고급 소프트웨어 실습(CSE4152)

3주차

Finonacci number

목차

- 실습 환경
- Fibonacci sequence
- Recursion
- Golden ratio
- Golden rectangle & Golden spiral
- 실습 및 과제

실습 환경

- Google Colaboratory (Colab)
- 브라우저 내에서 Jupyter Notebook 기반의 Python 스크립트를 작성하고 실행 가능
- GPU 무료 제공
- https://colab.research.google.com/?hl=ko

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

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

print(factorial(4))
```

```
24
```

< 재귀를 이용한 factorial 구현(Python) >

factorial(4)

24

return 4 * factorial(3)

6

return 3 * factorial(2)

2

return 2 * factorial(1)

1

return 1 * $\boxed{\text{factorial(0)}}$

Base case

< 코드 흐름도 >

재귀의 장단점

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

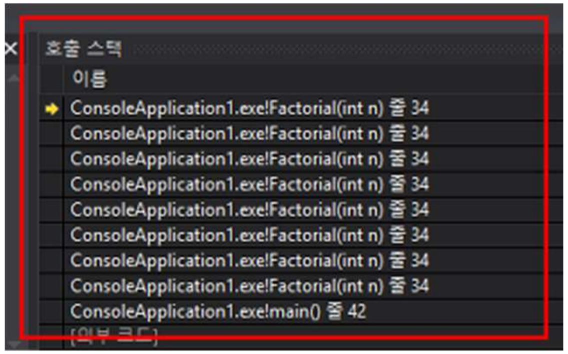
- 단점
  - 지속적으로 재귀 함수를 호출하게 됨에 따라 변수를 여러 번 호출 및 스택에 저장을 하게 됨
  - 이는 곧 **속도 저하 및 메모리 낭비**로 이어짐

- 해결책
  - **<u>꼬리 재귀(tail call recursion)</u>** 사용

    => 함수 return 시 재귀 호출 이후 추가적인 연산을 요구하지 않는 재귀

꼬리 재귀

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



< 일반 재귀 >

실행 코드

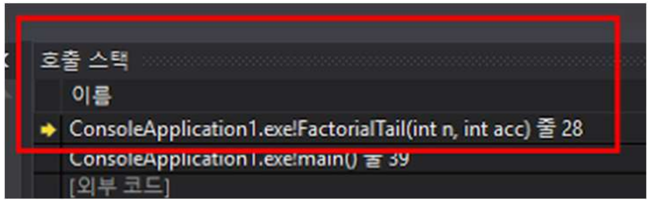
```
def factorialTail(n, acc):
    if n == 0:
        return acc
    return factorialTail(n-1, acc*n)
    # n * factorial()과 같은 return 부분에 추가적인 연산이 없음

