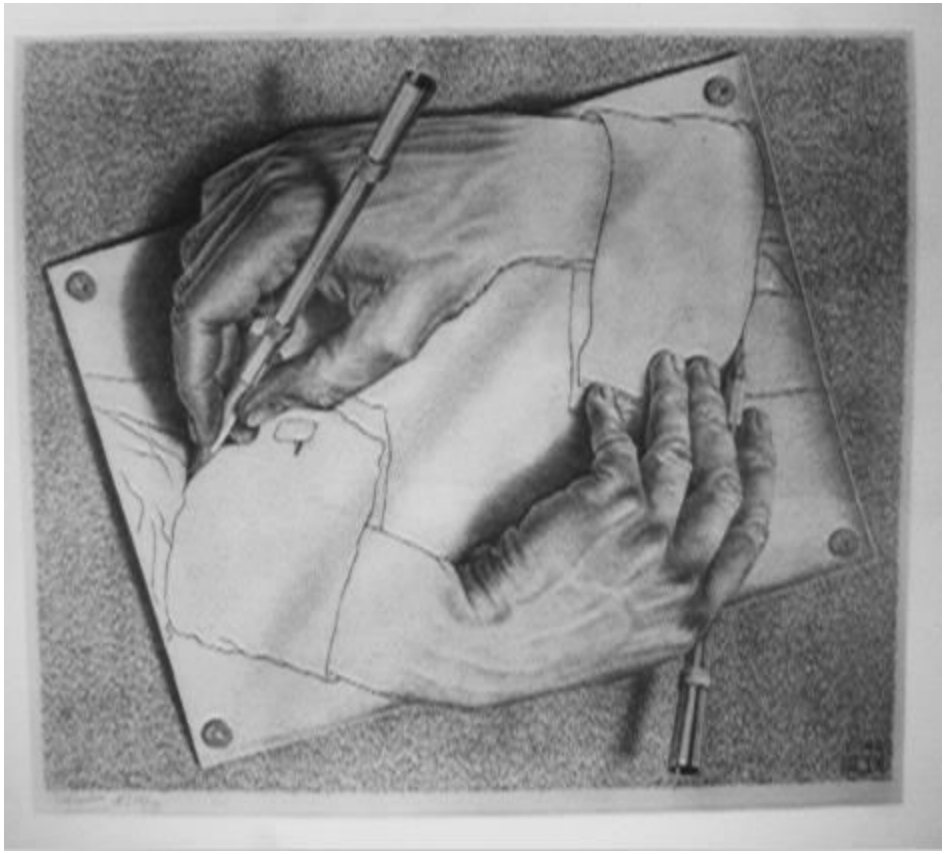
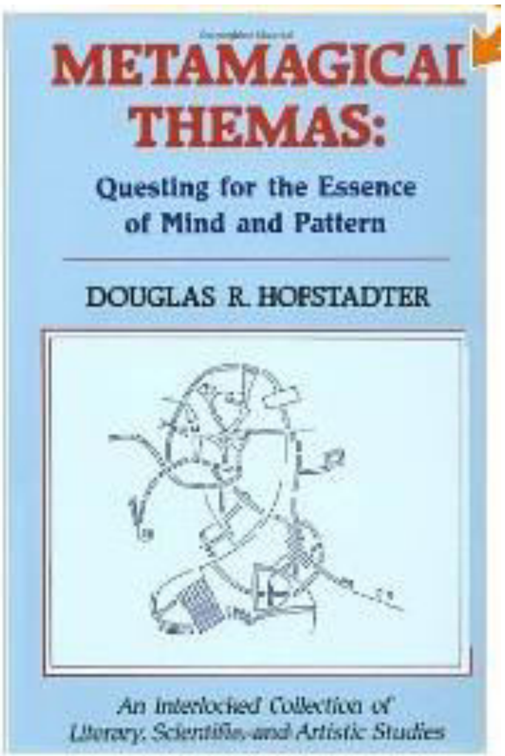
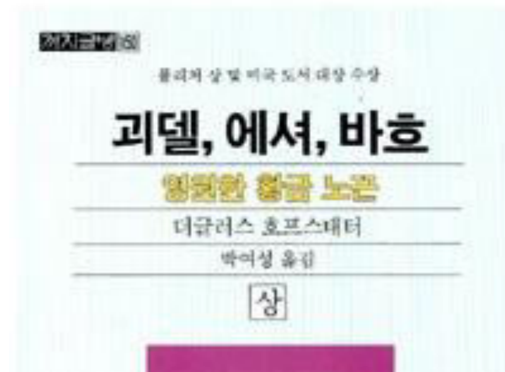
RECURSION

M.C. Escher's "Drawing Hands"



Gödel, Escher, Bach. Anniversary Edition:





Recursion Example 1

```python
def GetSome(count):
    if count == 0: return
    print("재귀 호출 %d" %count)
    GetSome(count-1)

GetSome(3)
```

Result

Main()

Recursion Example 1

```
1.  def GetSome(count):
2.      if count == 0: return
3.      print("재귀 호출 %d" %count)
4.      GetSome(count-1)

5.  GetSome(3)
```

Result

```
재귀 호출3
재귀 호출2
재귀 호출1
```

Recursion Example 2

```
def GetSome(count):
    if count == 0: return
    GetSome(count-1)
    print("재귀호출 %d" % count)


GetSome(3)
```

Result

Main()

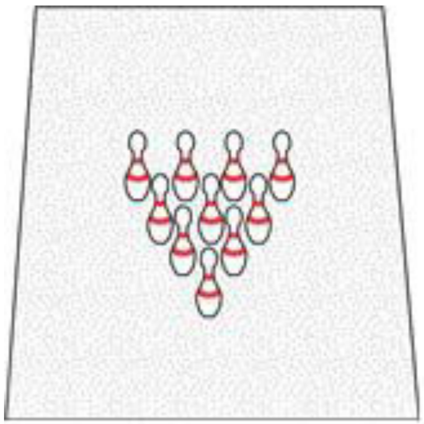
Recursion Example 2

```python
def GetSome(count):
    if count == 0: return
    GetSome(count-1)
    print("재귀호출 %d" % count)


GetSome(3)
```

