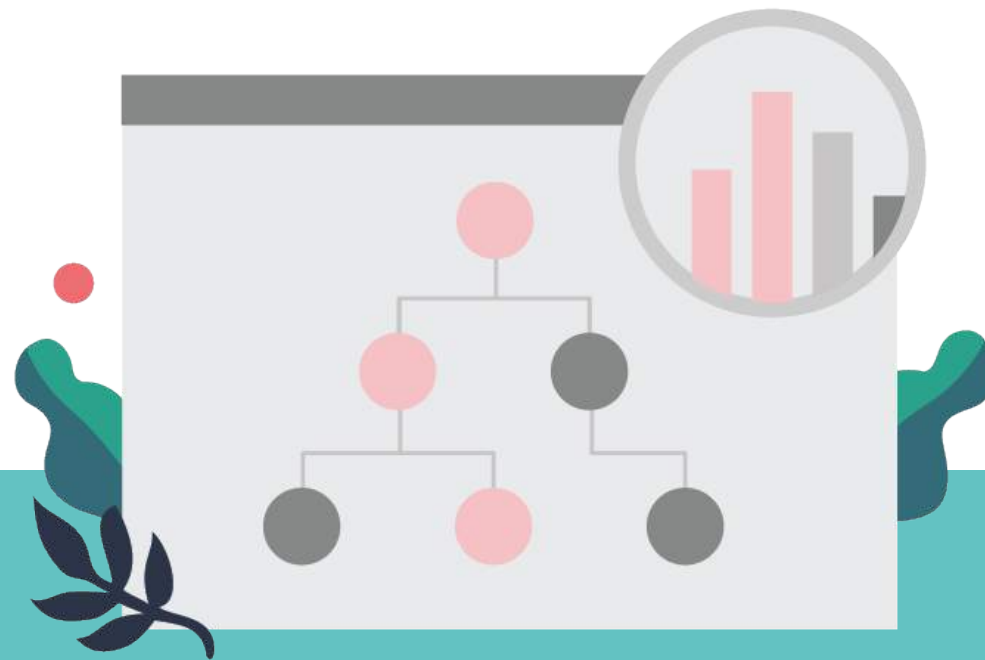


/* 데이터 구조 및 알고리즘 */

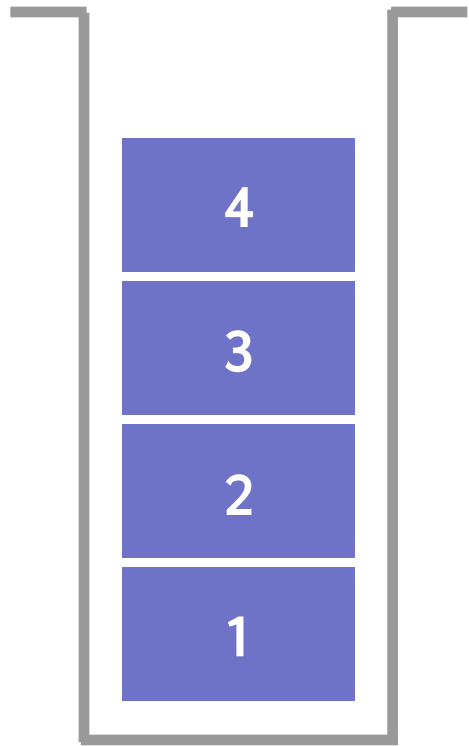
신현규 강사, 화/목 20:00

재귀적 계산 방법



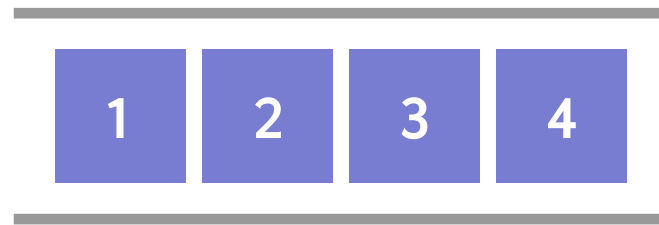
/* elice */

대표적인 자료구조



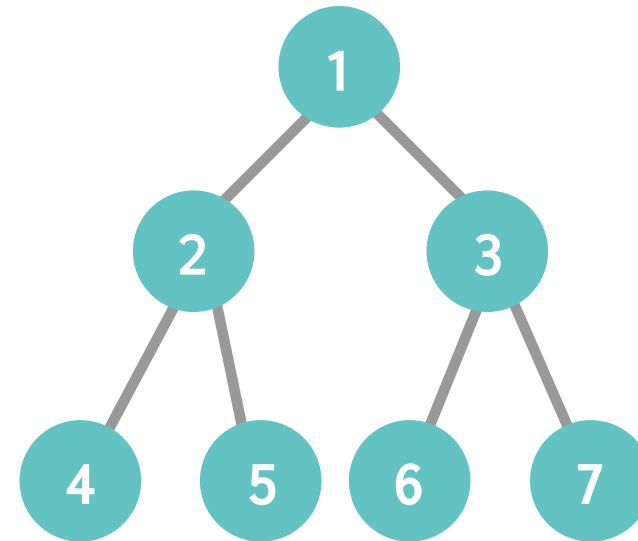
스택 (Stack)

Last In First Out

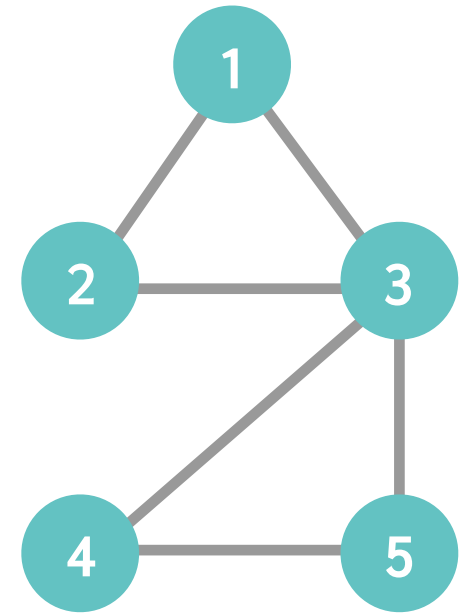


큐 (Queue)

First In First Out



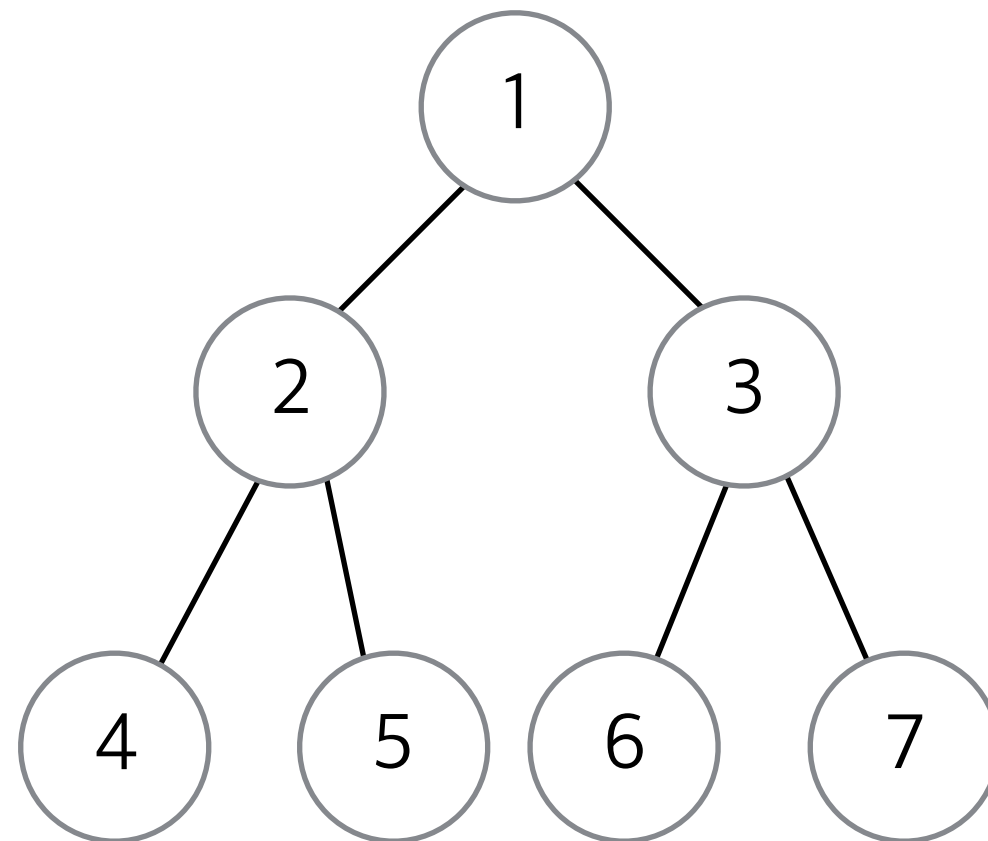
트리 (Tree)



그래프 (Graph)

트리의 재귀적 성질

트리는 그 안에 또 트리가 존재한다



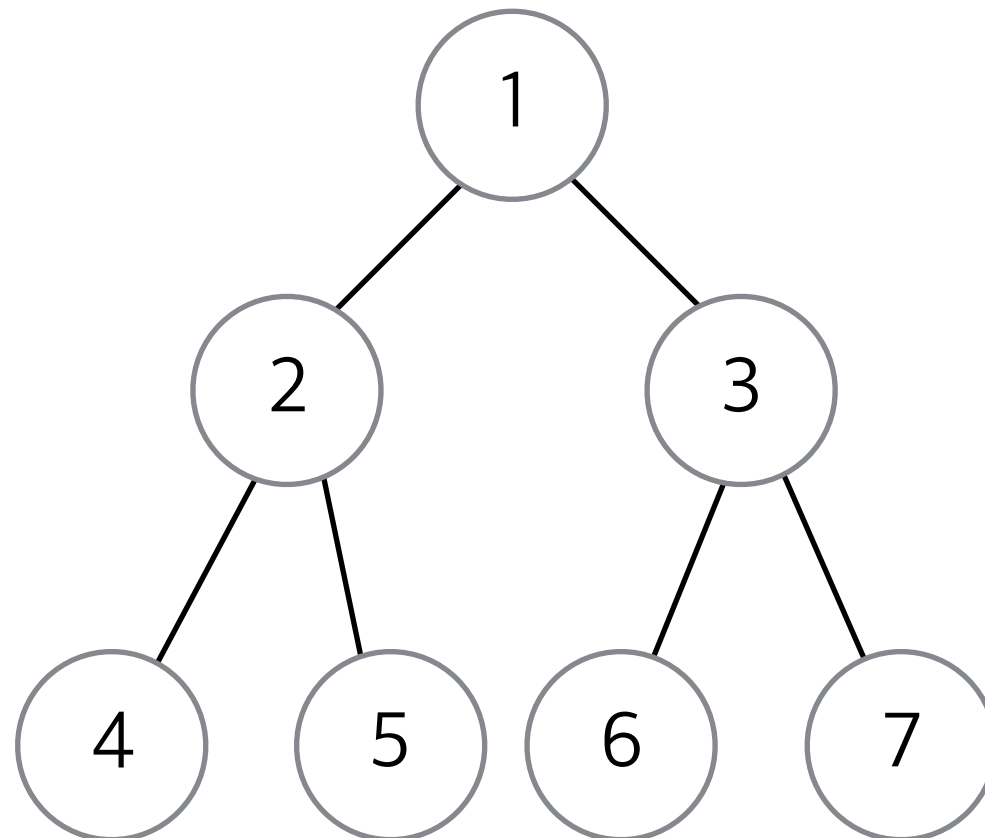
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root

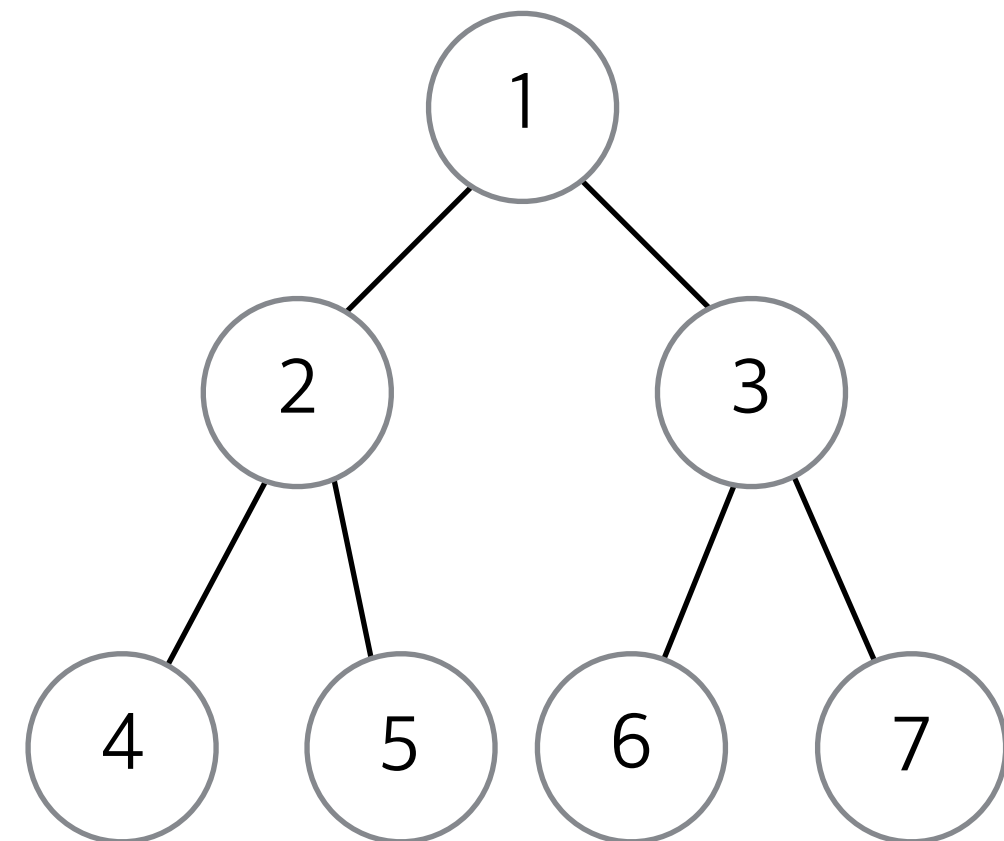


`/* elice */`

도대체 왜 하필 트리여야만 하는가?

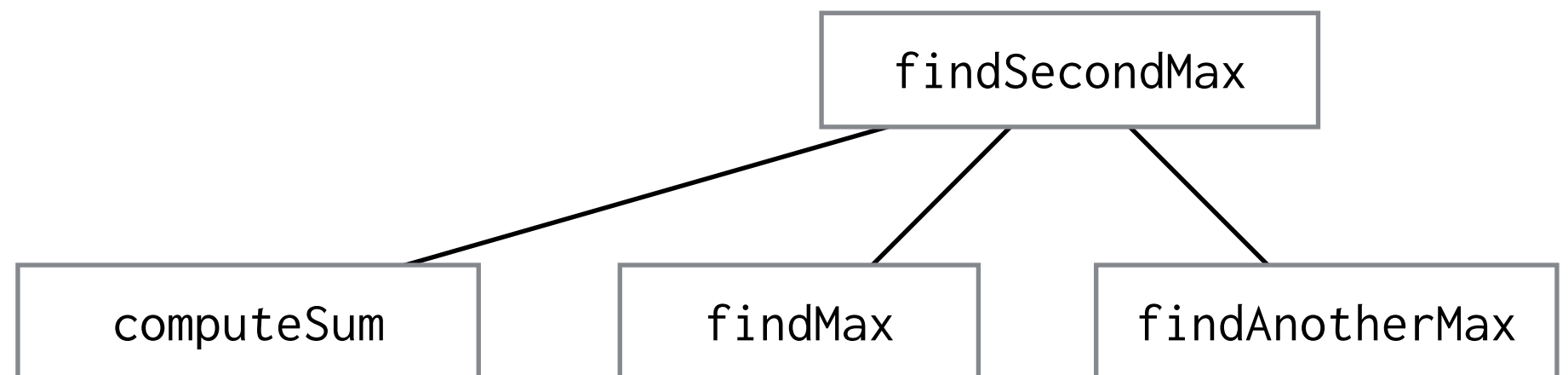
1) 정점에 무슨 자료를 담는가? 코드가 실행되는 상태

2) 간선은 어떤 의미인가? 코드 A가 코드 B를 부른다



의미 단위로 작성된 코드

```
def findSecondMax(myMatrix) :  
    sum = computeSum(myMatrix)  
  
    maxValue = findMax(myMatrix)  
    secondMaxValue = findAnotherMax(myMatrix, maxValue)  
  
    return (sum, maxValue, secondMaxValue)
```



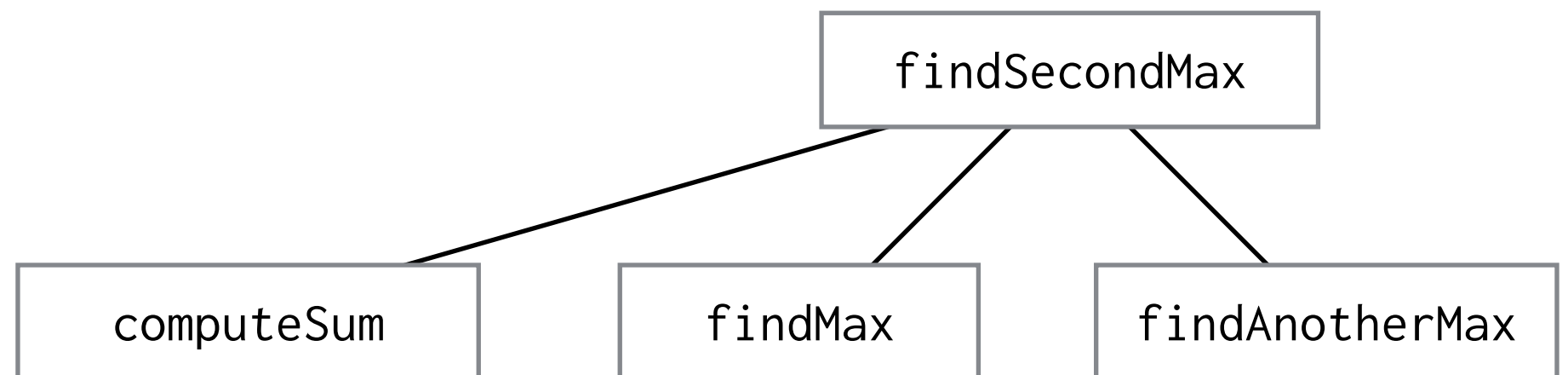
의미 단위로 작성된 코드

`findSecondMax(m)` : 주어진 행렬 `m`에서 (합, 최대값, 두 번째 최대값)을 반환하는 함수

`computeSum(m)` : 주어진 행렬 `m`에서 원소의 합을 반환하는 함수

`findMax(m)` : 주어진 행렬 `m`에서 최대값을 반환하는 함수

`findAnotherMax(m, v)` : 주어진 행렬 `m`에서 `v`를 제외한 최대값을 반환하는 함수



(저번시간) 요약

의미단위로 작성된 코드가 좋은 코드이다

→ 코드를 이해한다 = 각 함수가 무슨 일을 하는지 설명할 수 있다

트리는 코드가 실행되고 있는 상태를 나타내는 자료구조이다

→ 물론, 코드를 의미 단위로 나타냈을 때 파악이 가능한 사실이다

코드를 하나하나 따라가는 것은 컴퓨터가 해야 할 일이다

→ 우리는 앞으로 코드가 하는 일, 더 나아가 코드의 의미에 집중한다

왜 의미단위로만 생각하여 작성한 코드가
제대로 돌아가는가 ?

현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

Factorial(n) : n! 을 반환하는 함수

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

Factorial(n) : n! 을 반환하는 함수

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

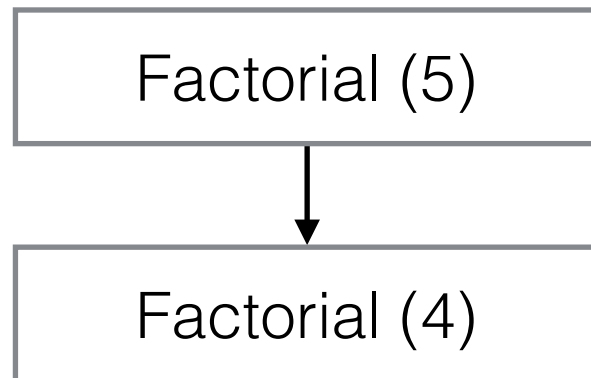
Factorial (5)

Factorial(n) : n! 을 반환하는 함수

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

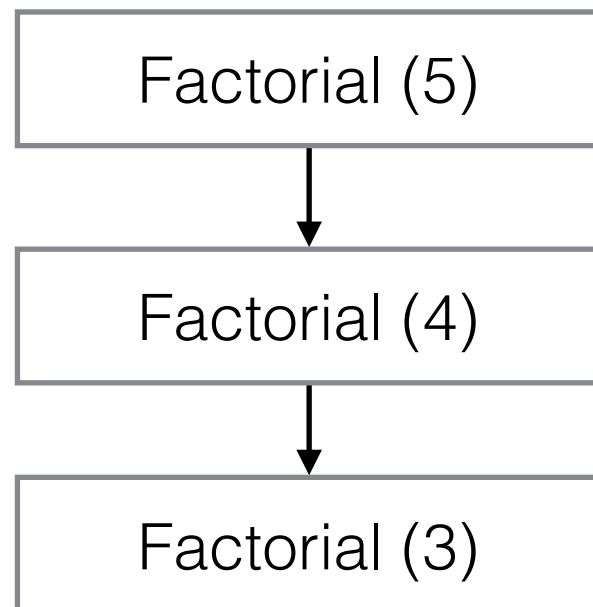


Factorial(n) : n! 을 반환하는 함수

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

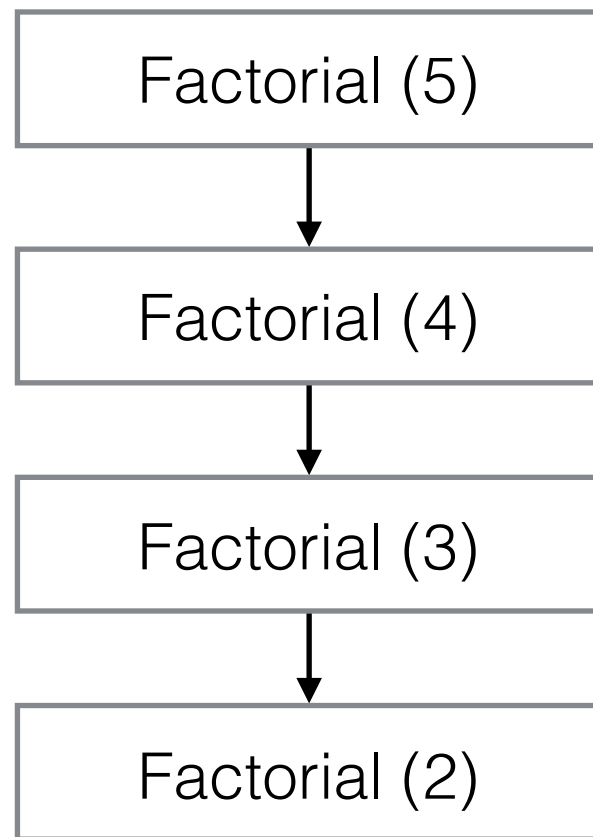


Factorial(n) : n! 을 반환하는 함수

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

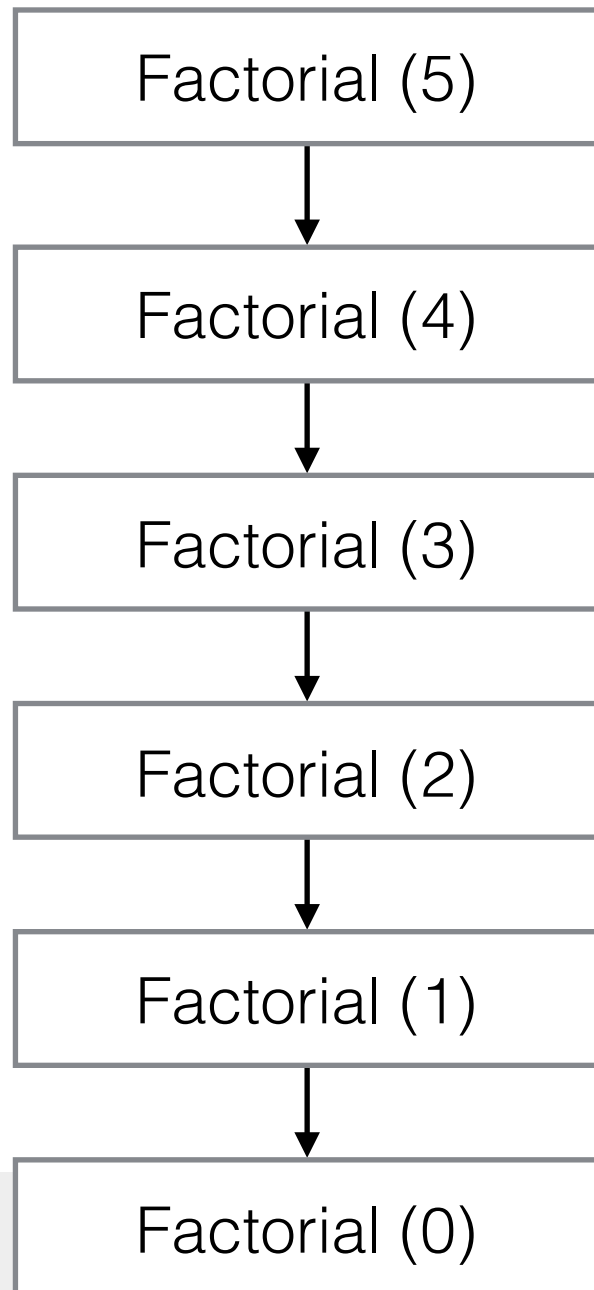


Factorial(n) : n! 을 반환하는 함수

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


팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

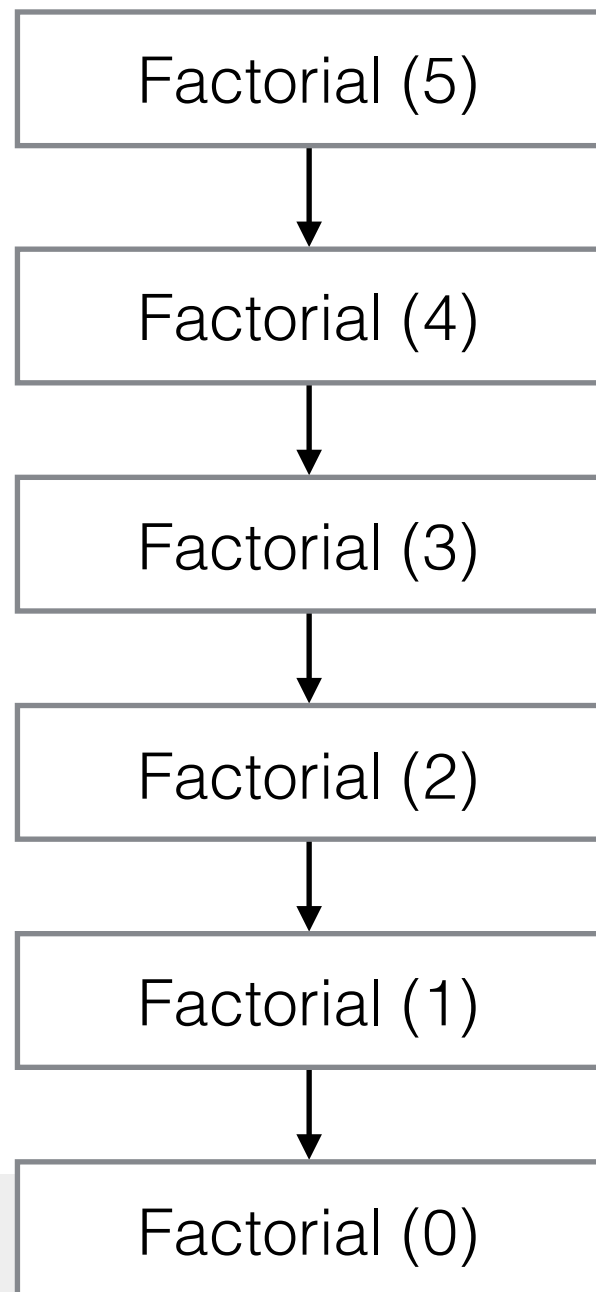


Factorial(n) : n! 을 반환하는 함수

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

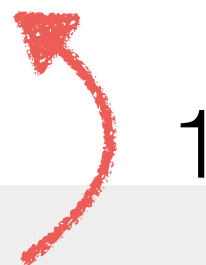
팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$



Factorial(n) : n! 을 반환하는 함수

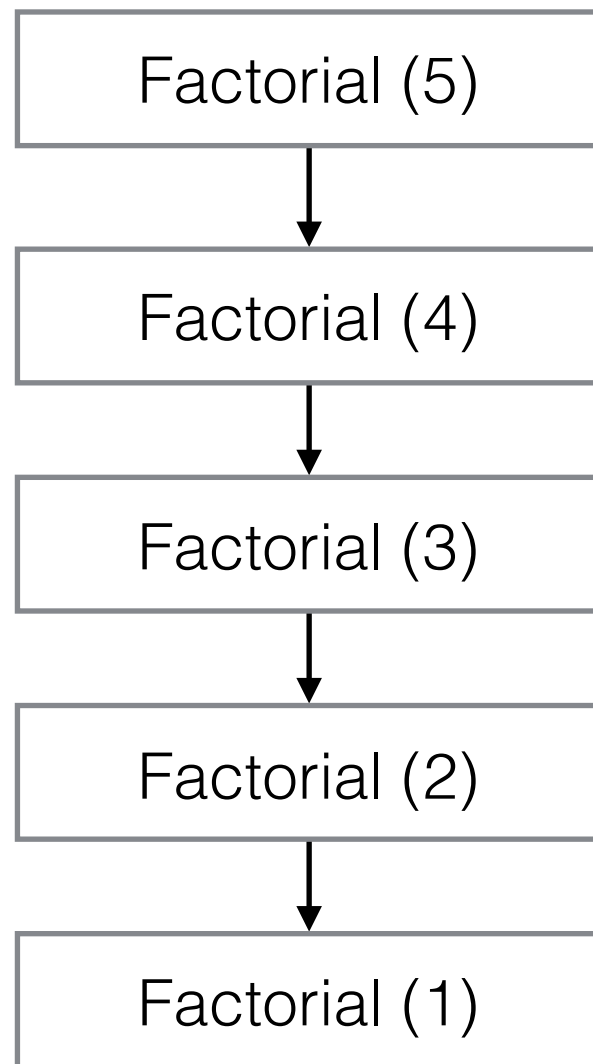
```
def Factorial(n) :  
    if n == 0 :  
        return 1  
    else :  
        return n * Factorial(n-1)
```



`/* elice */`

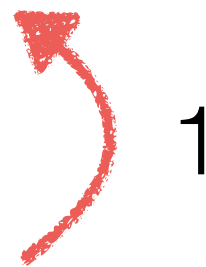
팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$



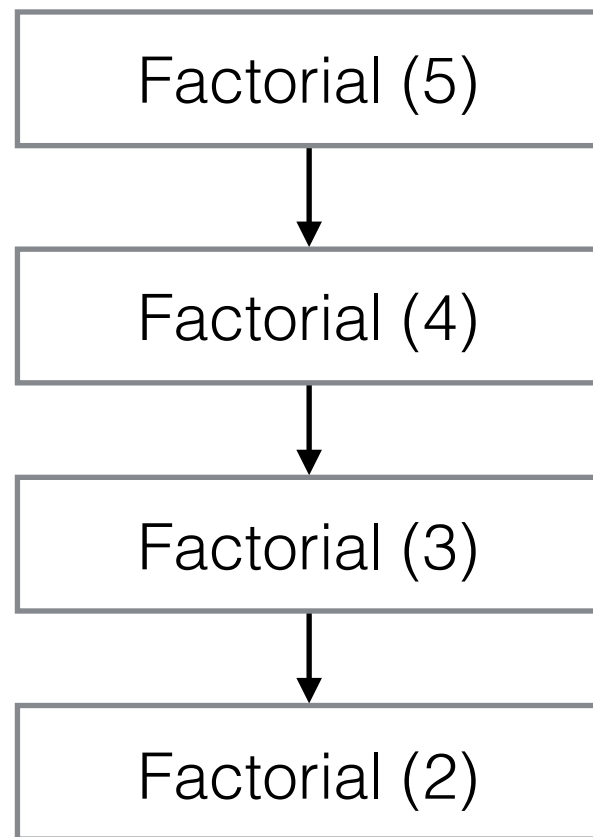
Factorial(n) : n! 을 반환하는 함수

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



팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$



Factorial(n) : n! 을 반환하는 함수

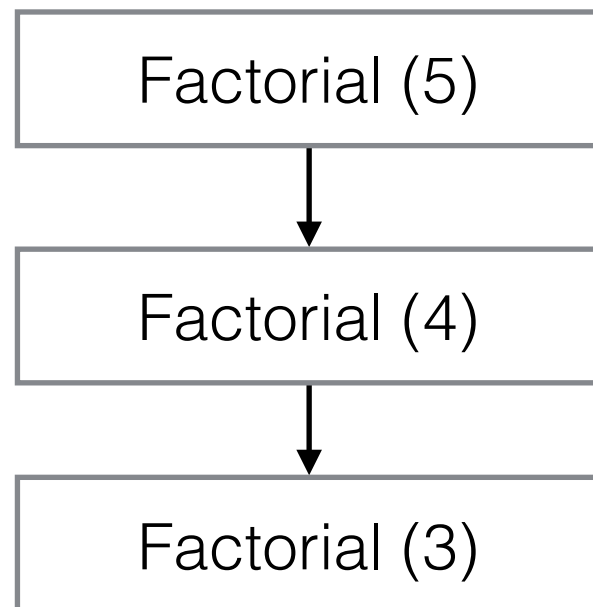
```
def Factorial(n) :  
    if n == 0 :  
        return 1  
    else :  
        return n * Factorial(n-1)
```




