문제해결프로그래밍실습 (CSE4152)

11주차

Fibonacci number

목차

- Fibonacci sequence
- Recursion
- Golden ratio
- Golden rectangle & Golden spiral
- 과제

Fibonacci sequence(피보나치 수열)

- 첫째, 둘째 수가 0, 1이며 그 뒤의 모든 수는 바로 앞 두 수의 합인 수열
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .....
- 음악에서의 튜닝, 차트 분석(엘리엇 파동) 등 여러 분야에서 활용됨
- 수식으로 표시하면 아래와 같음

$$fibo(n) = \begin{cases} 0 & \text{if } n \text{ is } 0, \\ 1 & \text{if } n \text{ is } 1, \\ fibo(n-1) + fibo(n-2) & \text{otherwise.} \end{cases}$$

재귀 (recursion)

- 자신을 정의할 때 자기 자신을 재 참조하는 방식
- 주로 같은 연산이 반복되는 작업을 수행할 때 사용함

  => 반복 처리 시 반복문(for, while) 또는 재귀 함수 사용

- 재귀가 종료되는 조건이나 경우를 **'기저 사례(base case)'**라 하는데 재귀 함수 정의 시 기저 사례 정의가 매우 중요함
- 재귀를 수행하며 선언되는 변수는 스택(stack)에 저장됨

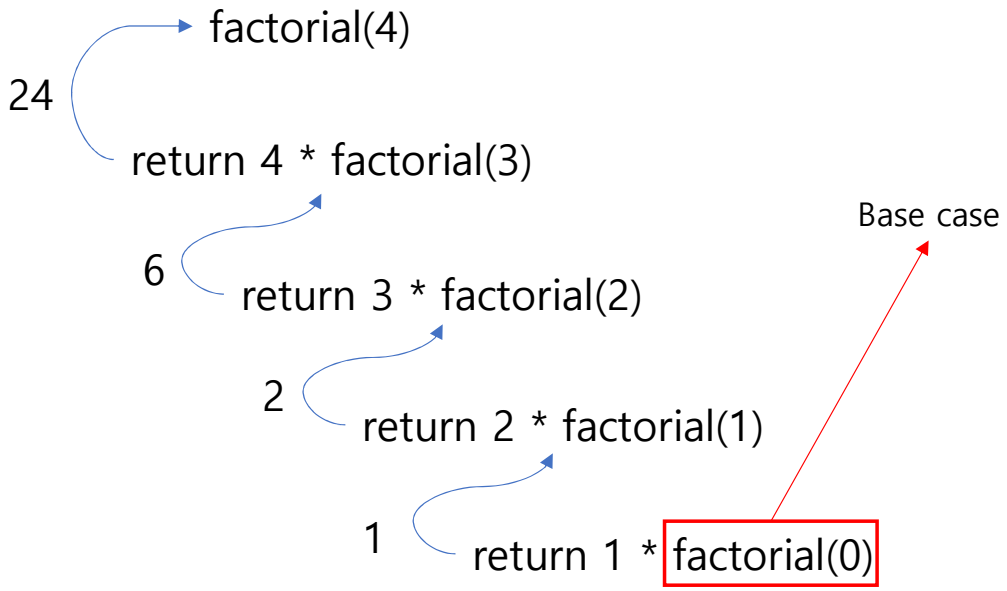
재귀 예시

- 대표적으로 factorial 연산을 재귀의 예시로 들 수 있음

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)

