Recursion

"To understand recursion, one must first understand recursion."

Element of Programming

Primitive expressions	which represent the simplest entities the language is concerned with
means of combination	by which compound elements are built from simpler ones, and
means of abstraction	by which compound elements can be named and manipulated as units.

Square, sum of square

f(a) = $(a + 1)^2 + (2a)^2$

```
def f(a):
    return sum_of_square(a + 1, a * 2)
```

sum_of_squre(x, y): $x^2 + y^2$

```
def sum_of_square(x, y):
    return square(x) + square(y)
```

square(x): x^2

```
def square(x):
    return x * x
```

Substitution model - Applicative order

To apply a function to arguments, evaluate the return expression of the function with each parameter replaced by the corresponding argument.

Applicative order vs *normal order*

f(5)

sum_of_squres(5+1, 5*2)

square(6) + square(10)

(6 * 6) + (10 * 10)

36 + 100

136

```
def f(a):
    return sum_of_square(a+1, a*2)
def sum_of_squares(x,y):
    return square(x) + square(y)
def square(x):
    return x * x
```

Substitution model – Normal order evaluation

f(5)

sum_of_squres(5+1, 5*2)

square(5+1) + square(5*2)

((5+1) * (5+1)) + ((5*2) * (5*2))

6 * 6 + 10 * 10

36 + 100

136

Test for normal order evaluation

```
def p():
	while true:
		pass

def test(x, y):
	return 0 if x == 0 else y

test(0, p())
```

Sum

sum(10) = $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$

```
# write a function to return sum of integers starting from 0 to n

def nsum(n):

    ...
```

Sum - Iteration

```python
# iteration 1
def nsum(n):
    total = 0
    for i in range(n+1):
        total += i
    return total
```

```python
# iteration 2
def nsum(n):
    return sum([i for i in range(n + 1)])


# iteration 3
def nsum(n):
    return sum(list(range(n+1)))
```

Sum - Recursion

sum(n) = n + sum(n-1)

sum(5)

5 + sum(4)

5 + (4 + (sum(3))

5 + (4 + (3 + sum(2)))

5 + (4 + (3 + (2 + sum(1))))

5 + (4 + (3 + (2 + (1 + sum(0)))))

5 + (4 + (3 + (2 + (1 + 0))))))

Sum – Recursion (I)

“I tried to write a recursive function… but I forgot the base case. I’m still waiting for it to return.”

```
# recursive version 1

def sum(n):

    return n + sum(n-1)
```

sum(5)

5 + sum(4)

5 + (4 + sum(3))

5 + (4 + (3 + sum(2)))

5 + (4 + (3 + (2 + sum(1))))

5 + (4 + (3 + (2 + (1 + (sum(0)))))

5 + (4 + (3 + (2 + (1 + (0 + sum(-1))))))

5 + (4 + (3 + (2 + (1 + (0 + (-1 + sum(-2)))))))

…

RecursionError

Sum - Recursion (II)

```
# write a function to return sum of integers
starting from 0 to n

def sum(n):

    if n == 0:

        return 0

    else:

        return n + sum(n-1)
```

sum(5)

5 + sum(4)

5 + (4 + sum(3))

5 + (4 + (3 + sum(2))) 1

5 + (4 + (3 + (2 + sum(1))))

5 + (4 + (3 + (2 + (1 + (sum(0))))) 2

5 + (4 + (3 + (2 + (1 + 0))))

5 + (4 + (3 + (2 + 1)))

5 + (4 + (3 + 3))

5 + (4 + 6)

5 + 10

15

Sum - Tail Recursion

```
def sum_iter(n, total):
    if n == 0:
        return total
    else:
        return sum_iter(n-1, total + n)
def sum(n):
  return sum_iter(n, 0)
```

sum(5)

sum_iter(5, 0)

sum_iter(4, 5)

sum_iter(3, 9)

sum_iter(2, 12)

sum_iter(1, 14)

sum_iter(0, 15)