Result

```
재귀 호출1
재귀 호출2
재귀 호출3
```

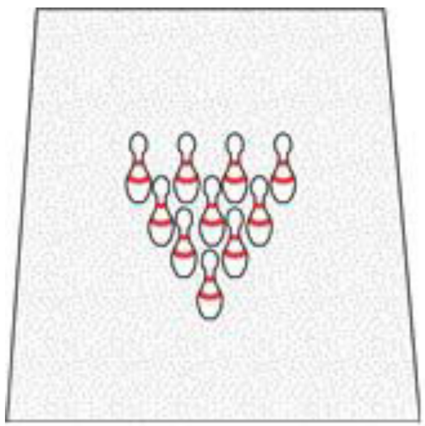
Recursion Example 3



line	pins number	Total	formula
1	1	1	1
2	2		
3	3		
4	4		
5	5		
6	6		
7	7		

Triangle(n) = {
// BASE CASE
Triangle(n) = ______________________________ ;
}

Recursion Example 3



line	pins number	Total	formula
1	1	1	1
2	2	3	2 + T(1)
3	3	6	3 + T(2)
4	4	10	4 + T(3)
5	5	15	5 + T(4)
6	6	21	6 + T(5)
7	7	28	7 + T(6)

$$\text{Triangle}(n) = \begin{cases} 1 & \text{If } n = 1 \quad \text{// BASE CASE} \\ \text{Triangle}(n) = n + \text{Triangle}(n - 1); & \text{otherwise} \end{cases}$$

Recursion Example 4

rev_prt(“", 0, 2);

```
void rev_prt(char S[ ], int a, int b ) {
        if (a > b) return;
        else {
                rev_prt (S, a+1, b);
                printf("%c", S[a]);
        }
}
```

Result

S[ ]

A	B	C	₩n

b	
a	
S	
R.A	Main

rev_prt(“ABC", 0, 2);

b	
a	
S	
R.A	

rev_prt(“ABC", __, __);

b	
a	
S	
R.A	

rev_prt(“ABC", __, __);

b	
a	
S	
R.A	

rev_prt(“ABC", __, __);

Recursion Example 4

```
void rev_prt(char S[ ], int a, int b ) {
        if (a > b) return;
        else {
                rev_prt (S, a+1, b);
                printf("%c", S[a]);
        }
}
```

Result

C
B
A

S[ ]	A	B	C	₩n
	0	1	2	

	printf(S[0])
	rev_prt (S, 1, 2)
b	2
a	0
S	0x100
R.A	Main

rev_prt("ABC", 0, 2);

	printf(S[1])
	rev_prt (S, 2, 2)
b	2
a	1
S	0x100
R.A	

rev_prt("ABC", __, __);

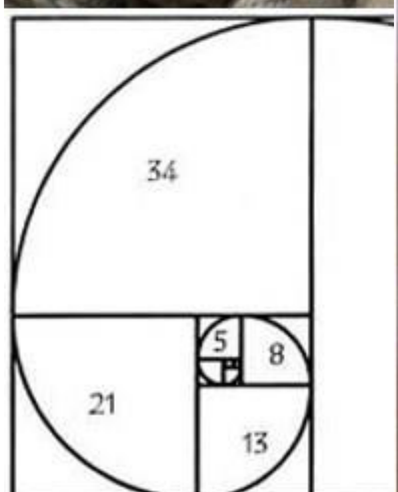
	printf(S[2])
	rev_prt (S, 3, 2)
b	2
a	2
S	0x100
R.A	

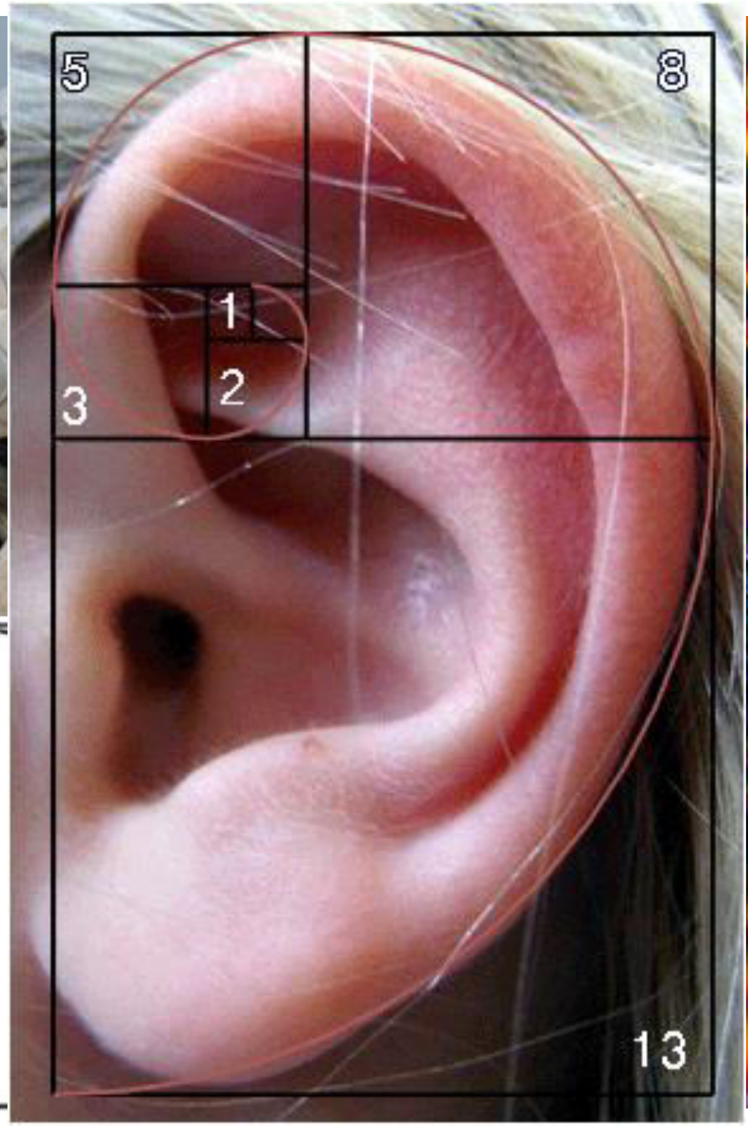
rev_prt("ABC", __, __);