print(factorial(4))
```

```
24
```

< 재귀를 이용한 factorial 구현(Python) >



< 코드 흐름도 >

재귀의 장단점

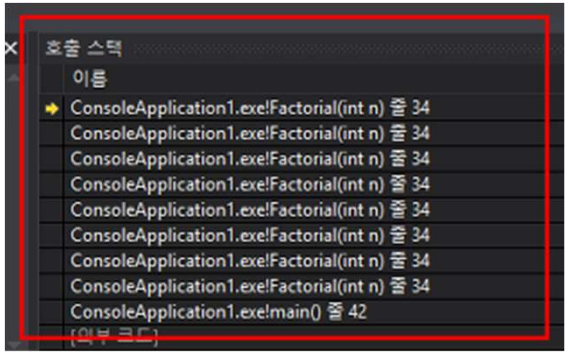
- 장점
  - 코드가 반복문에 비해 상대적으로 간결함
  - 변수 사용을 줄여줌

- 단점
  - 지속적으로 재귀 함수를 호출하게 됨에 따라 변수를 여러 번 호출 및 스택에 저장을 하게 됨
  - 이는 곧 **속도 저하 및 메모리 낭비**로 이어짐

- 해결책
  - **꼬리 재귀(tail call recursion)** 사용
    => 함수 return 시 재귀 호출 이후 추가적인 연산을 요구하지 않는 재귀

꼬리 재귀

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



< 일반 재귀 >

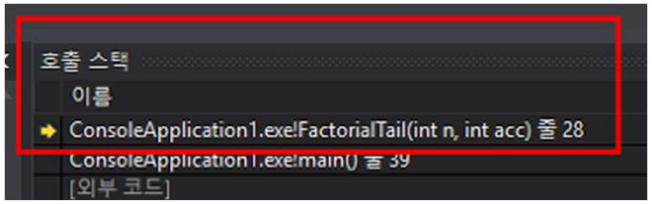
실행 코드

```python
def factorialTail(n, acc):
    if n == 0:
        return acc
    return factorialTail(n-1, acc*n)
    # n * factorial()과 같은 return 부분에 추가적인 연산이 없음

