓰면

$$D_i = \max_{0 \le j < i, A_j < A_i} (D_j + 1)$$

(길이 1 짜리 수열도 '증가하는 부분 수열' 이 맞긴 하니까 초기에 D 는 전부 1 로 초기화한다)

가장 긴 증가하는 부분 수열 BOJ #11053

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

int a[1000], d[1000];
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        d[i] = 1;
    }
```

입력 잘 받는다

가장 긴 증가하는 부분 수열 BOJ #11053

```cpp
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] >= a[i]) continue;
            d[i] = max(d[i], d[j] + 1);
        }
    }

    int mx = 0;
    for (int i = 0; i < n; i++) {
        mx = max(mx, d[i]);
    }
    cout << mx;

    return 0;
}
```

계산해 둔 점화식을 그대로 구현

(팁: continue를 쓰면 if 문이 깊어지지 않아서 깔끔하다)

가장 긴 증가하는 부분 수열 BOJ #11053

```cpp
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] >= a[i]) continue;
            d[i] = max(d[i], d[j] + 1);
        }
    }

    int mx = 0;
    for (int i = 0; i < n; i++) {
        mx = max(mx, d[i]);
    }
    cout << mx;

    return 0;
}
```

최대인 D_i를 출력

LCS[Longest Common Subsequence] 문제는 두 수열이 주어졌을 때, 모두의 부분 수열이 되는 수열 중 가장 긴 것을 찾는 문제이다.

예를 들어, ACAYKP와 CAPCAK의 LCS는 ACAK가 된다.

- 문자열의 길이 $N \leq 1000$

D_{ij}: 문자열 A의 i번째 위치, 문자열 B의 j번째 위치까지 확인했을 때의 LCS의 길이라고 하면,

- $A_i \neq B_j$라면, $D_{i(j-1)}$과 $D_{(i-1)j}$ 중에 큰 게 D_{ij}가 된다

D_{ij}: 문자열 A의 i번째 위치, 문자열 B의 j번째 위치까지 확인했을 때의 LCS의 길이라고 하면,

- $A_i = B_j$라면, D_{ij}는 $D_{(i-1)(j-1)} + 1$이 될 수 있다
- 하지만 $D_{i(j-1)}$ 혹은 $D_{(i-1)j}$가 더 크다면 이쪽을 택해야 한다

따라서

$$D_{ij} = \begin{cases} \max\left(D_{i(j-1)}, D_{(i-1)j}\right) & A_i \neq B_j \\ \max\left(D_{i(j-1)}, D_{(i-1)j}, D_{(i-1)(j-1)} + 1\right) & A_i = B_j \end{cases}$$

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

int d[1000][1000];

int main() {
    string a, b;
    cin >> a >> b;

    int n = a.length(), m = b.length();
```

C++에서는 string을 사용하면 char*를 쓸 필요가 없다

LCS BOJ #9251

```cpp
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (a[i] == b[j]) {
                d[i + 1][j + 1] = max({d[i + 1][j], d[i][j + 1], d[i][j] + 1});
            } else {
                d[i + 1][j + 1] = max(d[i + 1][j], d[i][j + 1]);
            }
        }
    }

    cout << d[n][m];

    return 0;
}
```

점화식 그대로 구현인데, D_{00}을 계산한다고 치면 $D_{(-1)(-1)}$ 같은 게 필요한데 이런 걸 하나하나 처리하는 건 곤란하므로 D는 1-based index를 쓴다

LCS BOJ #9251

```cpp
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (a[i] == b[j]) {
                d[i + 1][j + 1] = max({d[i + 1][j], d[i][j + 1], d[i][j] + 1});
            } else {
                d[i + 1][j + 1] = max(d[i + 1][j], d[i][j + 1]);
            }
        }
    }

    cout << d[n][m];

    return 0;
}
```

그러면 답은 D_{nm}

문제 풀어보고, 질문하는 시간 (–17시까지)