2

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$



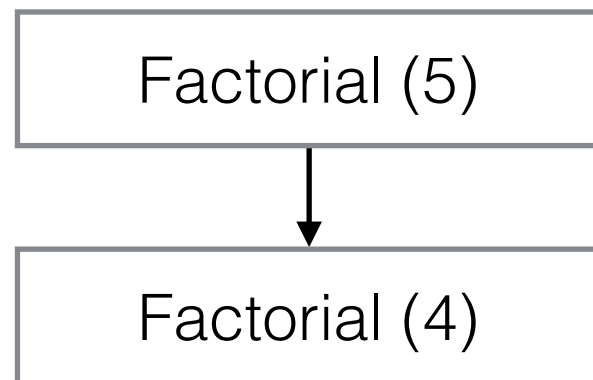
Factorial(n) : n! 을 반환하는 함수



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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$



Factorial(n) : n! 을 반환하는 함수

24

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

120

Factorial (5)

Factorial(n) : n! 을 반환하는 함수

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

팩토리얼의 재귀적 정의 및 구현

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

120

Factorial (5)

Factorial(n) : n! 을 반환하는 함수

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

O(n)

수학적 귀납법 = 재귀적 증명법

수학적 귀납법

명제 $P(n)$ 을 다음과 같이 증명하는 방법

수학적 귀납법

명제 $P(n)$ 을 다음과 같이 증명하는 방법

$n = 1$ 일 때 성립함을 보인다

수학적 귀납법

명제 $P(n)$ 을 다음과 같이 증명하는 방법

$n = 1$ 일 때 성립함을 보인다

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

수학적 귀납법

명제 $P(n)$ 을 다음과 같이 증명하는 방법

$n = 1$ 일 때 성립함을 보인다

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

따라서 모든 자연수 n 에 대하여 $P(n)$ 이 성립한다

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$n = 1$ 일 때 성립함을 보인다

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$n = 1$ 일 때 성립함을 보인다

$$1! \leq 1^1$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

$$k! \times (k+1) \leq k^k \times (k+1)$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

$$k! \times (k+1) \leq k^k \times (k+1)$$

$$(k+1)! \leq k^k (k+1)$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

$$k! \times (k+1) \leq k^k \times (k+1)$$

$$(k+1)! \leq k^k (k+1) \leq (k+1)^k \times (k+1)$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

$$k! \times (k+1) \leq k^k \times (k+1)$$

$$(k+1)! \leq (k+1)^{k+1}$$

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

$P(k)$ 가 성립한다고 가정할 때, $P(k+1)$ 이 성립함을 보인다

$$k! \leq k^k$$

$$k! \times (k+1) \leq k^k \times (k+1)$$

$$(k+1)! \leq (k+1)^{k+1}$$

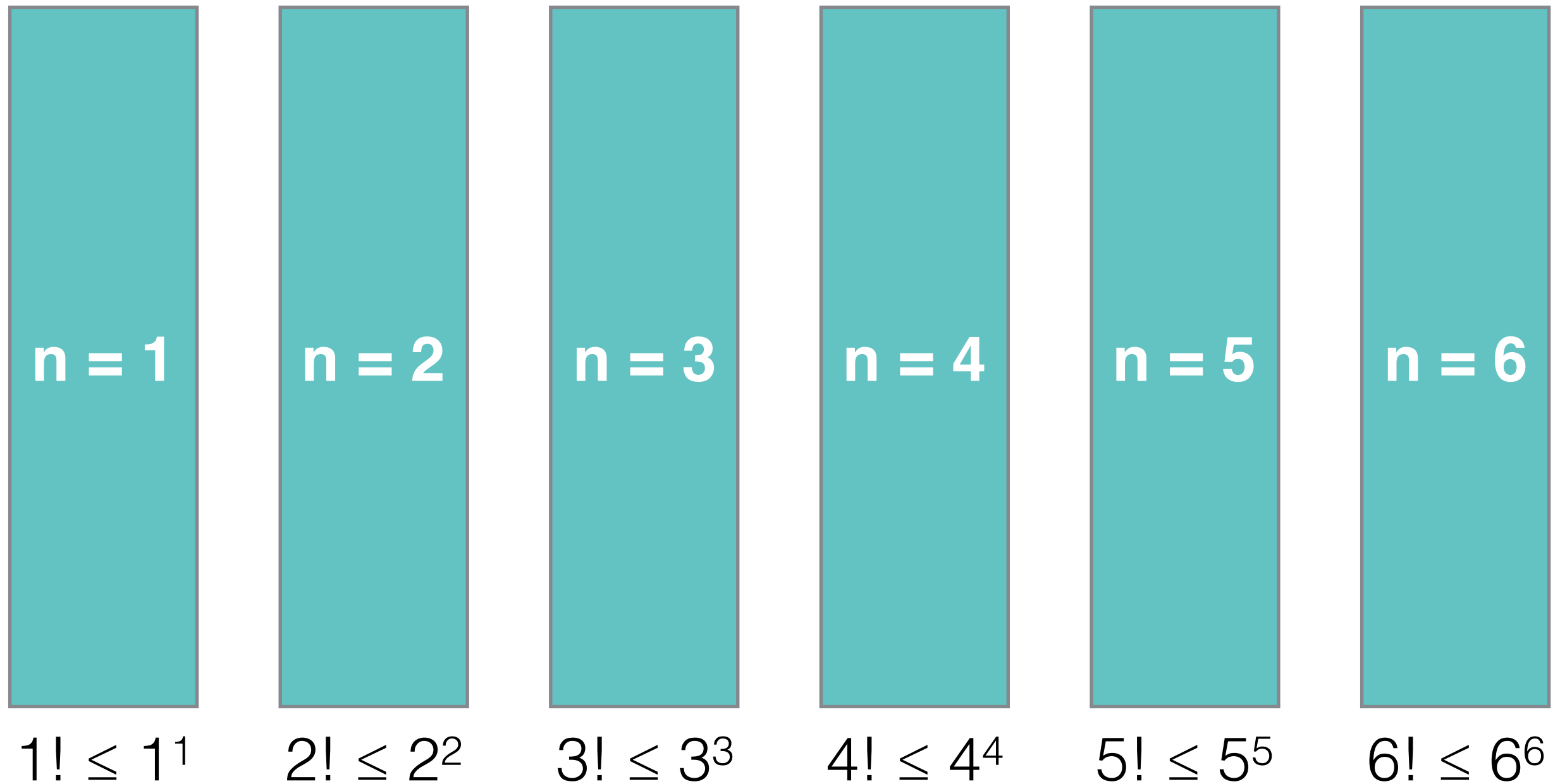

수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오

따라서 모든 자연수 n 에 대하여 위의 명제가 성립한다

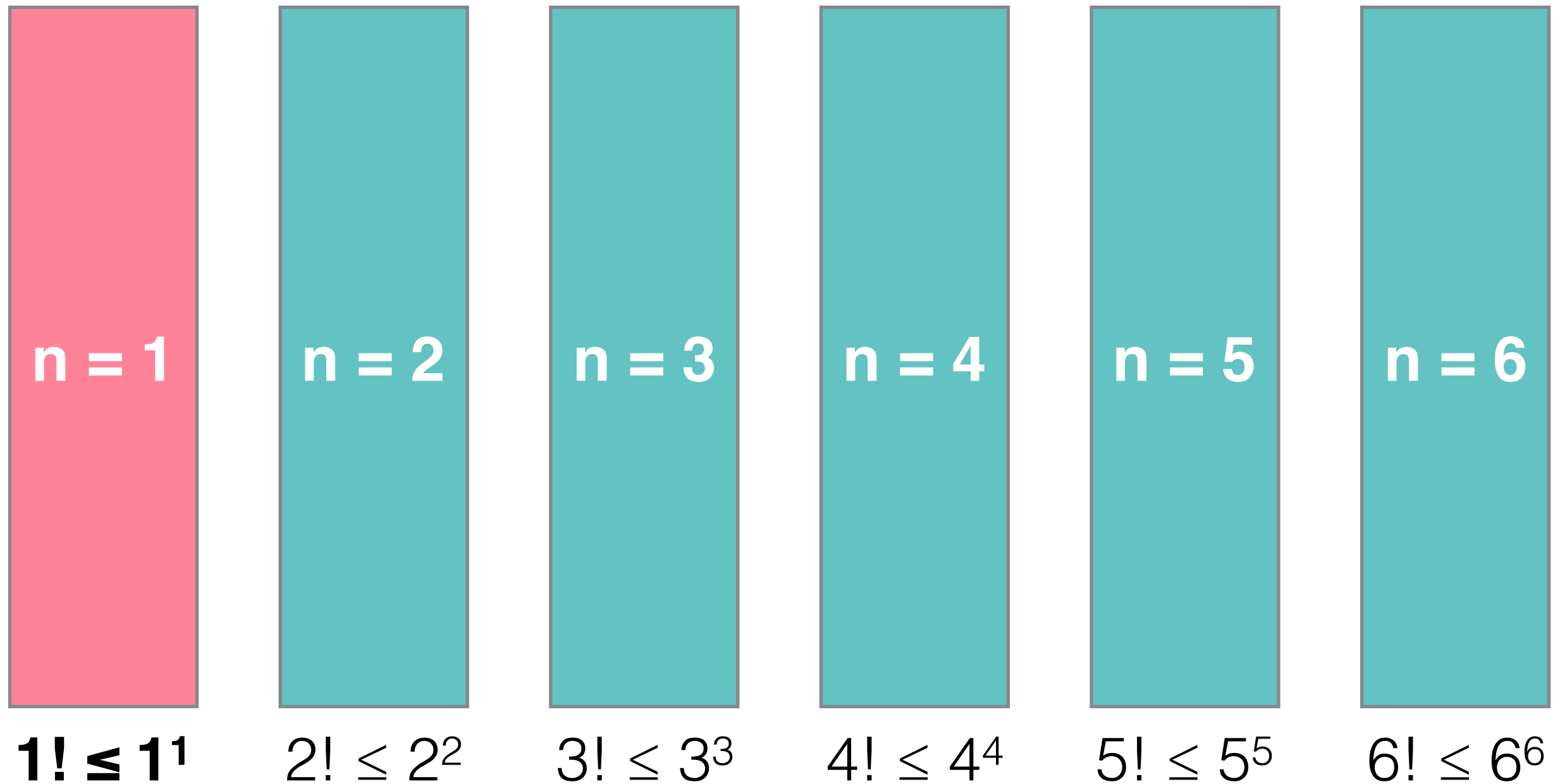
수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오



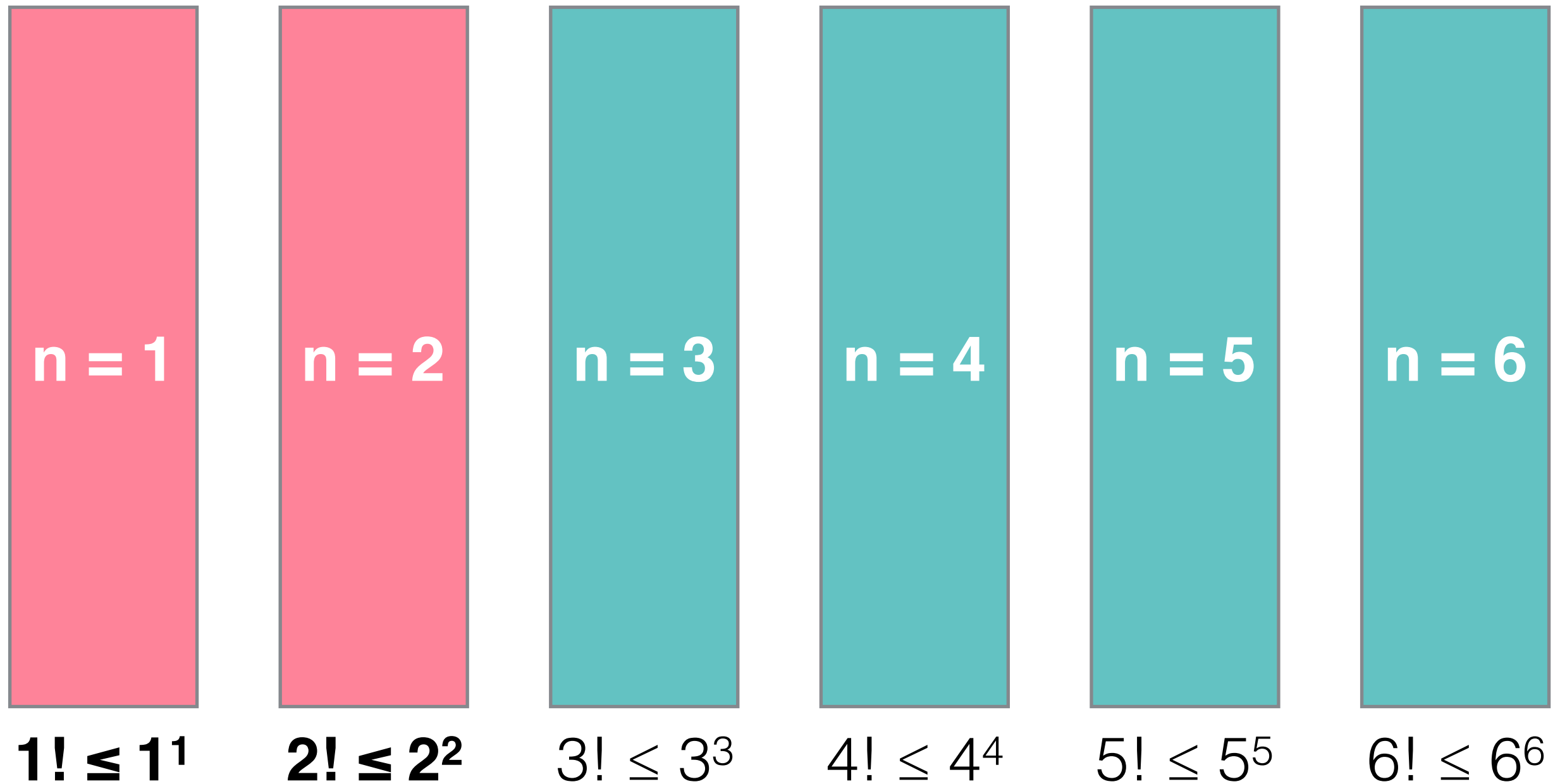
수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오



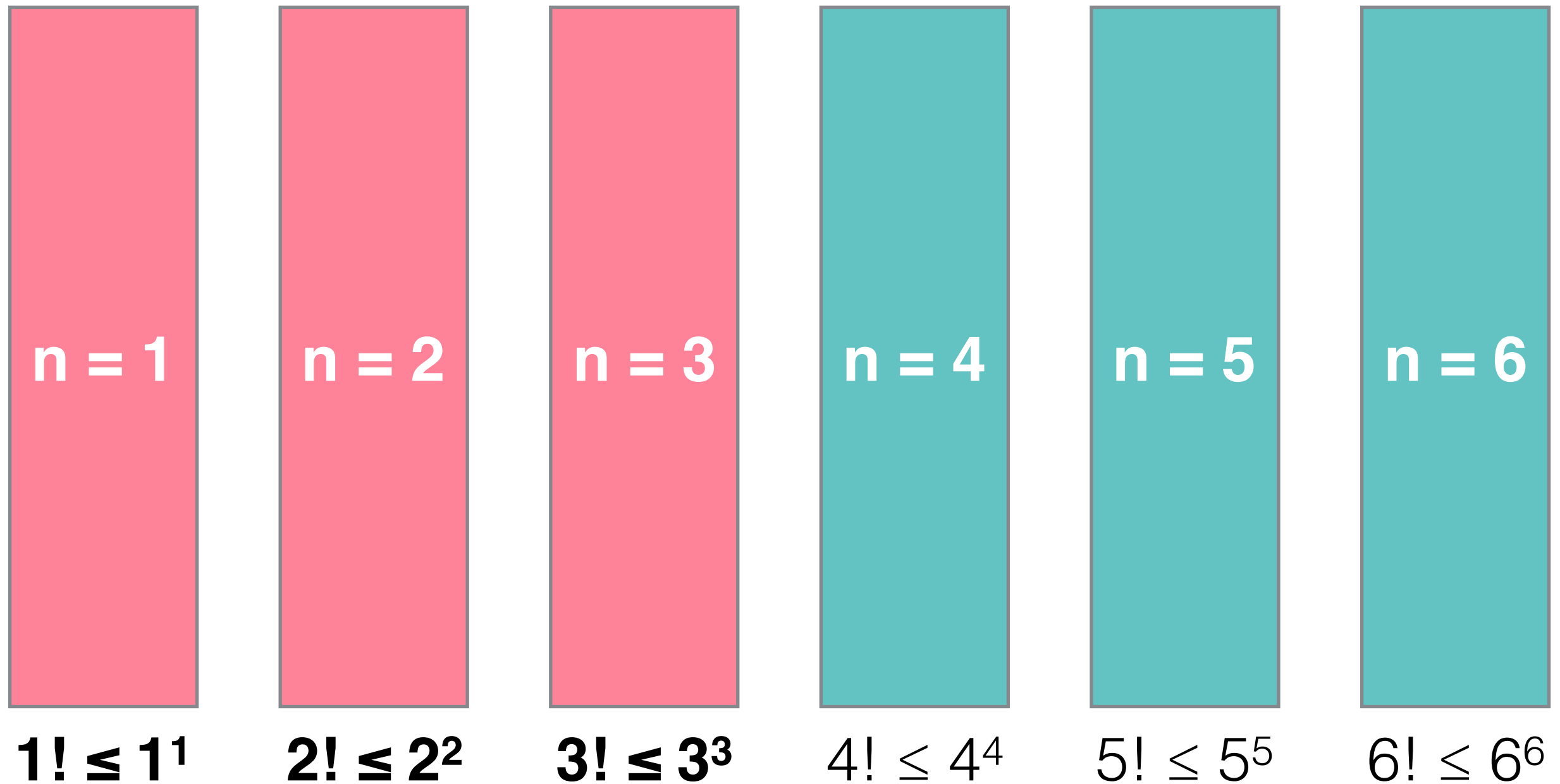
수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오



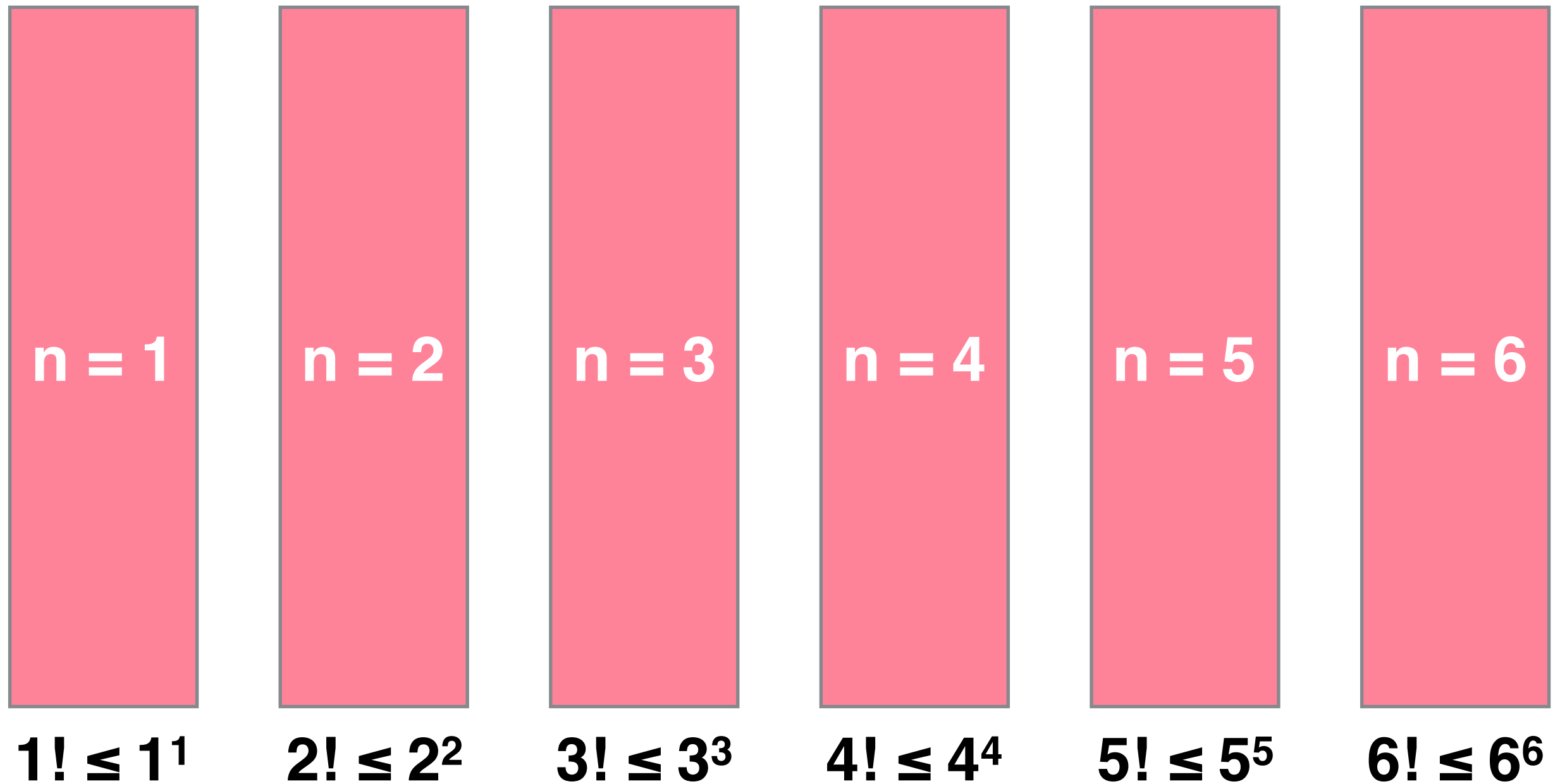
수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오



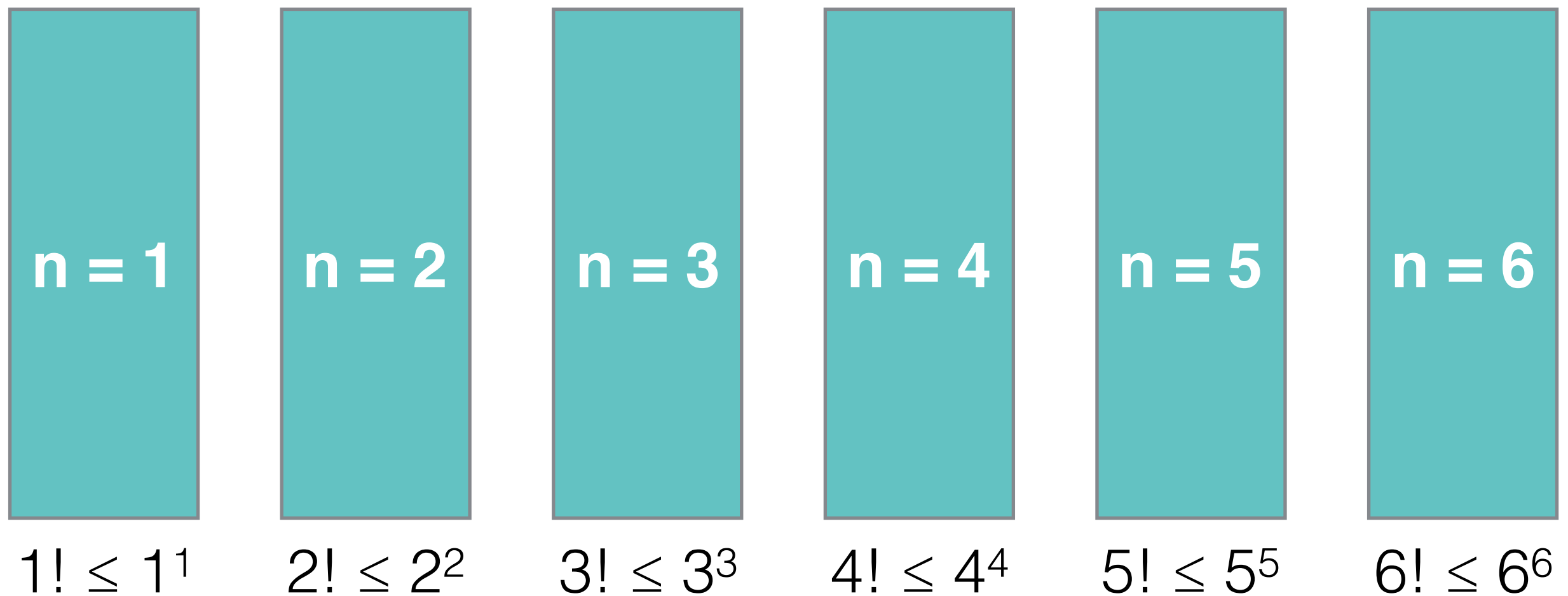
수학적 귀납법 예제

모든 자연수 n 에 대하여 $n! \leq n^n$ 임을 증명하시오



수학적 귀납법을 거꾸로 보기

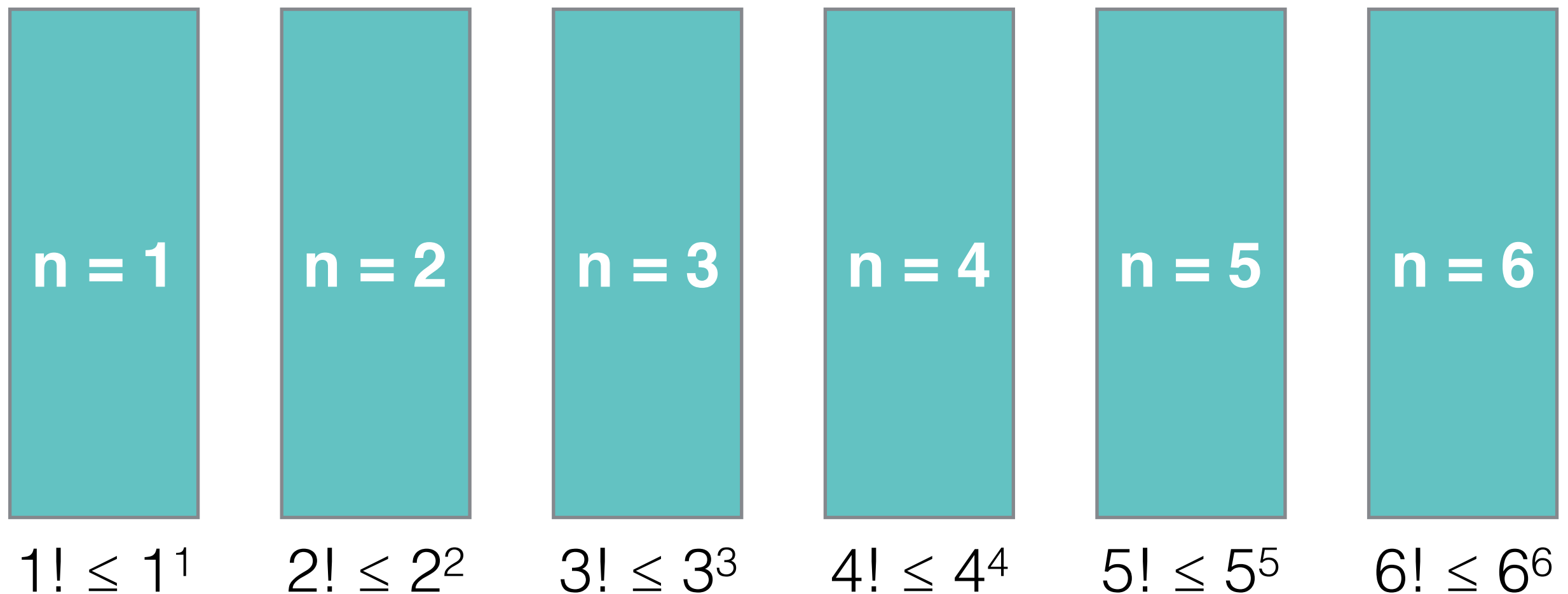
$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

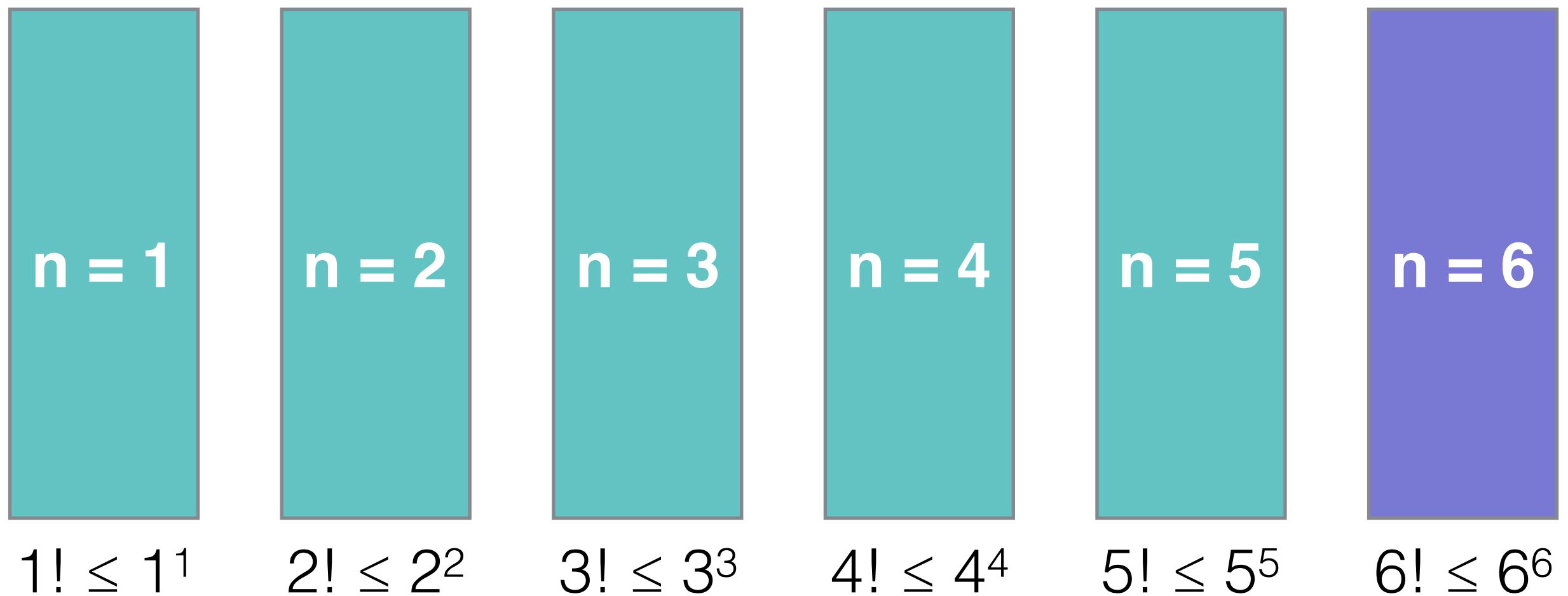
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

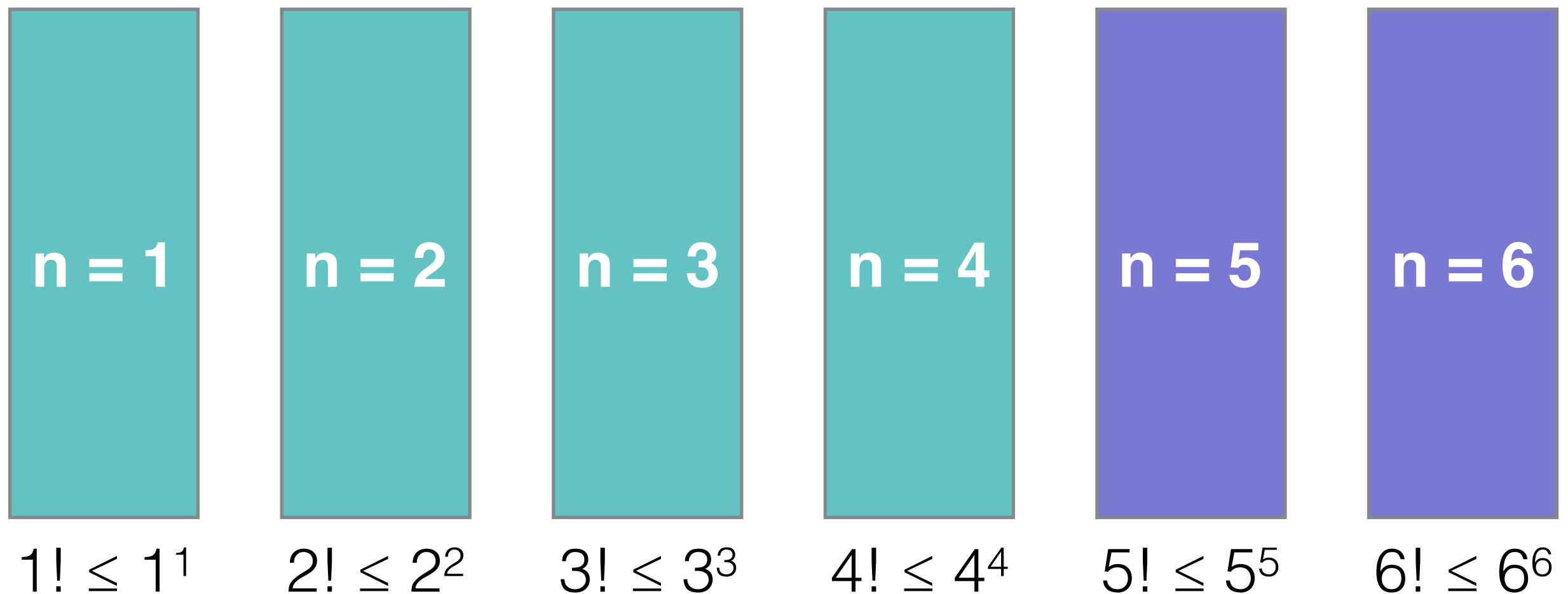
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