def factorial(n):
    return factorialTail(n, 1)
```

컴파일러가 해석하는 코드

```python
def factorialTail(n):
    acc = 1
    while True:
        if n == 1:
            return acc
        acc = acc * n
        n = n - 1
```

스택 호출 횟수가 1로 감소한 것을 확인할 수 있음

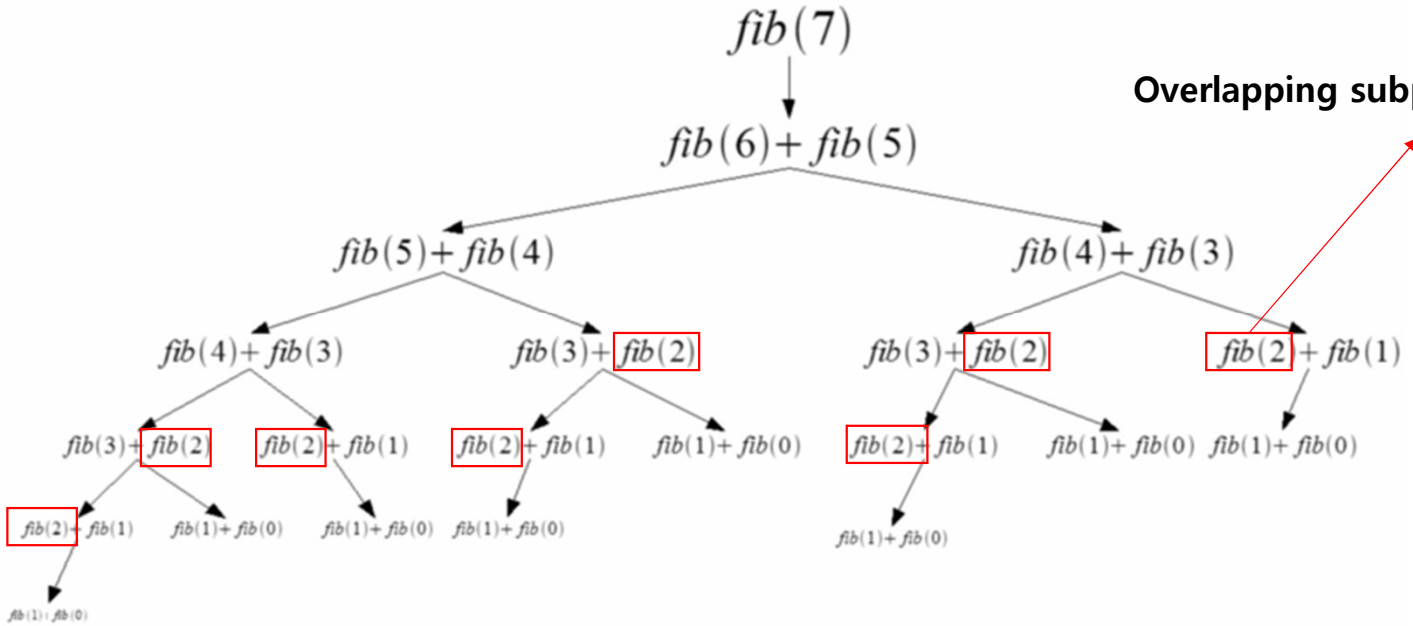


< 꼬리 재귀 >

- 꼬리 재귀를 사용할 경우 컴파일러는 코드를 반복문으로 해석하여 실행하기 때문에 실제 스택에 한 번만 호출하게 된다. (단, Java 컴파일러는 이 기능을 지원하지 않음)
- 따라서 꼬리 재귀를 사용하면 기존 재귀의 메모리 문제를 해결할 수 있다.

재귀로 표현한 피보나치 수열

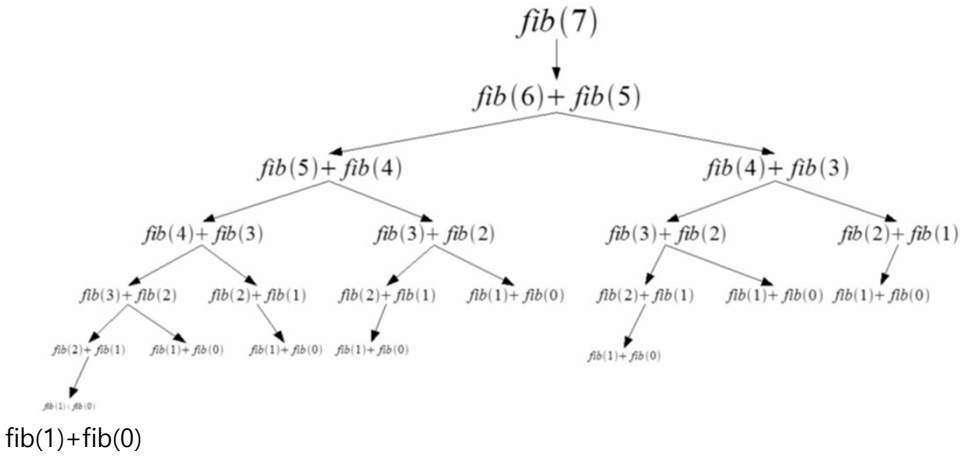
fib(0), fib(1)과 같은 중복되는 부분 문제의 값을 특정 테이블에 저장을 해놓고 필요할 때마다 불러오면 중복을 줄일 수 있지 않을까?



fib(1)+fib(0)

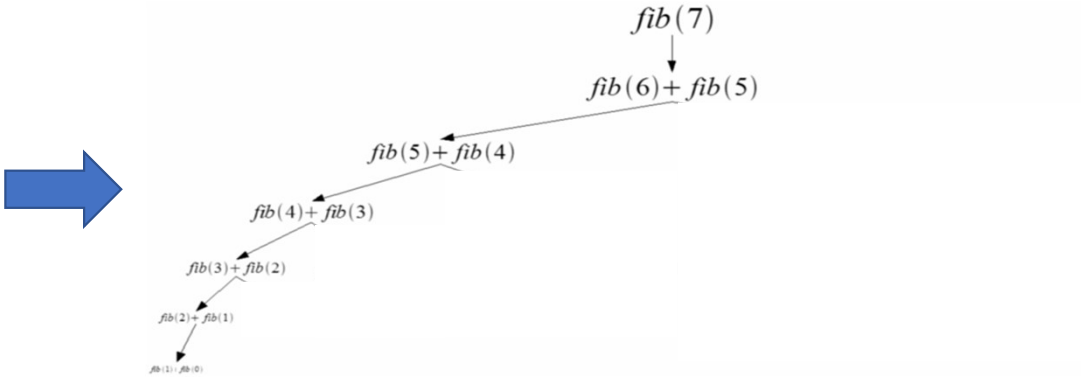
- 과연 이 방법 그대로 코드로 구현한다면 효율적일까?

부분문제 중복이 제거된 피보나치 수열



fib(1)+fib(0)

부분 문제 중복 제거 전



fib(1)+fib(0)

부분 문제 중복 제거 후

- Fib() 호출 수가 40에서 6으로 감소함을 알 수 있음
- 이와 같이 부분 문제의 값을 테이블에 저장 해놓고 필요할 때마다 불러오는 방식을 **'동적 계획법 (dynamic programming)'**이라고 함
- 동적 계획법은 반복문 풀이와 재귀적 풀이에 모두 활용 가능

실습

- 피보나치 수열을 Google Colaboratory 환경에서 코드로 구현
- 재귀, 반복문, 행렬 곱셈, 동적 계획법 등과 같은 다양한 방법으로 구현
- 행렬 연산을 위해 list 자료형 또는 NumPy 패키지 사용 예정
