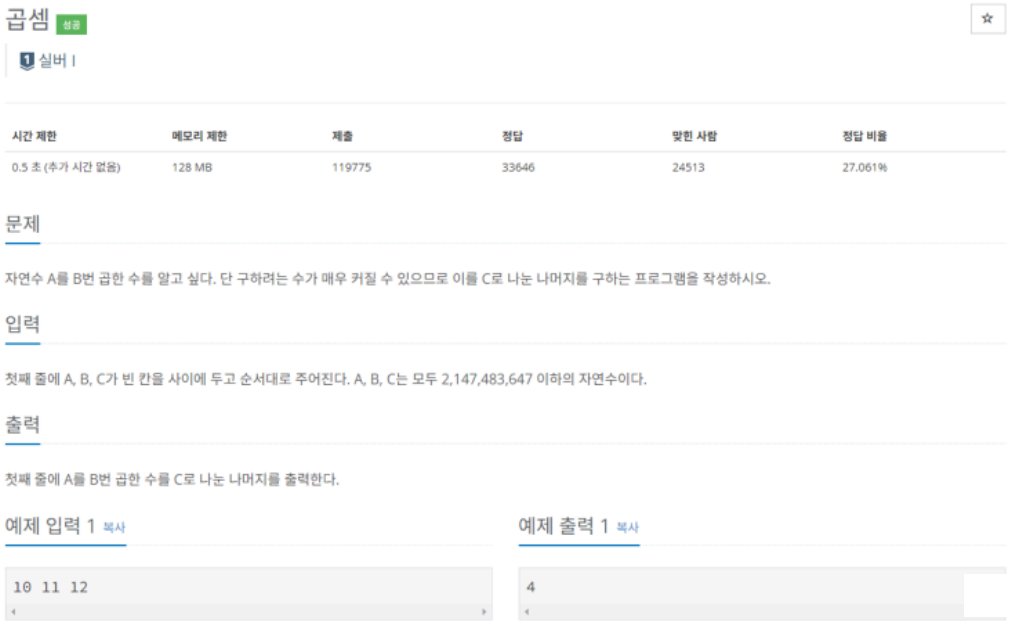
[Python] 파이썬 알고리즘 - 백준 28358번: …
윤년 한정 2월 29일을 맞아 오랜만에 윤년 관련 알고…
blog.naver.com

지난 윤일에 알고리즘 문제를 푼 이후, 이참에 공부를 좀 해둬야겠다는 생각이 들어 꾸준히 알고리즘 문제를 풀고 있습니다. 최대한 결정적이지 않을 법한 힌트만 골라 찾기도 하고, 때로는 혼자 눈 까뒤집고 분노 표출을 하며...

이런 말 하면 머피의 법칙으로 억까 당할까 봐 무섭긴 한데, 그래도 조금은 뇌가 풀렸는지 이제 자주 푼 유형의 문제를 보면 뭔가 감이 쉽게 잡히는 것 같기도 합니다.

여하튼 이번에는 '분할 정복을 이용한 거듭제곱'으로 분류되는 알고리즘 문제를 몇 가지 풀어보고, 해설을 남겨볼까 합니다. 관련하여 얽혀 있는 개념, 문제들이 많아 해당 유형을 정복하지 않고서는 알고리즘 문제 풀이에서 한 단계 더 나아가기가 쉽지 않겠더라고요.

오늘 다뤄볼 백준 문제는 '1619번: 곱셈', '11444번: 피보나치 수 6'입니다. 곱셈 문제부터 살펴보겠습니다.



백준 1619번: 곱셈 캡처, 클릭시 문제 페이지 이동

처음에는 조금 의아했습니다. 가볍게 접근한다면 아래 코드 두 줄로 끝인 문제입니다.

```
a, b, c = map(int, input().split())
print(a**b % c)
```

그러나 실상은 그렇지 않죠. A, B, C는 2147483647 이하의 자연수입니다. 만약 a, b, c에 최댓값의 자연수가 들어간다면, a는 어마어마하게 큰 숫자가 될 것입니다. 숫자를 저장하기 위한 어마어마한 메모리 사용과 동시에 이를 나누고 출력하기까지 매우 오랜 시간이 걸리게 되겠죠.