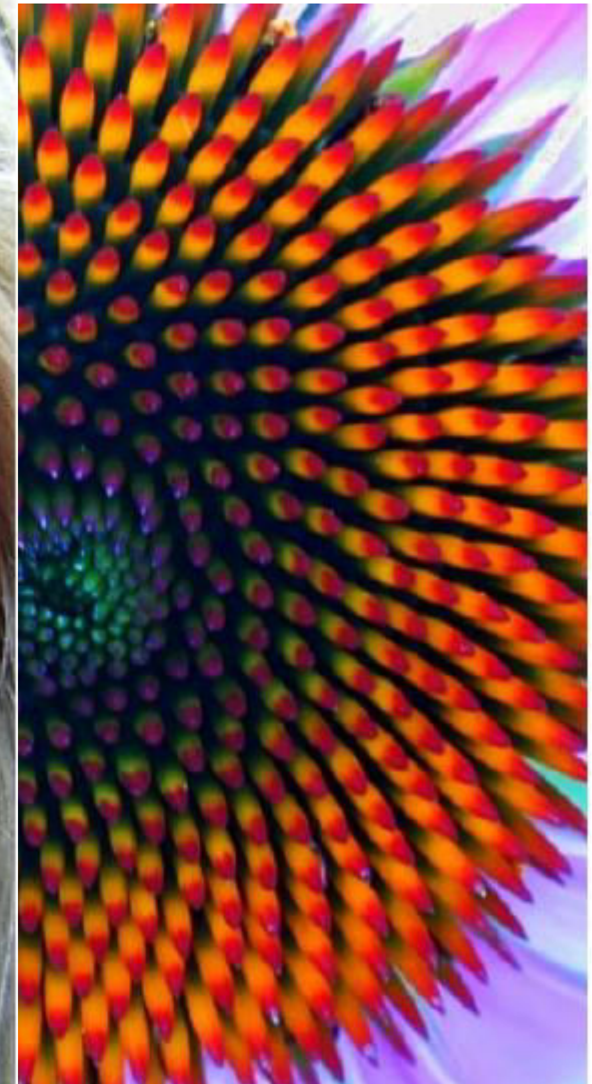
b	2
a	3
S	0x100
R.A	

rev_prt("ABC", __, __);

Fibonacci Number







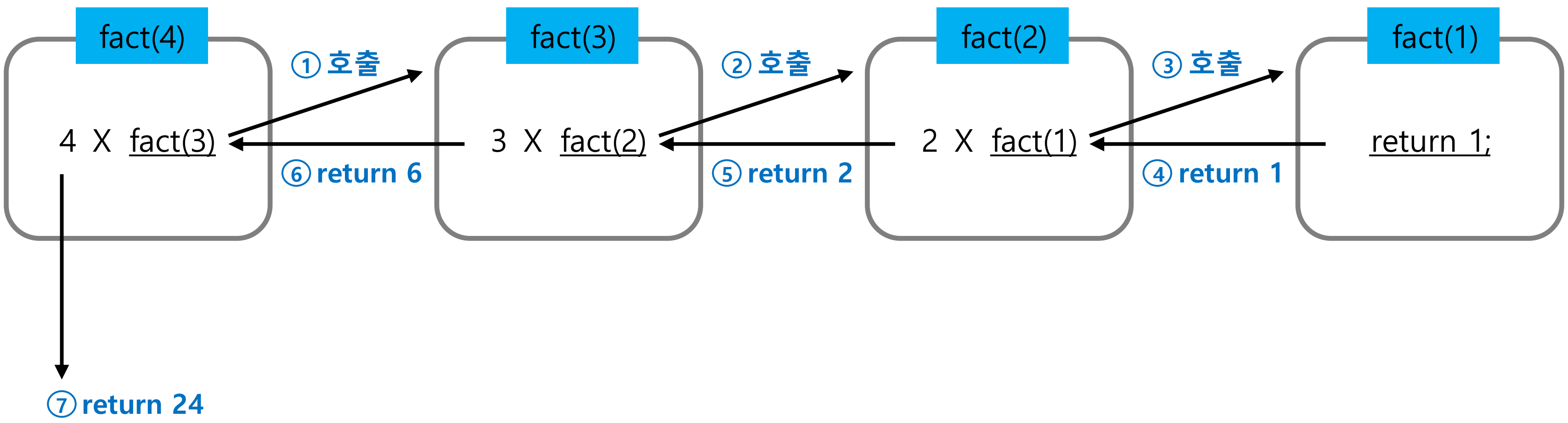
- **자기 자신을 호출하여 순환 수행되는 것**
- **함수에서 실행해야 하는 작업의 특성에 따라 일반적인 호출방식보다 재귀호출방식을 사용하여 함수를 만들면 프로그램의 크기를 줄이고 간단하게 작성**
  - 재귀 호출의 예) factorial

    – n에 대한 factorial : 1부터 n까지의 모든 자연수를 곱하여 구하는 연산

```
 n! = n x (n-1)!
   (n-1)! = (n-1) x (n-2)!
   (n-2)! = (n-2) x (n-3)!
…
   2! = 2 x 1!
   1! = 1
```

    – 마지막에 구한 하위 값을 이용하여 상위 값을 구하는 작업을 반복

factorial 함수에서 n=4 인 경우의 실행



- **0과 1로 시작하고 이전의 두 수 합을 다음 항으로 하는 수열을 피보나치라 한다**
  - 0, 1, 2, 3, 5, 8, 13, ...
- **피보나치 수열의 i번 째 값을 계산하는 함수 F를 정의 하면 다음과 같다.**
  - $F_0 = 0, F_1 = 1$
  - $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$
- **위의 정의로부터 피보나치 수열의 i번째 항을 반환하는 함수를 재귀함수로 구현할 수 있다.**