- 각 방법의 시간 복잡도와 메모리 효율성을 비교하여 최적의 알고리즘 도출이 목적

실습 1-1, 1-2

- **실습 1-1**. 주어진 피보나치 수열의 정의를 이용하여 기본적인 **재귀적 풀이 방식**으로 n번째 피보나치 수를 구하는 fibo_1 함수를 작성하시오.

  => 함수가 한 번 호출되면 두 번 더 호출되므로 시간 복잡도는 $O(2^n)$

$$fibo(n) = \begin{cases} 0 & \text{if } n \text{ is } 0, \\ 1 & \text{if } n \text{ is } 1, \\ fibo(n-1) + fibo(n-2) & \text{otherwise.} \end{cases}$$

- **실습 1-2**. 피보나치 수열의 특징을 이용하여 **반복적 풀이 방식**으로 n번째 피보나치 수를 구하는 fibo_2 함수를 직성하시오.

  => 피보나치 수열의 n+1번째 항은 n-1번째 항과 n번째 항을 더한 값이다.

  => n번만큼 루프를 반복하므로 시간 복잡도는 $O(n)$

실습 1-3

- **실습 1-3**. Numpy를 사용한 **행렬 곱셈 풀이**로 n번째 피보나치 수를 구하는 fibo_3 함수를 작성하시오.

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

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

When n=2,

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

- 시간 복잡도 : $O(n)$
- Hint : Numpy 모듈의 행렬 곱셈 함수(np.matmul)와 피보나치 수열의 행렬 곱셈 관계를 참고한다.

Numpy를 이용한 2차원 행렬 곱셈 연산

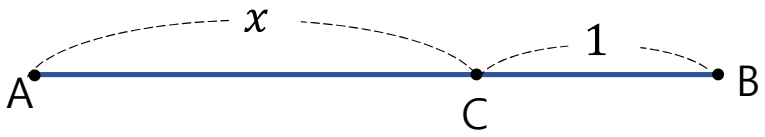
```python
# NumPy 모듈을 이용한 행렬 곱셈 구현
import numpy as np   # numpy 모듈을 np라는 이름으로 현재 소스로 불러오기