그래서 반복문을 이용해 곱할까 생각하더라도, 단순히 곱하기만을 반복하면 약 20억 회의 연산을 수행해야 합니다. O(N)으로는 어림도 없을 것 같은 0.5초의 시간제한을 고려해 본다면, 연산 횟수를 logN 정도로 획기적으로 줄여야 한다는 사실을 알 수 있습니다.

지금 생각해 보면 어렵지 않은 것 같은데, 문제를 풀 때는 머리가 잘 안 굴러가더라고요. 그래서 집단지성의 힘을 좀 빌렸습니다.

[Algorithm] 분할정복을 이용한 거듭제곱

2021. 3. 24. 22:54

 # 분할정복을 이용한 거듭제곱

- C**n연산은 x를 n번 곱하므로 O(N)이지만, 이 방법을 사용하면 O(logN)에 거듭제곱 값을 구할 수 있다.
- 아래 코드에서 fpow함수가 그 방법이다.
- n이 1이면 그냥 C의 1제곱이므로 return C를 해준다.
- n이 2이상일 때, C의 n제곱은 다음과 같다.

$$C^n = \begin{cases} C^{n/2}C^{n/2} \\ C^{(n-1)/2}C^{(n-1)/2}C \end{cases}$$

- n이 짝수이면 윗줄의 식을 만족하고, n이 홀수이면 아래줄의 식을 만족한다.

코드로 짜면 다음과 같다.

1. **재귀사용**

```
def fpow(C, n):
        if n == 1:
                return C
        else:
                x = fpow(C, n//2)
                if n % 2 == 0:
                        return x * x
                else:
                        return x * x * C
```

2. **반복문 사용**

- n & 1은 n이 짝수이면 0이므로 False이고, n이 홀수이면 1이어서 True이다.)
- n >> 1은 비트연산에서 오른쪽으로 비트를 하나씩 미는 것이다. 즉, n //= 2와 같다)
- 반복문이 약간 빠르다.

```
def fpow(C, n):
        res = 1
        while n:
                if n & 1:
                        res *= C
                C *= C;
                n >> 1

        return res
```

분할정복을 이용한 거듭제곱, 사진 클릭 시 원문 이동

예를 들어 설명해 보겠습니다. 분할정복을 이용한 거듭제곱으로 42의 98 제곱을 구한다면, 첫 두 단계에 다음 과정을 거치게 됩니다.

$$42^{98} = 42^{98\div 2} \times 42^{98\div 2}$$

$$42^{49} = 42^{48\div 2} \times 42^{48\div 2} \times 49$$

실제로는 42의 98 제곱에서 49 제곱으로, 49 제곱에서 24 제곱으로 트리 구조처럼 파생되어 나가기 때문에, logN 정도의 수행으로 결괏값을 구할 수 있습니다. 문제에서 제시된 최대 범위인 2147483647의 2147483647 제곱을 구한다면? 21474836470이 정수를 표현하는 4byte, 32bit int의 양수 최댓값이고, 음수 범위까지 고려하면 2의 31 제곱에서 1을 뺀 값이니, 30번 정도의 수행이면 끝난다는 사실을 알 수 있습니다.

이제 이걸 재귀, 반복문의 형태로 나타내기만 하면 됩니다. 위 사진의 재귀 코드에서는 5번째 줄 x = fpow(C, n//2)로 몫 연산자를 사용하여 홀수, 짝수 케이스 모두에서 x로 함수를 한 번만 호출하여 값을 구할 수 있도록 처리했습니다. 반복문 코드에서는 and(&) 연산, 시프트 연산(>>)을 사용한 점이 독특한데, 이 부분 조금만 설명하고 넘어가겠습니다.