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

컴파일러가 해석하는 코드

```
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



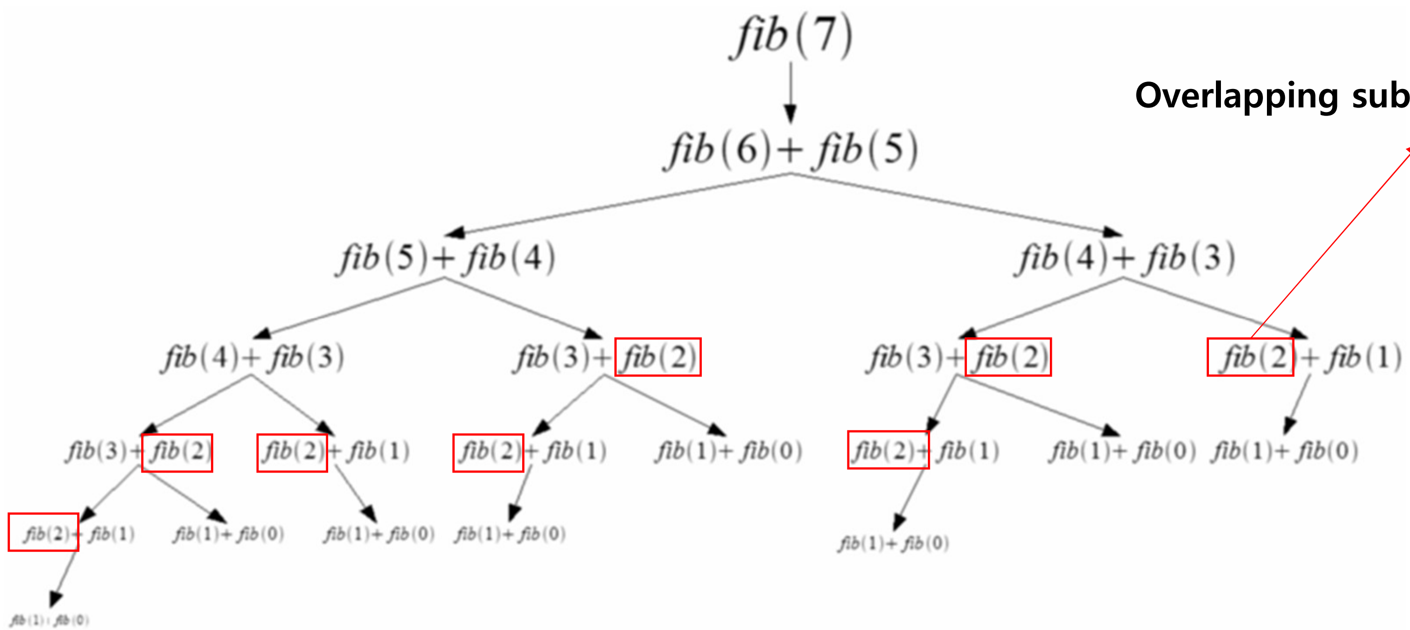
< 꼬리 재귀 >

- 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?

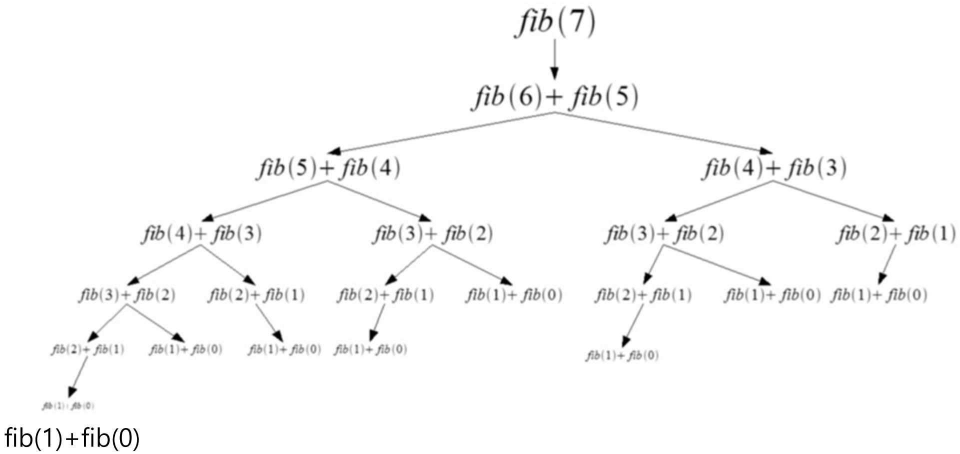
Overlapping subproblem (부분문제 중복)



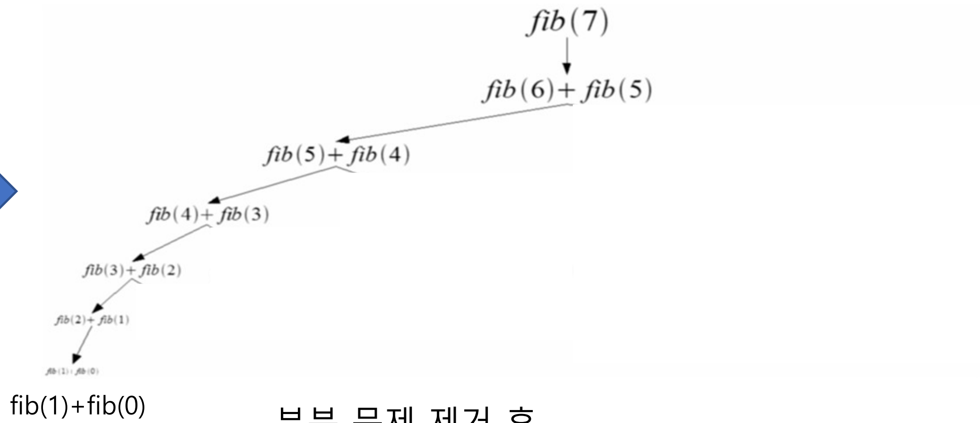
fib(1)+fib(0)

- 과연 이 방법 그대로 코드로 구현한다면 효율적일까?

부분문제 중복이 제거된 피보나치 수열



부분 문제 제거 전



부분 문제 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

문제 1

- 피보나치 수열을 Google Colaboratory 환경에서 코드로 구현
- 재귀, 반복문, 동적 계획법 등과 같은 7가지의 다양한 방법으로 구현
- 행렬 연산을 위해 list 자료형 또는 NumPy 패키지 사용 예정
- 각 방법의 시간 복잡도와 메모리 효율성을 비교하여 최적의 알고리즘 도출이 목적

문제 1-1, 1-2

- **문제 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 $O(2^n)$

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

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

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

  => n번만큼 루프를 반복하므로 시간 복잡도는 $O(n)$

문제 1-3

- **문제 1-3**. 동적 계획법과 재귀적 방법 또는 반복문을 이용하여 n번째 피보나치 수를 구하는 fibo_3 함수를 작성하시오.
- Hint : list나 array 같은 자료구조에 n번째 피보나치 수열의 값을 저장한다
- Hint(재귀적 방법) : 재귀함수의 재귀 종료조건(=base case)을 n<2, 즉 피보나치 수열의 1번째 항일 때로 설정한다.

문제 1-4

- **문제 1-4**. Numpy를 사용한 행렬 곱셈 풀이로 n번째 피보나치 수를 구하는 fibo_4 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.

문제 1-5에서 사용

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

문제 1-5

- **문제 1-5**. 행렬 곱셈을 이용한 풀이를 이용하여 문제 1-4 문제를 해결하는 fibo_5 함수를 구현하시오. 단, numpy를 사용하지말고 시간 복잡도를 $O(\log_2 n)$으로 줄일 것

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

Hint 1 : 문제 1-5

- 행렬을 표현할 자료구조로 Python 기본 자료구조인 list 사용
- List 사용법은 소스에 아래 사용 예시 참조