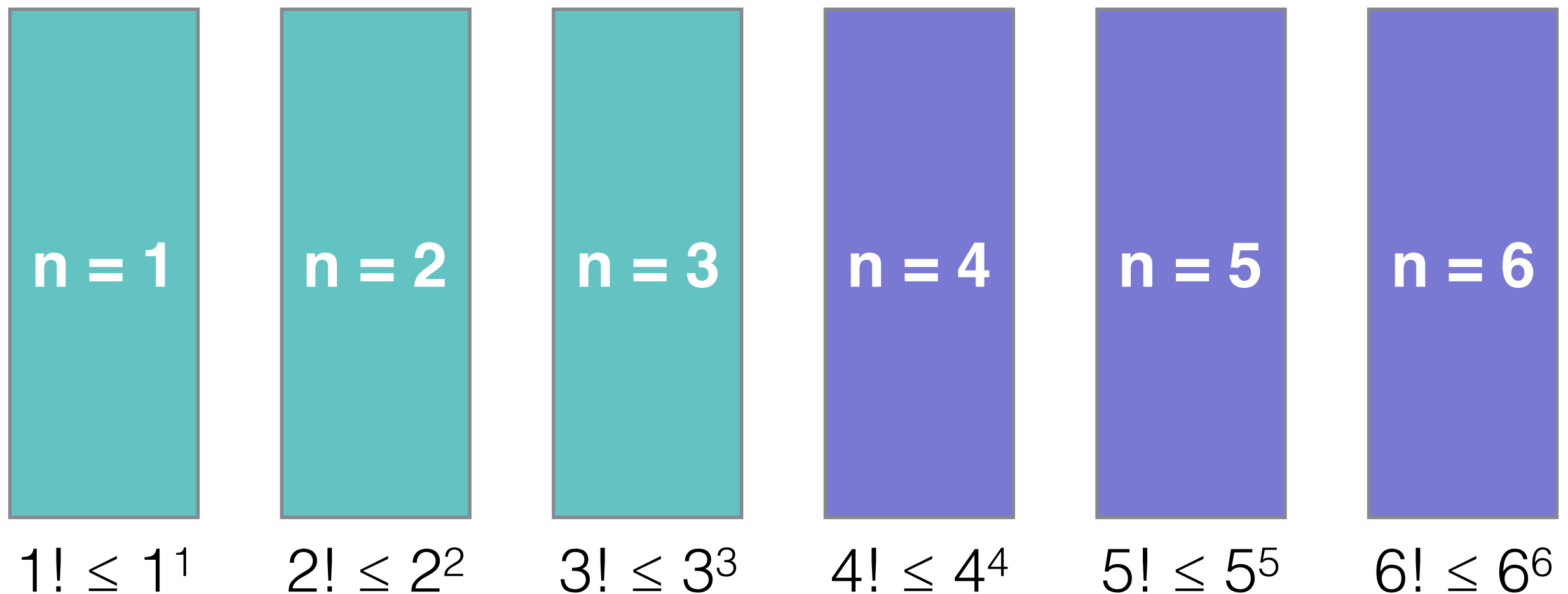
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

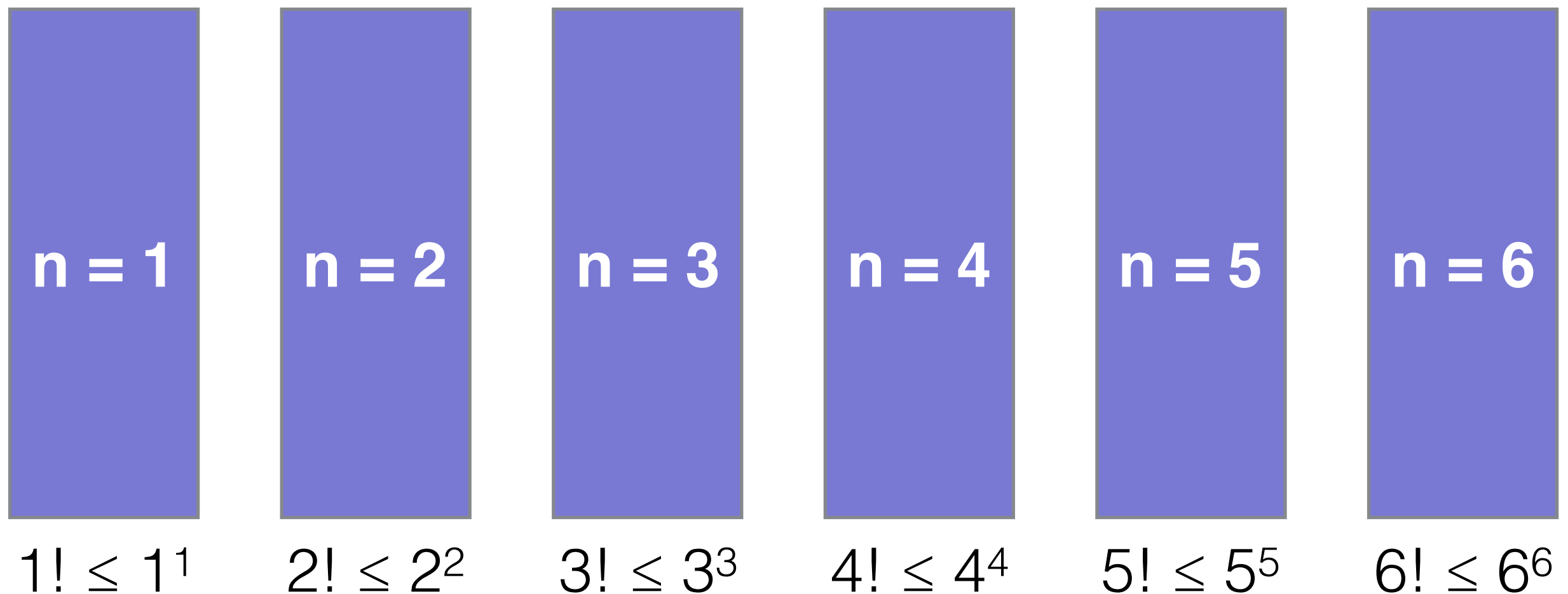
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

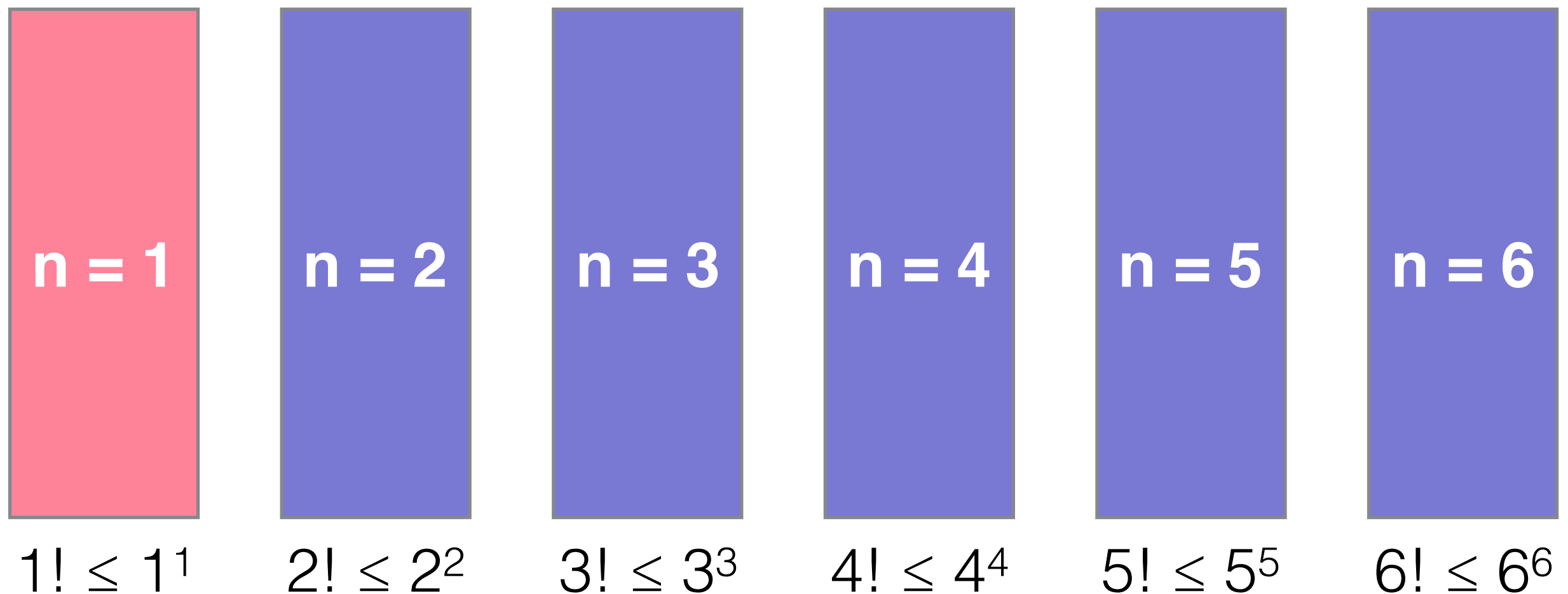
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

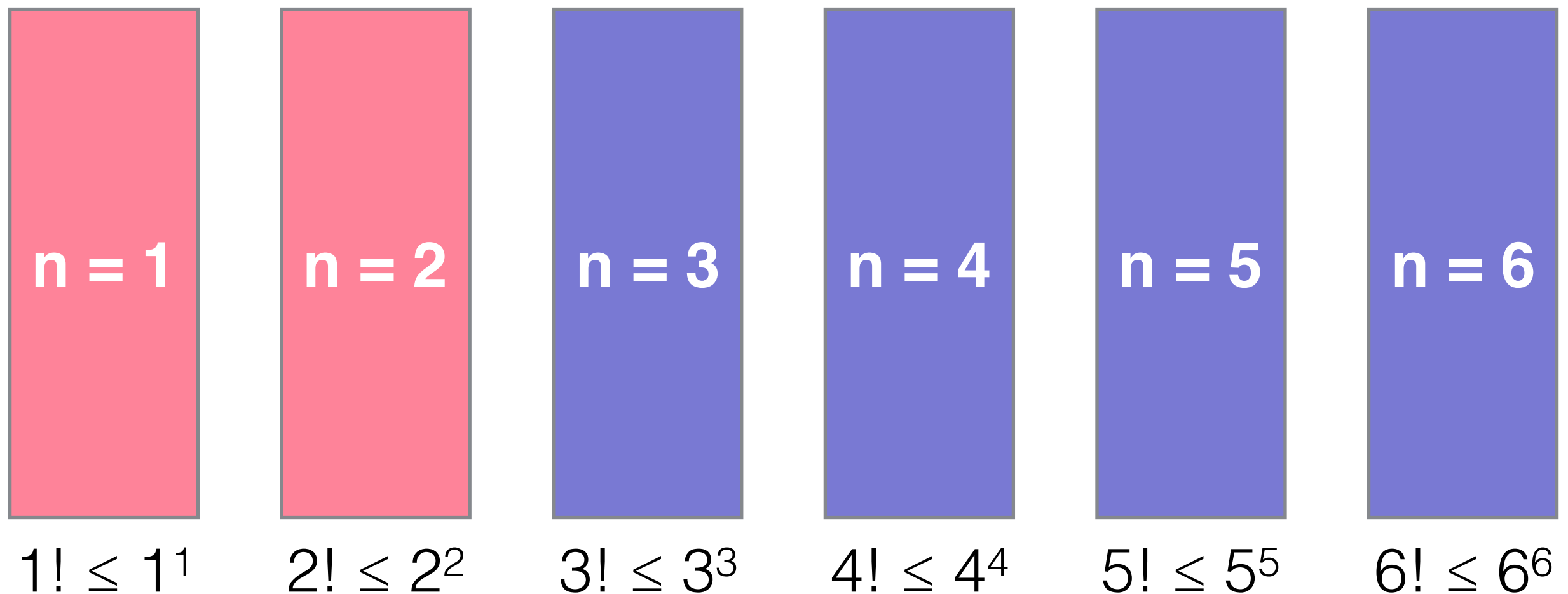
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

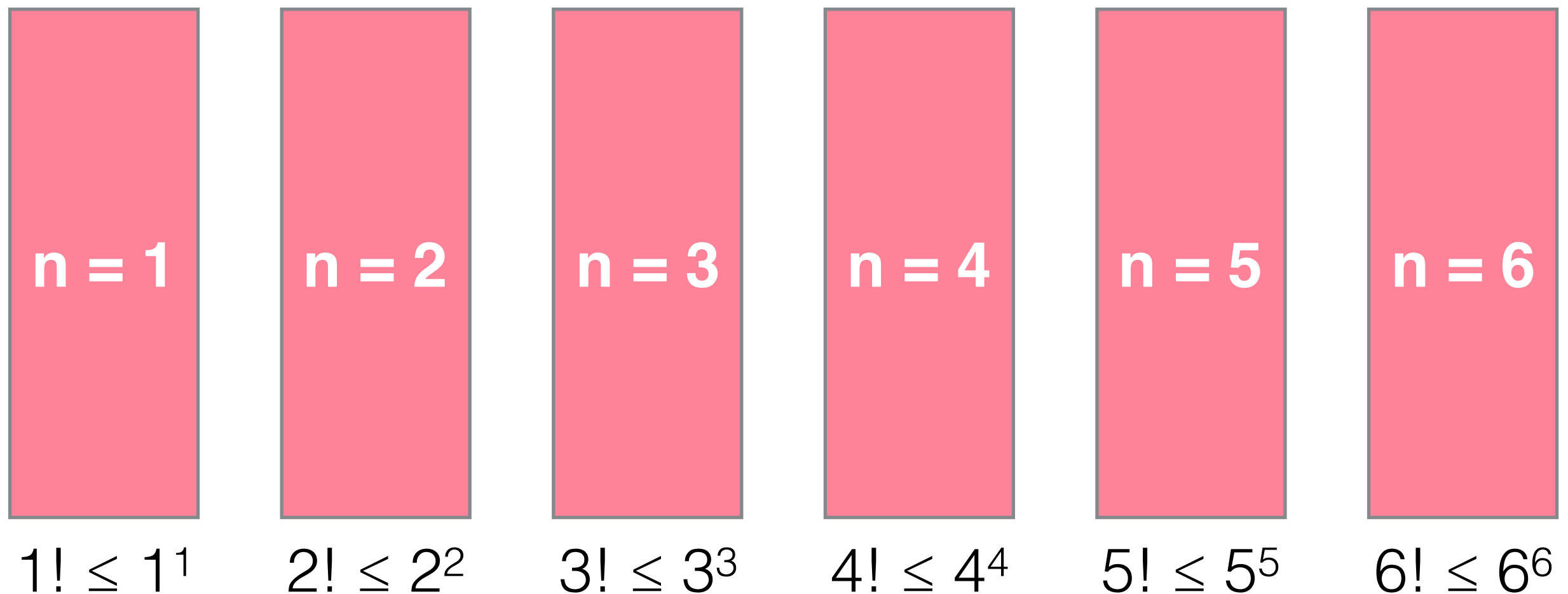
$n = 1$ 일 경우에는 명제가 성립한다



수학적 귀납법을 거꾸로 보기

$n! \leq n^n$ 이기 위해서는 $(n-1)! \leq (n-1)^{n-1}$ 이어야 한다

$n = 1$ 일 경우에는 명제가 성립한다



재귀적 계산 방법

factorial(n) : $n!$ 을 반환하는 함수

$n = 1$

$1!$

$n = 2$

$2!$

$n = 3$

$3!$

$n = 4$

$4!$

$n = 5$

$5!$

$n = 6$

$6!$

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n = 1

1!

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1!

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1!

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1!

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1!

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1! = 1

n = 2

2!

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1! = 1

n = 2

2! = 2

n = 3

3!

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1! = 1

n = 2

2! = 2

n = 3

3! = 6

n = 4

4!

n = 5

5!

n = 6

6!

재귀적 계산 방법

factorial(n) : n! 을 반환하는 함수

factorial(n) = factorial(n-1) * n

n=1일때는 factorial(n)이 제대로 작동한다

n = 1

1! = 1

n = 2

2! = 2

n = 3

3! = 6

n = 4

4! = 24

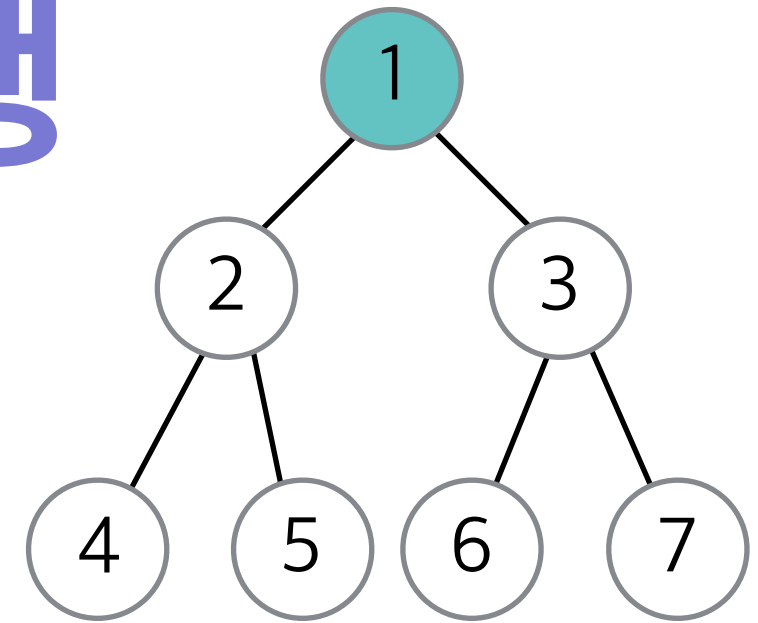
n = 5

5! = 120

n = 6

6! = 720

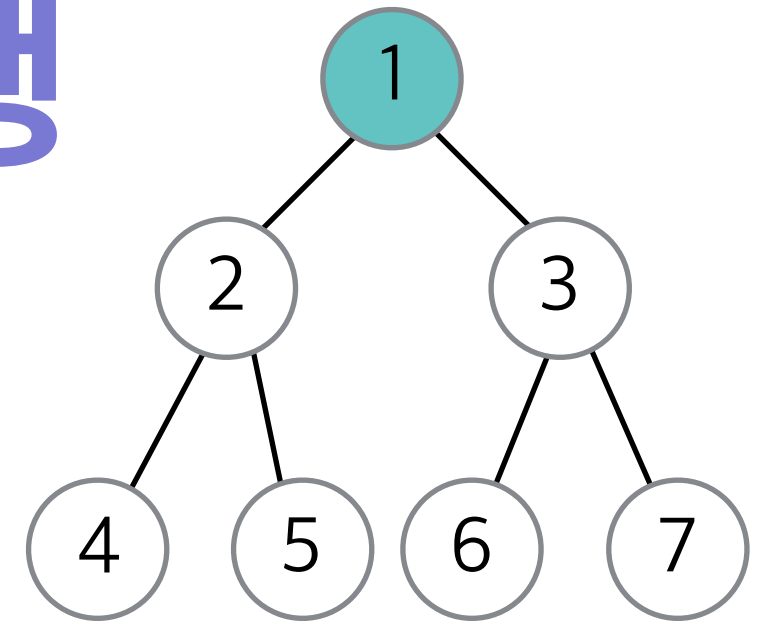
재귀함수의 실행



```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```

main:8, 1

재귀함수의 실행



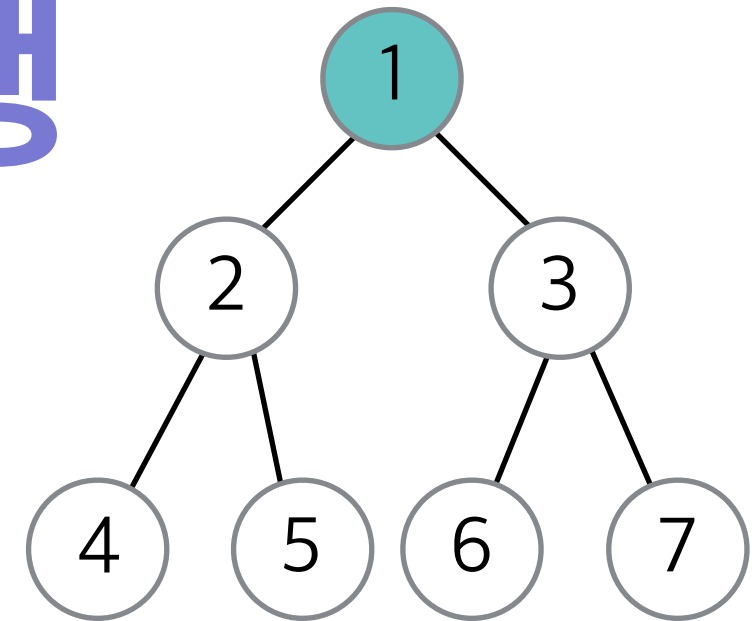
```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]

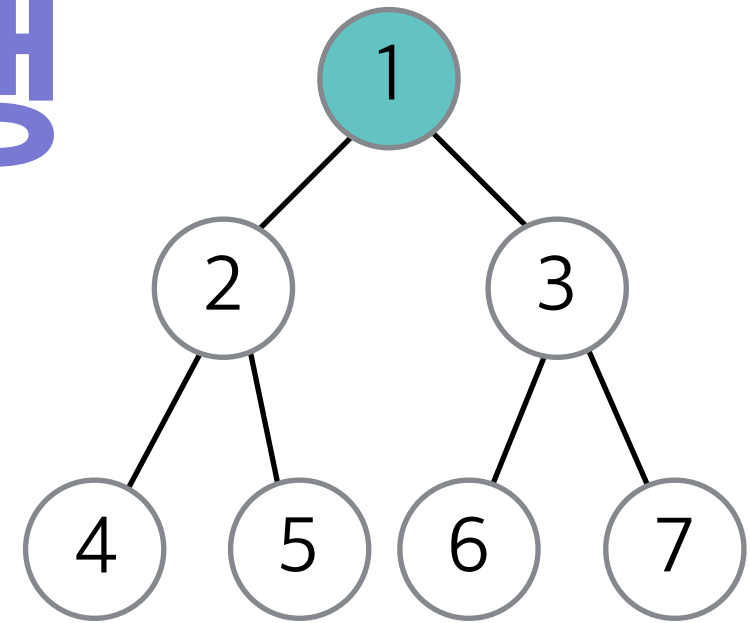


main:8, 1

재귀함수의 실행

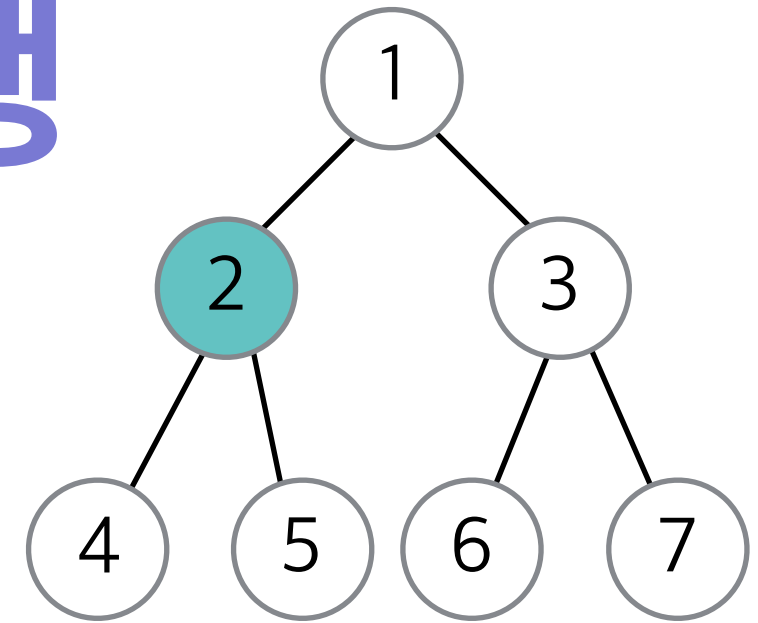
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



main:8, 1

재귀함수의 실행

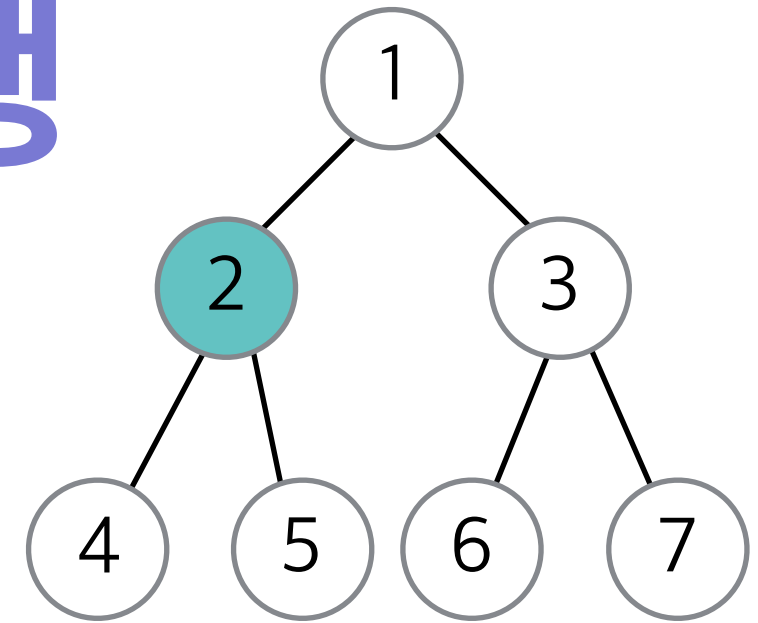


```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```

po:5, (1, [])

main:8, 1

재귀함수의 실행

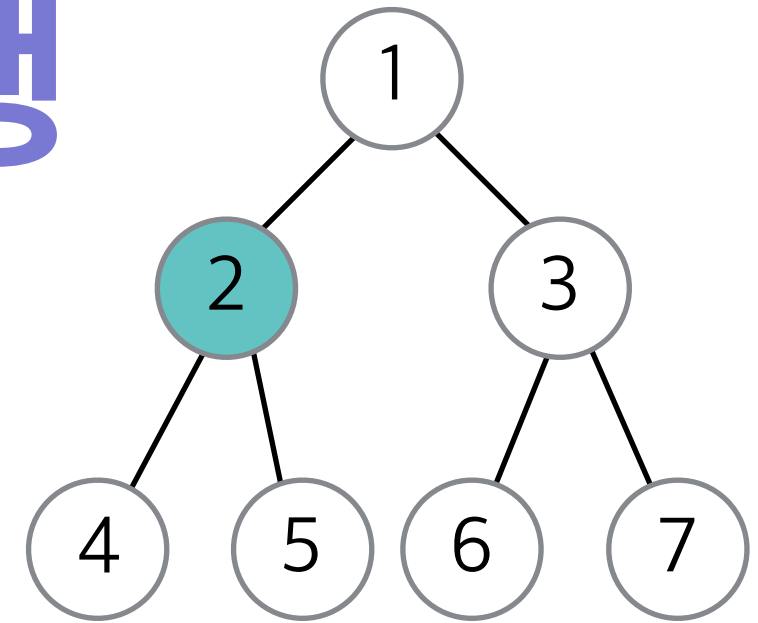


```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

po:5, (1, [])

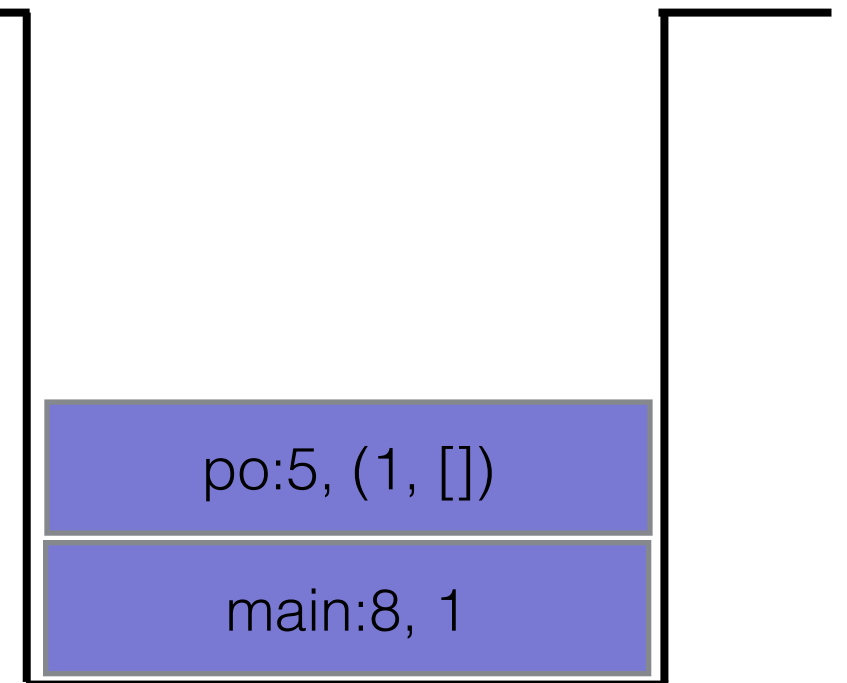
main:8, 1

재귀함수의 실행



```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

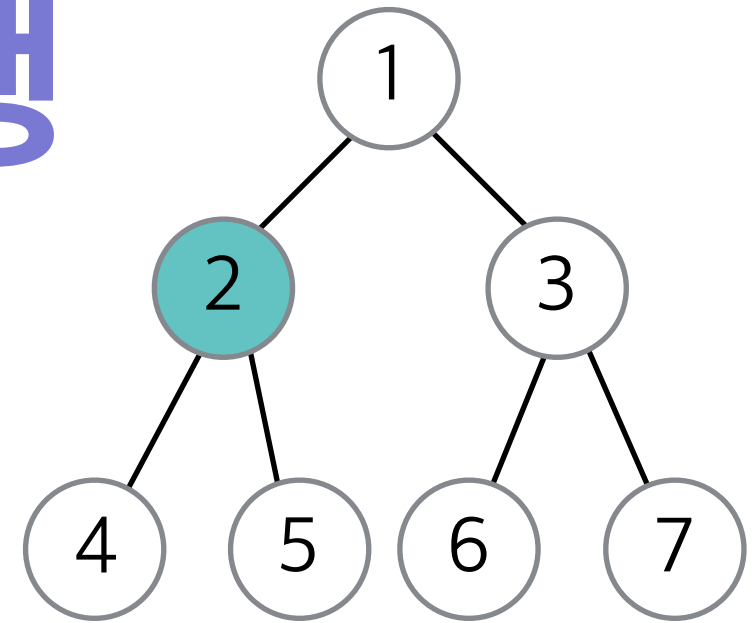
[]



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

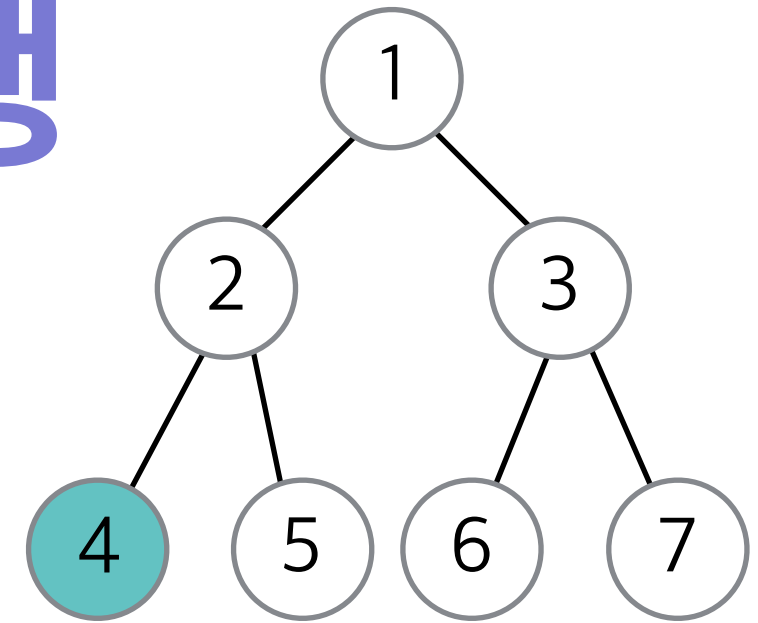
[]



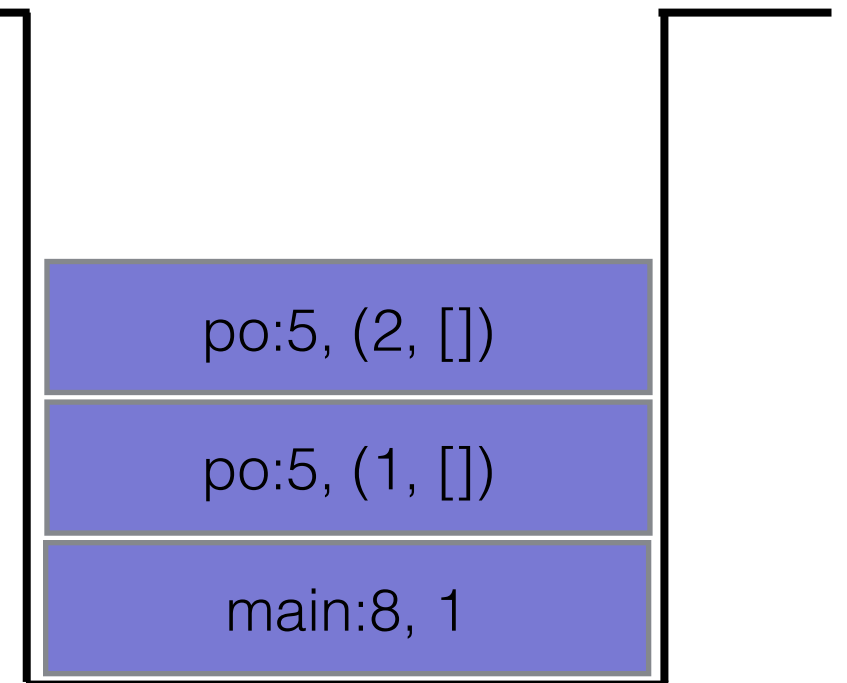
po:5, (1, [])

main:8, 1

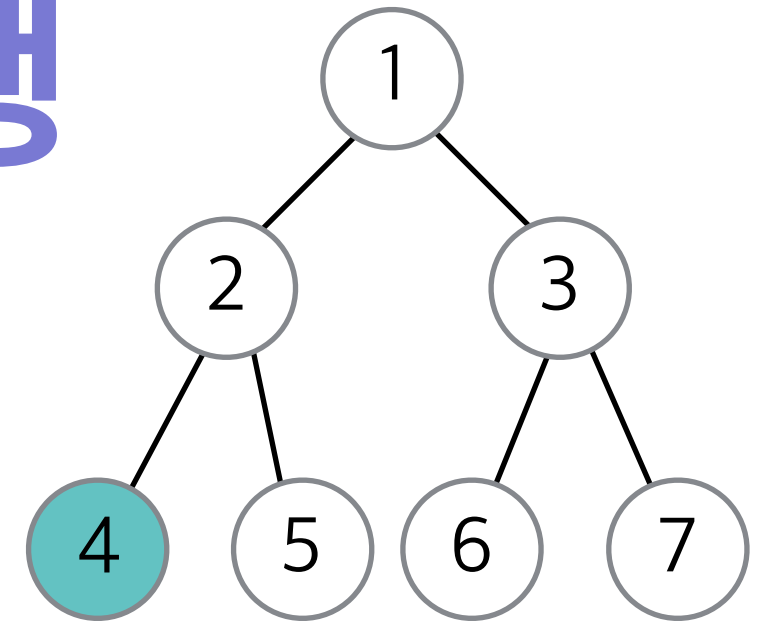
재귀함수의 실행



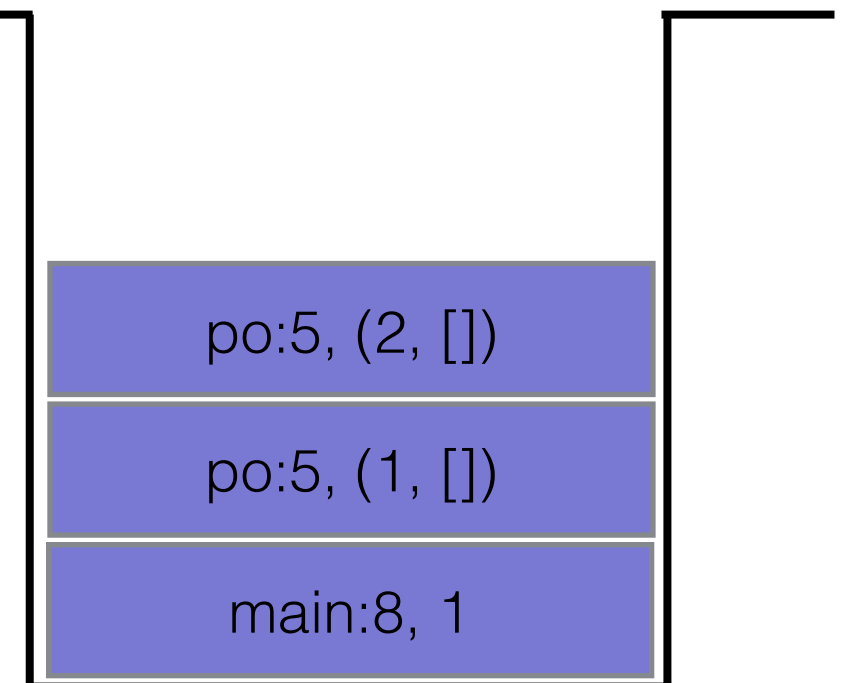
```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```