a = np.array([[1, 2],
              [3, 4]])  # 1행이 1, 2고 2행이 3, 4인 2x2 정방행렬 a 정의
b = np.array([[5, 6],
              [7, 8]])  # 1행이 5, 6이고 2행이 7, 8인 2x2 정방행렬 b 정의
c = np.matmul(a, b)   # np.matmul을 이용하여 axb 순으로 행렬곱
print(c)
```

```
[[19 22]
 [43 50]]
```

- 실습에 필요한 list 및 numpy 사용 방법은 실습 시 소스 파일로 제공 예정

Golden Ratio (황금비율)

- 한 선분을 두 부분으로 나눌 때에, 큰 부분에 대한 작은 부분의 비와 전체에 대한 큰 부분의 비가 같게 한 비

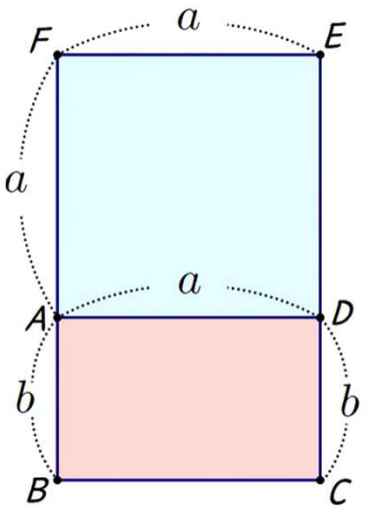


$$x : 1 = x + 1 : x$$

- 황금비 x = 1.6180339 ……..

Golden Rectangle (황금 직사각형)

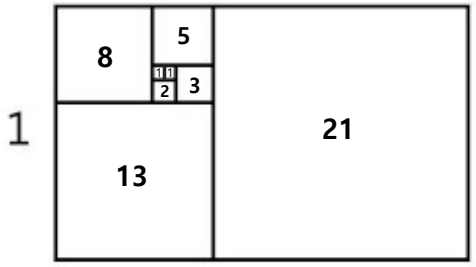
- 황금 직사각형이란 두 변 길이의 비가 1.618:1(=황금비)인 직사각형으로 다음과 같이 나타낼 수 있다.
- 두 변의 비가 황금비가 되도록 직사각형을 반복적으로 추가하면 오른쪽과 같은 결과를 얻을 수 있다.



$$a : b = a + b : a$$

황금사각형

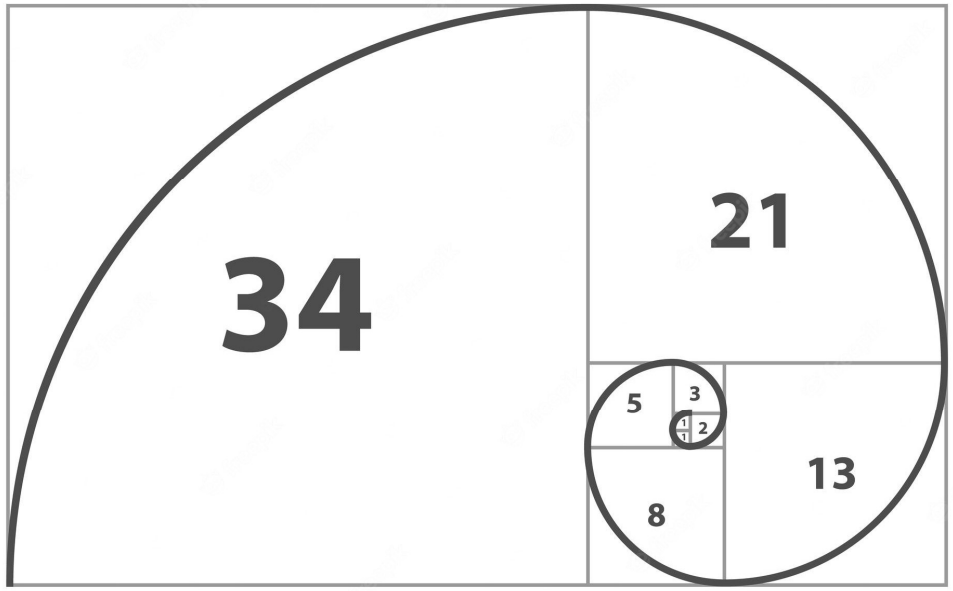
$$a : b = 1.618 : 1$$



1.618...

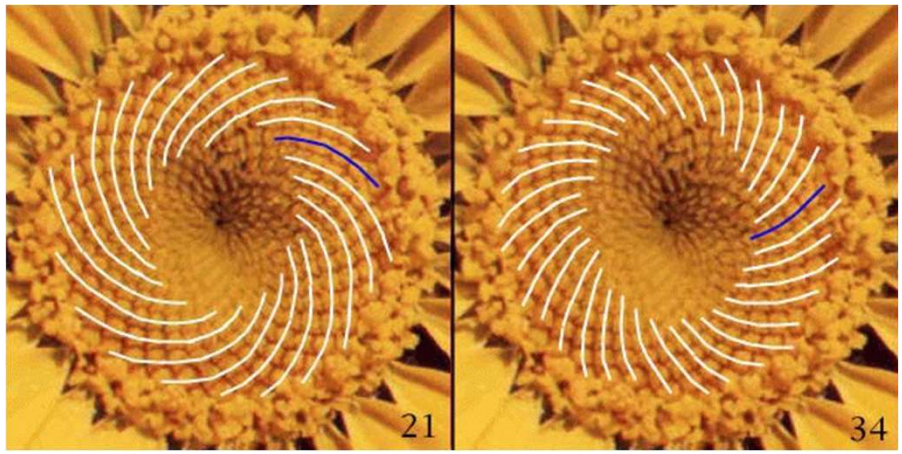
Golden spiral (황금 나선)

- 황금 직사각형의 대각선 위치의 꼭짓점을 호(arc)로 이은 나선을 golden spiral(황금 나선)이라고 한다.
- 황금 나선은 작은 구조가 전체 구조와 비슷한 형태로 끝없이 되풀이 되는 구조인 '프랙탈(Fractal)' 구조이다.



Golden Ratio

- 해바라기 꽃씨의 배열
- 솔방울씨의 배열, 국화 꽃잎의 배열
- 신용카드 가로 세로 길이의 비
- 책, 컴퓨터 모니터, 스마트폰, 영화 스크린 등



시계방향 21개, 반시계방향 34

실습 2

- 황금비 정의 방정식을 구현 후 풀이를 통해 황금비를 직접 구하는 실습
- 황금비와 피보나치 수열의 관계를 알아내는 것이 목표
- 방정식 정의 및 풀이를 위해 Sympy 패키지 사용