& 연산은 양쪽의 비트를 비교하여 둘 다 1(참)이면 1, 그렇지 않으면 0을 리턴합니다. 1은 이진수로 표현하면 마지막 자리만 1이 됩니다. 이것을 정수 n과 and 연산한다면, n이 홀수이면 1이 되고 짝수인 경우 0이 됩니다. 이진수의 첫째 자리는 1을 나타내는 자리이니, n이 홀수라면 1이고 짝수라면 0일 테니까요. 뭔가 말로 늘어놓으니 헷갈리네요. 저도 처음 볼 때 헷갈렸습니다.

시프트 연산도 비슷한 맥락입니다. n >> 1이 n //= 2와 같은 이유는, 정수 n을 이진수로 표현한 비트를 모두 오른쪽으로 한 칸씩 밀면, 1의 자리에 있던 값은 사라지고 나머지 모든 자리는 원래 자리보다 2로 나눈 값의 자리에 놓이게 됩니다. 즉, 2로 나눈 몫만 남고 나머지인 1의 자리는 사라지니 2로 나눈 몫만 구하는 것과 같은 결과가 나오는 것이죠. 참고로, 엄밀히 말하자면 n >> 1이 아니라 n >>= 1로 작성하는 것이 맞습니다. 그래야 n 값이 변경되거든요.

이제 배운 내용을 문제에 적용해 봅시다. 재귀 사용, 반복문 사용 순입니다.

```
a, b, c = map(int, input().split())

def ft_pow(a, b):
    if b == 1:
        return a
    x = ft_pow(a, b//2)
    if b & 1:
        return x * x * a % c
    else:
        return x * x % c

print(ft_pow(a, b) % c)
```

```
a, b, c = map(int, input().split())

res = 1
while b > 0:
    if b & 1:
        res *= a % c
        res %= c
    a *= a
    a %= c
    b >>= 1
print(res)
```

문제에서 곱한 수를 C로 나눈 나머지를 구하고 있기 때문에, 나머지 연산이 추가로 적용되었습니다. 마지막에 한 번 나머지 연산을 수행하는 것보다, 중간중간 일반적인 정수 범위를 넘어설 순간에 한 번씩 나머지 연산을 섞어주어야 숫자가 커지지 않고 빠든 연산 수행이 가능합니다. 나머지 연산은 분배 법칙이 성립하기 때문에 이것이 가능합니다.

시간, 메모리 제한으로 인해 난도가 높은 문제에서는 결괏값의 나머지 출력을 요구하는 경우가 많습니다. 개념을 제대로 잡아놓고 가면 좋을 것 같아, 증명 과정을 찾아봤습니다. 아래 증명 과정이 담긴 블로그 글을 남겨둡니다. 과정이 짧아서 직접 따라 해봤더니 금방 이해가 됐습니다.

이러한 나머지 연산이 가지는 분배법칙 성질이 있는데 이를 한번 알아보자.

분배 법칙

```
(A + B) % N = ((A % N) + (B % N)) % N
```

다음의 법칙이 성립한다는 것인데 이를 증명해보면 다음과 같다.

A 와 B에 해당하는 몫과 나머지를 각각 [`q1` `r1`], [`q2` `r2`] 로 가정해보자.
그러면

```
A = q1 X N + r1
B = q2 X N + r2
```

A, B를 나타낼 수 있고 이를 오른쪽 항에 대입하면

```
(q1 X N + r1 + q2 X N + r2) % N
= ((q1+q2) X N + r1 + r2) % N
```

로 치환할 수 있다. 그러면 결국 `modulo` 는 나머지 연산이기 때문에
몫에 해당하는 `q1` 와 `q2` 는 결국 나머지 연산 과정에서 `0` 이 되어
남는 것은

```
(r1 + r2) % N
```

이다. 이때 `r1` 과 `r2` 는 위에서 나타낸 대로 대한 A와 B의 나머지이므로
이는 결국

```
r1 = A % N
r2 = B % N
```

이 된다. 따라서

```
(A + B) % N = ((A % N) + (B % N)) % N
```

이 성립한다는 것을 증명할 수 있다.

같은 방식으로 덧셈 뿐만 아니라 뺄셈, 곱셈에도 마찬가지로 증명할 수 있다.

```
(A + B) % N = ((A % N) + (B % N)) % N
(A - B) % N = ((A % N) - (B % N)) % N
(A X B) % N = ((A % N) X (B % N)) % N
```

하지만 나눗셈은 위의 공식에 적용되지 않는다.