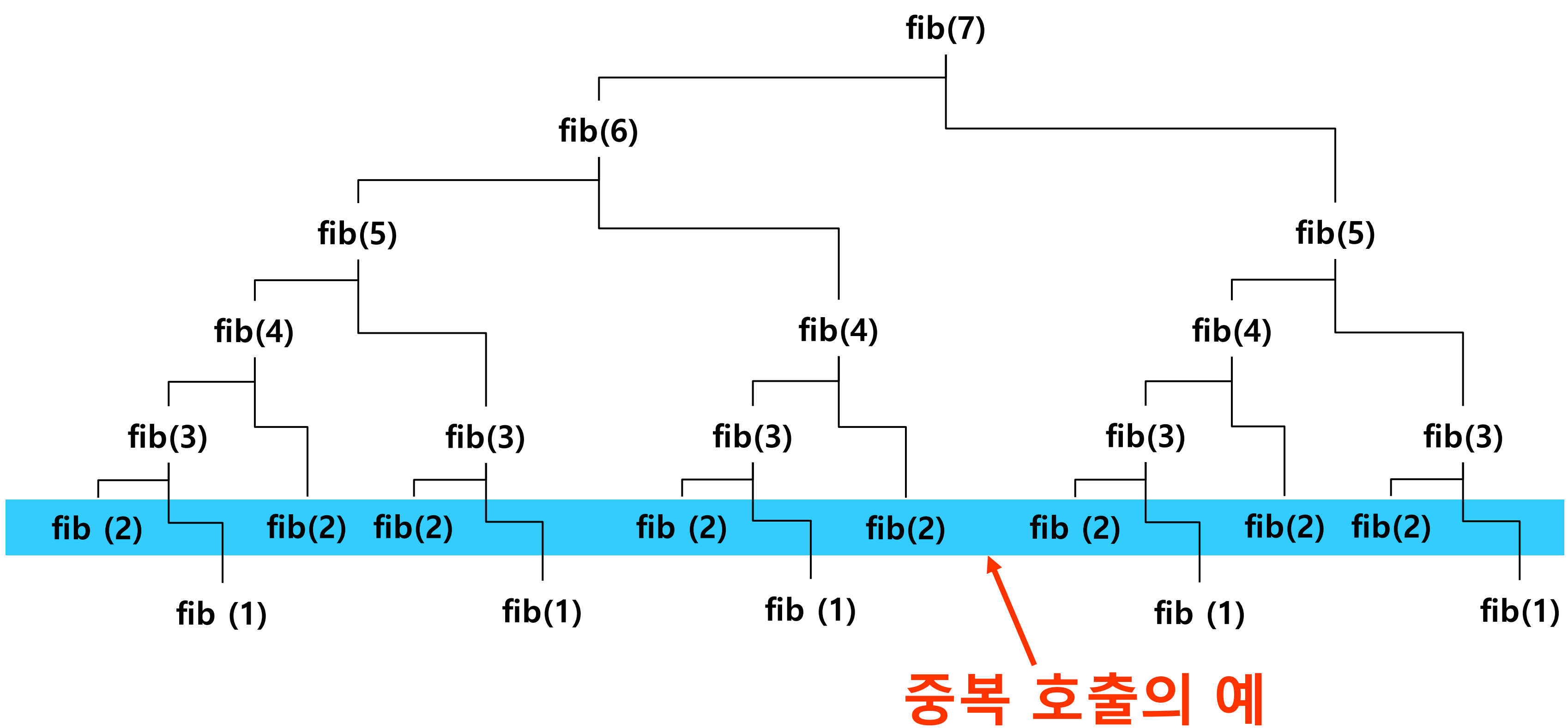
피보나치 수를 구하는 재귀함수

```
def fibo(n) :
    if n < 2 :
        return n
    else :
        return fibo(n-1) + fibo(n-2)
```

- 앞의 예에서 피보나치 수를 구하는 함수를 재귀함수로 구현한 알고리즘은 문제점이 있다.
- “엄청난 중복 호출이 존재한다”는 것이다.

피보나치 수열의 Call Tree

$\Theta(2^n)$



- 메모이제이션(memoization)은 컴퓨터 프로그램을 실행할 때 이전에 계산한 값을 메모리에 저장해서 매번 다시 계산하지 않도록 하여 전체적인 실행속도를 빠르게 하는 기술이다. 동적 계획법의 핵심이 되는 기술이다.

- ‘memoization’은 글자 그대로 해석하면 ‘메모리에 넣기(to put in memory)’라는 의미이며‘기억되어야 할 것’이라는 뜻의 라틴어 memorandum에서 파생되었다. 흔히‘기억하기’,‘암기하기’라는 뜻의 memorization과 혼동하지만, 정확한 단어는 memoization이다. 동사형은 memoize이다.

Fibonacci number- MeMoiZation

Fibonacci(5) =Fibonacci(3)+Fibonacci(4)

Fibonacci(3)?

Fibonacci(1)? **Return 1**

Fibonacci(2)? **Return 1**

F(3) = 1+ 1=2 **Return 2**

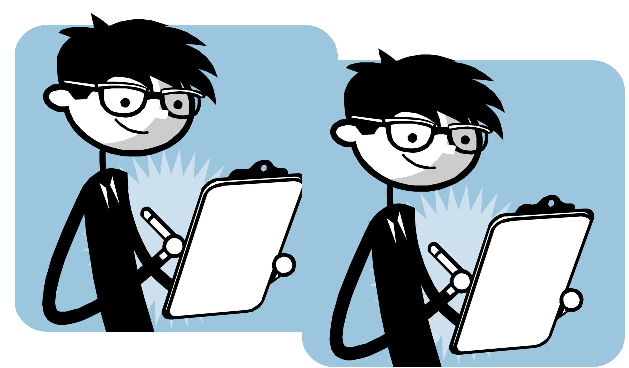
Fibonacci(4)?

Fibonacci(3)? **Return 2**

Fibonacci(2)? **Return 1**

F(4) = 2 + 1 = 3 **Return 3**

F(5) = 3 + 2 = 5 **Return 5**



- 앞의 예에서 피보나치 수를 구하는 알고리즘에서 fibo(n)의 값을 계산하자마자 저장하면(memoize), 실행시간을 Θ(n)으로 줄일 수 있다.

- Memoization 방법을 적용한 알고리즘은 다음과 같다.

```
memo를 위한 배열을 할당하고, 모두 0으로 초기화 한다;
memo[0]을 0으로 memo[1]는 1로 초기화 한다;

def fibo1(n) :
    global memo
    if n >= 2 and len(memo) <= n :
        memo.append(fibo1(n-1) + fibo1(n-2))
    return memo[n]

memo = [0, 1]
```

- 동적 계획 (Dynamic Programming) 알고리즘은 그리디 알고리즘과 같이 최적화 문제를 해결하는 알고리즘이다.

- 동적 계획 알고리즘은 먼저 입력 크기가 작은 부분 문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여, 최종적으로 원래 주어진 입력의 문제를 해결하는 알고리즘이다.

피보나치 수 DP 적용

- 피보나치 수는 부분 문제의 답으로부터 본 문제의 답을 얻을 수 있으므로 최적 부분 구조로 이루어져 있다

1) 문제를 부분 문제로 분할한다.

- Fibonacci(n) 함수는 Fibonacci(n-1)과 Fibonacci(n-2)의 합
- Fibonacci(n-1)은 Fibonacci(n-2)와 Fibonacci(n-3)의 합
- Fibonacci(2)는 Fibonacci(1)과 Fibonacci(0)의 합
- Fibonacci(n)은 Fibonacci(n-1), Fibonacci(n-2), ... Fibonacci(2), Fibonacci(1), Fibonacci(0) 의 부분집합으로 나뉜다

2) 부분 문제로 나누는 일을 끝냈으면 가장 작은 부분 문제부터 해를 구한다.