15

Exponentiation(거듭제곱)

$$
\begin{aligned}
b^n &= b \cdot b^{n-1} \\
b^0 &= 1
\end{aligned}
$$

```
# recursion version
def expt(b, n):
        if n == 0:
                return 1
        else:
                return n * expt(b, n -1)
```

```
# iteration version

def expt_iter(b, n):
        product = 1
        for i in range(n,1):
                product *= n
        return product
```

```
# tail recursion version
def expt_iter(b, counter, product):
        if counter == 0
                return product
        Else:
                return expt_iter(b, counter - 1, b * product)
```

Fast exponentiation

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

$$\begin{aligned} b^n &= (b^{n/2})^2 && \text{if } n \text{ is even} \\ b^n &= b \cdot b^{n-1} && \text{if } n \text{ is odd} \end{aligned}$$

```
def fast_expt(b, n):
    if n == 0:
        return 1
    else:
        if is_even(n):
            return square(fast_expt(b, n/2))
        else:
            return b * fast_expt(b, n-1)
```

Fast exponentiation - iteration

```
def fast_expt_iter(b, n):
    product = 1
    while n > 0:
        if is_even(n):
            b = square(b)
            n = n/2
        else:
            product *= b
            n = n-1
    return product
```

Tree Recursion

- Fibonacci number

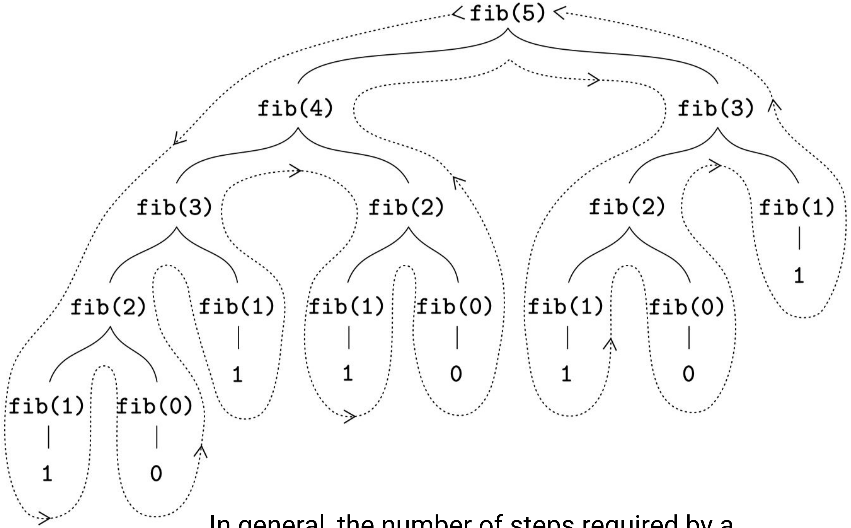
  0, 1, 1, 2, 3, 5, 8, 13, 21,...

- Fib definition

$$
\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & \text{otherwise} \end{cases}
$$

Fibonacci - Recursion

```
# write a function to calculate the fibonacci number

def fib(n):

    if n == 0:

        return 0

    elif n == 1:

        return 1

    else:

        return fib(n-1) + fib(n-2)
```



In general, the number of steps required by a tree-recursive process will be proportional to <u>the number of nodes in the tree</u>, while the space required will be proportional to t<u>he maximum depth of the tree</u>.

Fibonacci - Tail Recursion

```
# write a function to calculate the fibonacci number

def fib(n):

    def fib_iter(a, b, count):

        if count == 0:

            return b

        else:

            return fib_iter(b, a+b, count-1)

    return fib_iter(0, 1, n)
```

fib(5)

fib_iter(0,1,5)

fib_iter(1, 1, 4)

fib_iter(1, 2, 3)

fib_iter(2, 3, 2)

fib_iter(3, 5, 1)

fib_iter(5, 8, 0)

8

Function as black-box abstraction

- Sqrt root definition in math
  - *declarative* (what is)

$$\sqrt{x} \;=\; \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 \;=\; x$$

- How to write the code?
  - *imperative* (how to)
  - Let have a guess y for the value of square root of a number x:
  - Better guess = (y + x/y) / 2

Guess	Quotient	Average
1	$\frac{2}{1} = 2$	$\frac{(2+1)}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{(1.3333+1.5)}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{(1.4167+1.4118)}{2} = 1.4142$
1.4142	...	...