```
# 빈 리스트 정의 방법
a = list()
a = []

# 2차원 행렬 형태의 리스트 정의
b = [[1, 2], [3, 4]]
print(b)

# 리스트 원소 접근법
b[0]
print(b[0]) # 리스트 b의 1행 원소 출력
b[0][0]
print(b[0][0])# 리스트 b의 1행 1열의 원소 출력

# 반복문을 이용한 리스트 정의
c = [0 for i in range(0, 10)]
print(c)
# i의 초기값은 0이고 루프마다 i값이 1씩 증가하는데 i값이 10이 되면 루프 종료.
# 루프가 한 번 돌 때마다 리스트 c에 원소 0을 추가  => 0이 10개인 리스트 c 정의

d = [j for j in range(0, 10)]
print(d)
# j의 초기값은 0이고 루프마다 i값이 1씩 증가하는데 j값이 10이 되면 루프 종료.
# 루프가 한 번 돌 때마다 리스트 c에 원소 j를 추가  => 0, 1, 2, 3, 4, ... 9를 원소로 가지는 리스트 d 정의
```

Hint 2 : 문제 1-5

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

- 자연수 2가 아닌 행렬 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$의 거듭제곱을 위 방식으로 표현하면 어떨까?

Hint 2 : 문제 1-5

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

- 1) $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64}$을 계산하는 방법도 2^{64}와 같이 아래 두 가지로 나타낼 수 있다.

  1) $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$를 64번 곱한다. $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \dots \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64}$

  => 64번의 연산, 시간 복잡도 : $O(n)$

  2) $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4 \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^8 \dots \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{32} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{32} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64}$

  => 6번의 연산, 시간 복잡도 : $O(\log_2 n)$

- 2) 모든 자연수는 2의 제곱수로 표현 가능하다.

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