3) 그 결과는 테이블에 저장하고, 테이블에 저장된 부분 문제의 해를 이용하여 상위 문제의 해를 구한다.

테이블 인덱스	저장되어 있는 값
[0]	0
[1]	1
[2]	1
[3]	2
[4]	3
…	…
[n]	fibo(n)

피보나치 수 DP 적용 알고리즘

```
def fibo2(n) :
    f = [0, 1]

    for i in range(2, n + 1) :
        f.append(f[i-1] + f[i-2])

    return f[n]
```

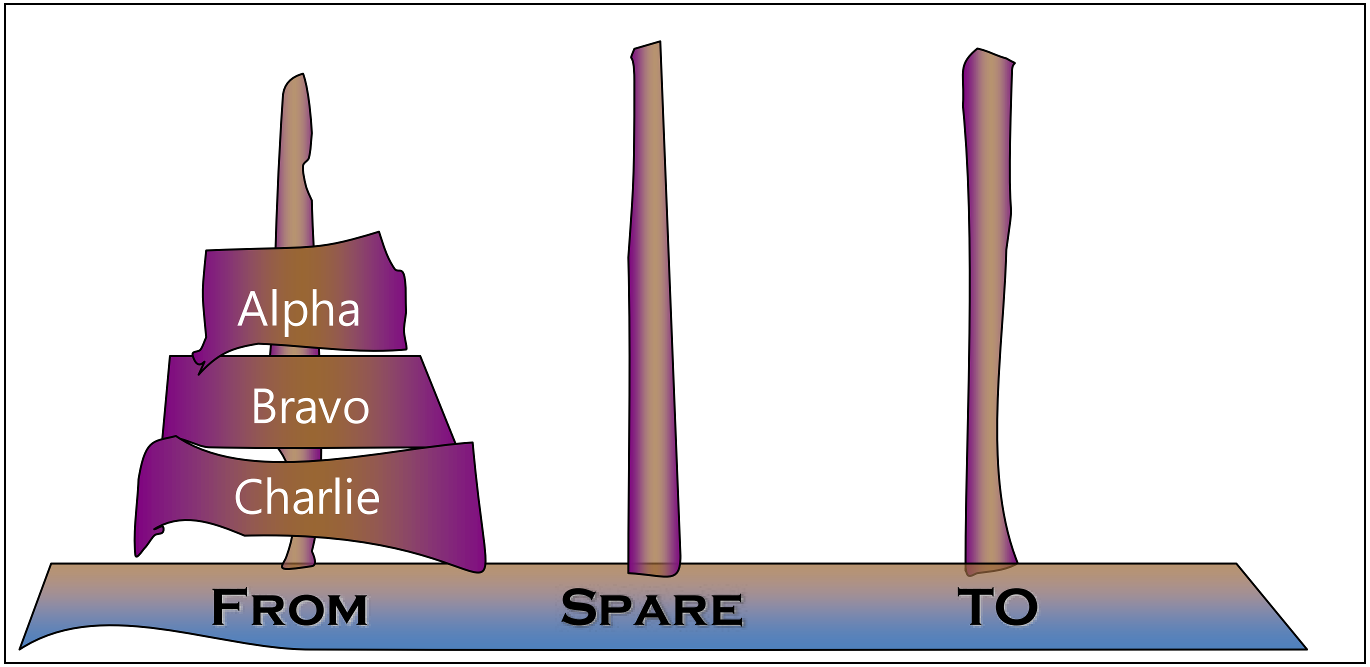
DP의 구현 방식

- recursive 방식 : fib1()
- iterative 방식 : fib2()

- memoization을 재귀적 구조에 사용하는 것보다 반복적 구조로 DP를 구현한 것이 성능 면에서 보다 효율적이다.
- 재귀적 구조는 내부에 시스템 호출 스택을 사용하는 오버헤드가 발생하기 때문이다.

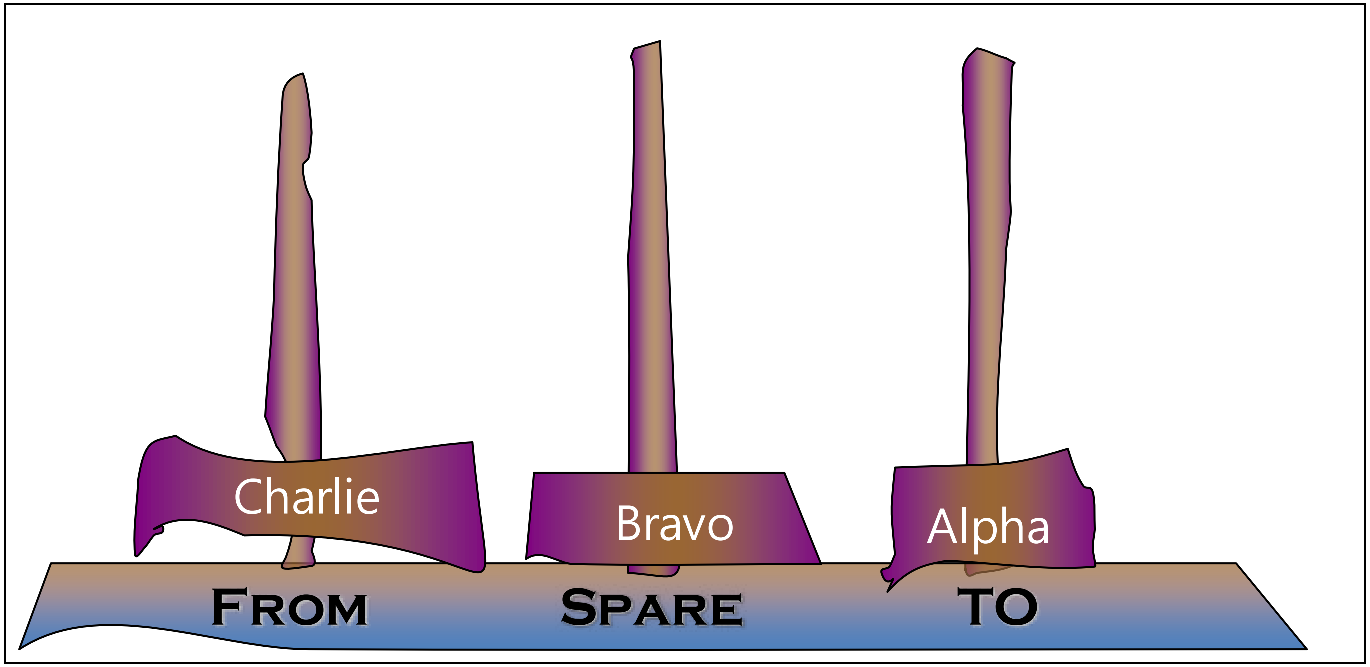
Hanoi Tower

- Move n (3) disks from From to to
  - Move n-1 (2) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (2) disks from **SPARE** to **TO**