Sqrt-root

```
def sqrt_iter(guess, x):
        if is_good_enough(guess, x):
                return guess
        else:
                return sqrt_iter(improve(guess,x), x)
```

```
def improve(x):
        return average(guess, x / guess)
```

```
def average(x, y):
        return (x + y ) / 2
```

```
def is_good_enough(guess, x):
        return abs(square(guess) - x) < 0.001
```

Recursion: Break the problem into smaller subproblems

재귀는 큰 문제를 작은 문제로 나누고, 그 작은 문제가 자기 자신과 구조가 동일할 때 적용할 수 있는 해결 방법

- **문제 파악**: 이 문제를 자기 자신보다 작은 하위 문제로 표현할 수 있는가?
- **기저 조건 정의**: 언제 재귀 호출을 멈출 것인가?
- **하위 문제 호출**: 더 작은 인풋에 대해 자기 자신을 호출
- **결과 조합**: 하위 문제의 결과를 조합하여 전체 문제 해결

Counting change problem

문제 : 주어진 금액을, 사용 가능한 동전 단위를 이용해 얼마나 많은 방법으로 거슬러 줄 수 있는가?

예시: 금액 4, 동전 종류 [1, 2, 3] → 가능한 경우의 수는 4가지:

- 1+1+1+1
- 1+1+2
- 1+3
- 2+2

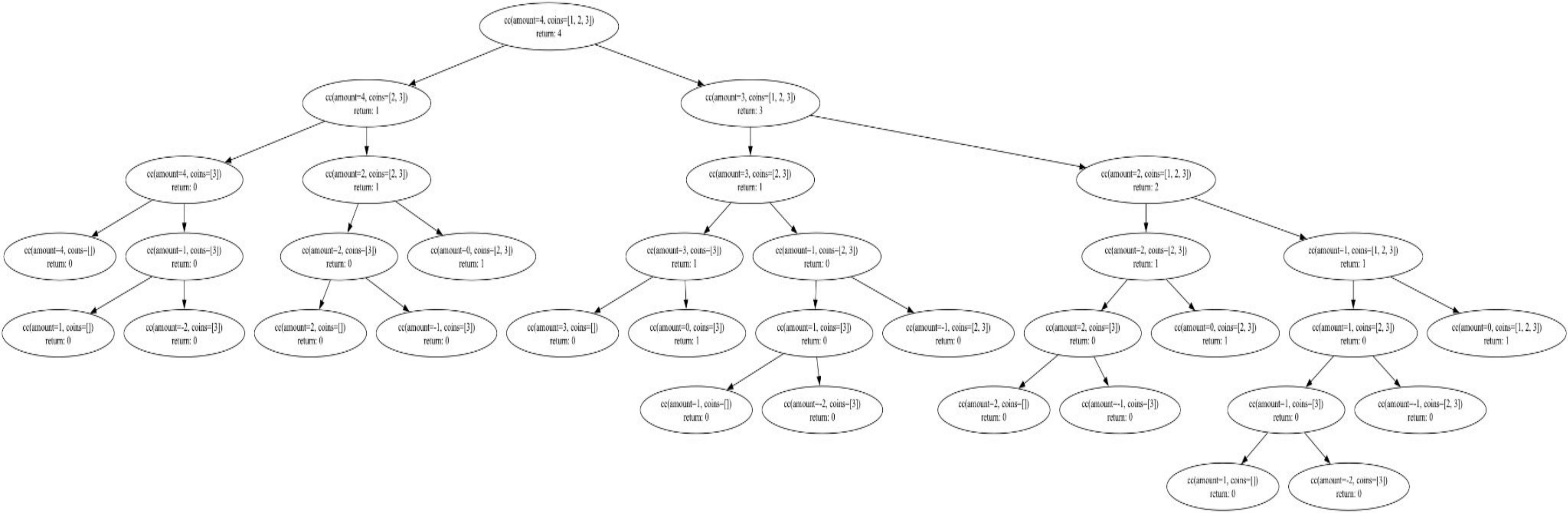
알고리즘 :