재귀함수의 실행



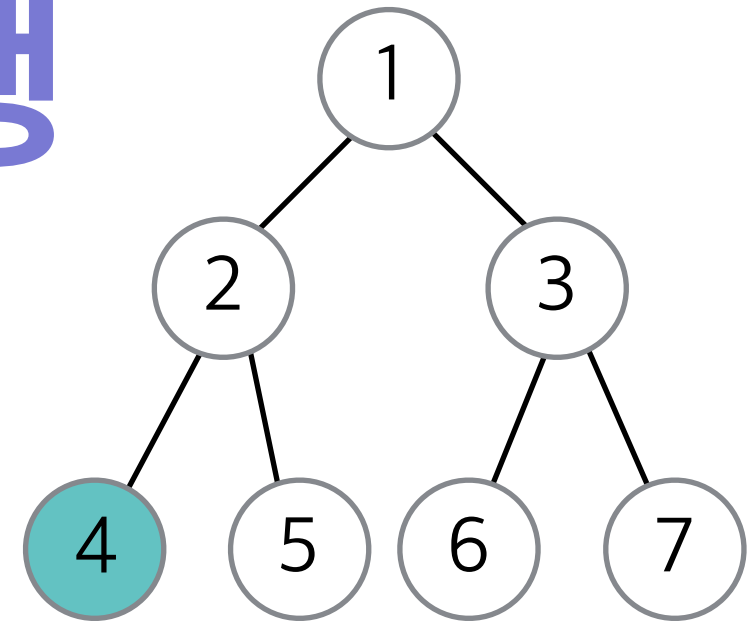
```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:5, (2, [])

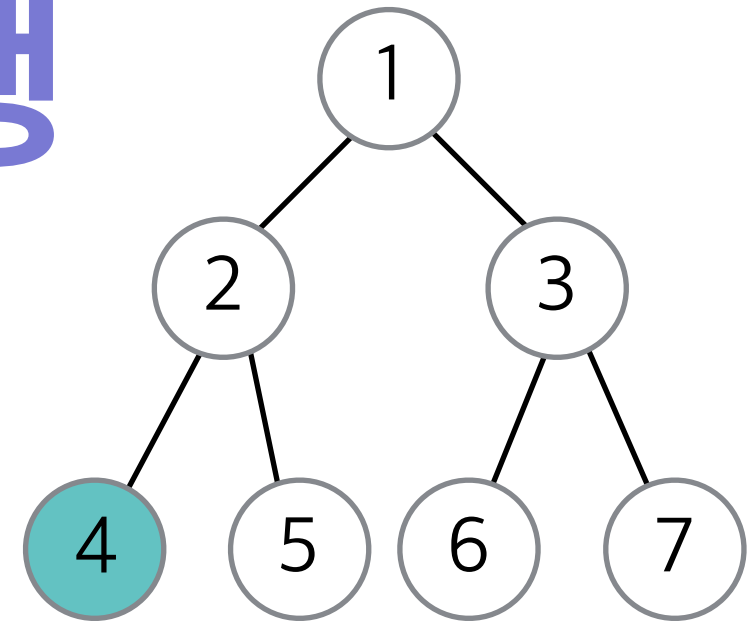
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]

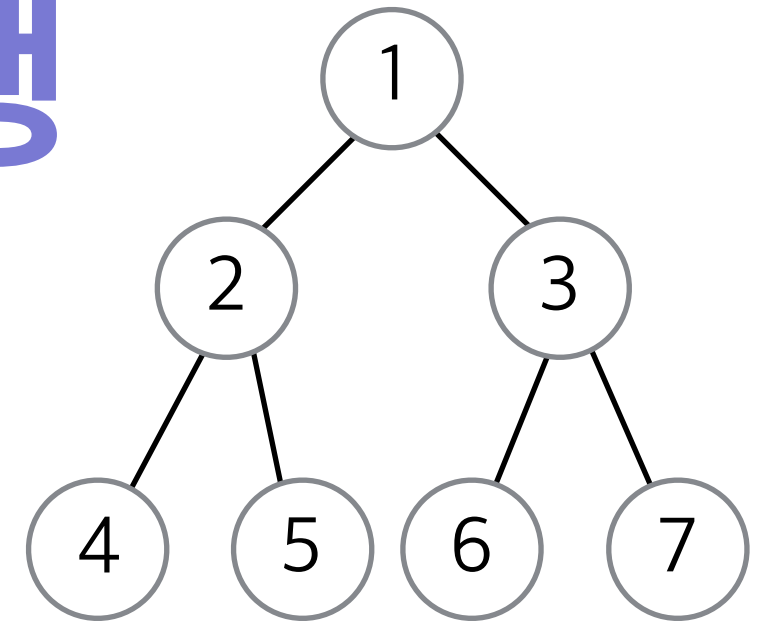


po:5, (2, [])

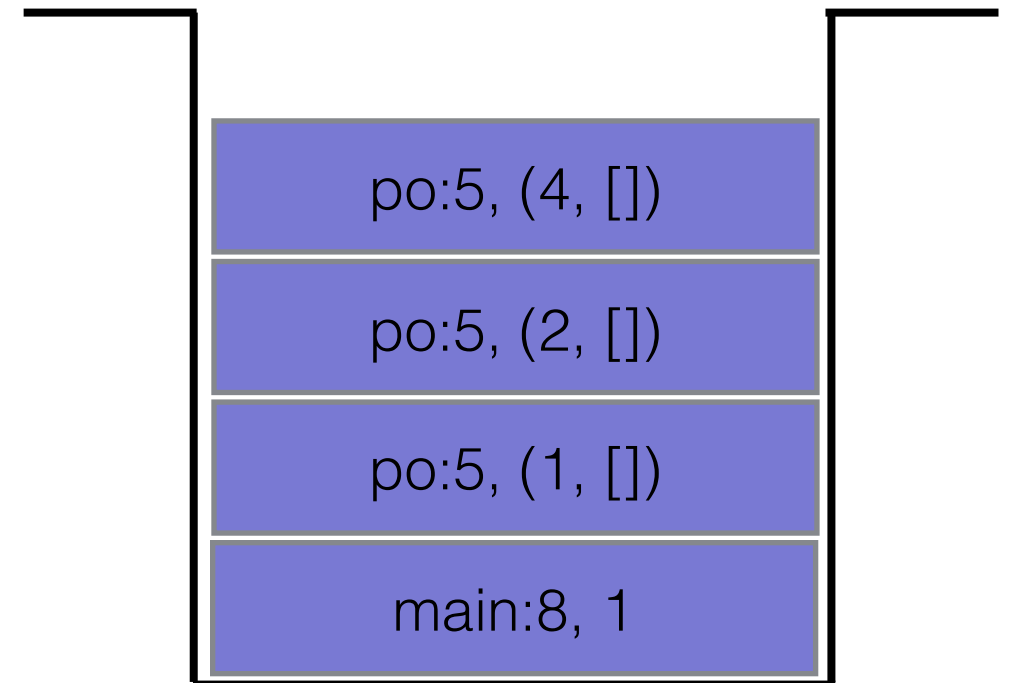
po:5, (1, [])

main:8, 1

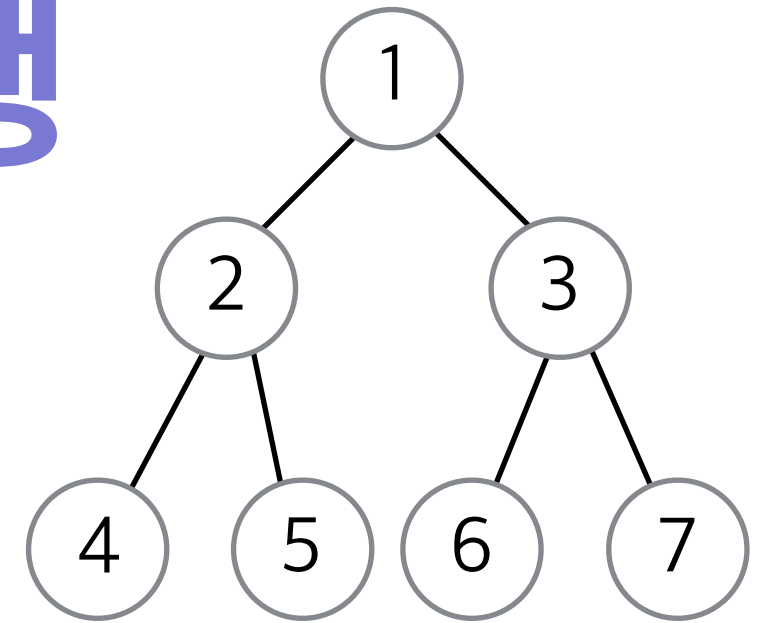
재귀함수의 실행



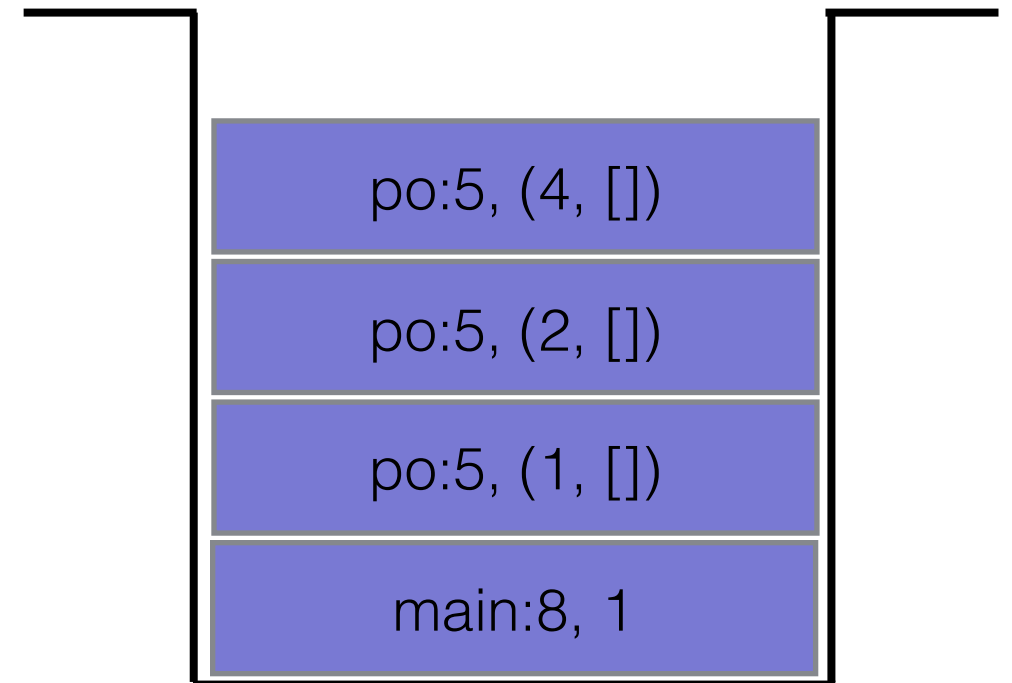
```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```



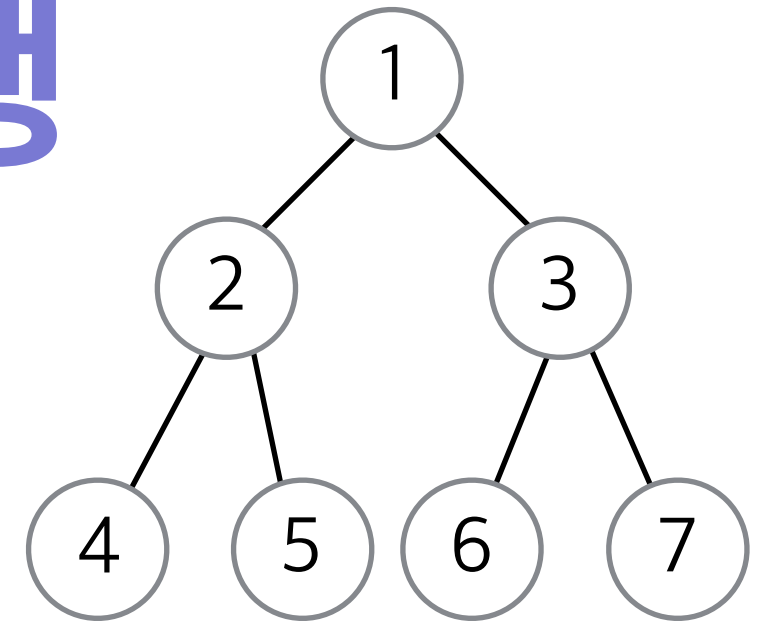
재귀함수의 실행



```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

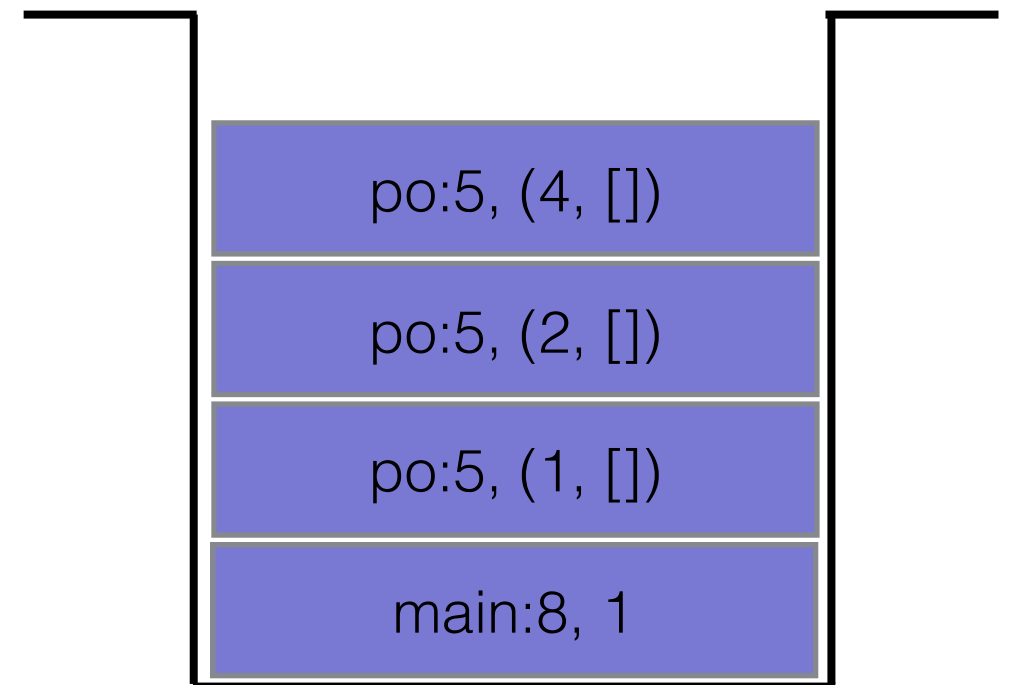


재귀함수의 실행



```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

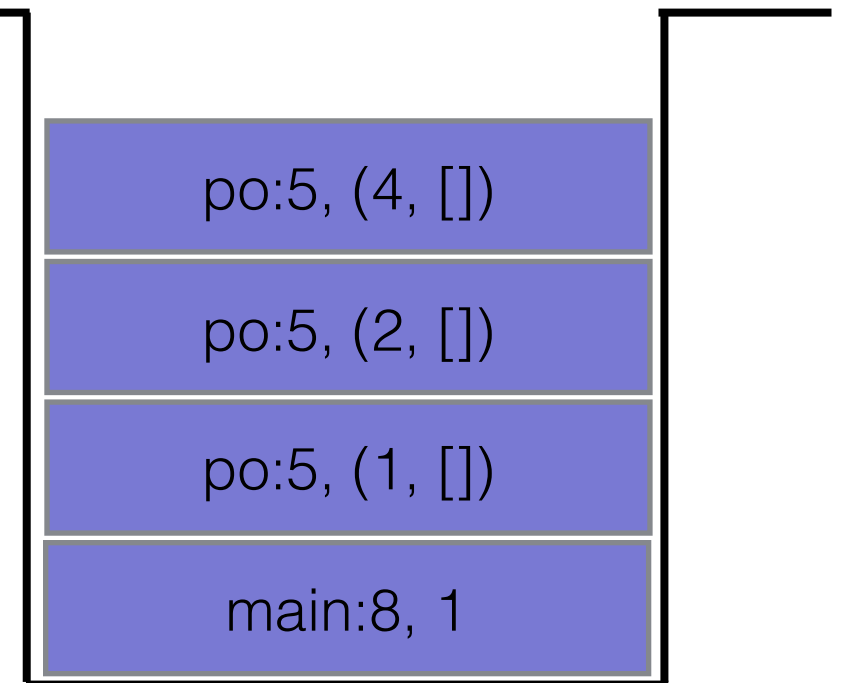
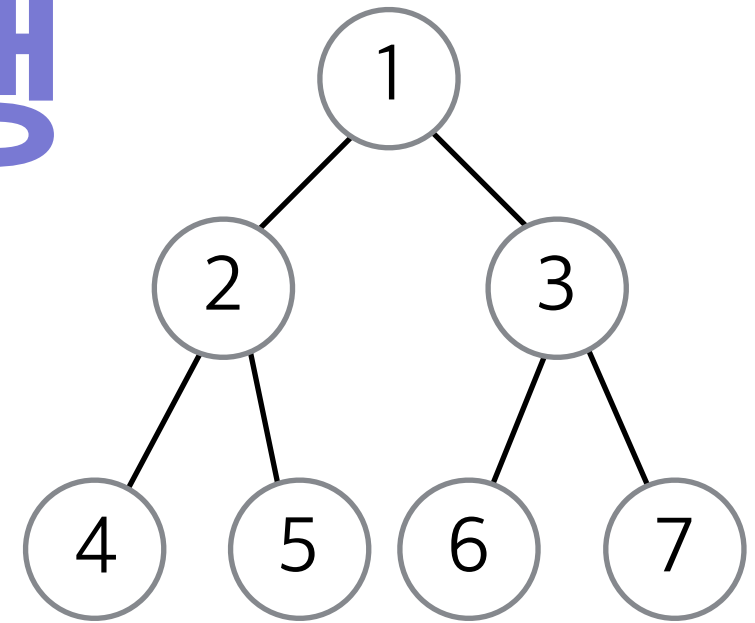
[]



재귀함수의 실행

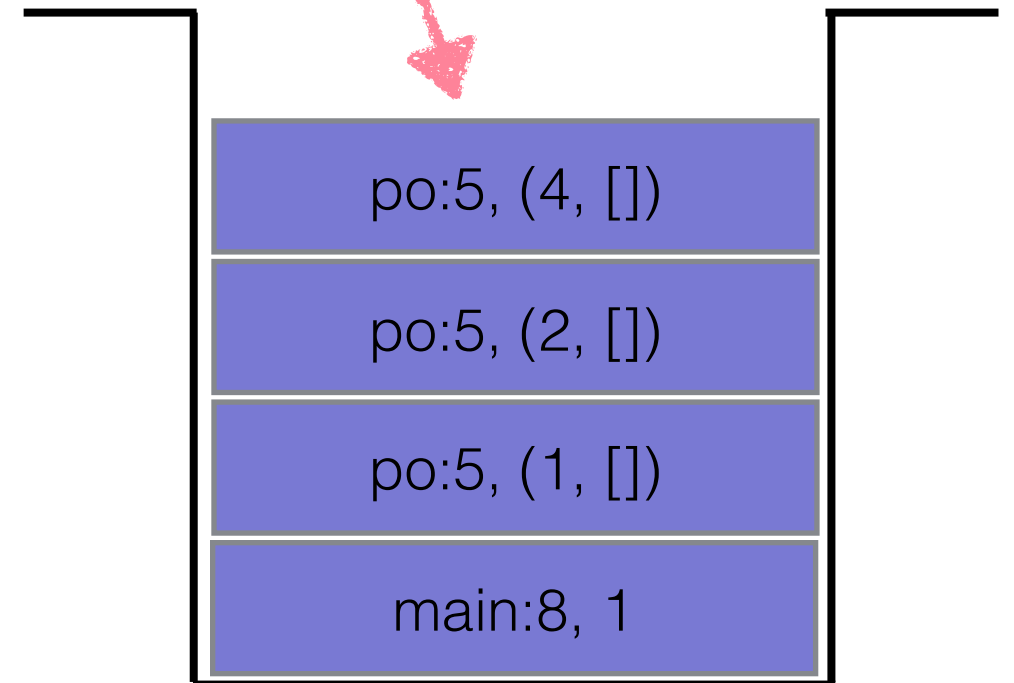
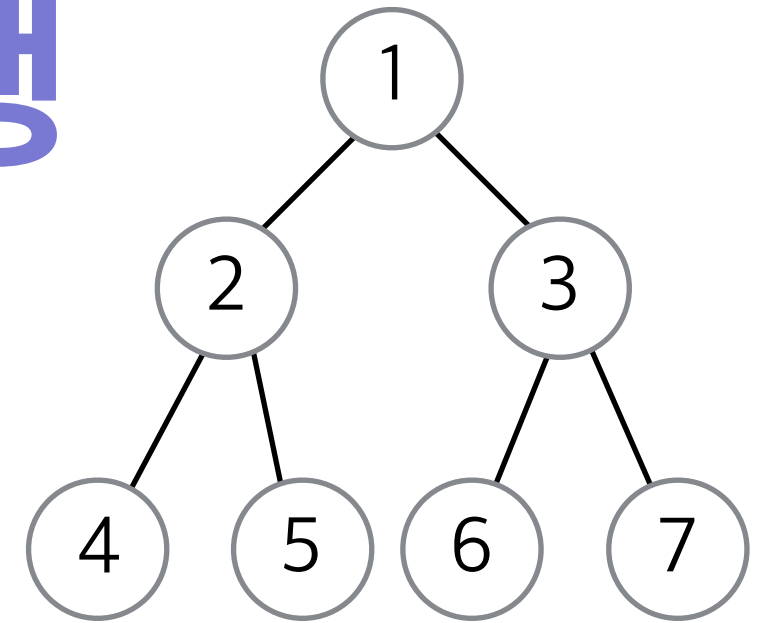
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
→ 4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



재귀함수의 실행

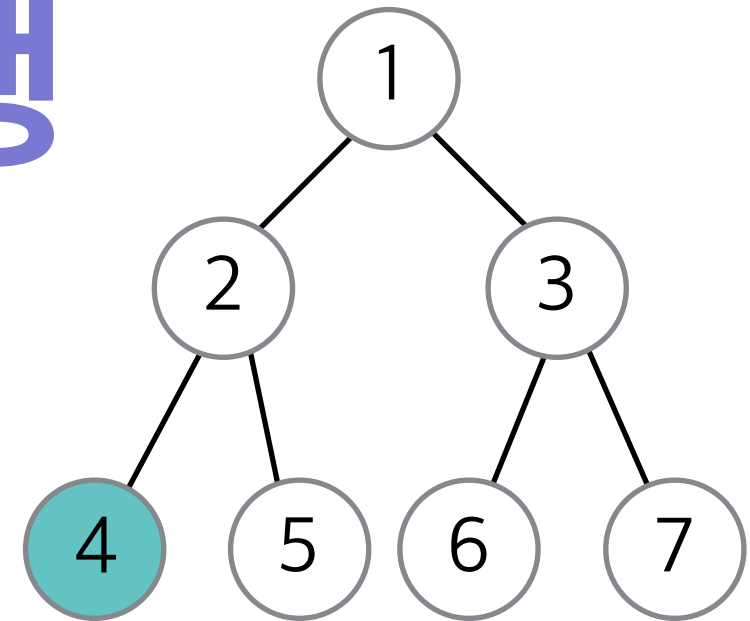
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:5, (2, [])

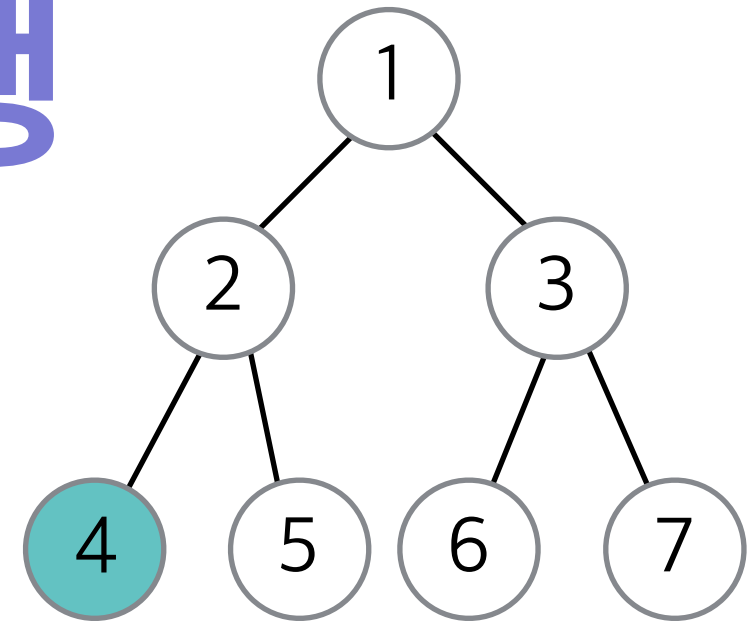
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]

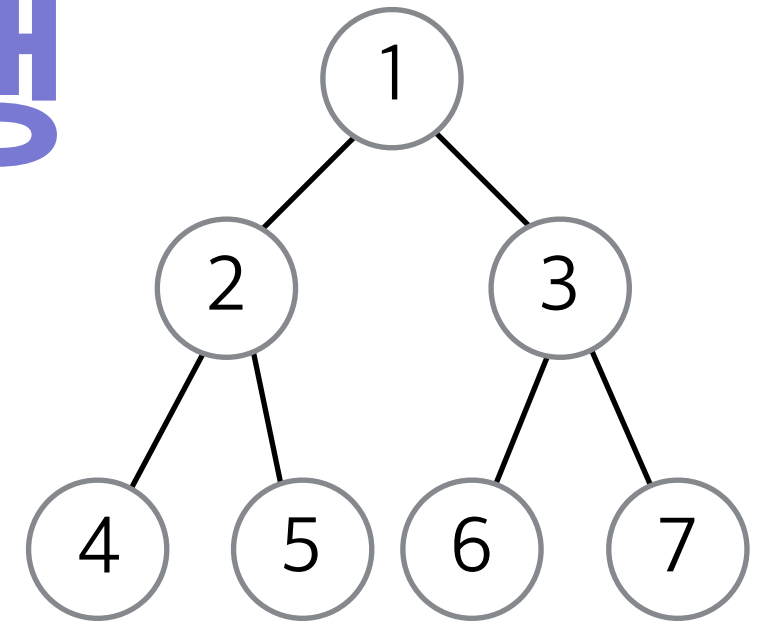


po:5, (2, [])

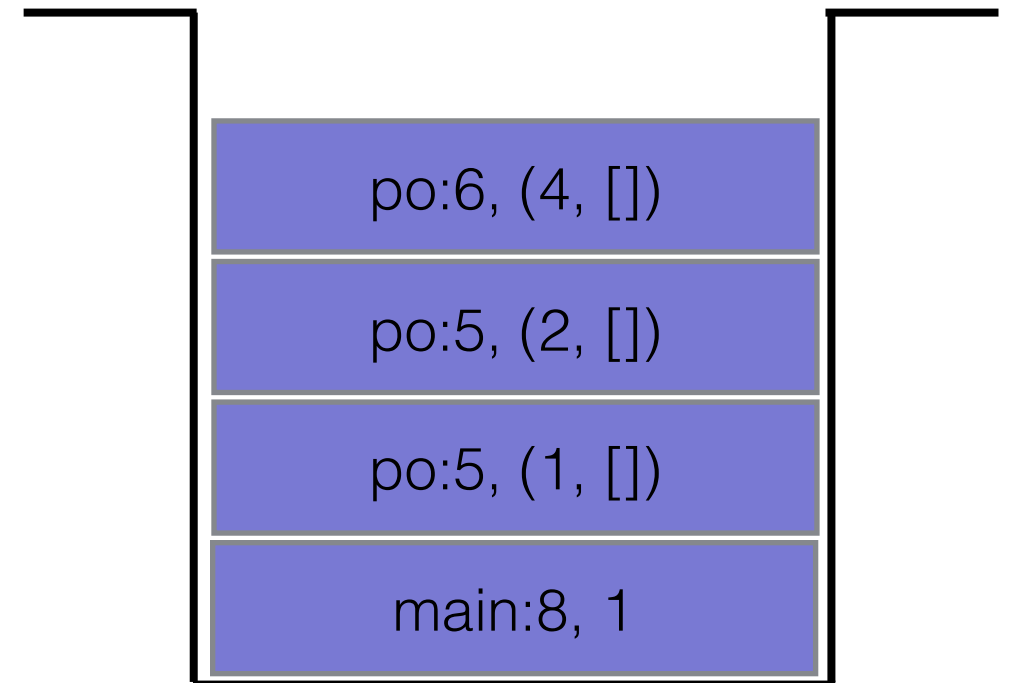
po:5, (1, [])

main:8, 1

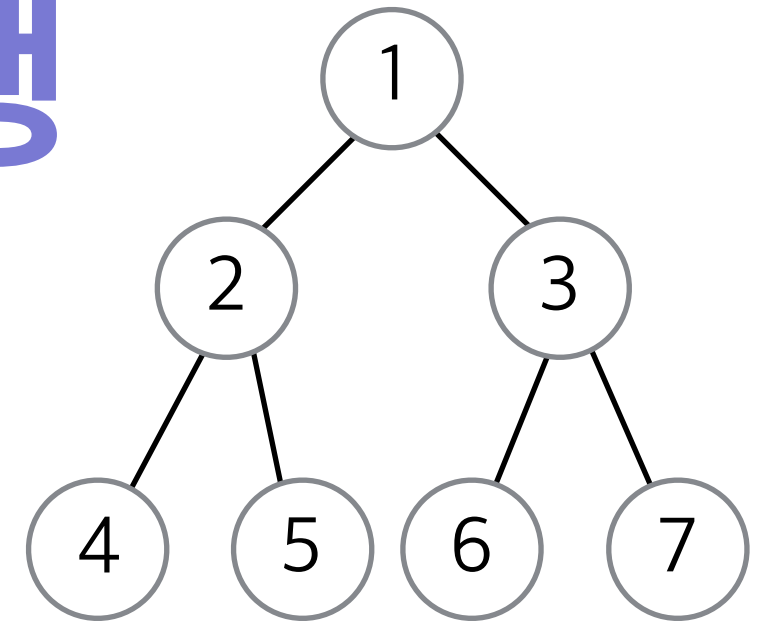
재귀함수의 실행



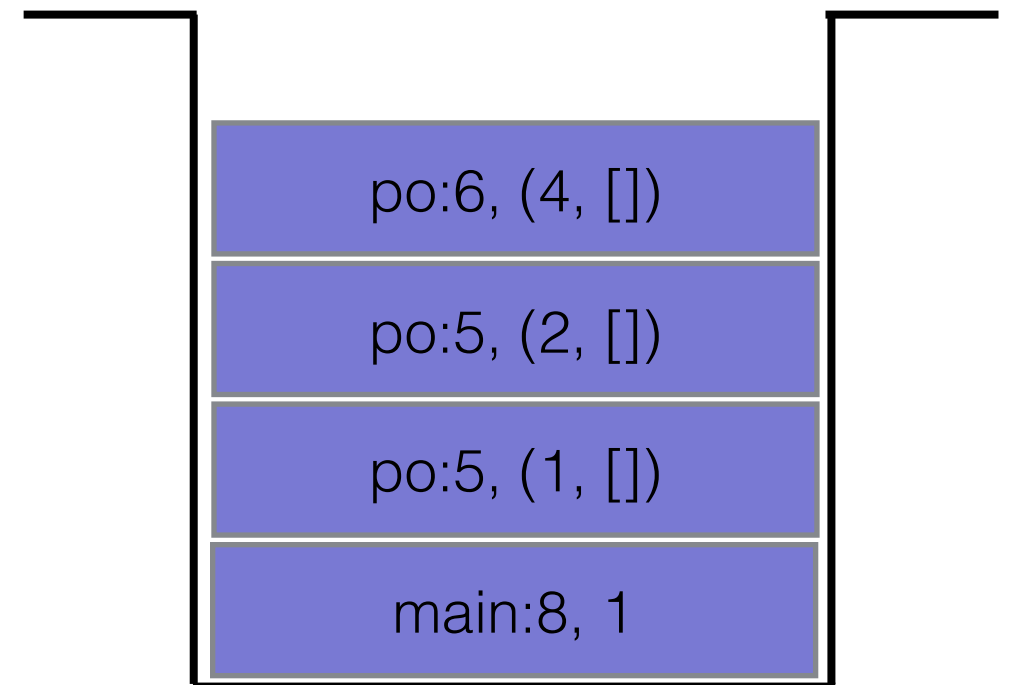
```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```



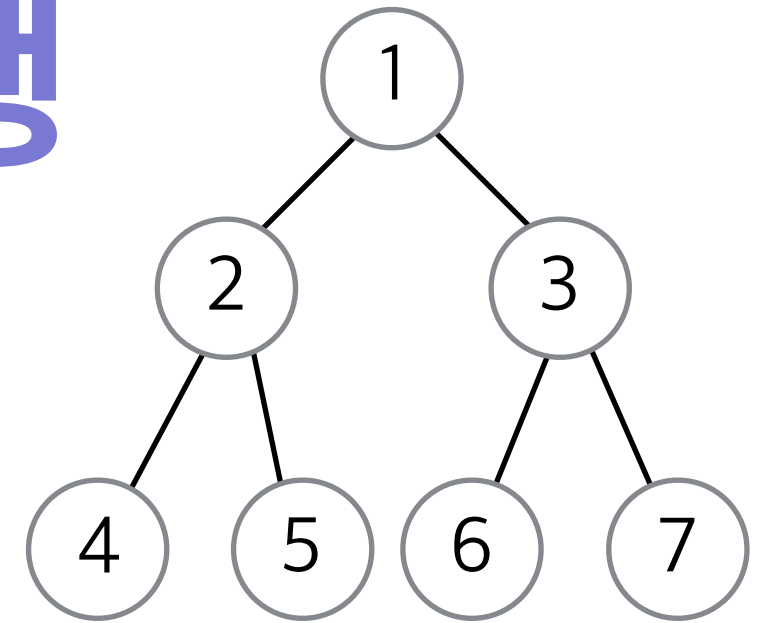
재귀함수의 실행