```
x:1 = x+1:x
식 f의 해 : [1/2 - sqrt(5)/2, 1/2 + sqrt(5)/2]
황금비 : 1:1.618033988749895
```

```
1과 2번째 피보나치 수열의 비율 : 1.0
2과 3번째 피보나치 수열의 비율 : 2.0
3과 4번째 피보나치 수열의 비율 : 1.5
4과 5번째 피보나치 수열의 비율 : 1.6666666666666667
5과 6번째 피보나치 수열의 비율 : 1.6
6과 7번째 피보나치 수열의 비율 : 1.625
7과 8번째 피보나치 수열의 비율 : 1.6153846153846154
8과 9번째 피보나치 수열의 비율 : 1.619047619047619
```

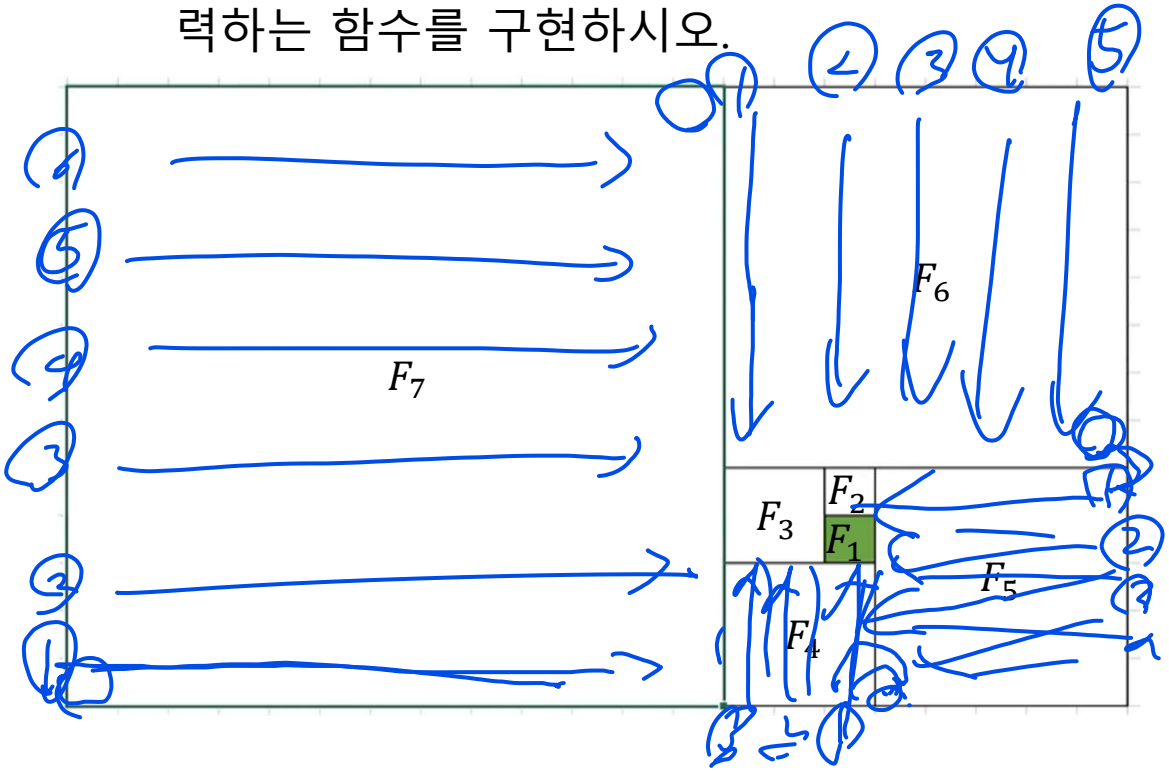
< 실습 2 결과 예시 >

코딩 과제 8-1

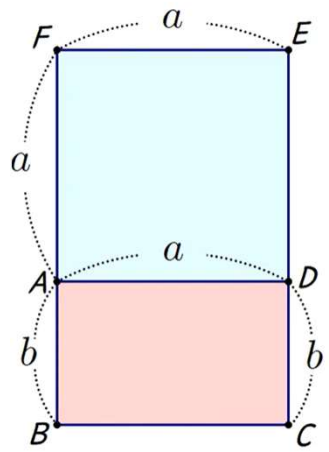
- **동적 계획법**을 이용하여 n번째 피보나치 수와 수열을 구하는 fibo 함수를 작성하시오.

코딩 과제 8-2

- 황금 직사각형은 피보나치 수열을 이용하여 아래 그림처럼 나타낼 수 있다. 구현한 피보나치 수열을 입력으로 받아 아래 그림과 같은 황금 직사각형을 배열 형태로 출력하는 함수를 구현하시오.



F_n
1
1
2
3
5
8
13



$a : b = a + b : a$

황금사각형

$a : b = 1.618 : 1$

코딩 과제 8-2 예시