나머지 연산의 분배 법칙 증명, 사진 클릭 시 원문 이동

풀이에 조금 자신이 생겼다면, 조금 더 매콤한 문제인 피보나치 수 6에 도전해 봅시다. 거듭제곱은 거듭제곱인데, 이제 숫자가 아니라 행렬을 곱합니다. 점화식 dp 문제를 O(N)이 아닌 O(logN)에 풀기 위해서...

피보나치 수 6 성공

골드 II

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	21359	9607	8011	47.973%

문제

피보나치 수는 0과 1로 시작한다. 0번째 피보나치 수는 0이고, 1번째 피보나치 수는 1이다. 그 다음 2번째 부터는 바로 앞 두 피보나치 수의 합이 된다.

이를 식으로 써보면 $F_n = F_{n-1} + F_{n-2}$ ($n \geq 2$)가 된다.

n=17일때 까지 피보나치 수를 써보면 다음과 같다.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

n이 주어졌을 때, n번째 피보나치 수를 구하는 프로그램을 작성하시오.

입력

첫째 줄에 n이 주어진다. n은 1,000,000,000,000,000,000보다 작거나 같은 자연수이다.

출력

첫째 줄에 n번째 피보나치 수를 1,000,000,007으로 나눈 나머지를 출력한다.

예제 입력 1 복사

```
1000
```

예제 출력 1 복사

```
517691607
```

평범한 피보나치 수 구하는 문제 같지만, n이 좀 이상합니다. 10억 제곱입니다. O(logN) 아니면 풀 생각도 하지 말라고 엄포를 놓습니다. 그래도 이런 문제들을 몇 번 접하고 나니, 시간 제한과 입력 범위로 풀이 알고리즘의 시간 복잡도를 유추하고 문제 해결의 단서로 삼게 되더라고요.

여하튼 1차원 dp 접근 같은 일반적인 방식으로는 풀 수 없는 문제입니다. 피보나치 수를 구하는 과정을 분할정복을 이용한 거듭제곱으로 바꾸어야 합니다. 이 문제에서는 피보나치 수열의 점화식을 만든 뒤, 만들어진 행렬에 곱해 다음 수열을 구할 행렬을 만들어야 합니다. 행렬 곱 연산을 할 함수도 필요하고요. 이 외에는 앞서 작성한 내용을 그대로 따라갑니다.

행렬 곱을 만드는 과정이 조금 어려운데, 관련 글 캡처와 링크를 첨부합니다. 설명하기엔 실력이 모자란 지라... 행렬이 익숙지 않다면 행렬 곱 계산 방법만 간단히 익히고, 피보나치 수로 이루어진 행렬에 곱할 행렬을 만드는 과정을 이해하면 됩니다. 향후 피보나치 수 이외의 점화식에도 같은 방식을 적용할 수 있을 테니, 미리 응용 연습한다는 느낌으로!

8.1. 분할 정복을 이용한 알고리즘 $\mathcal{O}(\log n)$ [편집]

피보나치 수로 이루어진 2x2 행렬 $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$를 생각하자. 이 행렬에 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$를 곱하면,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1F_{n+1} + 1F_n & 1F_n + 1F_{n-1} \\ 1F_{n+1} + 0F_n & 1F_n + 0F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

를 얻는다.

또한 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$이다.

따라서 다음과 같은 일반화된 공식을 얻을 수 있다:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$를 M이라고 하자. M을 n-1번 거듭제곱하면 F_n을 얻을 수 있다. 그러나 단순히 이를 n-1번 곱하는 방식은 시간 복잡도가 위의 반복문을 이용한
과 동일한 $\mathcal{O}(n)$이다. 그러나, $M^n M^m = M^{n+m}$ 라는 점을 이용하면 이를 $\mathcal{O}(\log n)$으로 줄일 수 있다.