```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

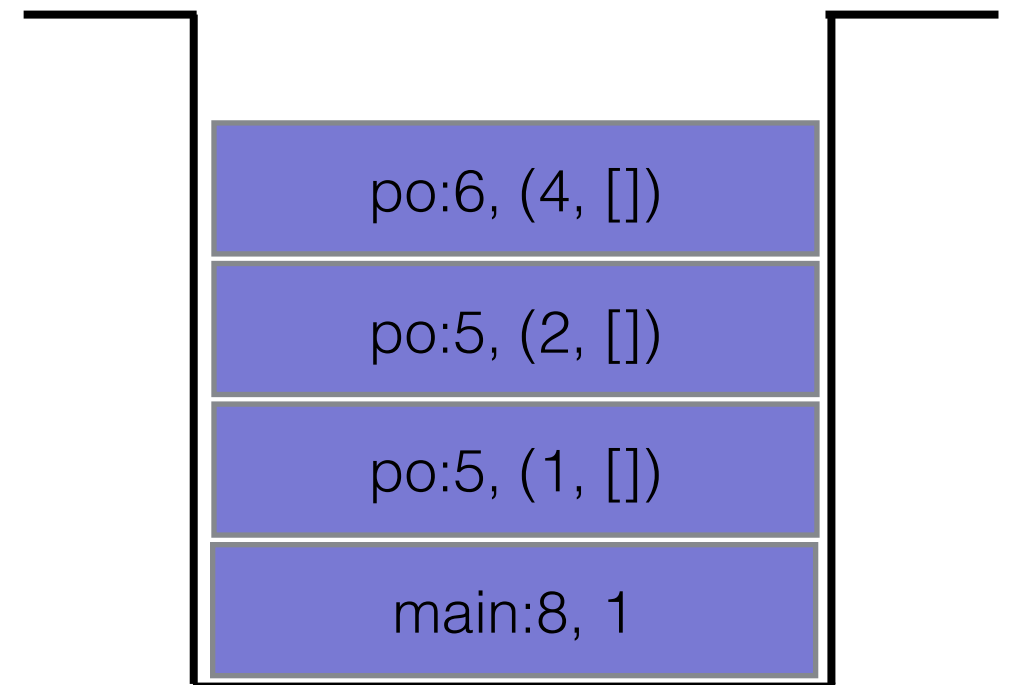


재귀함수의 실행

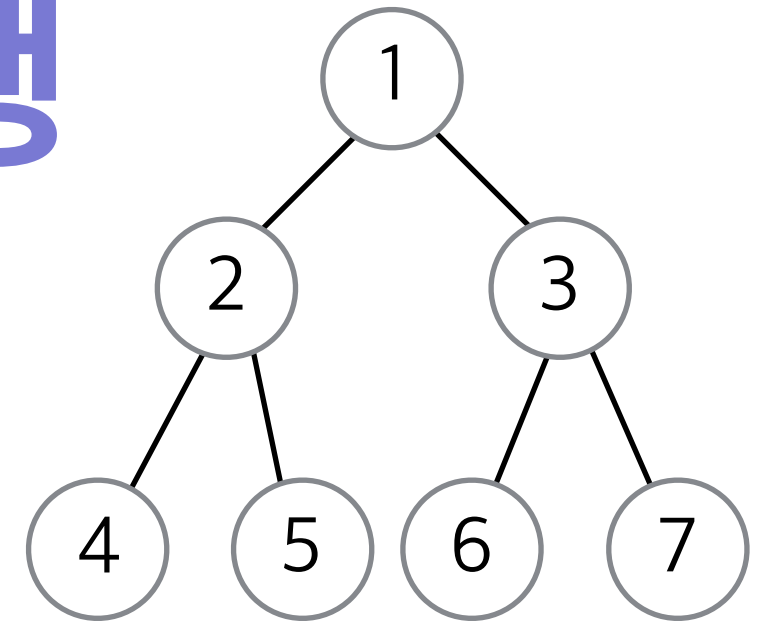


```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]

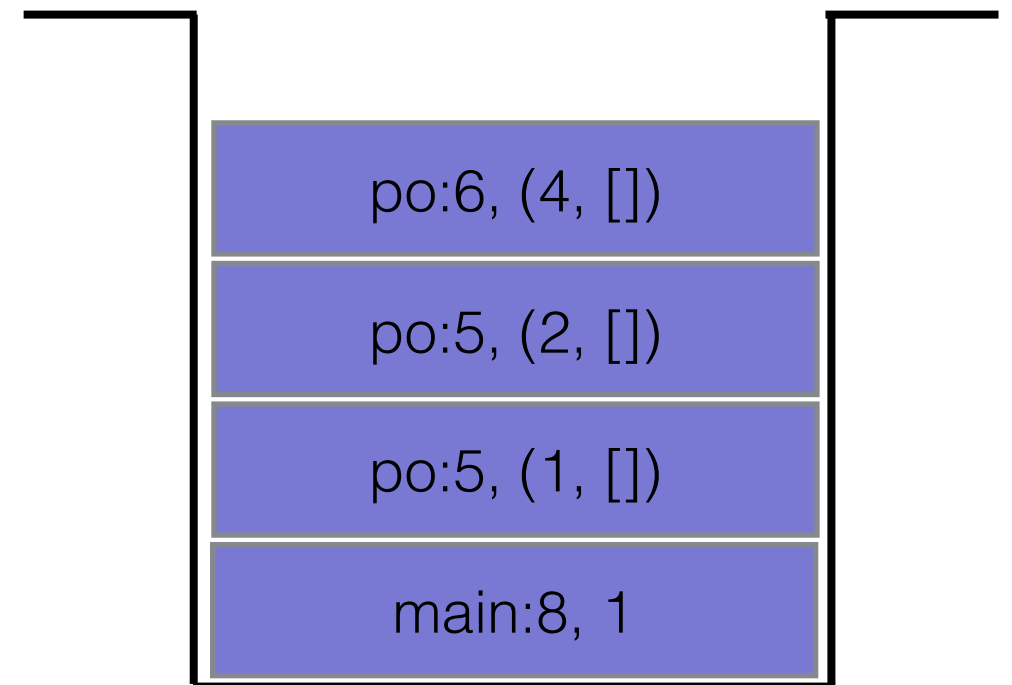


재귀함수의 실행



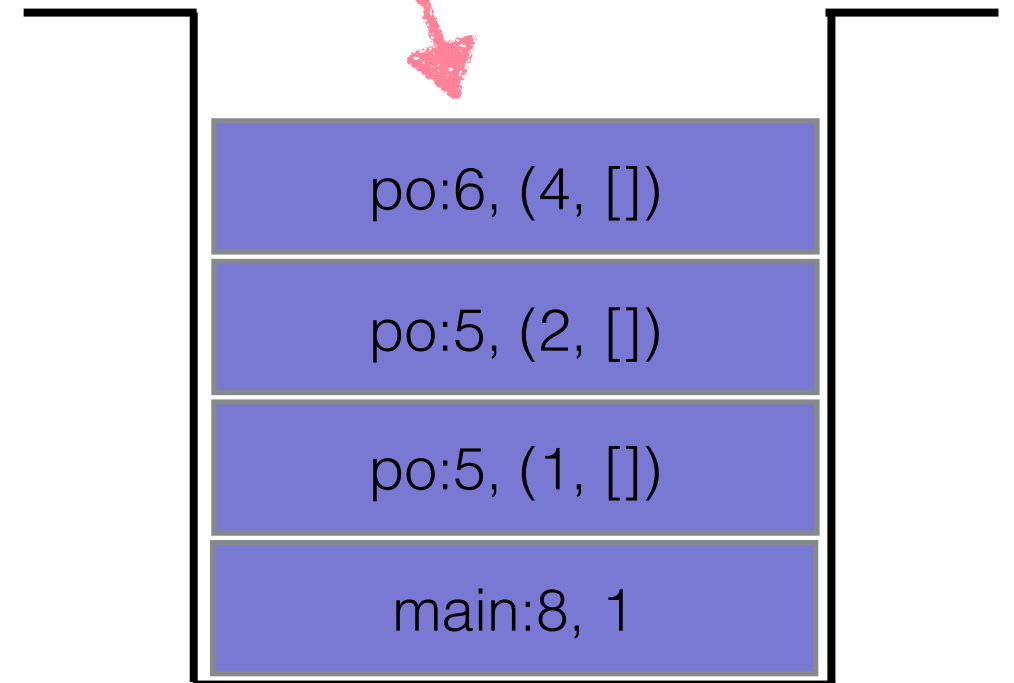
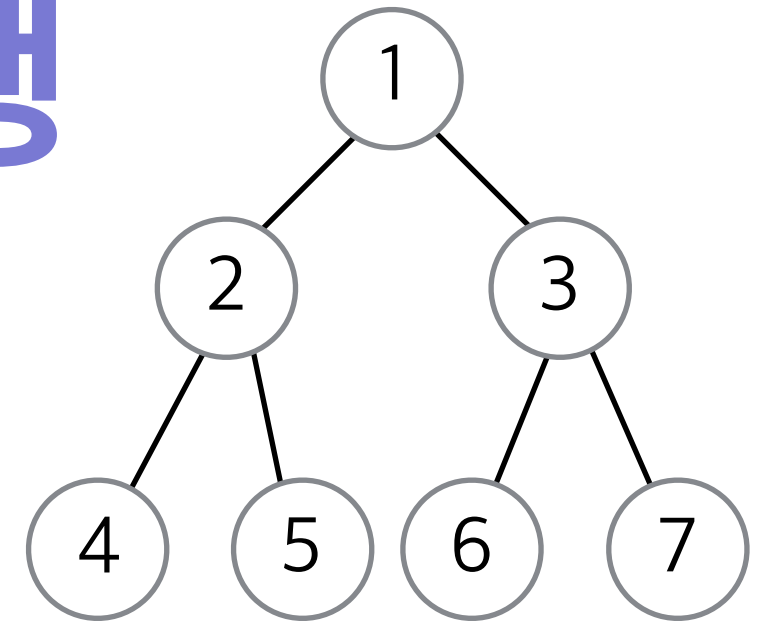
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
→ 4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



재귀함수의 실행

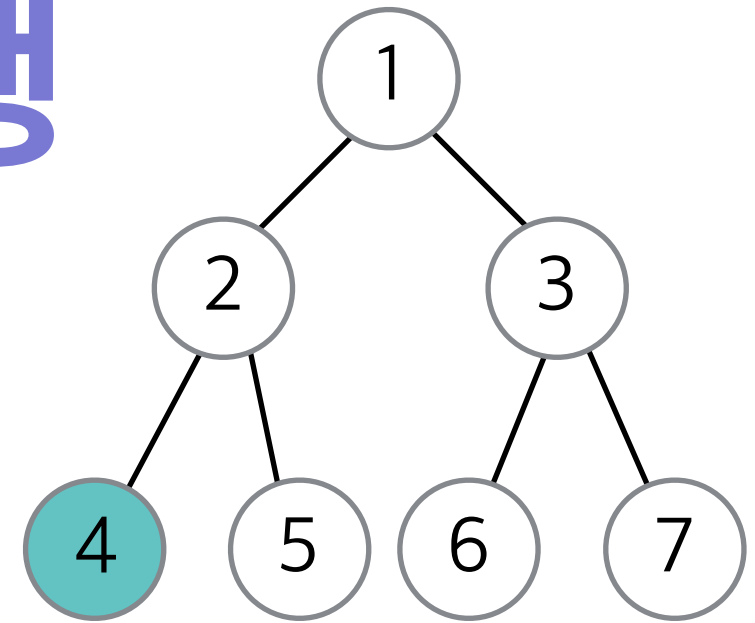
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
→ 4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:5, (2, [])

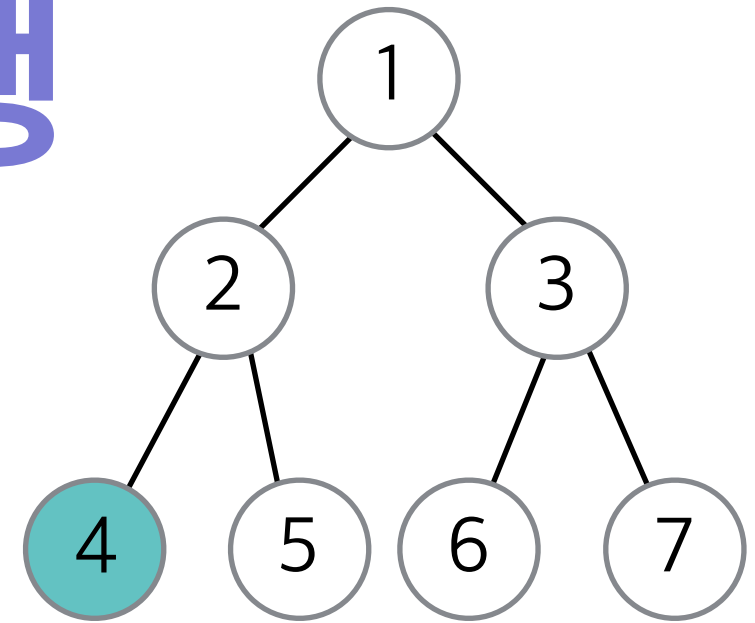
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
→ 7     result.append(tree.index)  
  
8     return result
```

[]



po:5, (2, [])

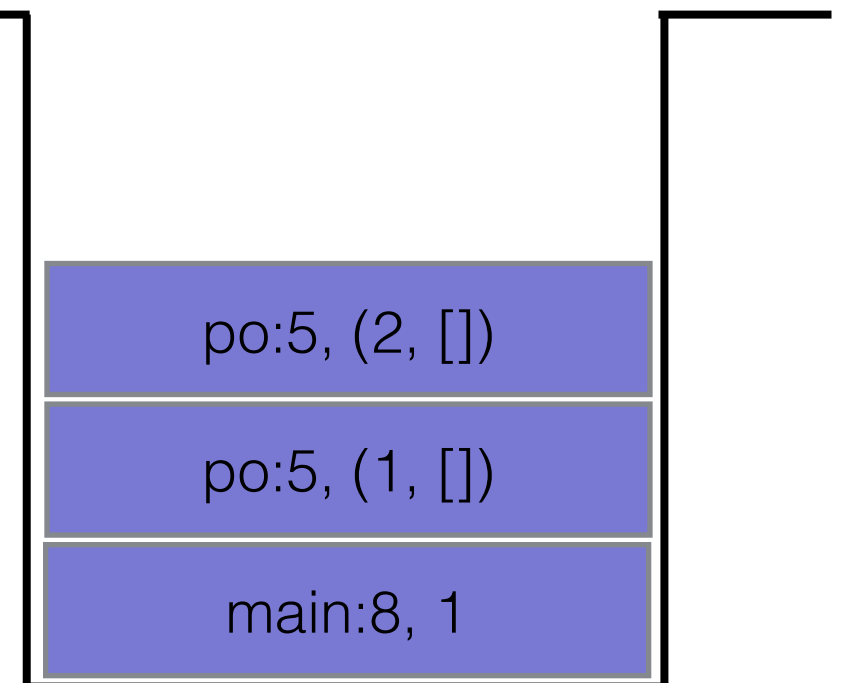
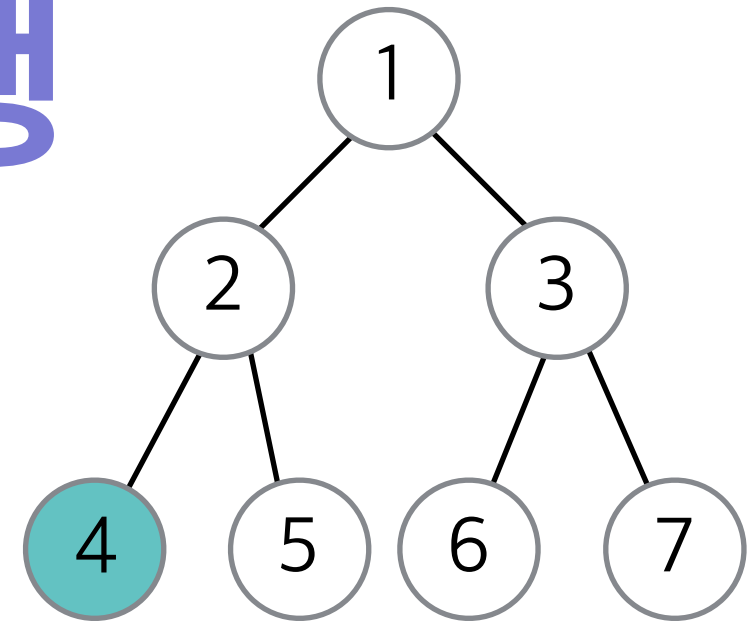
po:5, (1, [])

main:8, 1

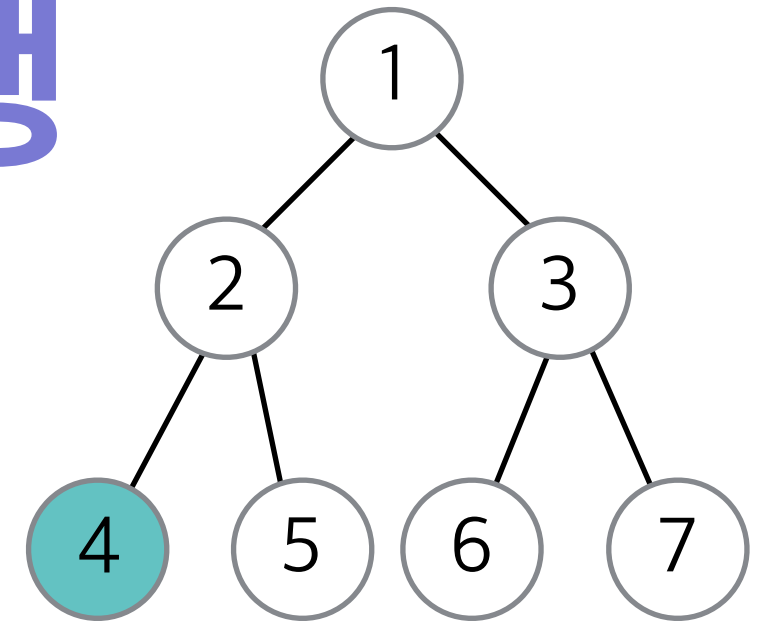
재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

[4]

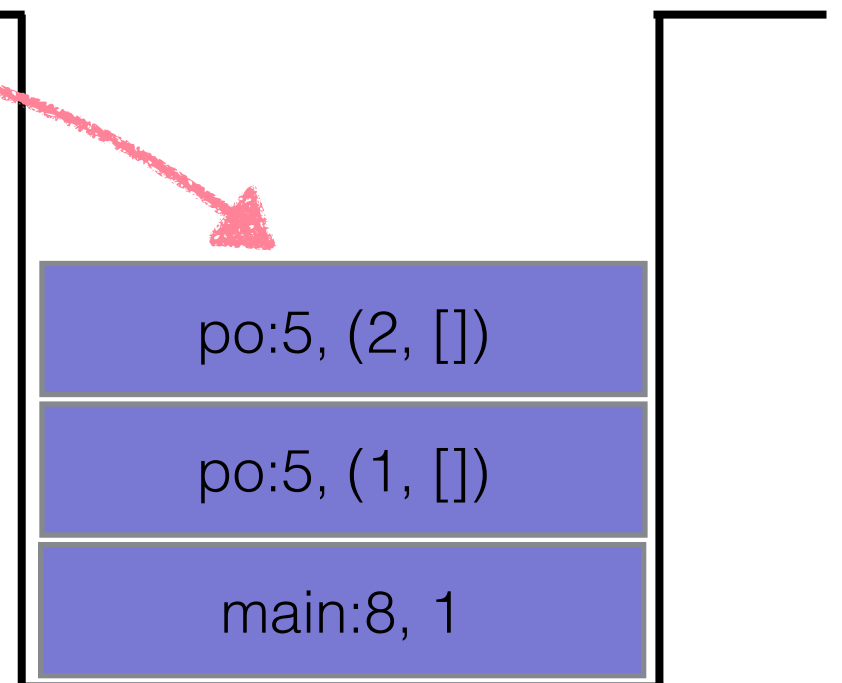


재귀함수의 실행



```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

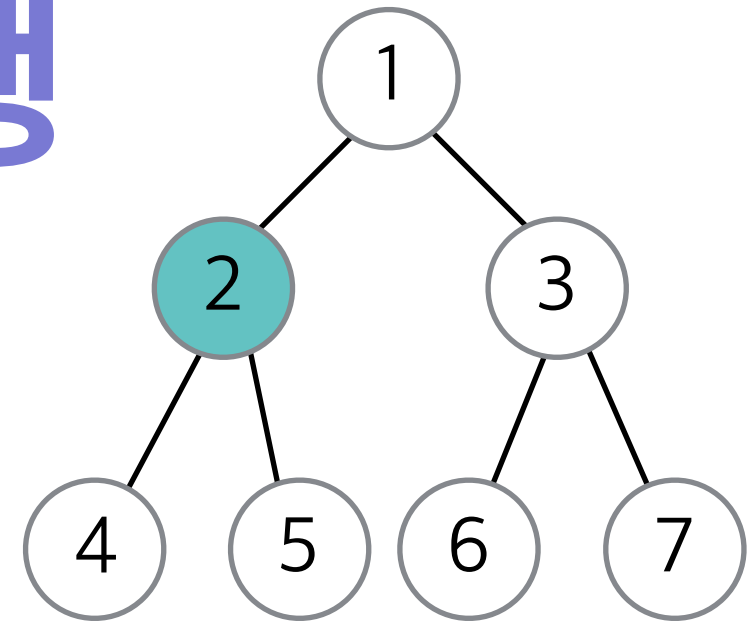
[4]



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



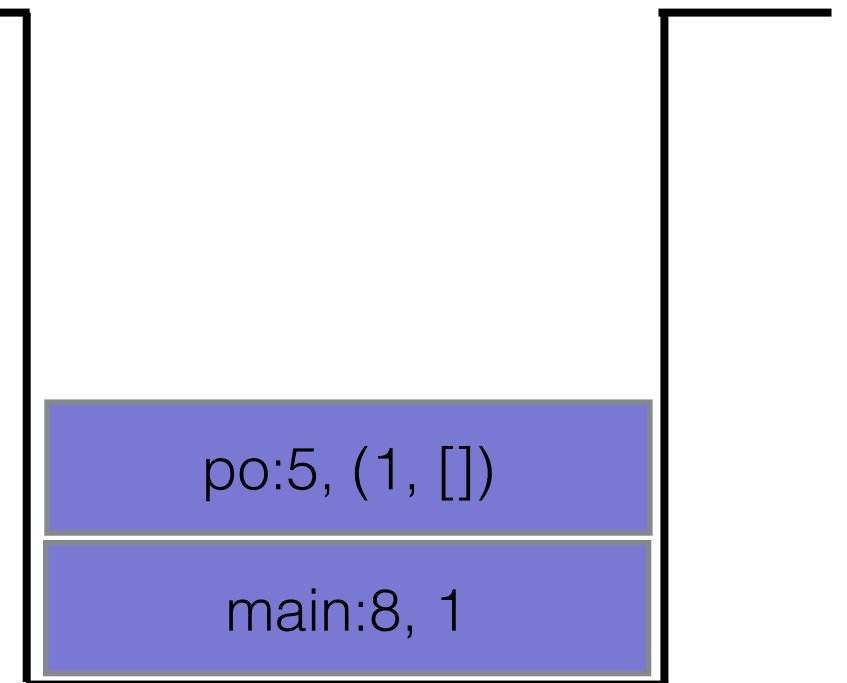
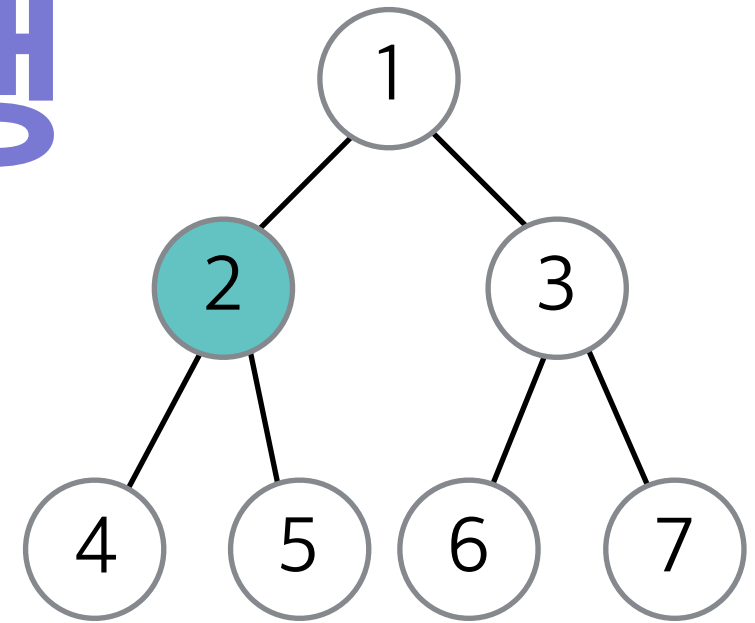
po:5, (1, [])

main:8, 1

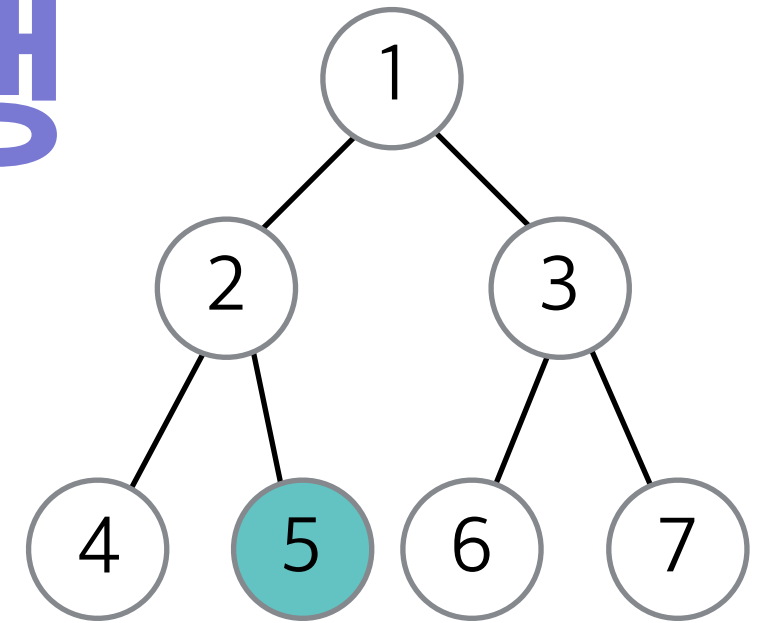
재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

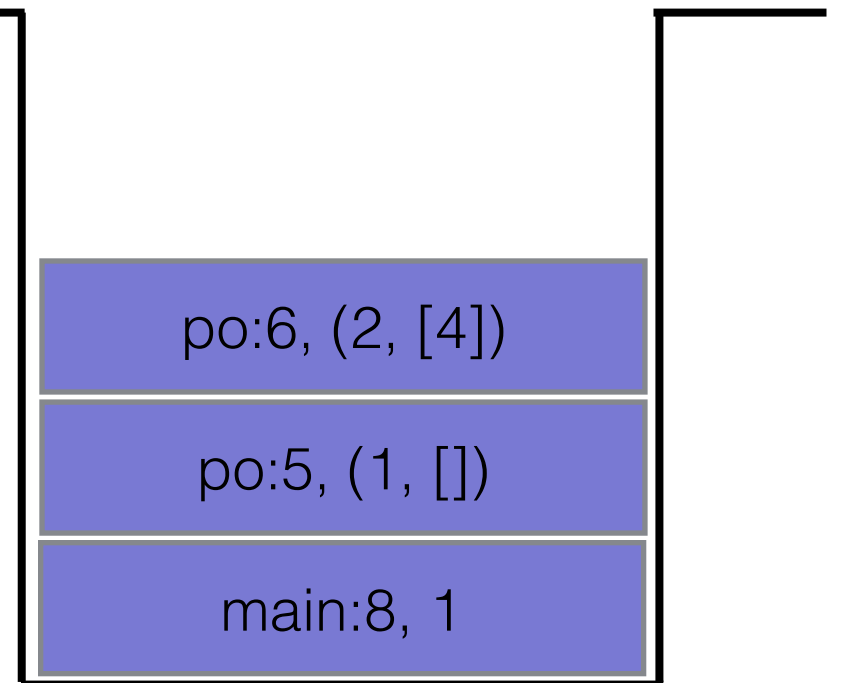
[4]



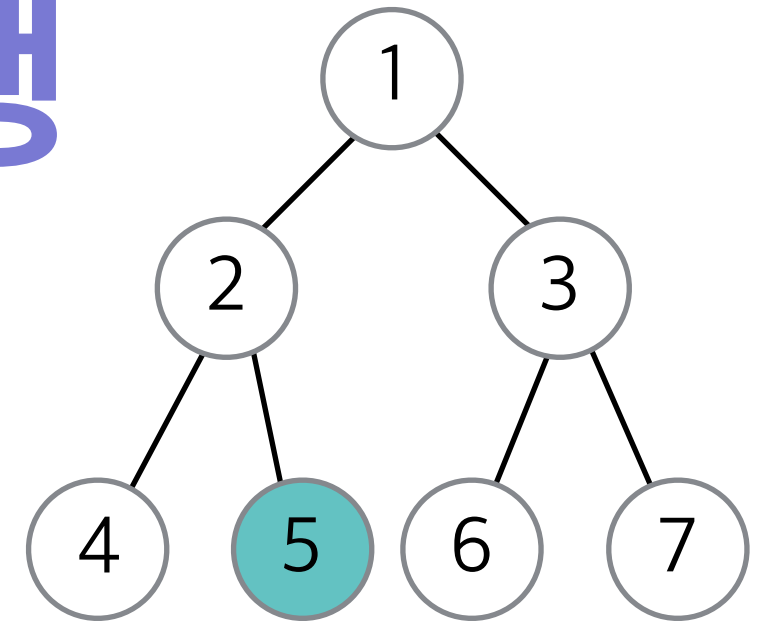
재귀함수의 실행



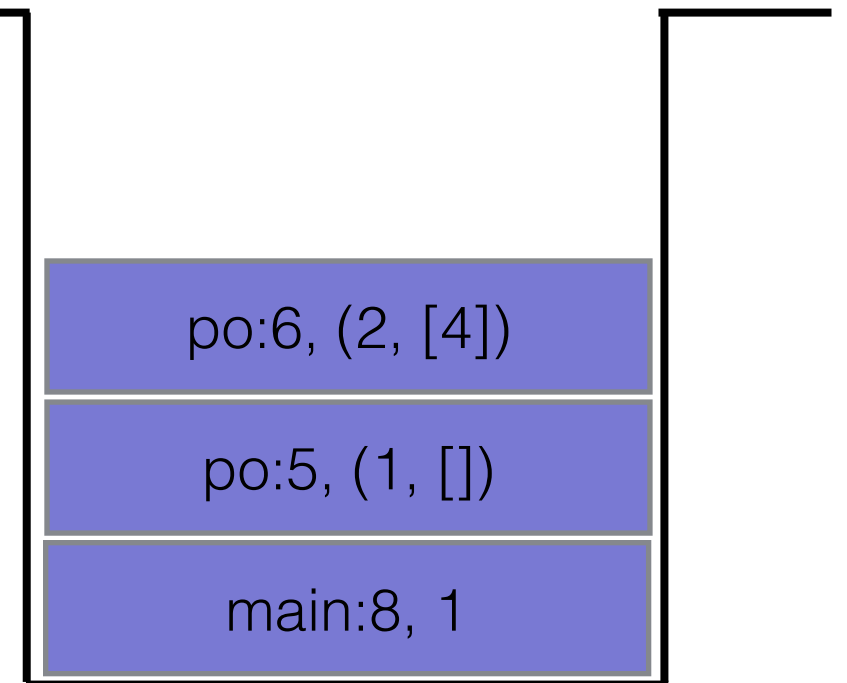
```
→ 1 def postorder(tree) :  
  2     result = []  
  