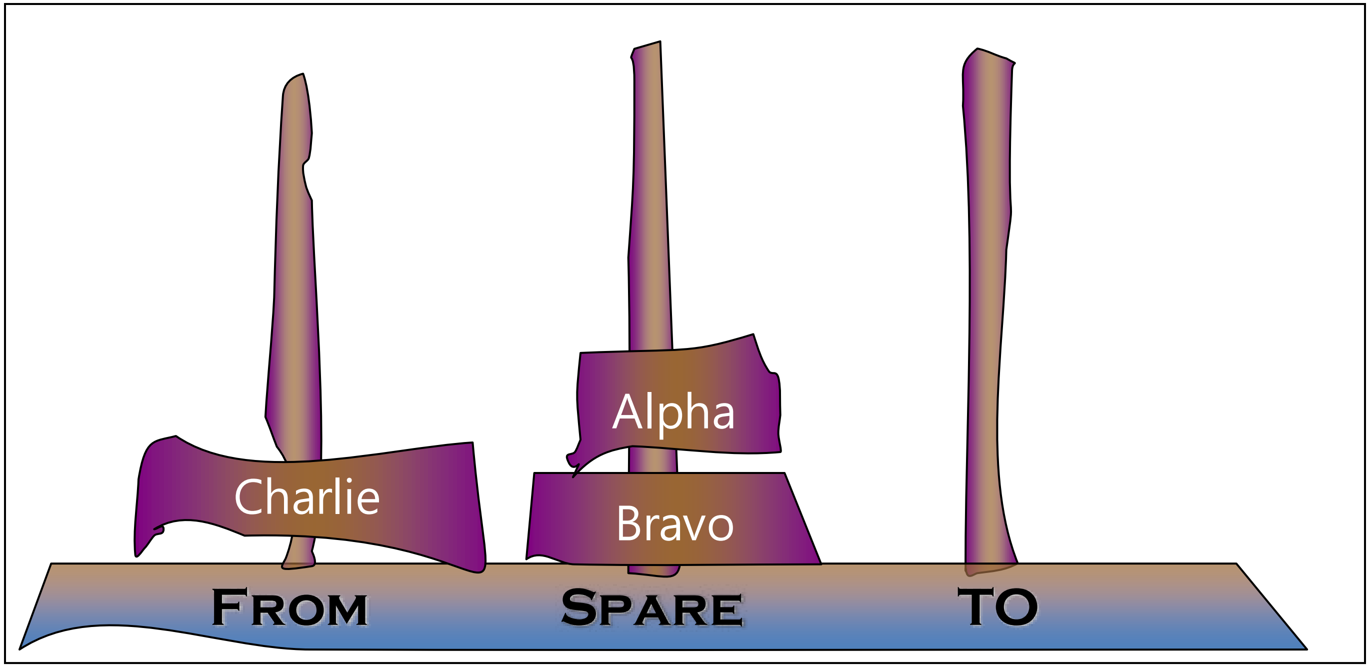
Hanoi Tower

- Move n (3) disks from From to to
  - Move n-1 (2) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (2) disks from **SPARE** to **TO**



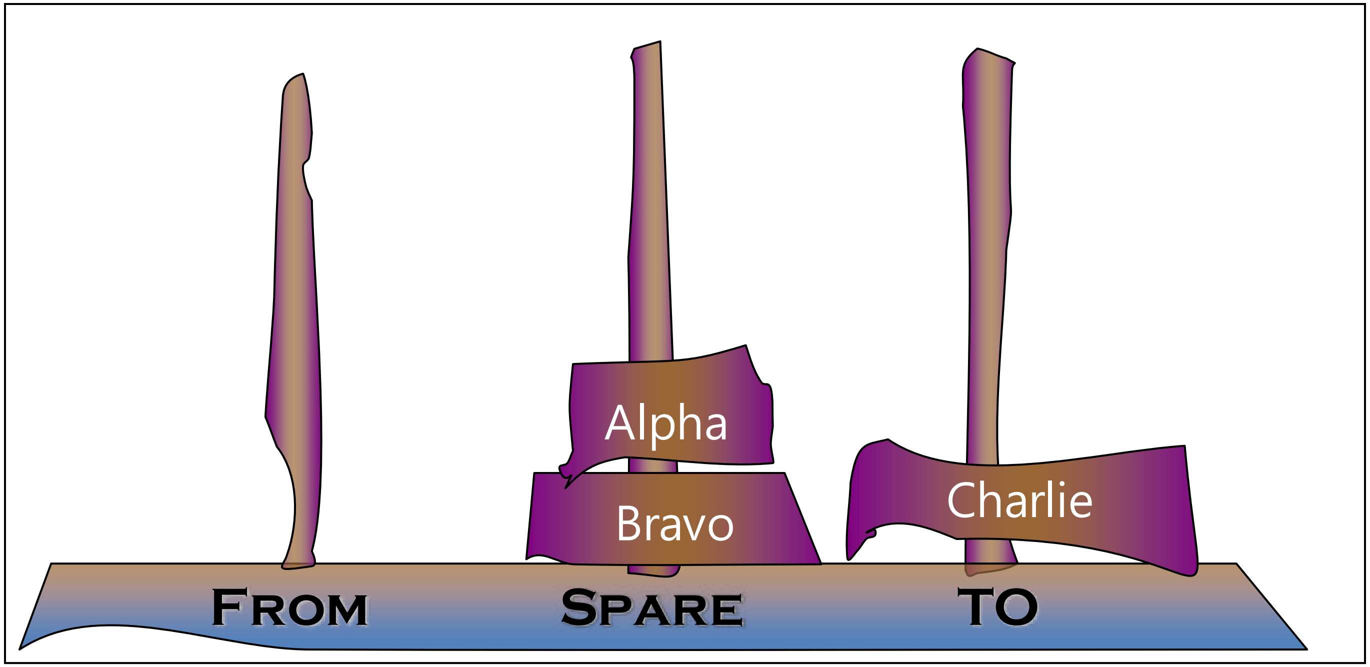
Hanoi Tower

- Move n (3) disks from From to to
  - Move n-1 (2) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (2) disks from **SPARE** to **TO**