피보나치 수를 구하는 여러가지 방법

피보나치 수는 다음과 같이 정의되는 수열입니다. $F_0 = 0$ $F_1 = 1$…

www.acmicpc.net

$$AB=\begin{pmatrix}\boxed{a_{11}\ a_{12}}\\ a_{21}\ a_{22}\end{pmatrix}\begin{pmatrix}\boxed{b_{11}}\ b_{12}\\ \boxed{b_{21}}\ b_{22}\end{pmatrix}=\begin{pmatrix}\boxed{a_{11}b_{11}+a_{12}b_{21}}\ a_{11}b_{12}+a_{12}b_{22}\\ a_{21}b_{11}+a_{22}b_{21}\ a_{21}b_{12}+a_{22}b_{22}\end{pmatrix}$$

$$AB=\begin{pmatrix}\boxed{a_{11}\ a_{12}}\\ a_{21}\ a_{22}\end{pmatrix}\begin{pmatrix}b_{11}\ \boxed{b_{12}}\\ b_{21}\ \boxed{b_{22}}\end{pmatrix}=\begin{pmatrix}a_{11}b_{11}+a_{12}b_{21}\ \boxed{a_{11}b_{12}+a_{12}b_{22}}\\ a_{21}b_{11}+a_{22}b_{21}\ a_{21}b_{12}+a_{22}b_{22}\end{pmatrix}$$

$$AB=\begin{pmatrix}a_{11}\ a_{12}\\ \boxed{a_{21}\ a_{22}}\end{pmatrix}\begin{pmatrix}\boxed{b_{11}}\ b_{12}\\ \boxed{b_{21}}\ b_{22}\end{pmatrix}=\begin{pmatrix}a_{11}b_{11}+a_{12}b_{21}\ a_{11}b_{12}+a_{12}b_{22}\\ \boxed{a_{21}b_{11}+a_{22}b_{21}}\ a_{21}b_{12}+a_{22}b_{22}\end{pmatrix}$$

$$AB=\begin{pmatrix}a_{11}\ a_{12}\\ \boxed{a_{21}\ a_{22}}\end{pmatrix}\begin{pmatrix}b_{11}\ \boxed{b_{12}}\\ b_{21}\ \boxed{b_{22}}\end{pmatrix}=\begin{pmatrix}a_{11}b_{11}+a_{12}b_{21}\ a_{11}b_{12}+a_{12}b_{22}\\ a_{21}b_{11}+a_{22}b_{21}\ \boxed{a_{21}b_{12}+a_{22}b_{22}}\end{pmatrix}$$

2*2 행렬의 곱연산

Remind: 피보나치 수

n번째 피보나치 수 F_n을 행렬을 이용해 로그 시간에 구하는 방법은 익히 알려져 있다.

$F_n = F_{n-1} + F_{n-2}$ 인데, 우선 F_n에 주목해 보자.

F_n을 F_{n-1}과 F_{n-2} 두 개의 항으로 나타내야 하므로, 좌변과 B 행렬을 적절히 잡아보자.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} \cdots \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}}_{B}$$

좌변에서 F_n을 점화식대로 맞춰주려면, A의 첫 행을 어떻게 채우면 될까?

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \square & \square \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

1, 1로 채우면, 좌변의 첫 행의 원소에 $F_n = F_{n-1} + F_{n-2}$ 점화식대로 결과값이 들어가게 된다.

좌변의 F_{n-1}이 남는데, 얘는 B에서 그대로 가져오면 된다. 그렇게 하려면 A 행렬의 나머지를 어떻게 채우면 될까?

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

1, 0으로 채우면 $F_{n-1} = F_{n-1}$이다. 말 그대로이다. 일단 기본적인 식이 완성되었다. 여기서 조금만 더 변형을 가해보자. **우변의 B의 차수를 하나씩 줄여나간다고 생각하면 편하다.**

우변의 맨 앞에 A를 하나 곱하면, B의 모든 항의 차수가 하나씩 줄어들게 된다.