  3     if tree == None :  
  4         return []  
  
  5     result = postorder(tree.left)  
  6     result = result + postorder(tree.right)  
  7     result.append(tree.index)  
  
  8     return result
```



재귀함수의 실행



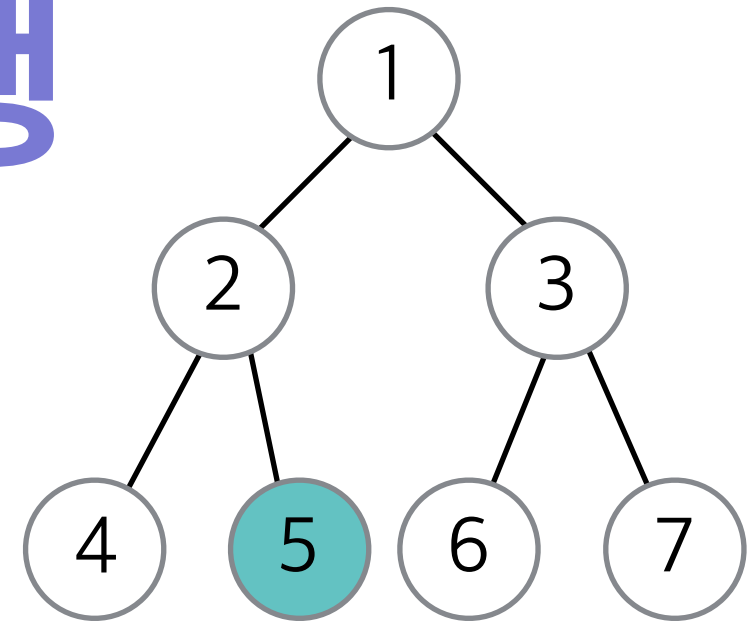
```
1 def postorder(tree) :  
→ 2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
→ 3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:6, (2, [4])

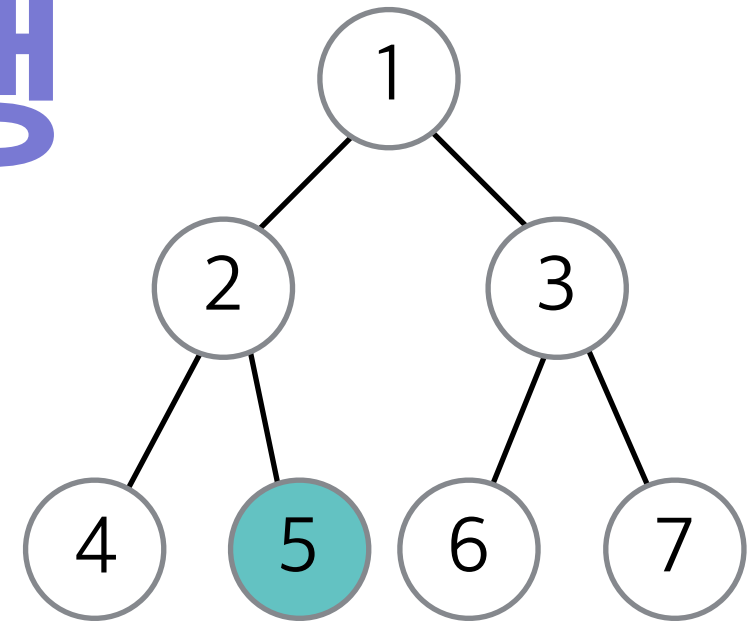
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:6, (2, [4])

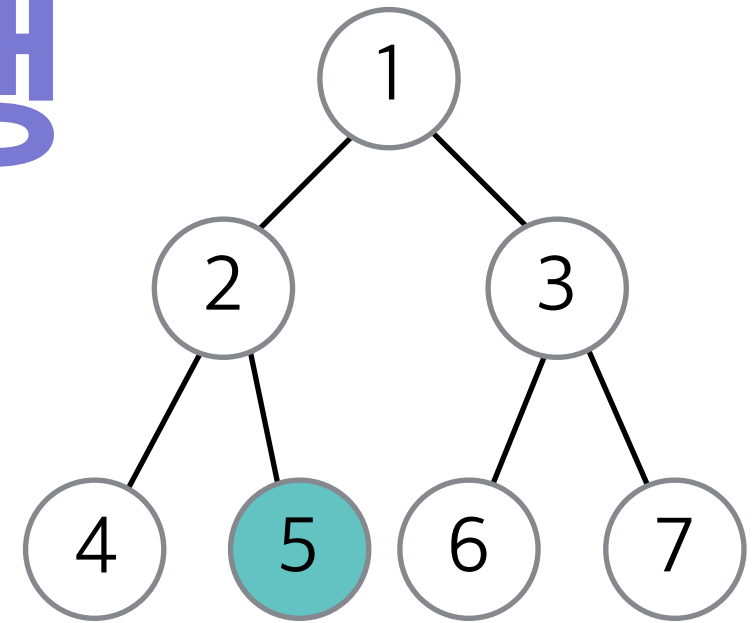
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]



po:6, (2, [4])

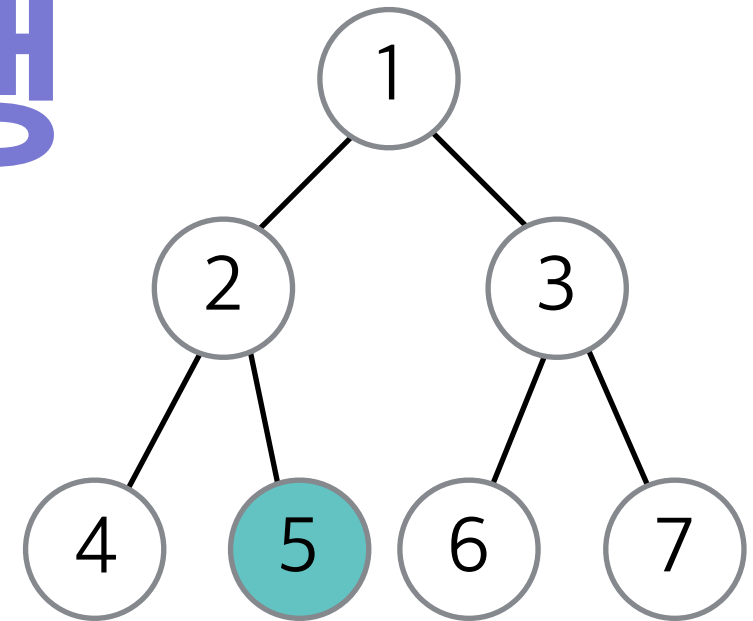
po:5, (1, [])

main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
→ 7     result.append(tree.index)  
  
8     return result
```

[]



po:6, (2, [4])

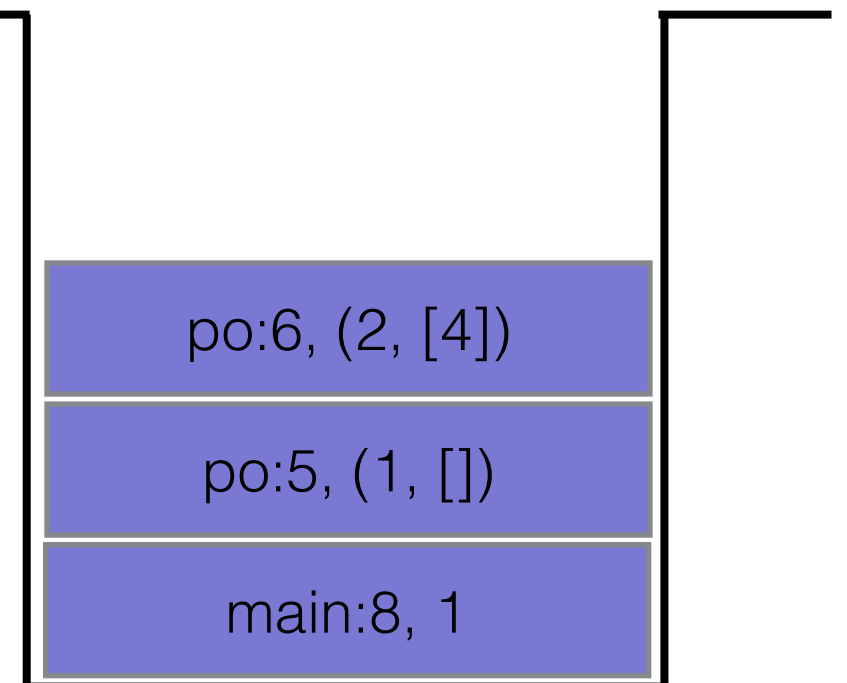
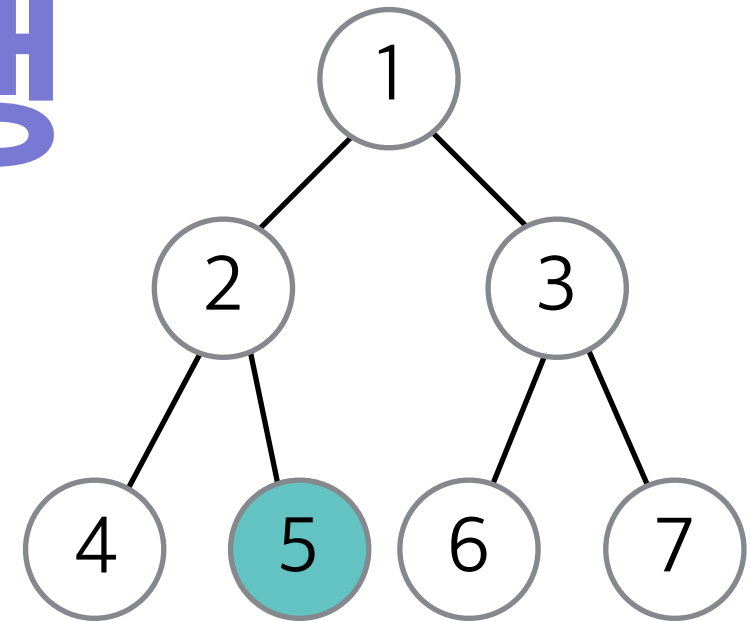
po:5, (1, [])

main:8, 1

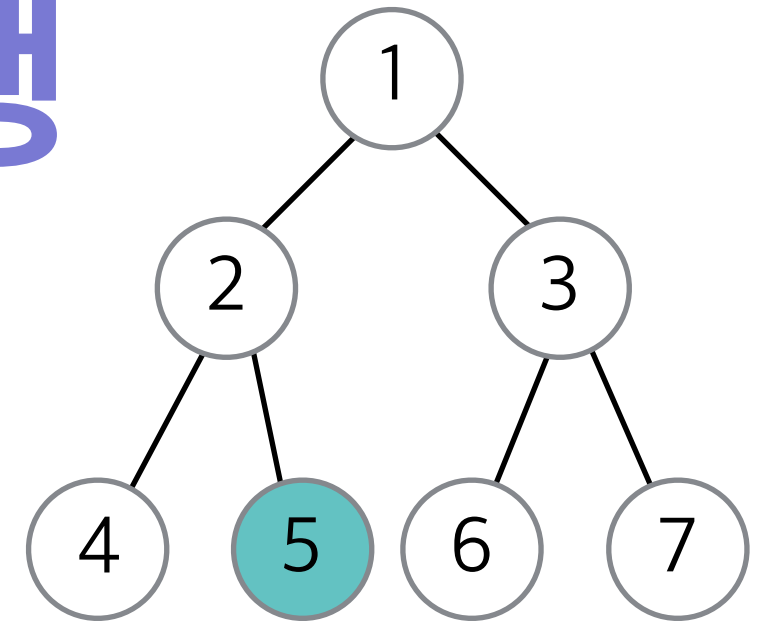
재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

[5]

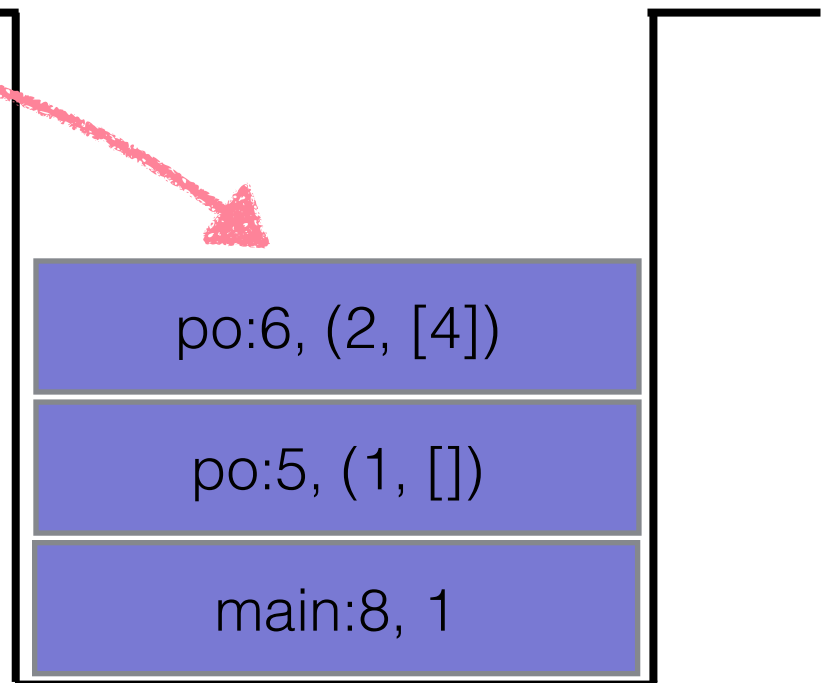


재귀함수의 실행



```
1 def postorder(tree) :  
2   result = []  
  
3   if tree == None :  
4     return []  
  
5   result = postorder(tree.left)  
6   result = result + postorder(tree.right)  
7   result.append(tree.index)  
→ 8 return result
```

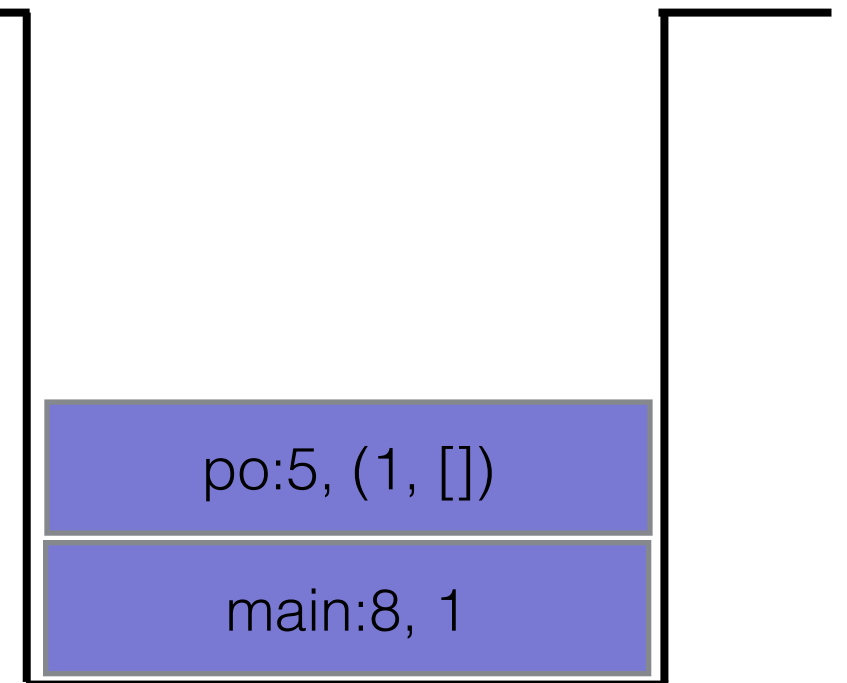
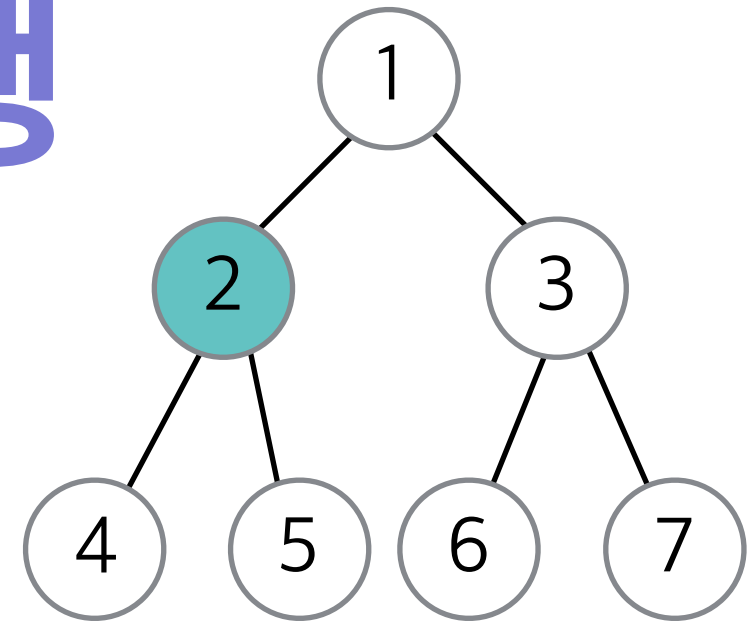
[5]



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

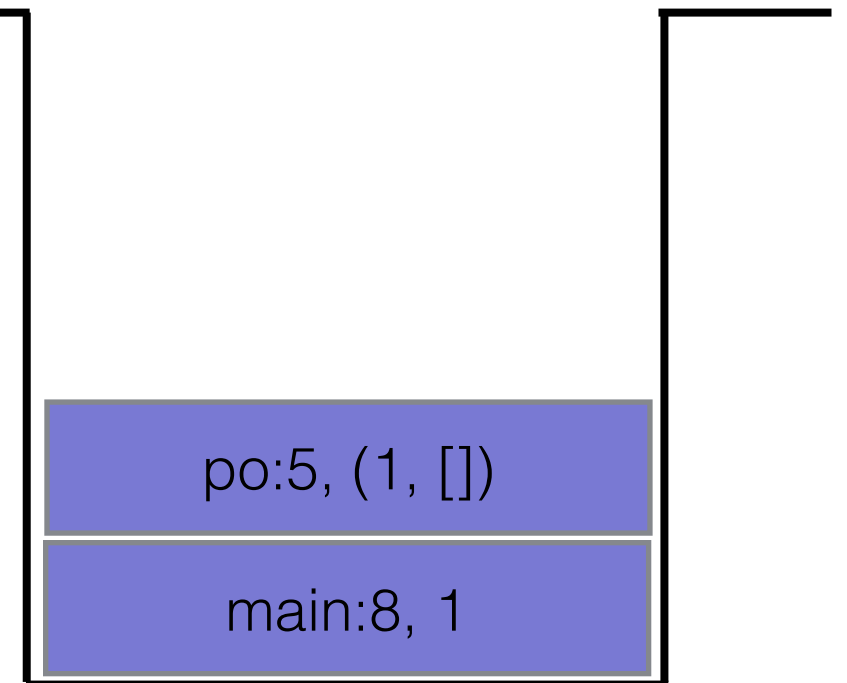
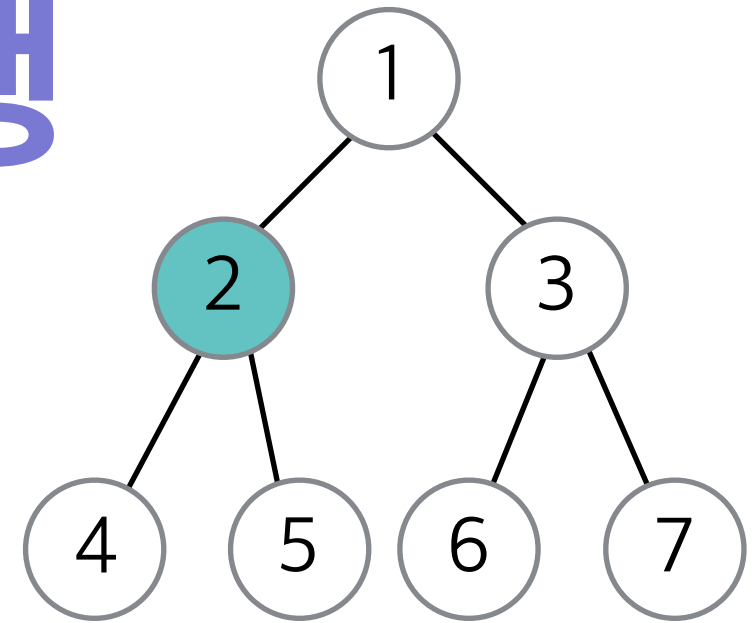
[4]



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
→ 7     result.append(tree.index)  
  
8     return result
```

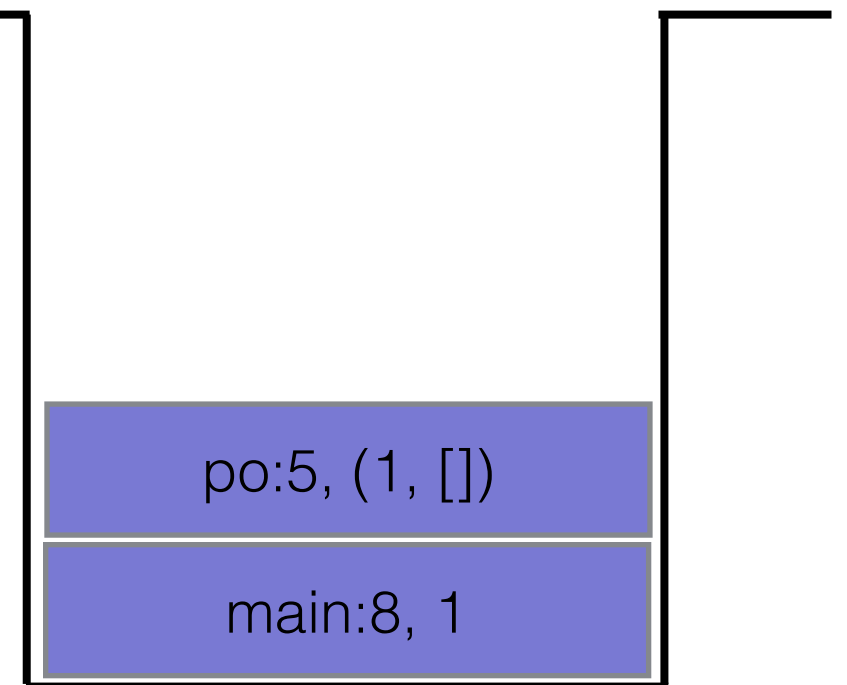
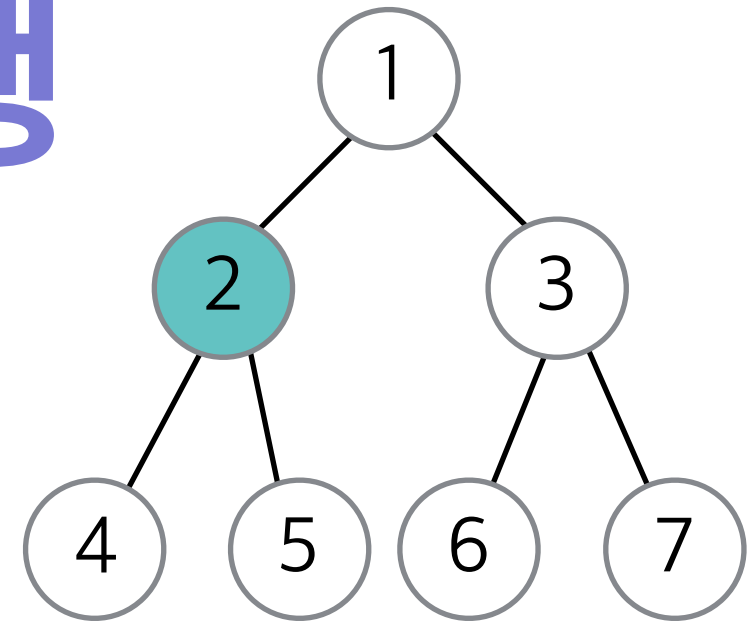
[4, 5]



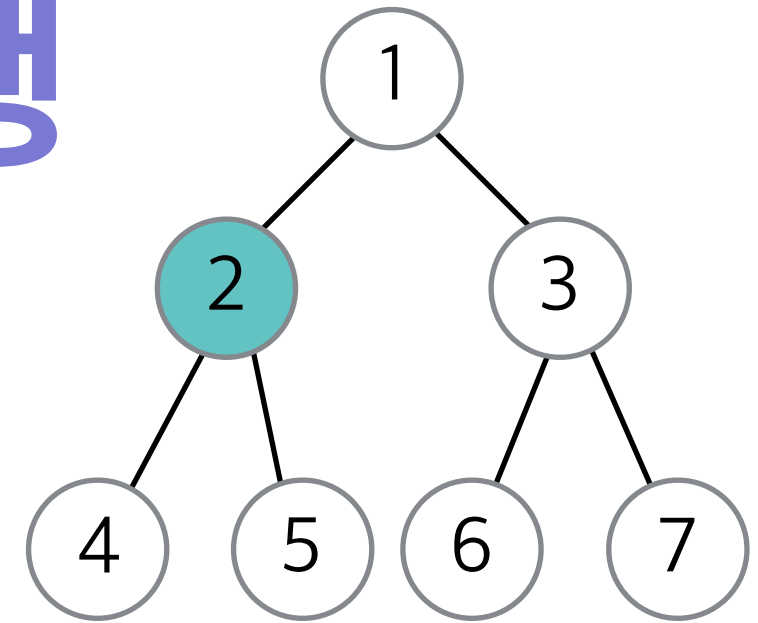
재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

[4, 5, 2]

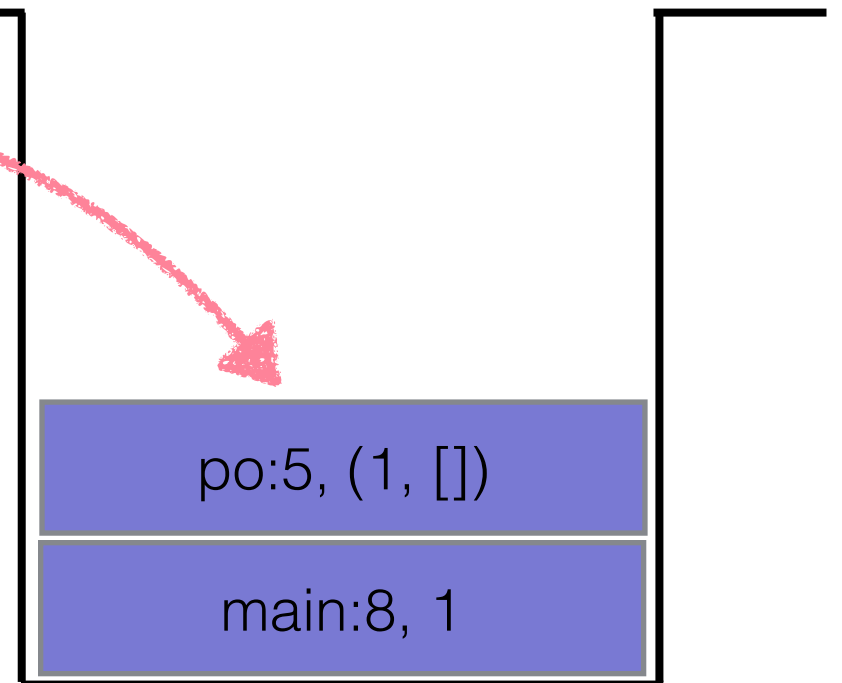


재귀함수의 실행



```
1 def postorder(tree) :  
2   result = []  
  
3   if tree == None :  
4       return []  
  
5   result = postorder(tree.left)  
6   result = result + postorder(tree.right)  
7   result.append(tree.index)  
→ 8   return result
```

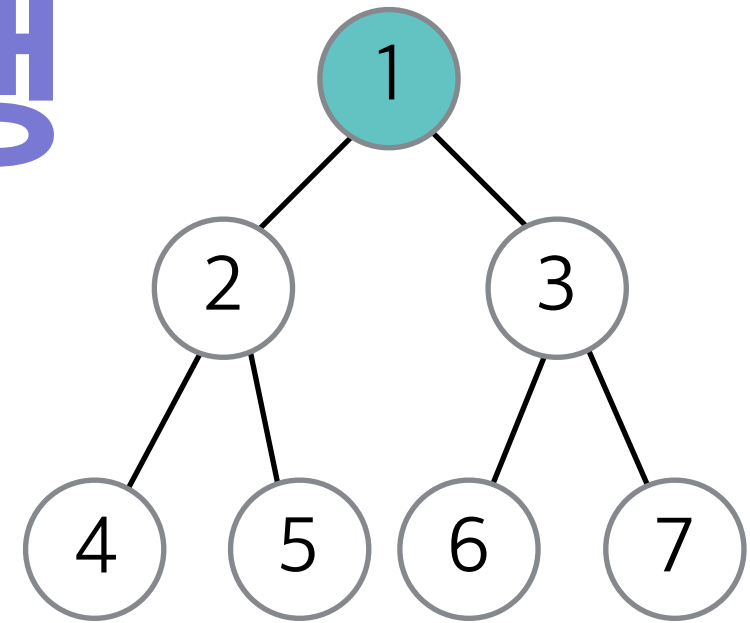
[4, 5, 2]



재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
→ 5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[]

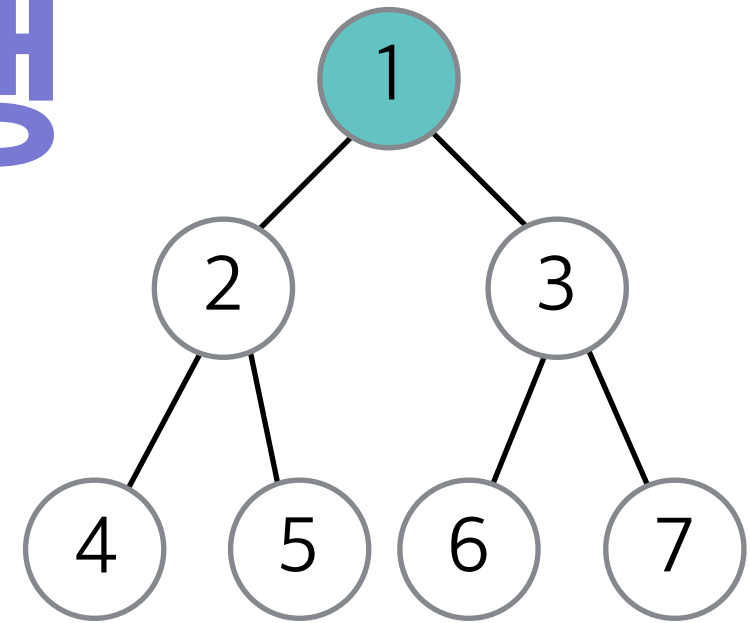


main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
→ 6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
  
8     return result
```

[4, 5, 2]

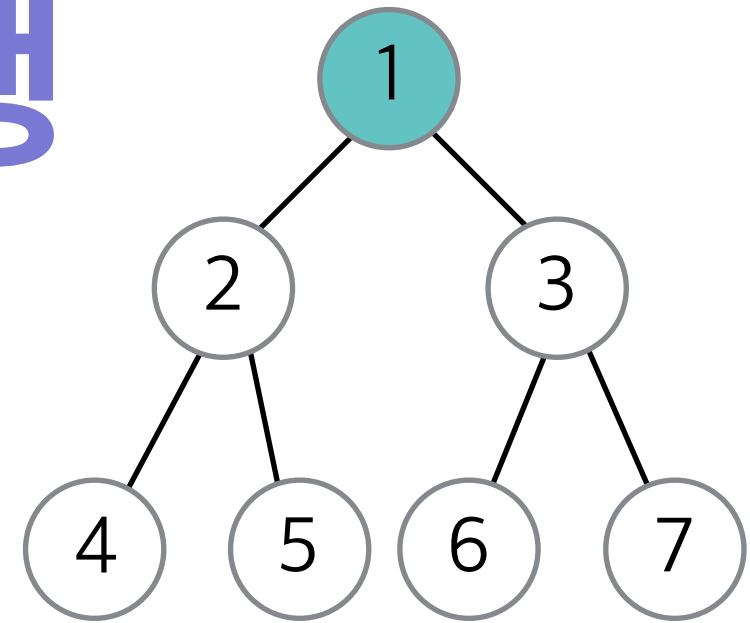


main:8, 1

재귀함수의 실행

```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
→ 7     result.append(tree.index)  
  