1. 주어진 금액(4)에 대해서 첫번째 동전(1)를 **제외**하고 거슬러줄 수 있는 경우의 수
   - 주어진 금액: 4
   - 사용 가능한 동전: [2,3] -> coins[1:]
2. 주어진 금액(4)에 대해서 첫번째 동전(1)를 **사용**하고 거슬러 줄 수 있는 경우 수
   - 주어진 금액: (4-1) = 3
   - 사용 가능한 동전: [1,2,3]
3. 위 1과 2의 합

Counting change problem (I)

```
def count_change(amount, coins):
    if amount == 0:
        return 1
    if amount < 0 or len(coins) == 0:
        return 0
    return
        count_change(amount, coins[1:]) + count_change(amount - coins[0], coins)
```

Counting change problem (II)



Hanoi Tower

문제

세 개의 기둥(A, B, C)**과 **n개의 크기가 다른 원판**이 있을 때, 모든 원판을 **A → C로 옮기되**, 다음 조건을 만족해야 합니다:

- 한 번에 하나의 원판만 옮길 수 있다.
- 큰 원판이 작은 원판 위에 올라갈 수 없다.
- 보조 기둥(B)을 이용할 수 있다.

해결책

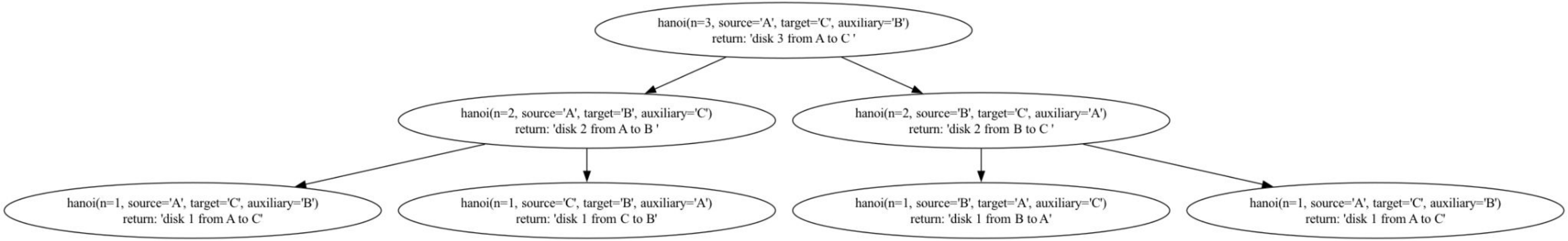
- 원판 1개일 경우: 바로 A → C
- 원판 2개 이상일 경우:

1. 원판더미 중 위에서 n-1개의 원판을 A → B (보조 기둥 B 이용)
2. 가장 아래에 있는 큰 원판 1개를 A → C
3. 위에서 n-1개의 원판을 B → C (보조 기둥 A 이용)

Hanoi Tower (I)

```
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
    else:
        hanoi(n - 1, source, auxiliary, target)
        print(f"Move disk {n} from {source} to {target}")
        hanoi(n - 1, auxiliary, target, source)
```

Hanoi Tower (II)



function_visualizer

- 재귀함수를 이해하기 쉽게 함수호출 과정을 도식화하는 라이브러리
- https://github.com/krafton-jungle/function_visualizer