※ 점화식 작성법

① n==1 테스트: 바닥조건(base case)

T(1) = 1 or C

② 재귀 호출: T(n) = 점화식.

→ 전개

# 2: Recursion (재귀) 맛보기

1. **예1: 1부터 n까지의 합을 계산해보자**

   a. 루프를 활용한 방법:

```
def sum(n):
    s = 0
    for i in range(1, n+1):
        s += i
    return s
```

   b. 재귀적으로 작성하는 법:

      1. 1 + 2 + 3 + … + n = (1 + 2 + … + n-1) + n
      2. 1부터 n까지의 합은 1부터 n-1까지의 합에 n을 더한 값이다 → 1부터 n-1까지의 합을 안다면 n만 더하면 된다 → 1부터 n-1까지의 합은 다시 같은 방식으로 (재귀적으로) 계산하면 된다
      3. sum(n) = sum(n-1) + n
         a. sum(n) = (sum(n-2) + n-1) + n
         b. sum(n) = ((sum(n-3) + n-2) + n-1) + n
         c. …
         d. sum(n) = (…(sum(1) + 2) + 3) + … )…) + n
      4. sum(1) = 1이므로 더 이상 전개할 필요 없다: 바닥 경우 (base case)

```
def sum(n):
    if n == 1: return 1
    return sum(n-1) + n
```

ex) Sum(4) = Sum(3)+4 = 10

재귀호출

Sum(2)+3

Sum(1)+2

return → 1

   c. **수행시간**

      i. 루프 함수: n번 반복되고, 매번 상수번의 기본연산만 수행 O(n)
      ii. 재귀 함수: sum(n-1)을 계산하는 시간 + (n을 더하는 기본연산 1회)
         1. sum(n-1)을 계산하는 시간 = sum(n-2) 계산 시간 + 1
         2. …
         3. 결국, 수행 시간도 재귀적으로 정의해야 함!
         4. T(n) : sum(n)의 수행시간을 나타내는 함수로 정의하면
            **T(n) = T(n-1) + 1  단, T(1) = 1**

   d. **점화식** **T(n) = T(n-1) + 1**

      i. 점화식을 전개하여 n에 관한 식으로 표현해야 함
      ii. 바닥 경우에 이를 때까지만 전개하면 됨 (T(1)의 값은 알고 있으므로)
      iii. T(n) = T(n-1) + 1 = (T(n-2) + 1) + 1
            = …
            = T(1) + (1 + … + 1) = n = **O(n)**

수행시간: $T(n)$

↳ Sum(n)의 수행시간

$T(n) = T(n-1) + C$

$= (T(n-2)+c) + C$

$= T(n-2) + 2c$

$= \boxed{T(n-(n-1)) + (n-1)C}$ $= Cn$

$T(1) = C$

$T(n) = cn = O(n)$

## 2. 예 2: 1부터 n까지의 합을 계산해보자. 좀 다르게~

a. 문제를 조금 수정해 a부터 b까지의 합을 구해보자

b. `sum(a, b)` 함수는 a부터 b까지 더한 값을 리턴 (1부터 n까지 합은 `sum(1,n)` 호출)

c. a와 b의 중간의 값을 m 이라 하자 (`m = (a+b)//2`)

d. `sum(a, b) = sum(a, m) + sum(m+1,b)`의 재귀 형식으로 정의가능

   i. **한 번이 아닌 두 번의 재귀 호출**이 이루어짐: `sum(a,m)`과 `sum(m+1,b)`

   ii. 여기서 base 경우는? `sum(a,b)`에서 `a == b`인 경우

      1. `a == b`인 경우엔 a를 리턴하면 됨

e. sum(2, 7)의 함수 호출 순서를 따라가보자

```
sum(2,7) = (sum(2,4) + sum(5,7))
         = ((sum(2,3)+sum(4,4)) + (sum(5,6)+sum(7,7)))
         = (((sum(2,2)+sum(3,3)) + 4) + ((sum(5,5)+sum(6,6))+7))
         = (((2+3)+4) + ((5+6)+7))
         = 27
```

f. 코드

바닥조건

```python
def sum(a, b):
    if a == b: return a
    if a > b: return 0
    m = (a+b)//2
    return sum(a, m) + sum(m+1, b)
```