8     return result
```

[4, 5, 2, 6, 7, 3]

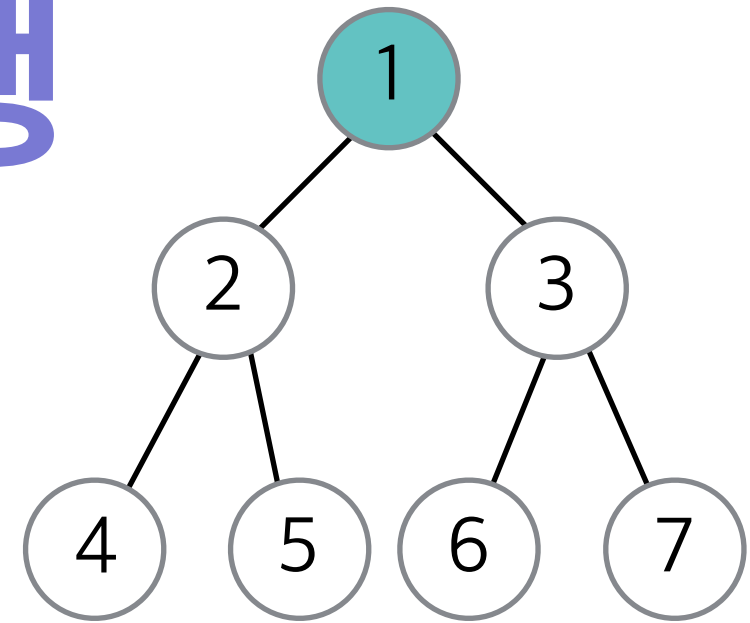


main:8, 1

재귀함수의 실행

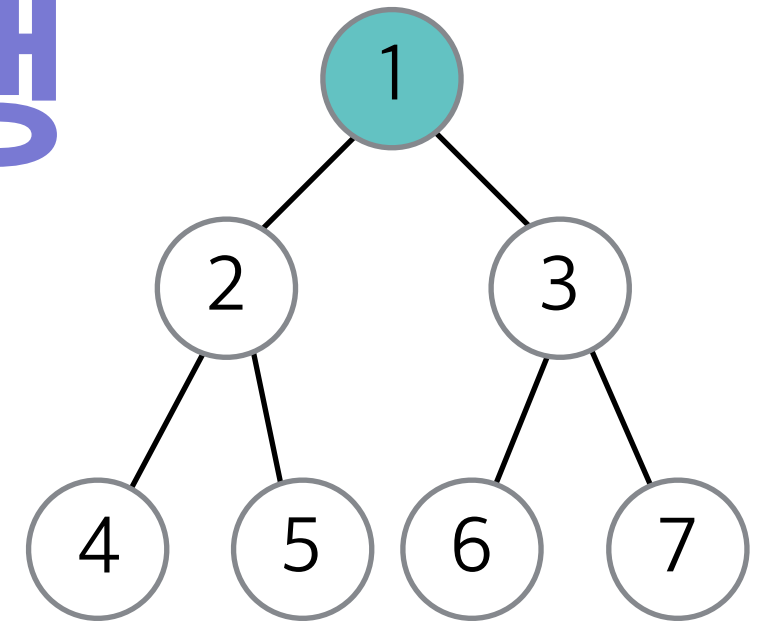
```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

[4, 5, 2, 6, 7, 3, 1]



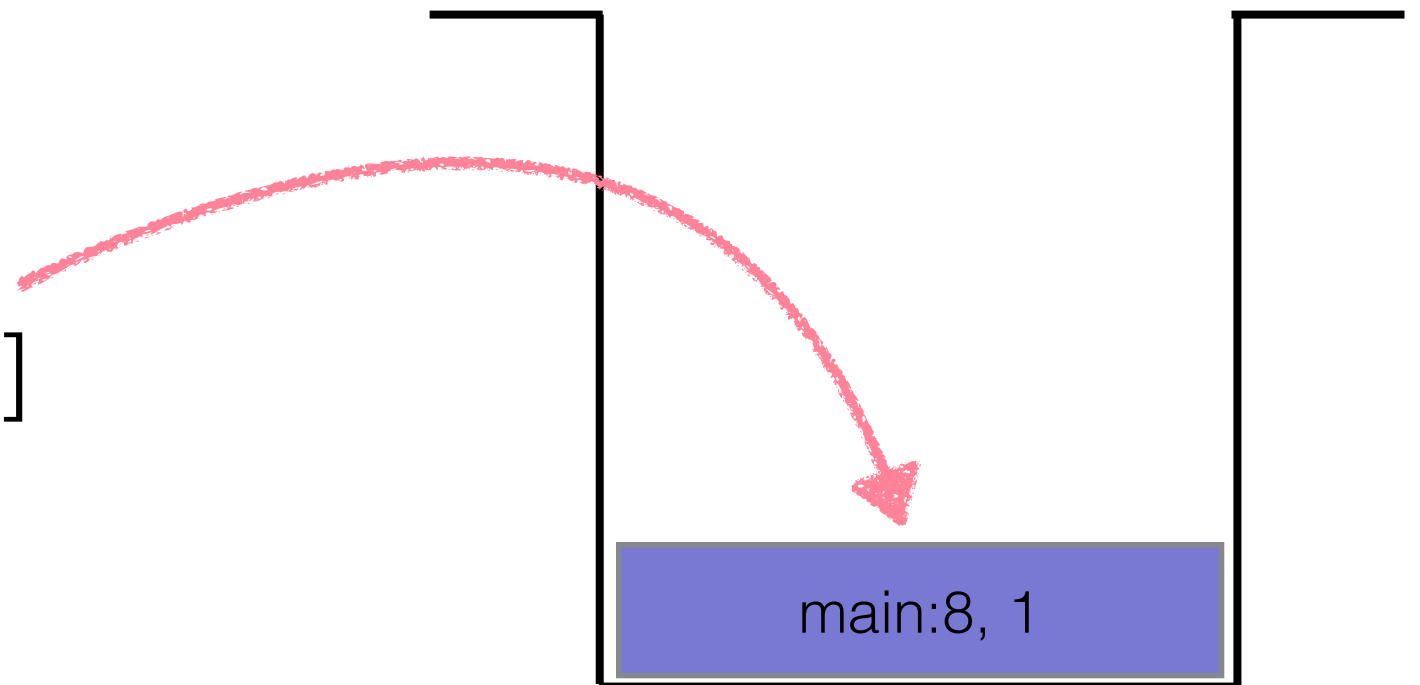
main:8, 1

재귀함수의 실행



```
1 def postorder(tree) :  
2     result = []  
  
3     if tree == None :  
4         return []  
  
5     result = postorder(tree.left)  
6     result = result + postorder(tree.right)  
7     result.append(tree.index)  
→ 8     return result
```

[4, 5, 2, 6, 7, 3, 1]



재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

1. 함수의 정의를 명확히 한다.

재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

1. 함수의 정의를 명확히 한다.

2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.

재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

1. 함수의 정의를 명확히 한다.
2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.
3. 그 후, 함수가 (작은 input에 대하여) 제대로 동작한다고 가정하고 함수를 완성한다.

[예제 3] 거듭제곱 구하기

m^n 을 구하시오

(단, $1 \leq n \leq 1,000,000,000$)

입력의 예

3 4

출력의 예

81

재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

1. 함수의 정의를 명확히 한다.
2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.
3. 그 후, 함수가 (작은 input에 대하여) 제대로 동작한다고 가정하고 함수를 완성한다.

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

1. 함수의 정의를 명확히 한다.

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

1. 함수의 정의를 명확히 한다.

getPower(m, n) : m^n 을 반환하는 함수

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.

$$\text{getPower}(m, 0) = 1$$

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

3. 그 후, 함수가 (작은 input에 대하여) 제대로 동작한다고 가정하고 함수를 완성한다.

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

3. 그 후, 함수가 (작은 input에 대하여) 제대로 동작한다고 가정하고 함수를 완성한다.

$$\text{getPower}(m, n) = m \times \text{getPower}(m, n-1)$$

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

getPower(m, 0) = 1

getPower(m, n) = m x getPower(m, n-1)

정의

점화식

기저조건
(Base condition)

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :  
    if n == 0 :  
        return 1  
    else :  
        return m * getPower(m, n-1)
```

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :  
    if n == 0 :  
        return 1  
    else :  
        return m * getPower(m, n-1)
```

O(n)

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

$$m^n = (m^{(n/2)})^2 \quad n \text{이 짝수일 경우}$$

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

$$m^n = (m^{(n/2)})^2 \quad n \text{이 짝수일 경우}$$

$$(m^{n-1}) \times m \quad n \text{이 홀수일 경우}$$

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

getPower(m, n) =

n이 짝수

getPower(m, n) =

n이 홀수

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

$$\text{getPower}(m, n) = (\text{getPower}(m, n//2))^2 \quad n0 \text{이 짝수}$$

$$\text{getPower}(m, n) = m \times \text{getPower}(m, n-1) \quad n0 \text{이 홀수}$$

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :  
    if n == 0 :  
        return 1  
    elif n % 2 == 0 :  
        temp = getPower(m, n//2)  
        return temp * temp  
    else :  
        return getPower(m, n-1) * m
```

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \dots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :  
    if n == 0 :  
        return 1  
    elif n % 2 == 0 :  
        temp = getPower(m, n//2)  
        return temp * temp  
    else :  
        return getPower(m, n-1) * m
```

$O(\log n)$

[문제 2] 거듭제곱 구하기



```
/* elice */
```

[예제 1] 퀵정렬 구현하기

숫자 n개를 오름차순으로 정렬하시오
(단, $1 \leq n \leq 1,000,000$)

입력의 예

10 2 3 4 5 6 9 7 8 1

출력의 예

1 2 3 4 5 6 7 8 9 10

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

4	7	4	2	10	19	2	4	5	3	1	5
---	---	---	---	----	----	---	---	---	---	---	---

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

4	7	4	2	10	19	2	4	5	3	1	5
---	---	---	---	----	----	---	---	---	---	---	---

pivot

퀵정렬 (Quick Sort)

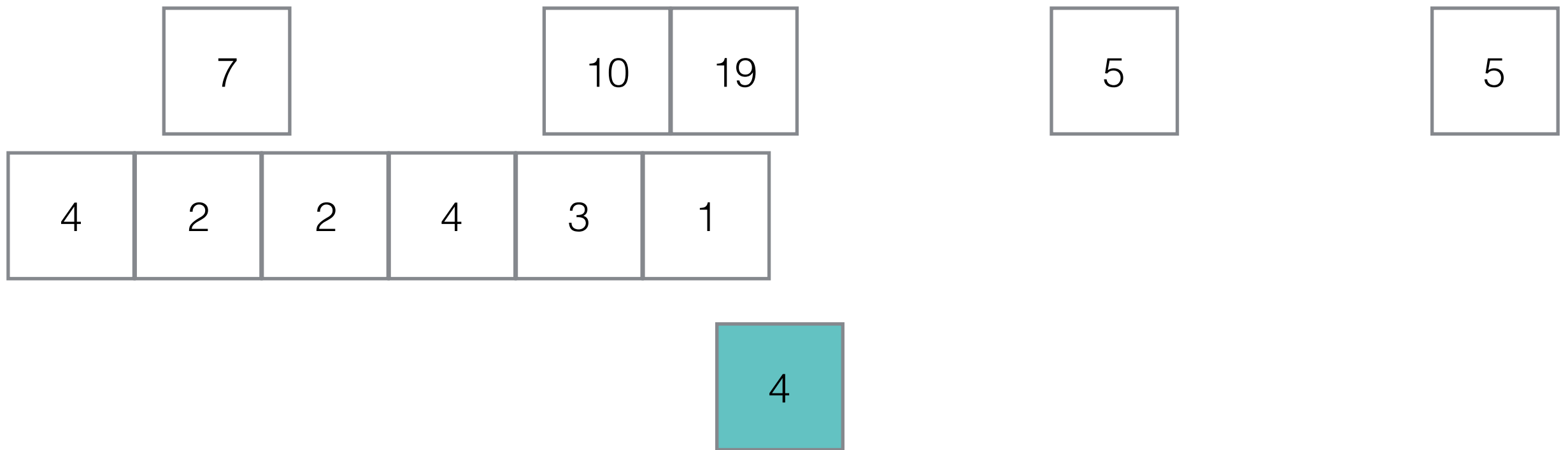
재귀호출을 이용한 대표적인 정렬

7	4	2	10	19	2	4	5	3	1	5
---	---	---	----	----	---	---	---	---	---	---



퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬



퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

4	2	2	4	3	1
---	---	---	---	---	---

7	10	19	5	5
---	----	----	---	---

4

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

4	2	2	4	3	1	4	7	10	19	5	5
---	---	---	---	---	---	---	---	----	----	---	---

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

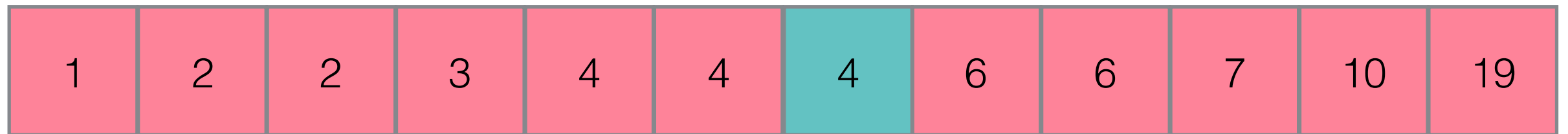
4	2	2	4	3	1	4	7	10	19	5	5
---	---	---	---	---	---	---	---	----	----	---	---

Quicksort!

Quicksort!

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬



Quicksort!

Quicksort!

퀵정렬 (Quick Sort)

재귀호출을 이용한 대표적인 정렬

1	2	2	3	4	4	4	6	6	7	10	19
---	---	---	---	---	---	---	---	---	---	----	----

Quicksort!

Quicksort!