$n = 4$인 경우를 생각해보자. 써 보면 다음과 같다.

$$\begin{bmatrix} F_4 \\ F_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_3 \\ F_2 \end{bmatrix}$$

여기서 우변의 맨 앞에 A를 하나 더 곱하면?

$$\begin{bmatrix} F_4 \\ F_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

뭐 이렇게 되겠다. B의 원소들의 차수가 하나씩 줄어든 것을 확인할 수 있다. 한번 더 하면?

$$\begin{bmatrix} F_4 \\ F_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

그런데, F_1과 F_0는 각각 1과 0으로 상수값이다. 따라서 마지막 식을 다시 써 보면,

$$\begin{bmatrix} F_4 \\ F_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

모든 과정이 끝났다. 우변의 (세제곱된) A는 빠른 행렬 거듭제곱을 이용하여 로그 시간에 구할 수 있고, 우변의 계산 결과인 2×1 행렬의 첫 행의 원소가 우리가 구하는 F_4가 된다.

방금은 $n = 4$인 경우를 예로 들었는데, 이를 일반화하면 다음과 같은 식을 도출할 수 있다.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

재귀적이지 않은, 바로 계산 가능한 일반화된 식이 만들어진 것이다. 이 문단에서 설명한 방법으로 BOJ#2749 - 보나치 수 3 (https://www.acmicpc.net/problem/2749)을 풀 수 있다.

재귀와 반복문을 사용한 풀이 코드는 다음과 같습니다.

```python
import sys
input = lambda: sys.stdin.readline().rstrip()

def cal_m(a, b):
    c = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            c[i][j] = (a[i][0]*b[0][j] + a[i][1]*b[1][j]) % 1000000007
    return c

def fibo(n, m):
    if n < 2:
        return m
    l = cal_m(m, m)
    if n & 1:
        return cal_m(fibo(n//2, l), m)
    else:
        return fibo(n//2, l)

n = int(input())
print(fibo(n, [[1, 1], [1, 0]])[0][1])
```

```python
import sys
input = lambda: sys.stdin.readline().rstrip()

def cal_m(a, b):
    if a == []:
        return b
    c = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            c[i][j] = (a[i][0]*b[0][j] + a[i][1]*b[1][j]) % 1000000007
    return c

m = [[1, 1], [1, 0]]
n = int(input())
res = []

while n > 1:
    if n & 1:
        res = cal_m(res, m)
    m = cal_m(m, m)
    n >>= 1
res = cal_m(res, m)
print(res[0][1])
```

예쁜 코드는 아니지만, 어떻게 통과만 되게끔 작성했습니다. 함수의 리턴으로 피보나치 수열 값을 바로 받도록 했으면 좋았을 텐데, 통과 코드 작성하는 것만으로도 아직은 벅차네요.

cal_m()에서 2*2 행렬 곱셈이 이루어집니다. 행렬 곱셈은 교환 법칙이 성립하지 않기 때문에, 인자 순서에도 유의해야 합니다. 저는 행렬 곱셈을 반복문으로 처리했지만, 어차피 4개만 채우면 되는 거라 그냥 하나씩 대입해 주는 게

오히려 깔끔합니다. 나머지 연산은 대입 과정에서 처리해 주고, 배열에서 필요한 값이 있는 위치의 인덱스를 출력하도록 처리했습니다.

분할 정복을 이용한 거듭제곱 관련 문제 두 가지를 풀어보았습니다. 이후 타일 채우기 문제(링크)도 같은 방식으로 풀이했는데, 이건 시리즈가 많아서 나중에 한 번에 정리하려고 합니다.

국토 종주 글 포함 밀린 포스팅 정리 및 영상 편집, 토이 프로젝트 몇 가지, 영어 공부, 알고리즘 공부, 독서, 운동 등등 해야 할 일만 산더미 같습니다. 당장 때려 고쳐야 할 습관인데 이건 뭐 몇 년째 변하지를 않네요. 마음만큼 문제 풀이가 잘 안돼서 속상하기도 하고 괜스레 쪽팔리기도 한데, 무지를 아는 것이 곧 앎의 시작이라고 테스형이 그랬답니다. 계속 손써가며 공부하면 뭐라도 남겠지요. 응원의 의미에서 스스로에게 칭찬 좀 해보겠습니다.