$T(\frac{n}{2})$ $T(\frac{n}{2})$

g. `sum(1, n)`을 호출한 경우의 수행시간 `T(n)`의 점화식

   i. `T(n) = T(n/2) + T(n/2) + c = 2T(n/2) + c,  T(1) = 1`

   ii. 전개

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + c \\
&= 2(2T(\frac{n}{2^2}) + c) + c \\
&= 2^2T(\frac{n}{2^2}) + 2c + c \\
&\vdots \\
&= 2^kT(\frac{n}{2^k}) + c(1 + 2 + \cdots + 2^{k-1}) \\
&= 2^k + c\frac{2^k - 1}{2 - 1} \\
&= O(n)
\end{aligned}
$$

## 3. 예3:

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

예):

$Sum(a,b) = a + (a+1) + \cdots + (b-1) + b$ $(a \le b)$

$Sum(3,8)$

$= 3+4+5+6+7+8$

$\rightarrow Sum(3,7)+8$

이렇게 함수로 읽어!

반으로 나눠계산

$Sum(3,8)$

$= Sum(3,5) + (6,8)$

$\frac{3+8}{2}$

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + c$ (기타 연산)

$= 2T(\frac{n}{2}) + c$ $\quad n = 2^k$ (가정)

$= 2 \times (2T(\frac{n}{2^2}) + c) + c$

$= 2^2 \times T(\frac{n}{2^2}) + 2c + c$

$\vdots$

$= 2^k \times T(\frac{n}{2^k}) + c(1 + 2 + 2^2 + \cdots + 2^{k-1})$

=1 $\quad$ 공비2 등비수열

1이 될때까지 계산

$= c \cdot 2^k + c(\frac{2^k - 1}{2-1})$

$= c \cdot 2^k + c \cdot 2^k - c$

$= 2 \cdot c \cdot 2^k - c$

$\quad 2^k = n$

$= 2cn - c \Rightarrow O(n)$

※ 등비수열 합

$S_n = \frac{a(r^n - 1)}{r - 1}$

$S_n = a + ar + \cdots + ar^{n-1}$

호출과정:

```
factorial(4) =    4 * factorial(3)
                  4 * (3 * factorial(2))
                  4 * (3 * (2 * factorial(1)))
                  4 * (3 * (2 * 1))
                  4 * (3 * 2)
                  4 * 6
                  24
```

a. 리턴되는 값을 가지고 계속 계산을 하며 완성해야 함 → recursion stack 메모리 사용

b. [💪] [해보기-easy] `list A`에 있는 가장 큰 수를 찾는 재귀 함수 `find_max(A, n)`을 작성해보자

```
def find_max(A, n): # A[0] ~ A[n-1] 중 최대값을 찾아 리턴
      # 이를 위한 재귀 코드는 ?
```

4. **예4: 리스트의 값을 거꾸로 배치하기**

a. A의 값을 반대방향으로 재 배치하는 함수 reverse(A) 재귀적으로 작성하기

b. **첫 번째 방법**: 이 방법의 단점은 무엇일까?

```
def reverse(A):
    if len(A) == 1: return A
    return reverse(A[1:]) + A[:1]
```

reverse(A)
↳ reverse(A[2~6]) + 1
[3~6] + 2
[4~6] + 3
[5+6] + 4
6 + 5

c. **두 번째 방법**: `A[start … stop-1]`까지의 값을 거꾸로 배치
# 이 방법과 첫 번째 방법을 비교해보자!

```
def reverse(A, start, stop):
    if start < stop-1: # 2개 이상의 값이 있는 경우만 의미 있으므로
        A[start], A[stop-1] = A[stop-1], A[start]
        reverse(A, start+1, stop-1)
```