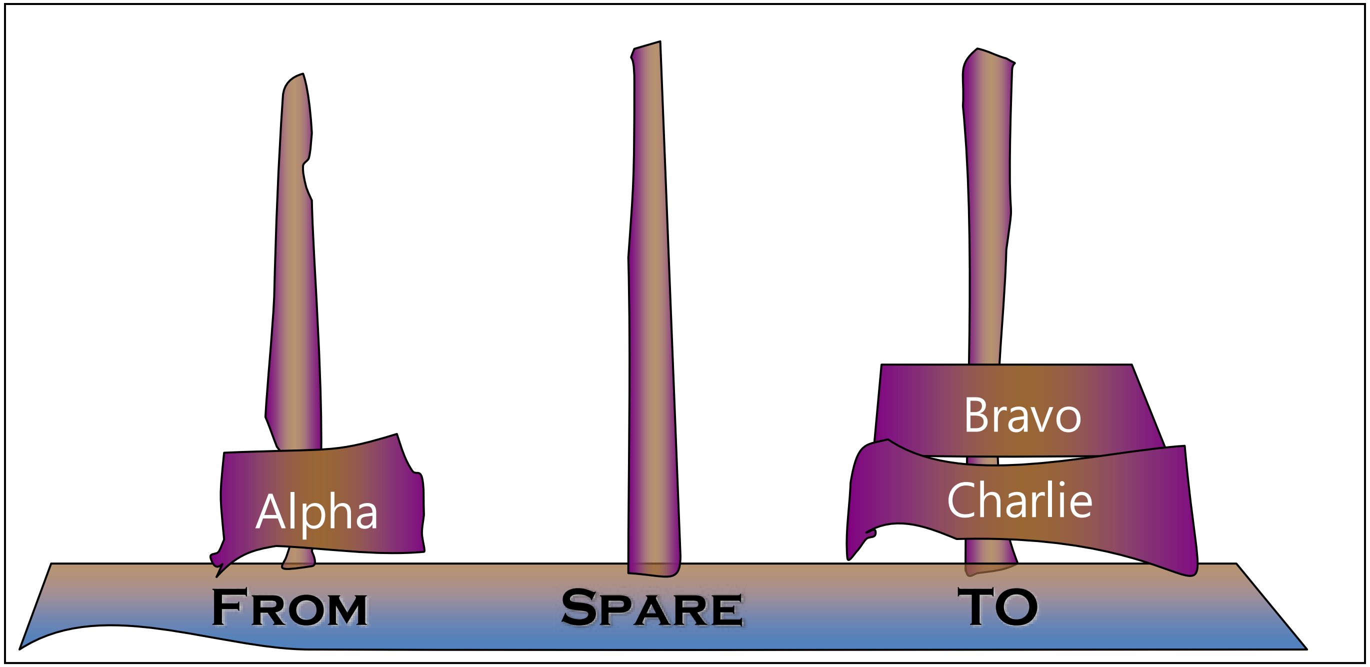
Hanoi towers

- Move n (3) disks from From to to
  - Move n-1 (2) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (2) disks from **SPARE** to **TO**



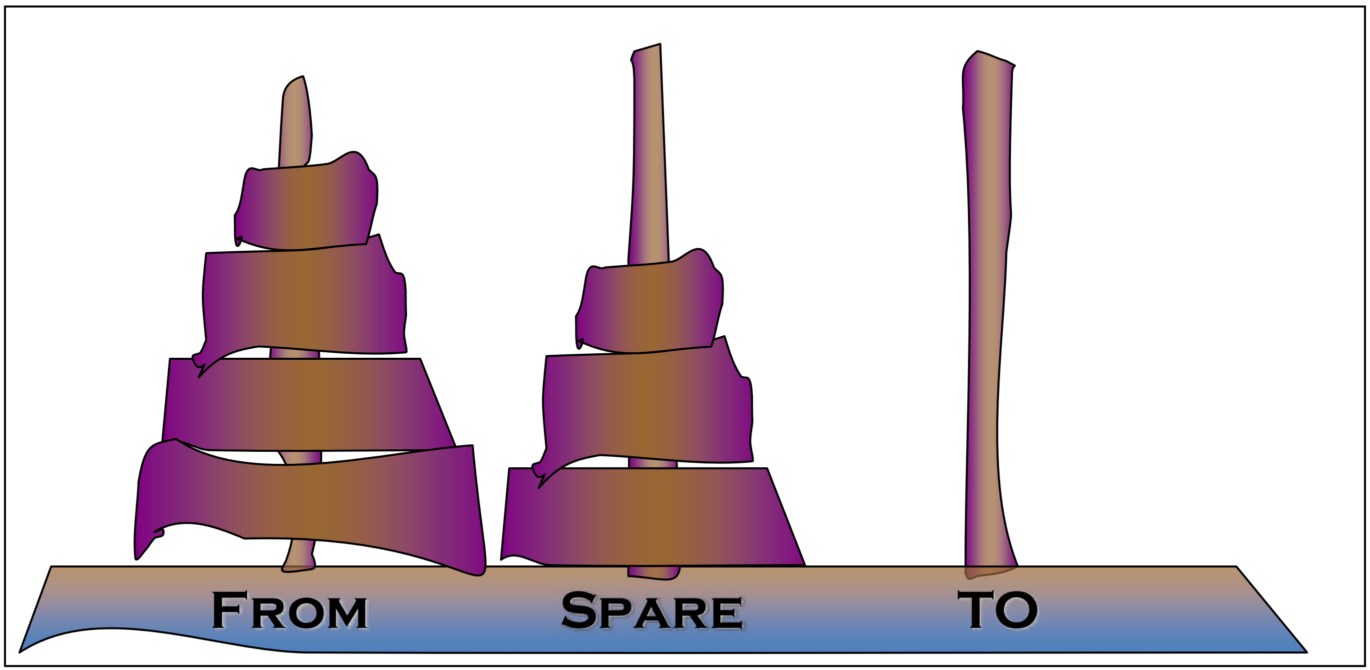
Hanoi towers

- Move n (3) disks from From to to
  - Move n-1 (2) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (2) disks from **SPARE** to **TO**