- 2)를 이용하여 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{100} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64+32} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{32} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{4}$ 와 같이 표현이 가능하고 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{64}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{32}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{4}$ 은 1)을 통하여 $O(\log_2 n)$의 시간 복잡도로 계산할 수 있다.

- 이 원리를 오른쪽의 피보나치 수열을 나타내는 행렬의 거듭제곱에 이용하여 $O(\log_2 n)$의 시간 복잡도로 피보나치 수열을 구현할 수 있다.

  ex) 100번째 피보나치 수열의 값 = $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{100}$의 <u>1행 2열</u> or <u>2행 1열의 값</u>

Hint 3 : 문제 1-5

- 0 이상의 정수 k에 대해 자연수 n이 2^k를 포함하는지는 <u>**'비트 연산'**</u>을 통해 쉽게 알 수 있다.

ex) 100 = 64 + 32 + 4 = $2^6 + 2^5 + 2^2 = \mathbf{2^6} + \mathbf{2^5} + 2^0 + 2^0 + \mathbf{2^2} + 2^0 + 2^0 = \mathbf{1100100}_2$

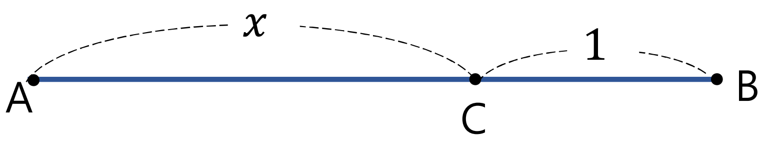
- 거듭제곱의 지수승을 n이라 하면, n을 1씩 left shift 연산을 하여 위 조건을 확인할 수 있다.

	K = 0	K = 1	K = 2	K = 3		K = 8
	1100100_2	1100100_2	1100100_2	1100100_2	……	1100100_2
Left shift (k만큼 shift)	0000001_2	0000010_2	0000100_2	0001000_2	……	1000000_2
and 연산 결과	0000000_2	0000000_2	0000100_2	0000000_2	……	1000000_2
	0	0	2^2	0		2^6

$$=> 2^2 + 2^5 + 2^6 = 1100100_2 = 100$$

Golden Ratio (황금비율)

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

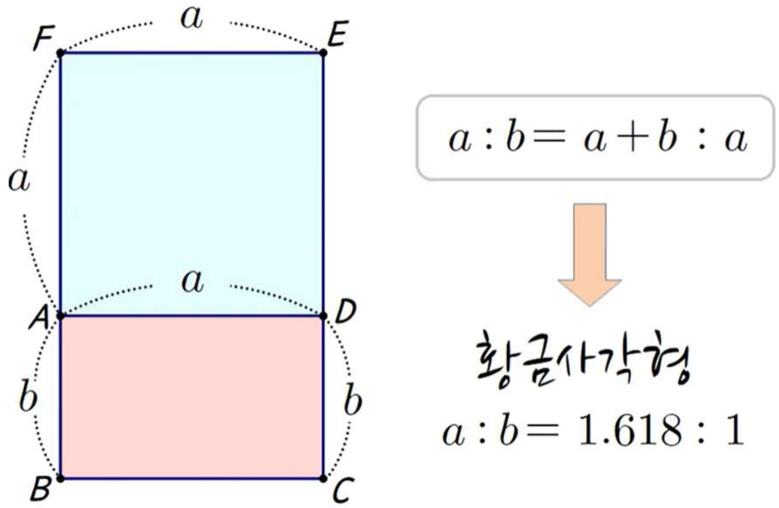


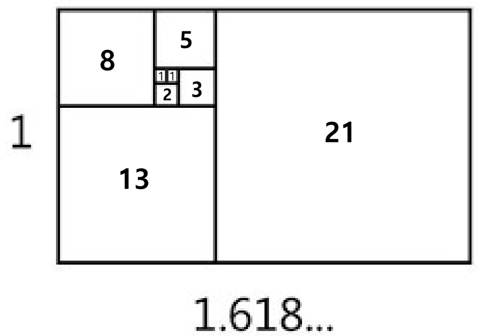
$$x : 1 = x + 1 : x$$

- 황금비 x = 1.6180339 ……..

Golden Rectangle (황금 직사각형)

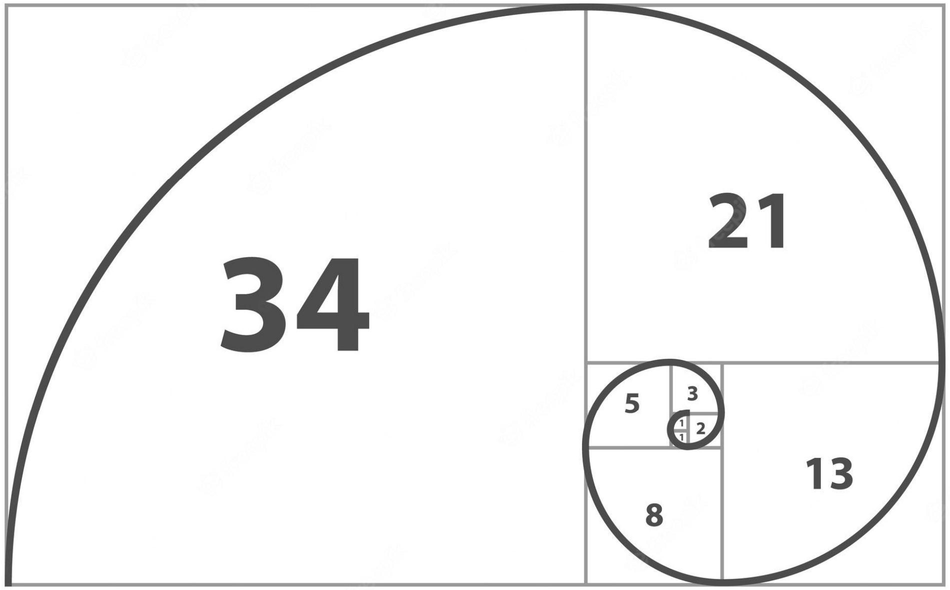
- 황금 직사각형이란 두 변 길이의 비가 1.618:1(=황금비)인 직사각형으로 다음과 같이 나타낼 수 있다.
- 두 변의 비가 황금비가 되도록 직사각형을 반복적으로 추가하면 오른쪽과 같은 결과를 얻을 수 있다.





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.



문제 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
```