퀵정렬의 자세한 단계

4	7	4	2	10	19	2	4	5	3	1	5
---	---	---	---	----	----	---	---	---	---	---	---

퀵정렬의 자세한 단계

4	7	4	2	10	19	2	4	5	3	1	5
---	---	---	---	----	----	---	---	---	---	---	---

pivot

퀵정렬의 자세한 단계

4	2	2	4	3	1	4	7	10	19	5	5
---	---	---	---	---	---	---	---	----	----	---	---

퀵정렬의 자세한 단계

						4	7	10	19	5	5
4	2	2	4	3	1						

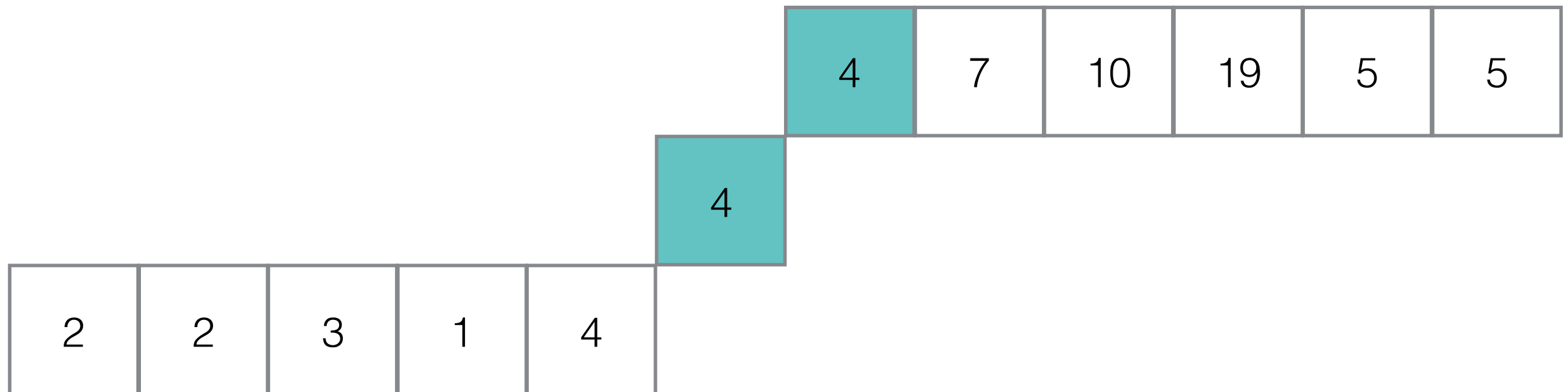
퀵정렬의 자세한 단계

						4	7	10	19	5	5
4	2	2	4	3	1						

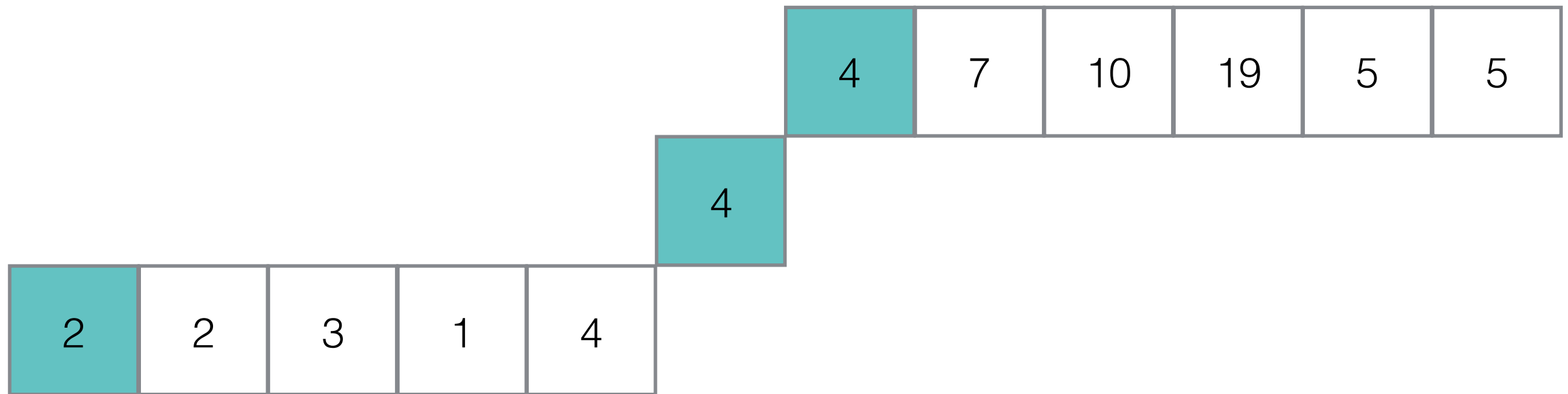
퀵정렬의 자세한 단계

						4	7	10	19	5	5
2	2	3	1	4	4						

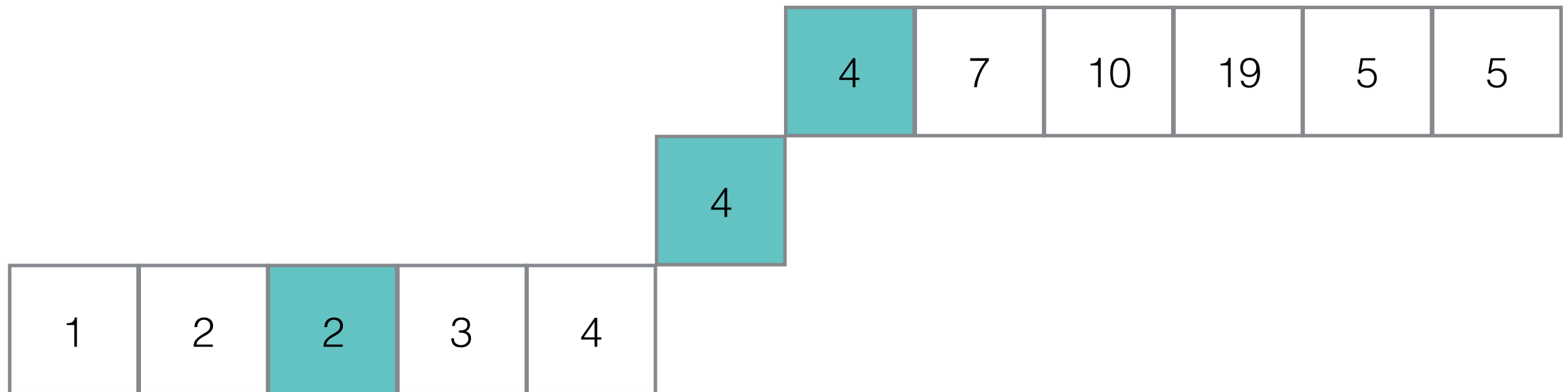
퀵정렬의 자세한 단계



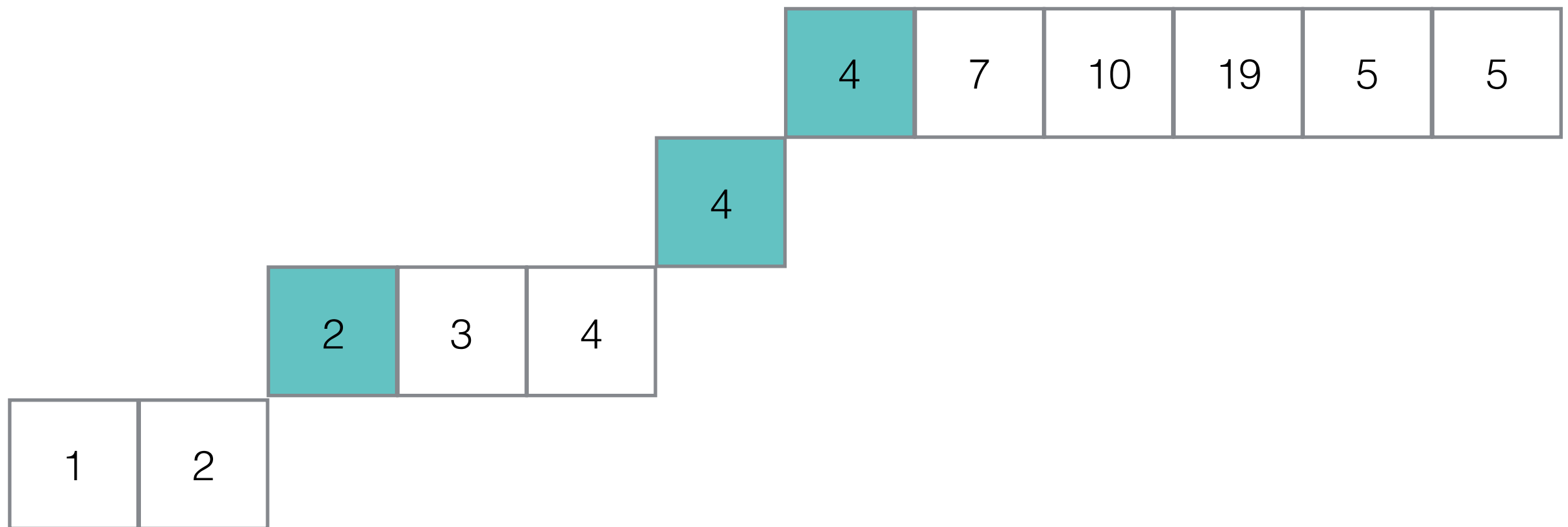
퀵정렬의 자세한 단계



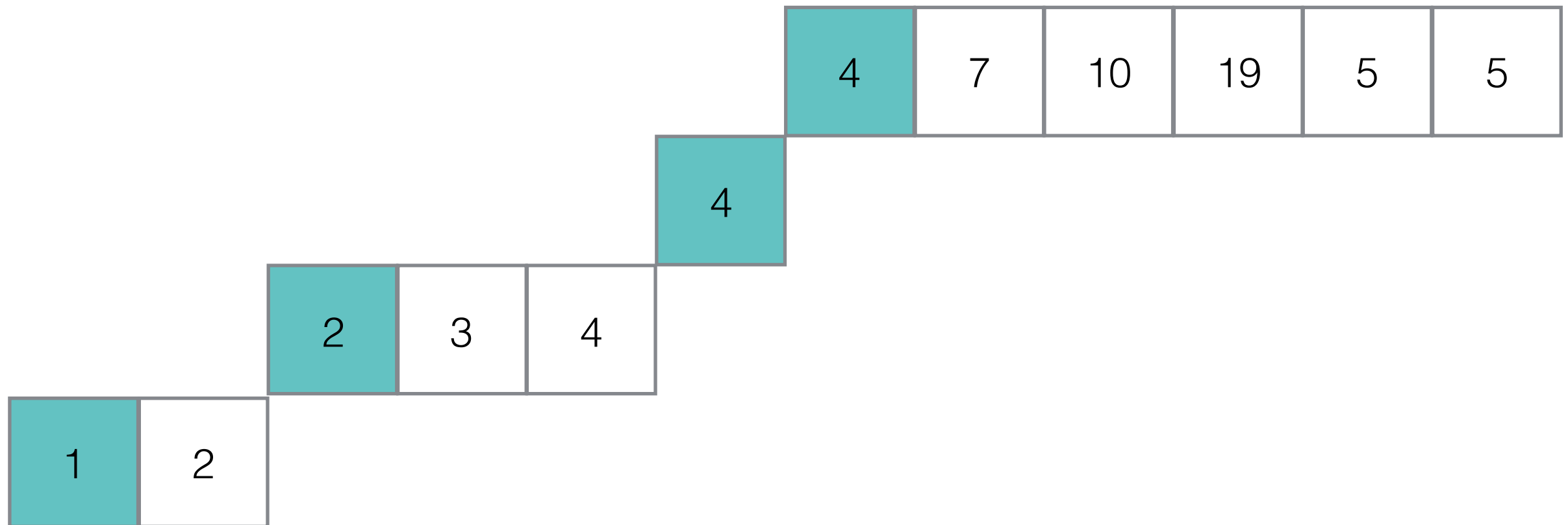
퀵정렬의 자세한 단계



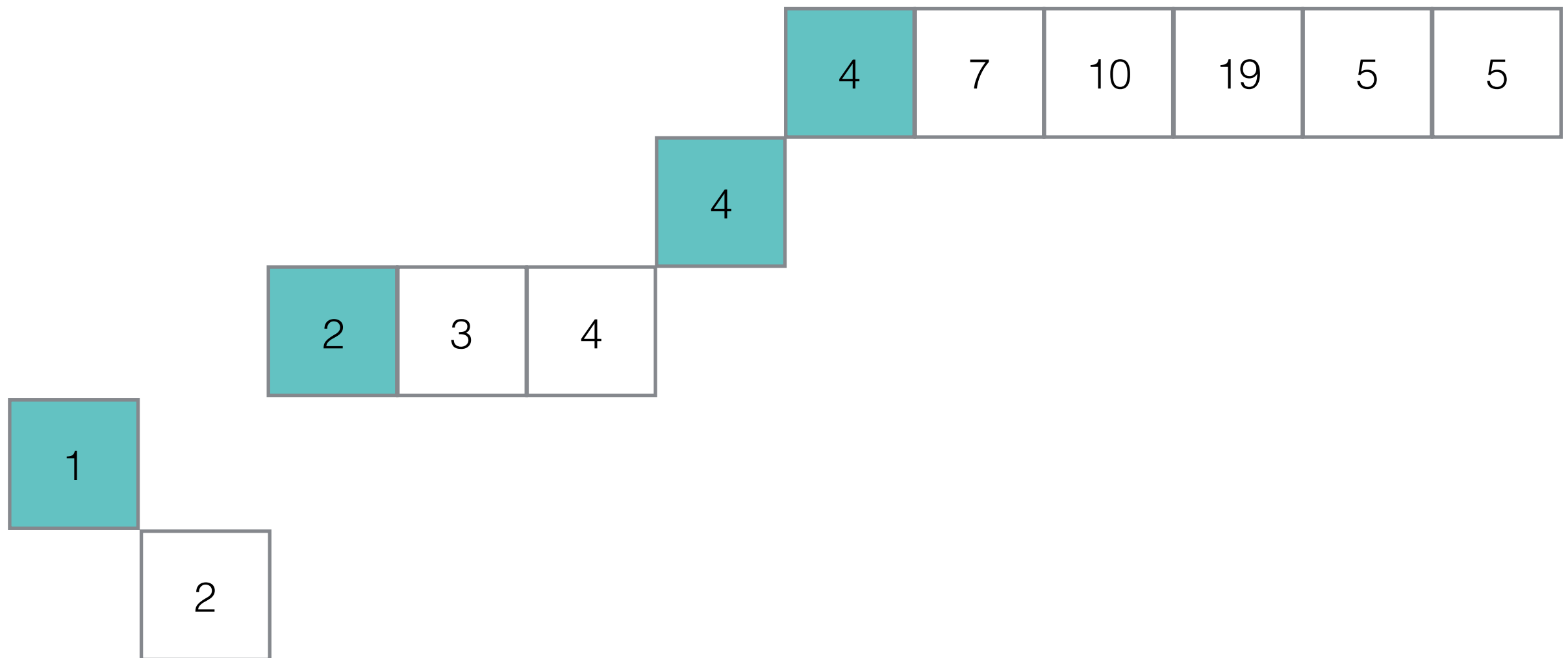
퀵정렬의 자세한 단계



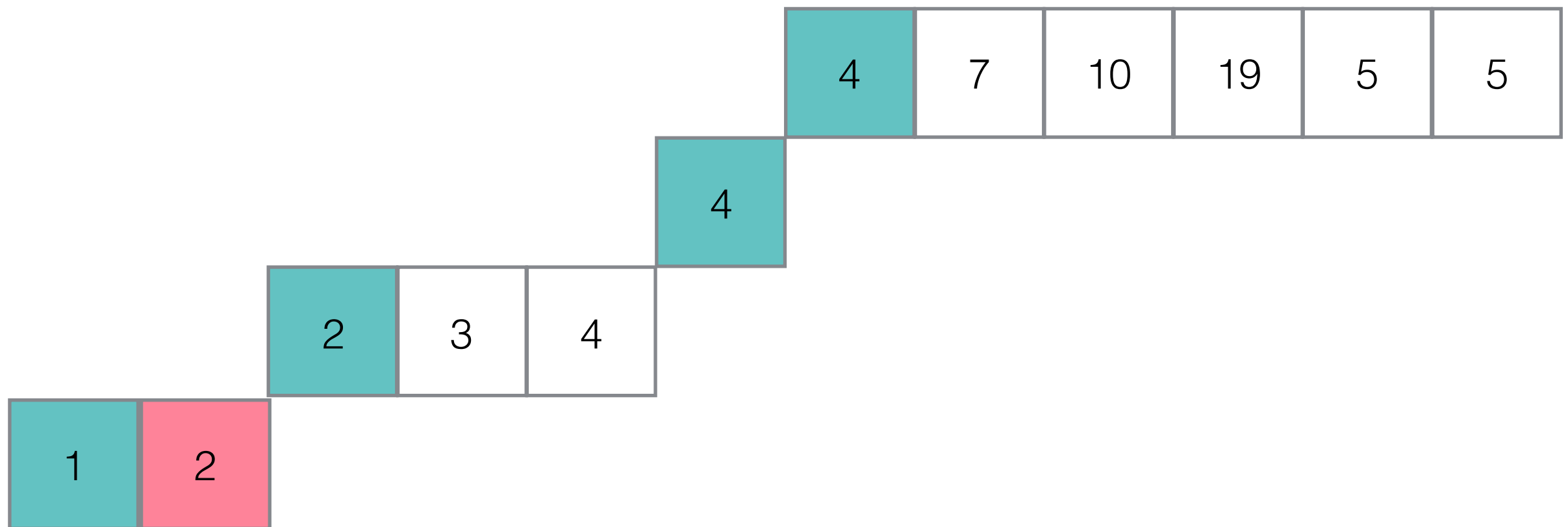
퀵정렬의 자세한 단계



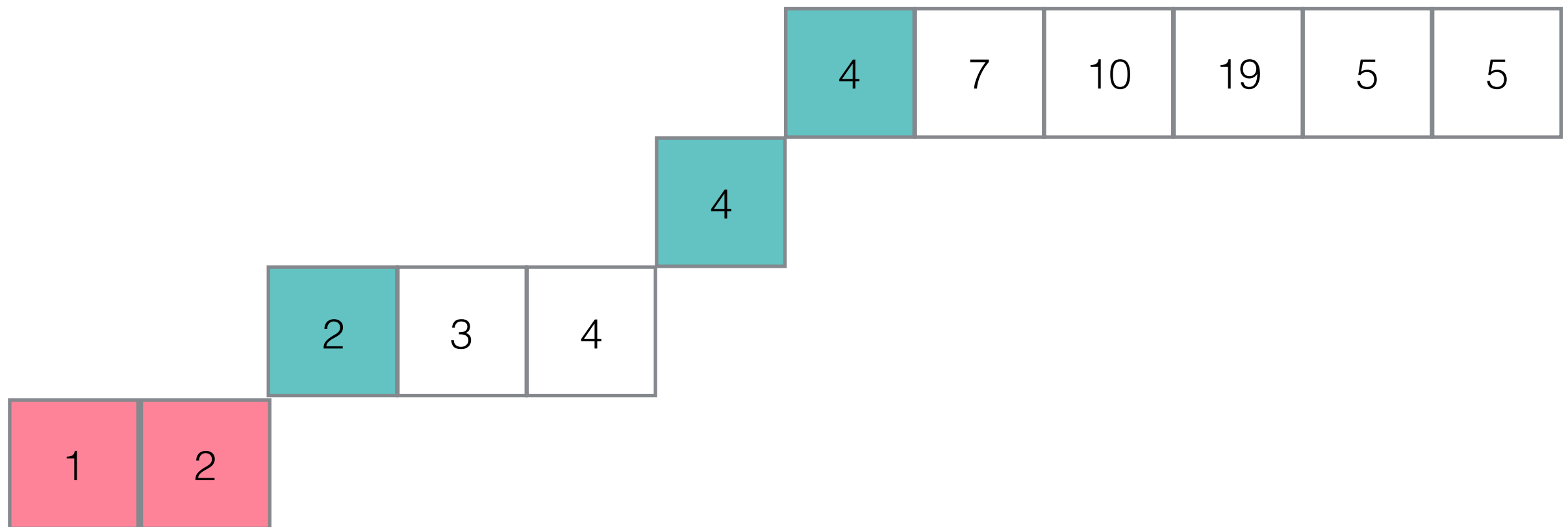
퀵정렬의 자세한 단계



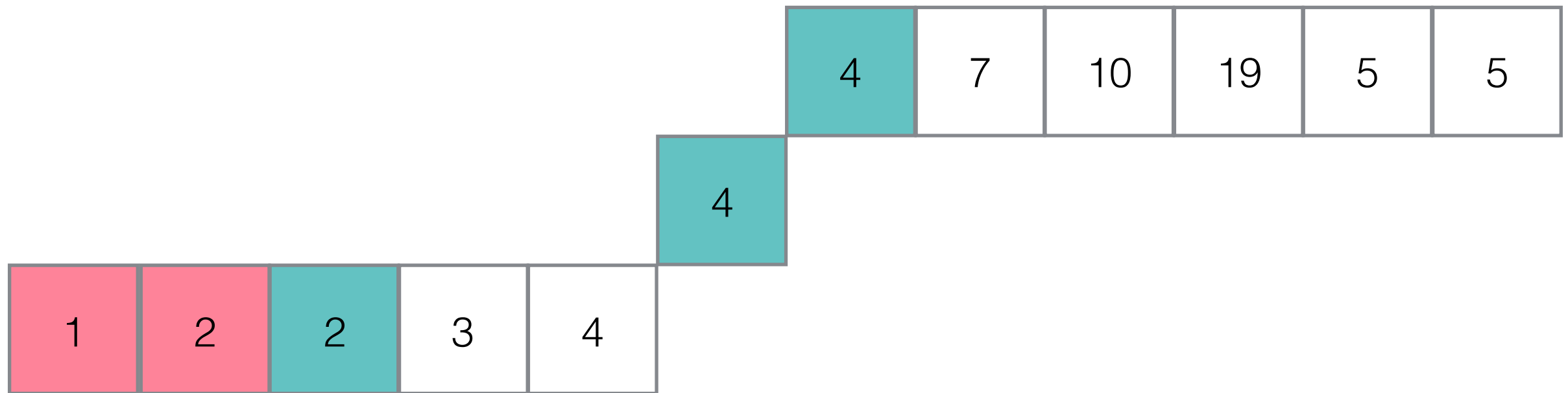
퀵정렬의 자세한 단계

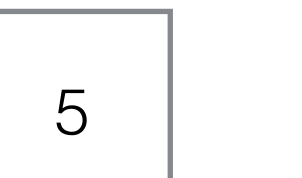


퀵정렬의 자세한 단계

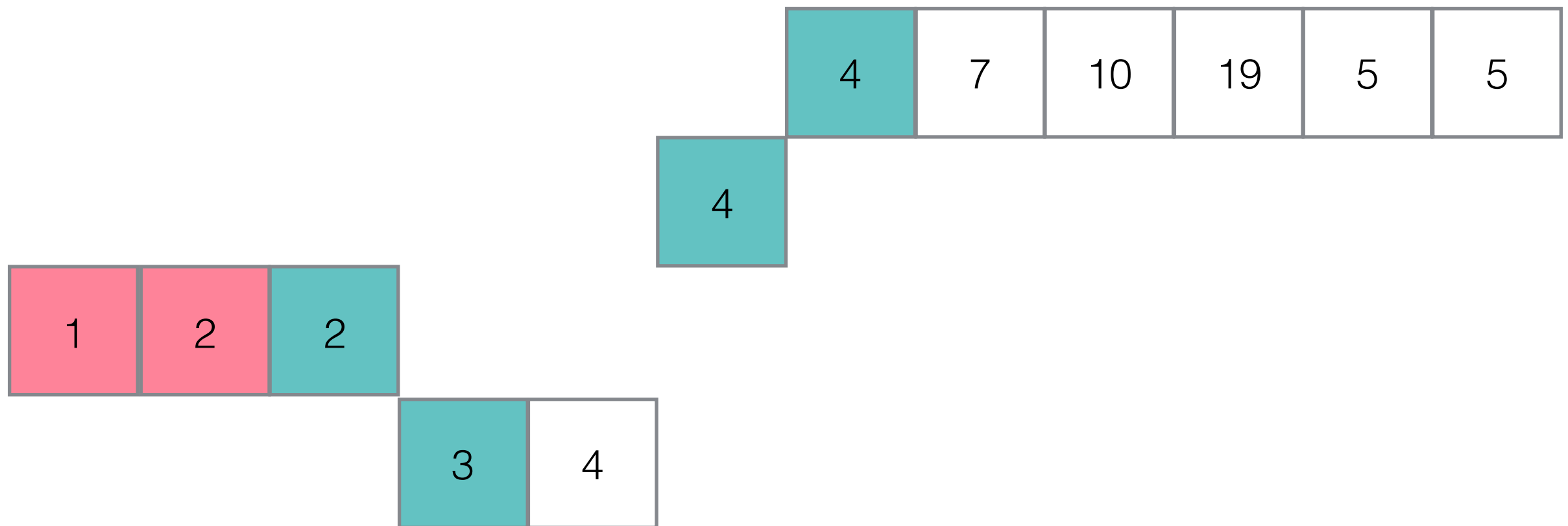


퀵정렬의 자세한 단계

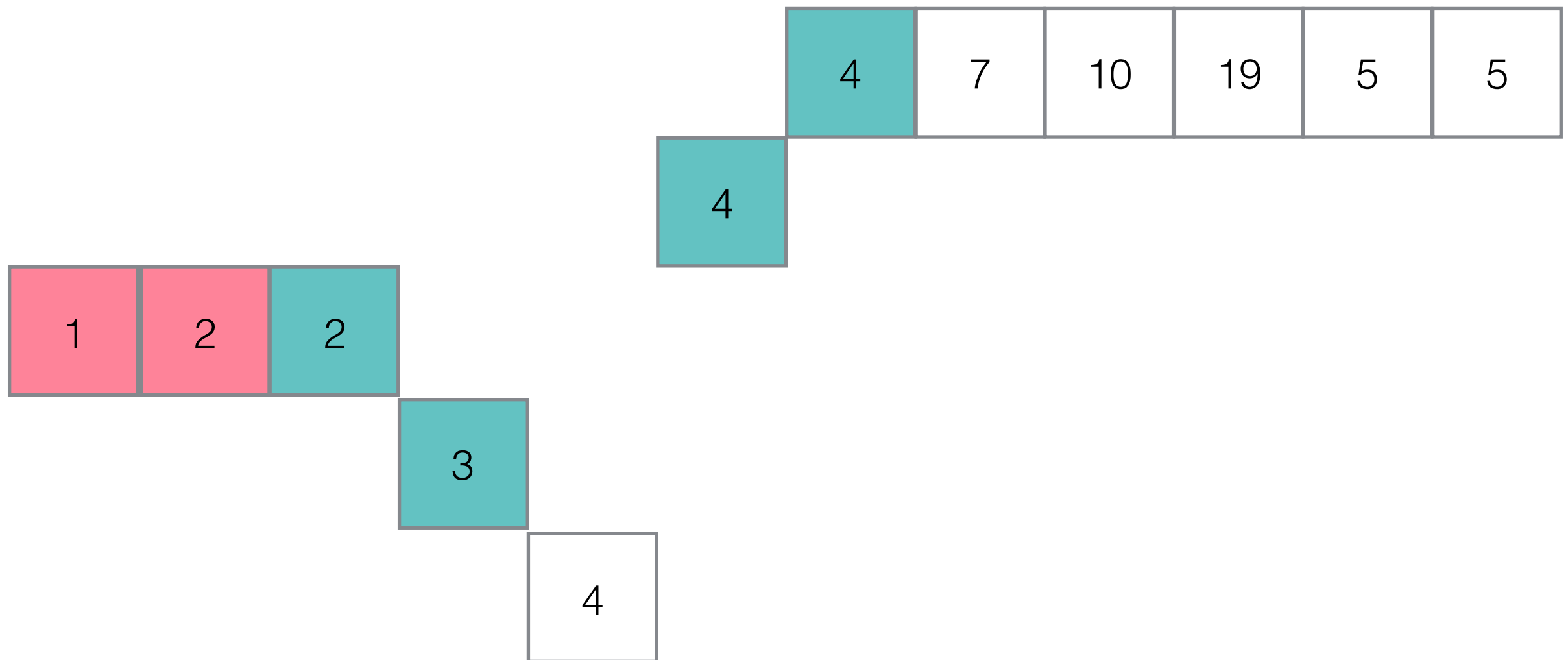




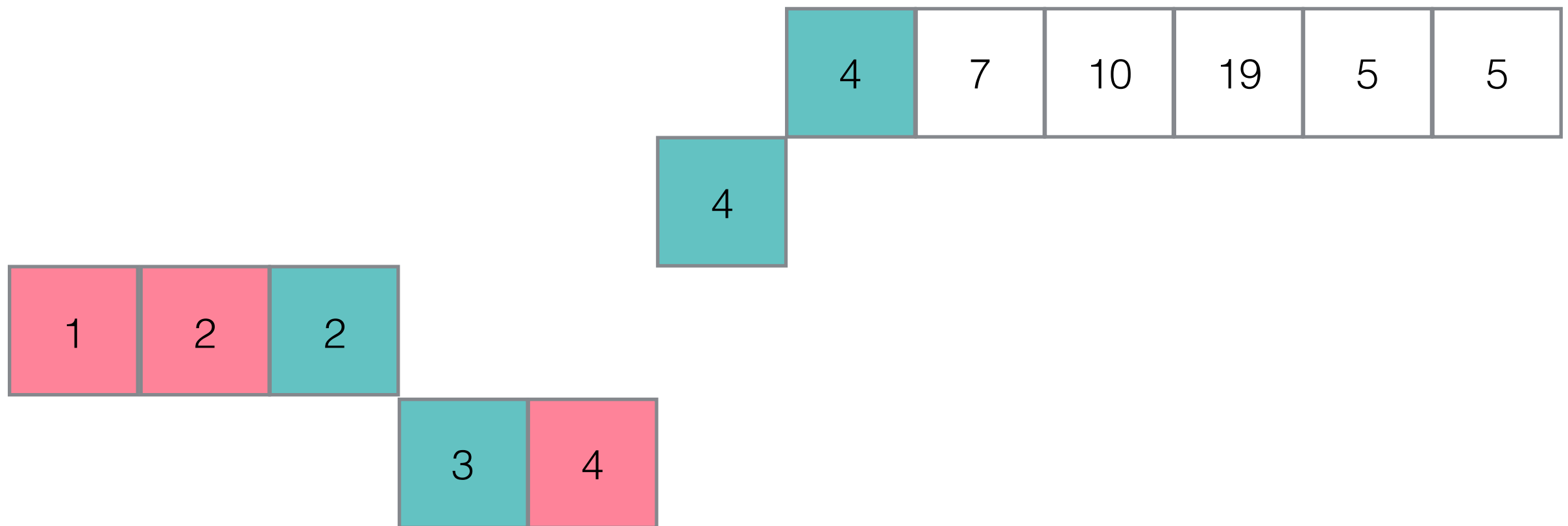
퀵정렬의 자세한 단계



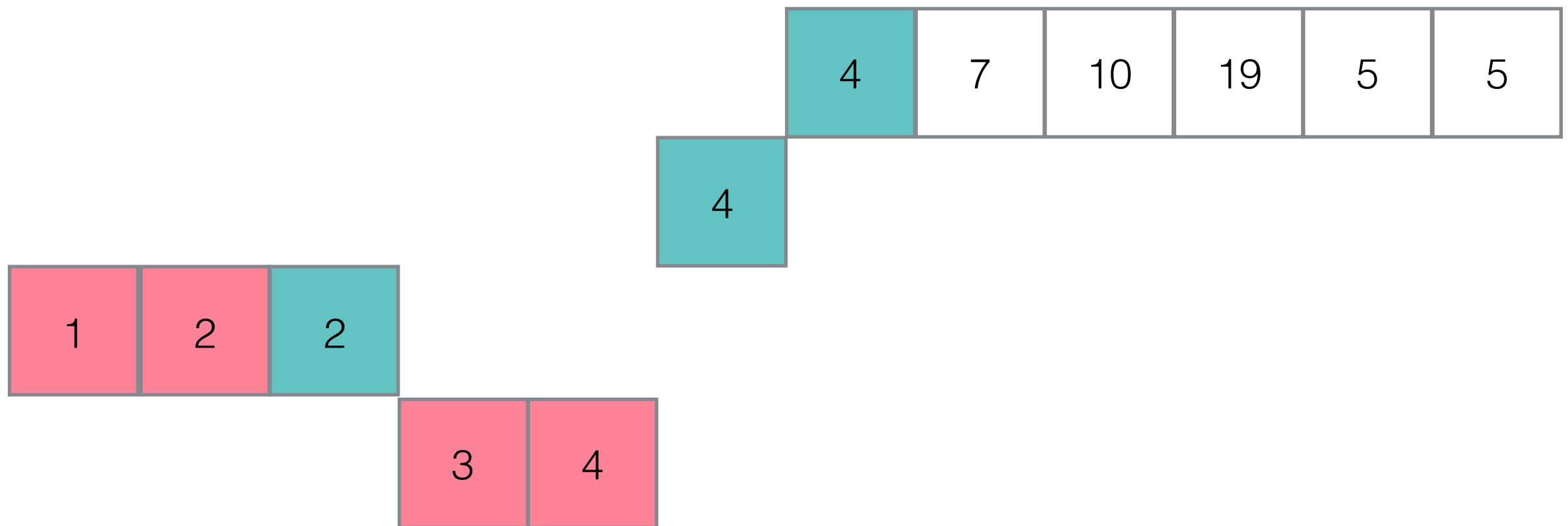
퀵정렬의 자세한 단계



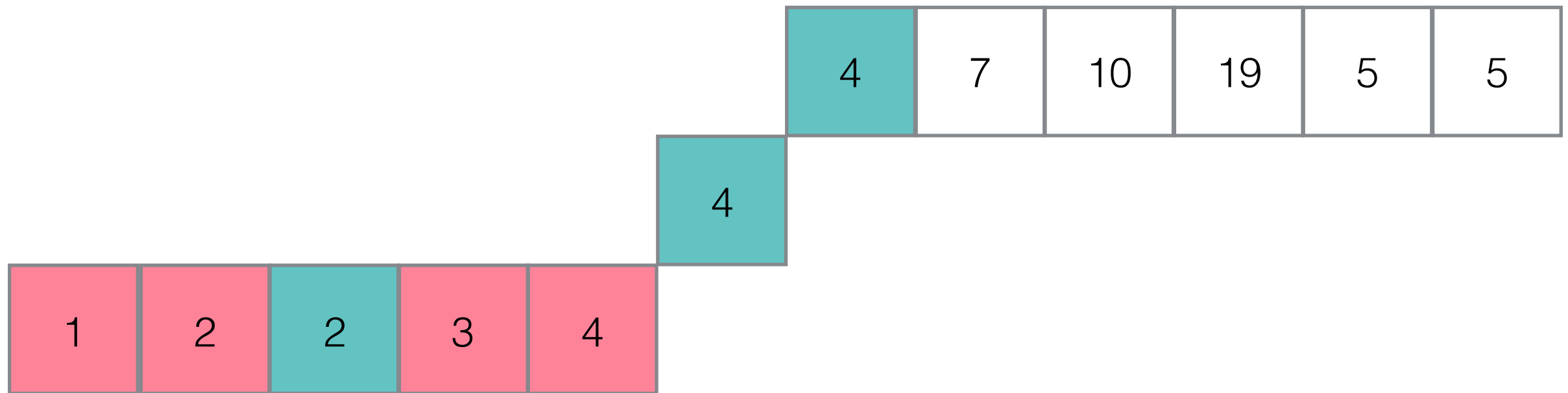
퀵정렬의 자세한 단계



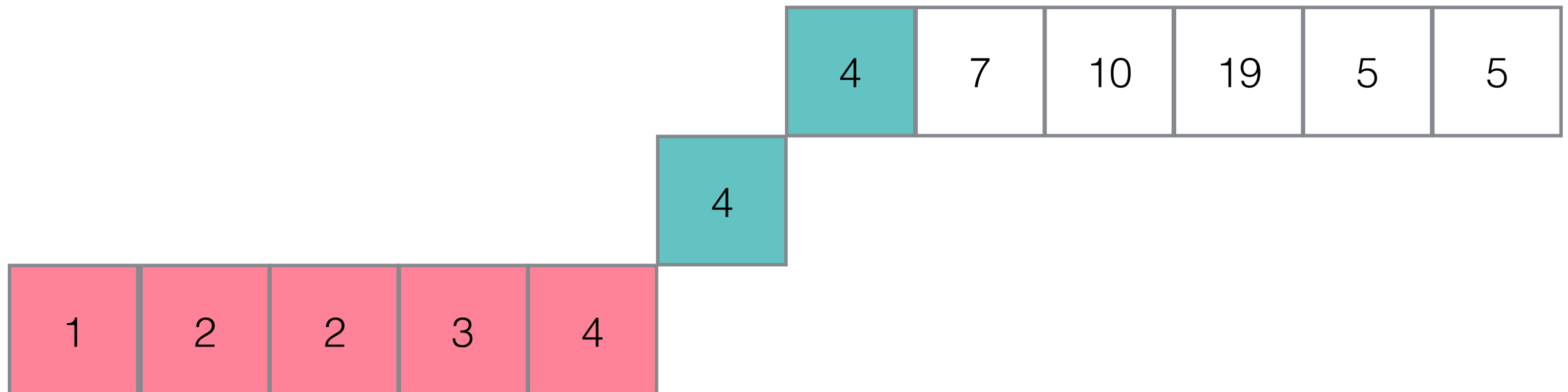
퀵정렬의 자세한 단계



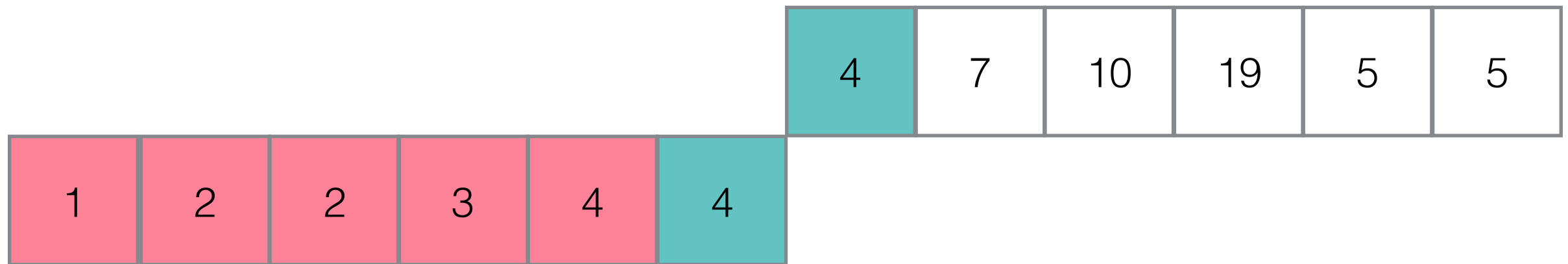
퀵정렬의 자세한 단계



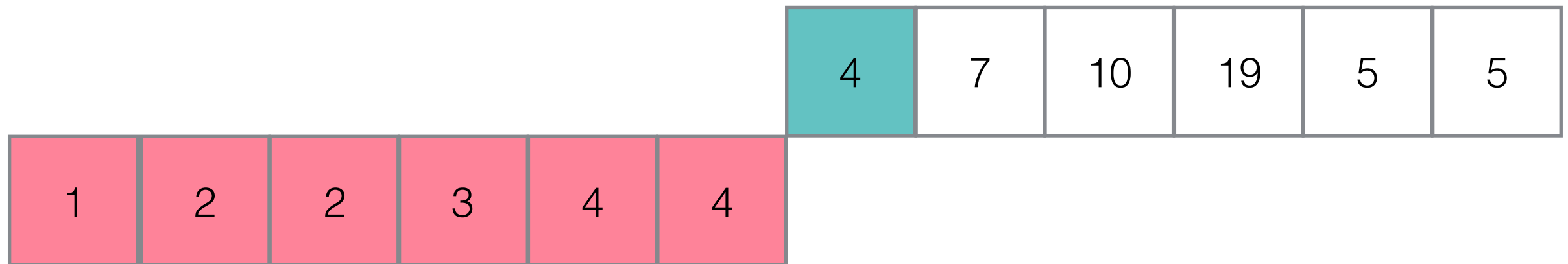
퀵정렬의 자세한 단계



퀵정렬의 자세한 단계



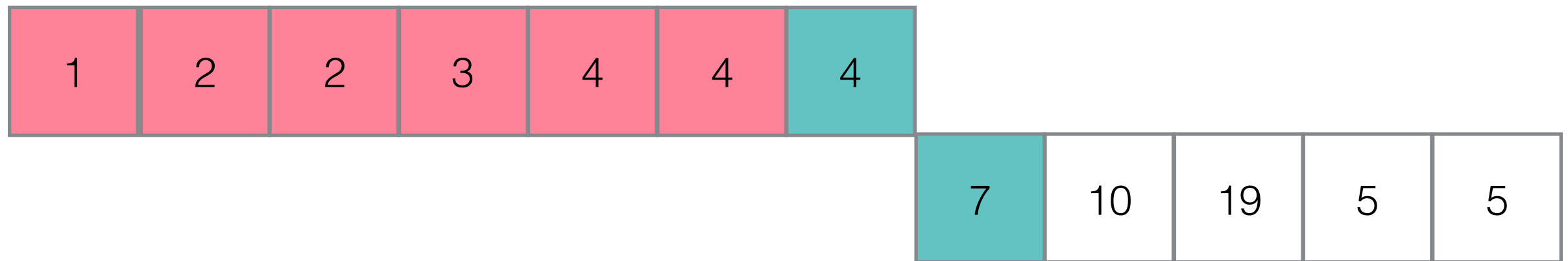
퀵정렬의 자세한 단계



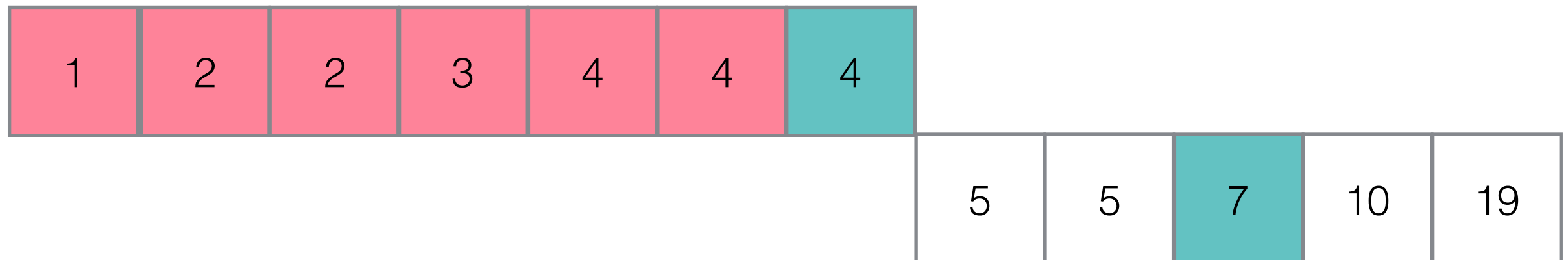
퀵정렬의 자세한 단계

1	2	2	3	4	4	4	7	10	19	5	5
---	---	---	---	---	---	---	---	----	----	---	---

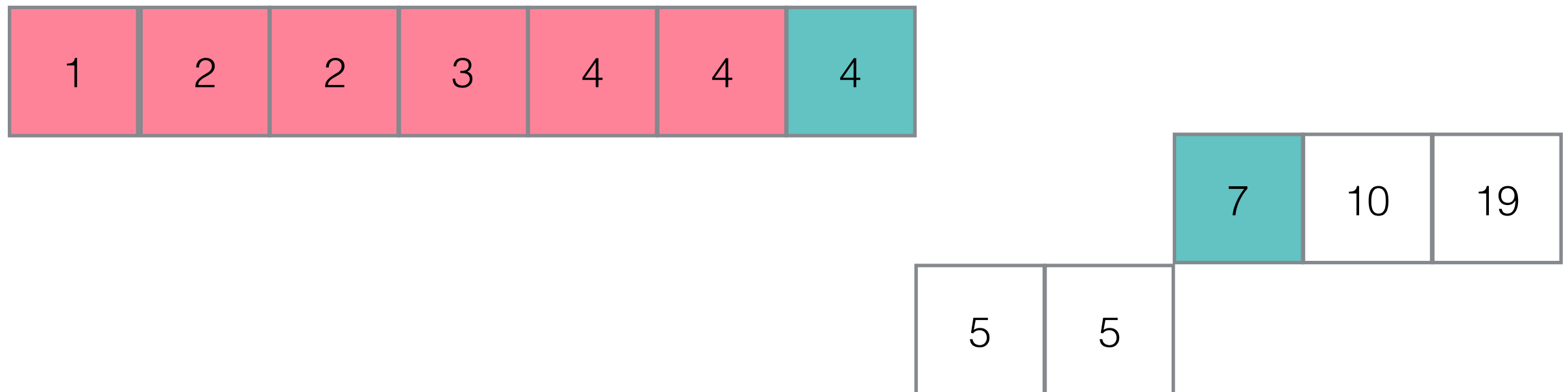
퀵정렬의 자세한 단계



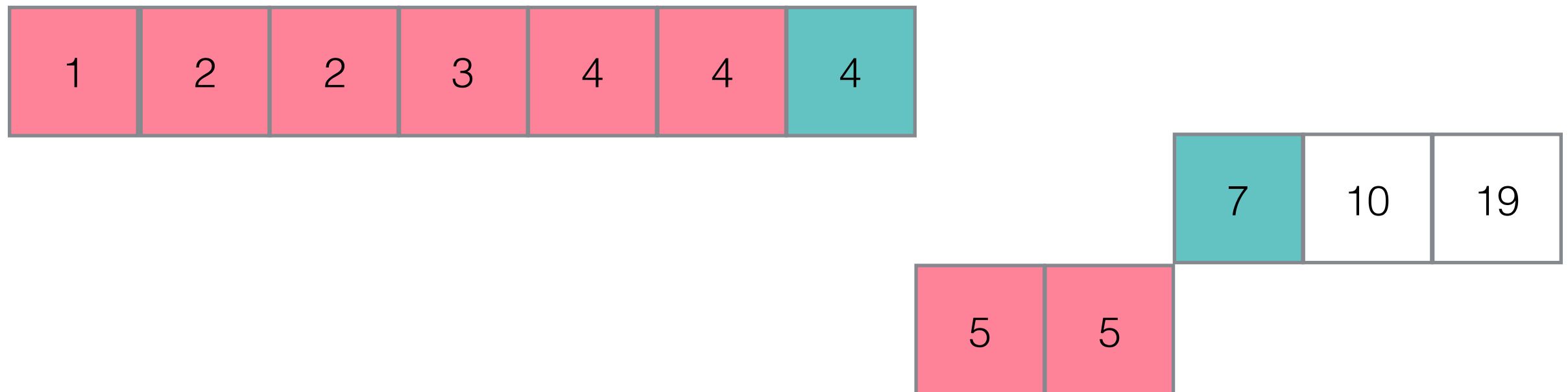
퀵정렬의 자세한 단계



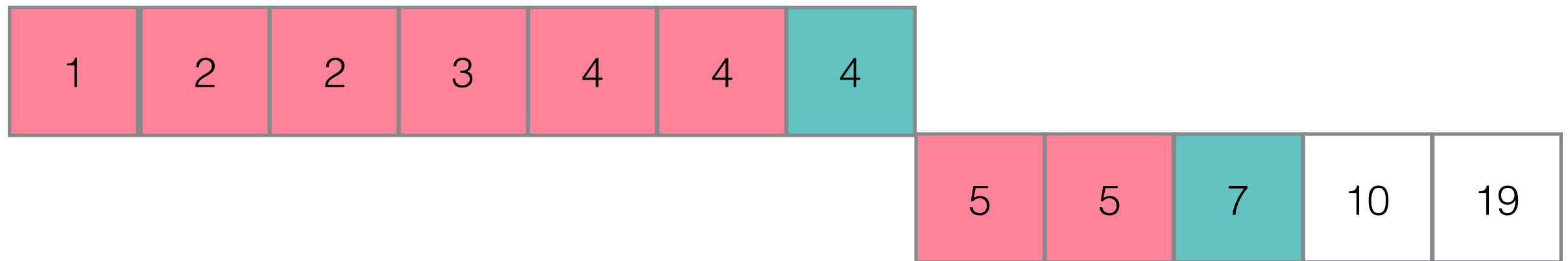
퀵정렬의 자세한 단계



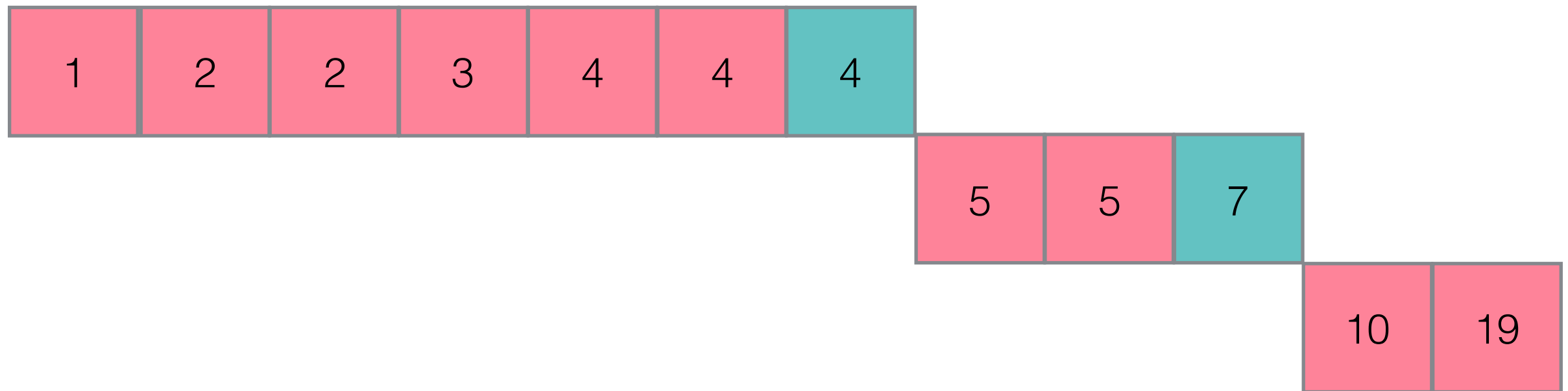
퀵정렬의 자세한 단계



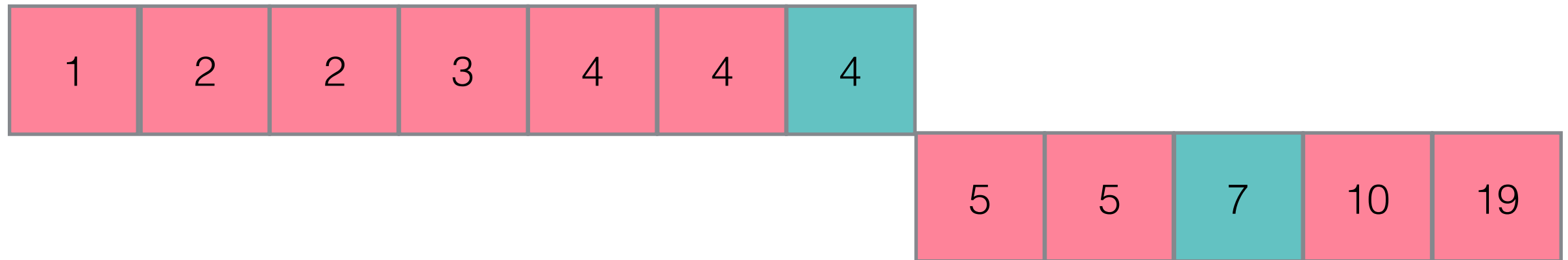
퀵정렬의 자세한 단계



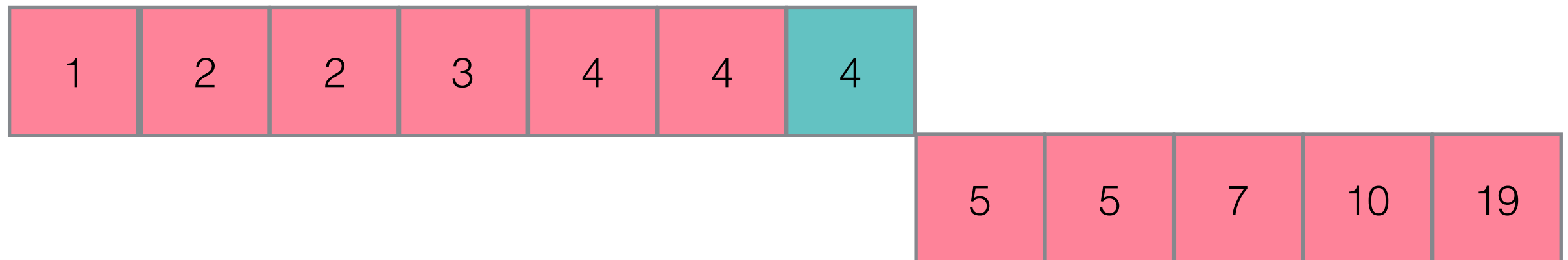
퀵정렬의 자세한 단계



퀵정렬의 자세한 단계



퀵정렬의 자세한 단계



퀵정렬의 자세한 단계

1	2	2	3	4	4	4	5	5	7	10	19
---	---	---	---	---	---	---	---	---	---	----	----

퀵정렬의 자세한 단계

1	2	2	3	4	4	4	5	5	7	10	19
---	---	---	---	---	---	---	---	---	---	----	----

퀵정렬의 자세한 단계

1	2	2	3	4	4	4	5	5	7	10	19
---	---	---	---	---	---	---	---	---	---	----	----

[예제 1] 쿼정렬 구현하기



```
/* elice */
```

요약

의미단위로 작성된 코드가 좋은 코드이다

→ 코드를 이해한다 = 각 함수가 무슨 일을 하는지 설명할 수 있다

트리는 코드가 실행되고 있는 상태를 나타내는 자료구조이다

→ 물론, 코드를 의미 단위로 나타냈을 때 파악이 가능한 사실이다

코드를 하나하나 따라가는 것은 컴퓨터가 해야 할 일이다

→ 우리는 앞으로 코드가 하는 일, 더 나아가 코드의 의미에 집중한다

퀴즈 및 강의평가



`/* elice */`

감사합니다!

신현규

E-mail : hyungyu.sh@kaist.ac.kr

Kakao : yougatup

/* elice */

문의 및 연락처

academy.elice.io

contact@elice.io

facebook.com/elice.io

blog.naver.com/elicer