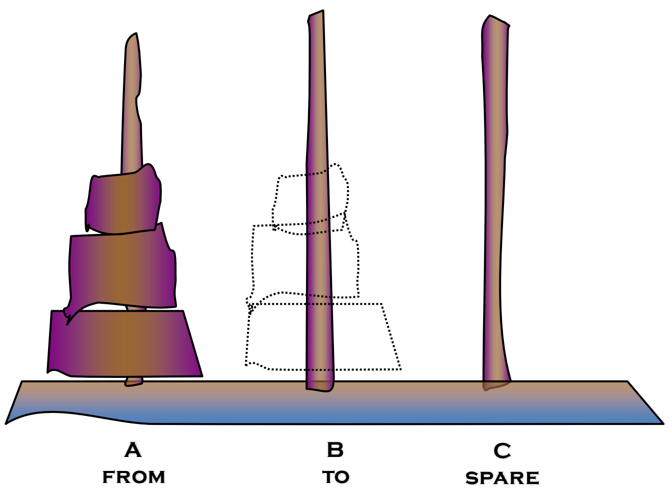
Hanoi towers

- Move n (4) disks from **FROM** to **TO**
  - Move n-1 (3) disks from **FROM** to **SPARE**
  - Move 1 disk from **FROM** to **TO**
  - Move n-1 (3) disks from **SPARE** to **TO**



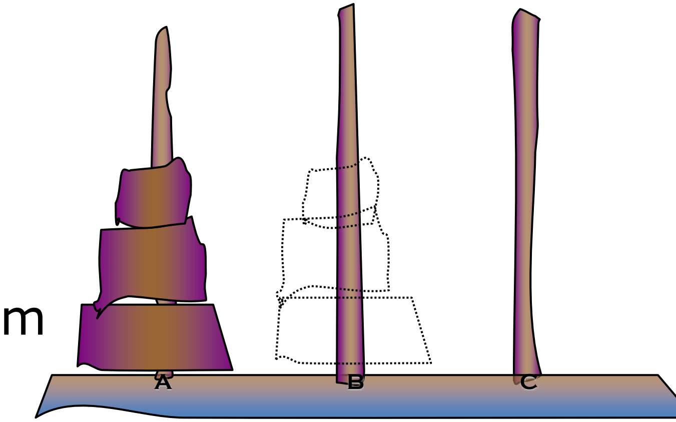
Hanoi towers

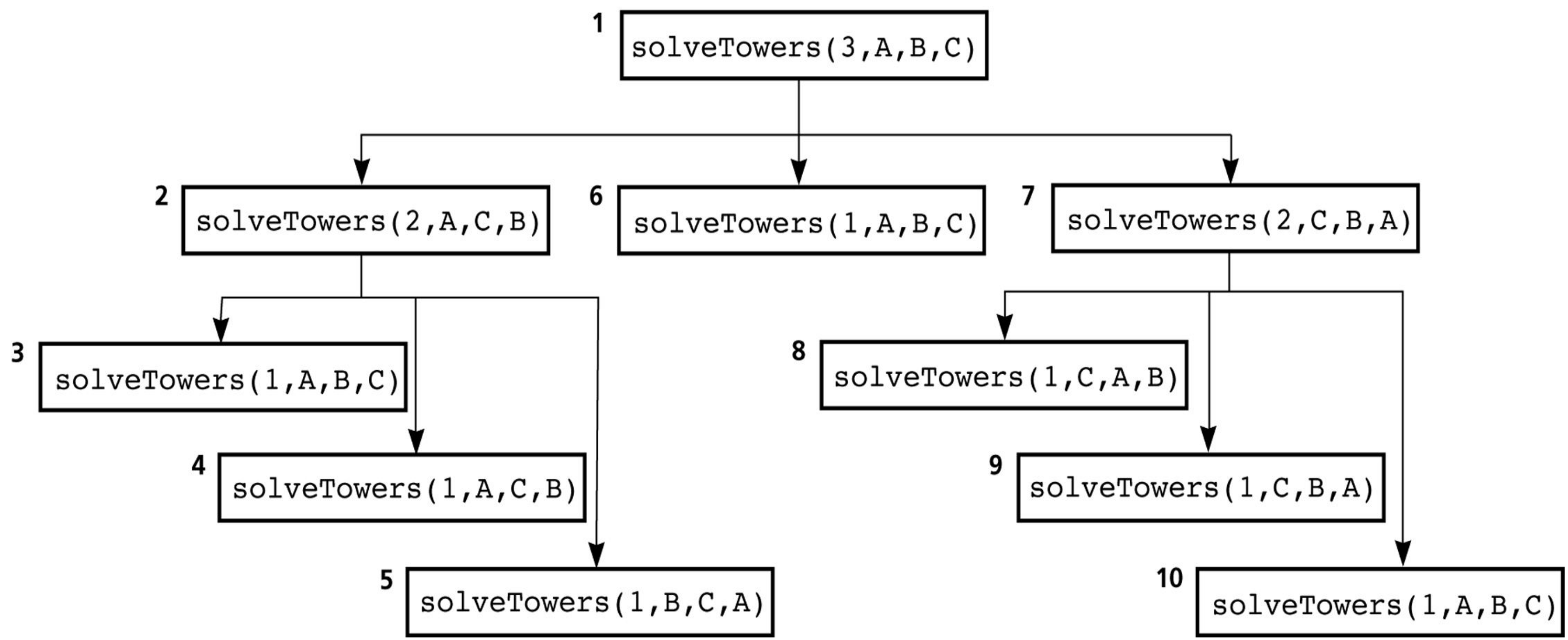
```
def hanoi(n, ffrom, to, spare):
    if n > 0:
        hanoi(n-1, ffrom, spare, to)
        print("%d번 원반을 %s에서 %s로 옮김" %(n, ffrom, to))
        hanoi(n-1, spare, to, ffrom)

print("원반의 갯수 : ")
n = int(input())
hanoi(n, 'from', 'to' , 'spare' )
```



Hanoi towers

- Recursion tree:
- The order of recursive calls that results from ***solveTowers(3,A,B,C)***





Exercise
Going up the Stairs

- Gildong tries to climb n steps.
- Gildong can climb first or second steps at a time, depending on the mood as he goes up the stairs.
- When the height n of the stairs is given, write a program to find the number of cases in which Gildong can climb this step.
- If there are three stairs, Gildong will go up 1, 1, 1, or 1,2 or 2, 1.
- There are three different ways to get up to.
- Input
  - Number of stairs n is entered (only n is a natural number less than 20).
- Output
  - Gil-Dong prints out all the ways to climb the stairs.
- Input Output

Input	Output
3	3

Going up the Stairs