0 1 2 3 4 5

d. A = [1, 2, 3, 4, 5, 6]에 위의 두 가지 방법을 적용해보자

reverse(A, 0, 6)
A=[6, 2, 3, 4, 5, 1]
reverse(A, 1, 4)
A=[6, 5, 3, 4, 2, 1]
↳ reverse(A, 2, 3)
A=[6, 5, 4, 3, 2, 1]
reverse(A, 3, 2)

e. 수행시간을 위해 두 방법의 수행시간 점화식을 세워보자

i. 첫 번째 방법: T(n) = $T(n-1)+1 = (T(n-2)+1)+1$
$= \cdots$
$= T(1) + (1+1+\cdots+1) + 1 = n = O(n)$

ii. 두 번째 방법: T(n) =

↳ $T(n) = T(n-2) + C$
$T(n-4) + C$
$T(n-6) + C$
$\vdots$
$T(1) + C$
$= \frac{n}{2}C \Rightarrow O(n)$

예: Reverse 함수

$A = [1, 2, 3, 4, 5] \Rightarrow [5, 4, 3, 2, 1]$

$\text{reverse}(A) = \text{reverse}(\quad) + A[0]$

$= \underbrace{\text{reverse}(A[1:])}_{\text{처음을 제외한 나머지}} + A[:1]$

$\text{reverse}(A, \text{start}, \text{stop}) = A[\text{start}] \cdots A[\text{stop}-1]$

$= A[\text{stop}-1] \boxed{\phantom{xx}} A[\text{start}]$

$\hookrightarrow \text{reverse}(A[\text{start}+1] \cdots A[\text{stop}-2])$

$= \text{reverse}(A, \text{start}+1, \text{stop}-2)$

$T(n) = T(n-2) + c$

$= T(n-4) + c + c$

$\vdots$

$T(1) = \frac{n}{2} c$

$\Rightarrow O(n)$

## [시간 복잡도 #2] Big-O 표기 제출완료

5점 ★★

주관식

- 아래 코드에 나타난 함수의 수행 시간은 Big-O로 무엇인가? (**최악의 경우**의 수행시간을 의미)
  - 답 예: `O(n^2), O(logn), O(n^0.5), O(n), O(nlogn)`, ...
  - `log`의 밑이 생략되어 있다면, 밑이 `2`라는 의미임

```python
def f(n):
    count = 0
    while n > 0:
        if n % 2:
            count += 1
        n = n // 2
    return count
```

k

$T(n) = 6k+1$

k= n의 이진비트수

$2 \lfloor 9$
$2 \lfloor 4 \quad 1$
$2 \lfloor 2 \quad 0$
$2 \lfloor 1 \quad 0$
$0 \quad 1$

수행시간 = 최악의 입력에 대한 기본연산 횟수

$\log_3 n = O(\log_2 n)$

$\frac{\log_2 n}{\log_2 3}$

$O(1) < O(\log_2 n) < O(\sqrt{n}) < O(n)$

소수인지 판별하는

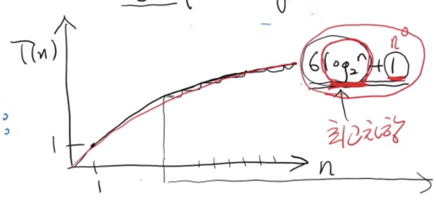
$T(n) = 6k+1$
$= 6 \cdot \log_2 n + 1$

k = n의 이진비트수
$= \lceil \log_2 n \rceil$ ceiling

def f(n):
  1 count = 0
  while n > 0
  반복할지 계속 결정
  for i in range(n):
  if n % 2: 2
  count += 1 2
  k 6
  n = n // 2 2 # n = n
  return count

print(f(3) + f(8)) # 3

11 1000
2 + 1




수행시간 = 최악의 입력에 대한 기본연산 횟수
worst case input