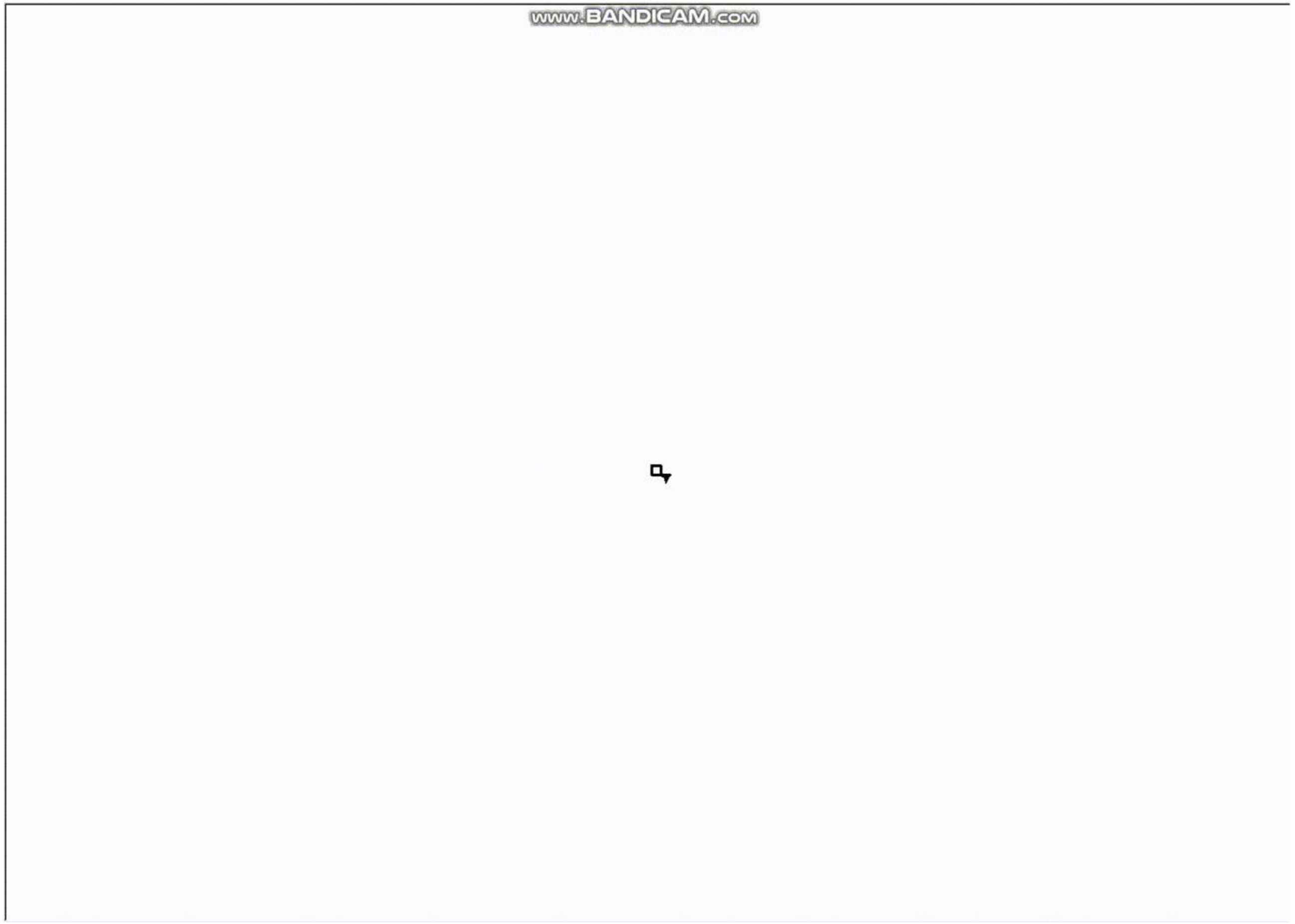
< 문제 2 결과 예시 >

문제 3

- n개의 피보나치 수열에 대한 황금 직사각형과 황금 나선을 구현
- 피보나치 수열과 황금비의 특성을 이용하면 황금 직사각형과 나선의 길이 및 각도를 유추할 수 있음
- Python의 그래픽 모듈 turtle 사용
- 전체 코드 작성이 아닌 코드 빈 칸 채우기 형식으로 실습 예정

* 코드 관련 자료는 각 분반 실습 전날 업로드 예정

문제 3 – 결과 예시



과제

- 황금 직사각형과 황금 나선이 응용되는 예를 각각 두 가지씩 설명하시오.