```
Golden rectangle(length 6):
[[8. 8. 8. 8. 8. 8. 8. 8. 5. 5. 5. 5. 5.]
 [8. 8. 8. 8. 8. 8. 8. 8. 5. 5. 5. 5. 5.]
 [8. 8. 8. 8. 8. 8. 8. 8. 5. 5. 5. 5. 5.]
 [8. 8. 8. 8. 8. 8. 8. 8. 5. 5. 5. 5. 5.]
 [8. 8. 8. 8. 8. 8. 8. 8. 5. 5. 5. 5. 5.]
 [8. 8. 8. 8. 8. 8. 8. 8. 1. 1. 3. 3. 3.]
 [8. 8. 8. 8. 8. 8. 8. 8. 2. 2. 3. 3. 3.]
 [8. 8. 8. 8. 8. 8. 8. 8. 2. 2. 3. 3. 3.]]
```

```
Golden rectangle(length 8):
 [[21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13. 13.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  3.  3.  3.  2.  2.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  3.  3.  3.  2.  2.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  3.  3.  3.  1.  1.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  5.  5.  5.  5.  5.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  5.  5.  5.  5.  5.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  5.  5.  5.  5.  5.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  5.  5.  5.  5.  5.  8.  8.  8.  8.  8.  8.  8.  8.]
 [21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21. 21.  5.  5.  5.  5.  5.  8.  8.  8.  8.  8.  8.  8.  8.]]
```

보고서 과제 8

- 행렬 곱셈을 사용하여 실습 1-3을 해결했다. 이를 활용하여 비트연산을 추가해 fibonacci를 구하는 알고리즘을 수도 코드로 작성하고 설명하시오. 시간 복잡도는 $O(\log_2 n)$으로 줄일 것.
- 글씨 크기 10, 2 페이지 내외

Hint.

1) 2^{64}을 계산하는 방법은 아래 두 가지로 나타낼 수 있다.

1) 2를 64번 곱한다. $2 \times 2 \times 2 \times 2 \times \ldots \times 2 \times 2 = 2^{64}$

=> 64번의 연산, 시간 복잡도 : $O(n)$

2) $2^1 \times 2^1 = 2^2$, $2^2 \times 2^2 = 2^4, 2^4 \times 2^4 = 2^8 \ldots 2^{32} \times 2^{32} = 2^{64}$

=> 6번의 연산, 시간 복잡도 : $O(\log_2 n)$

2) 모든 자연수는 2의 제곱수로 표현 가능하다.

ex) $100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2 = 1100100_2$

2)를 이용하여 $2^{100} = 2^{64+32+4} = 2^{64} \times 2^{32} \times 2^4$ 와 같이 표현이 가능하고 2^{64}, 2^{32}, 2^4 은 1)을 통하여 $O(\log_2 n)$의 시간 복잡도로 계산할 수 있다.