Recursion

Recursion

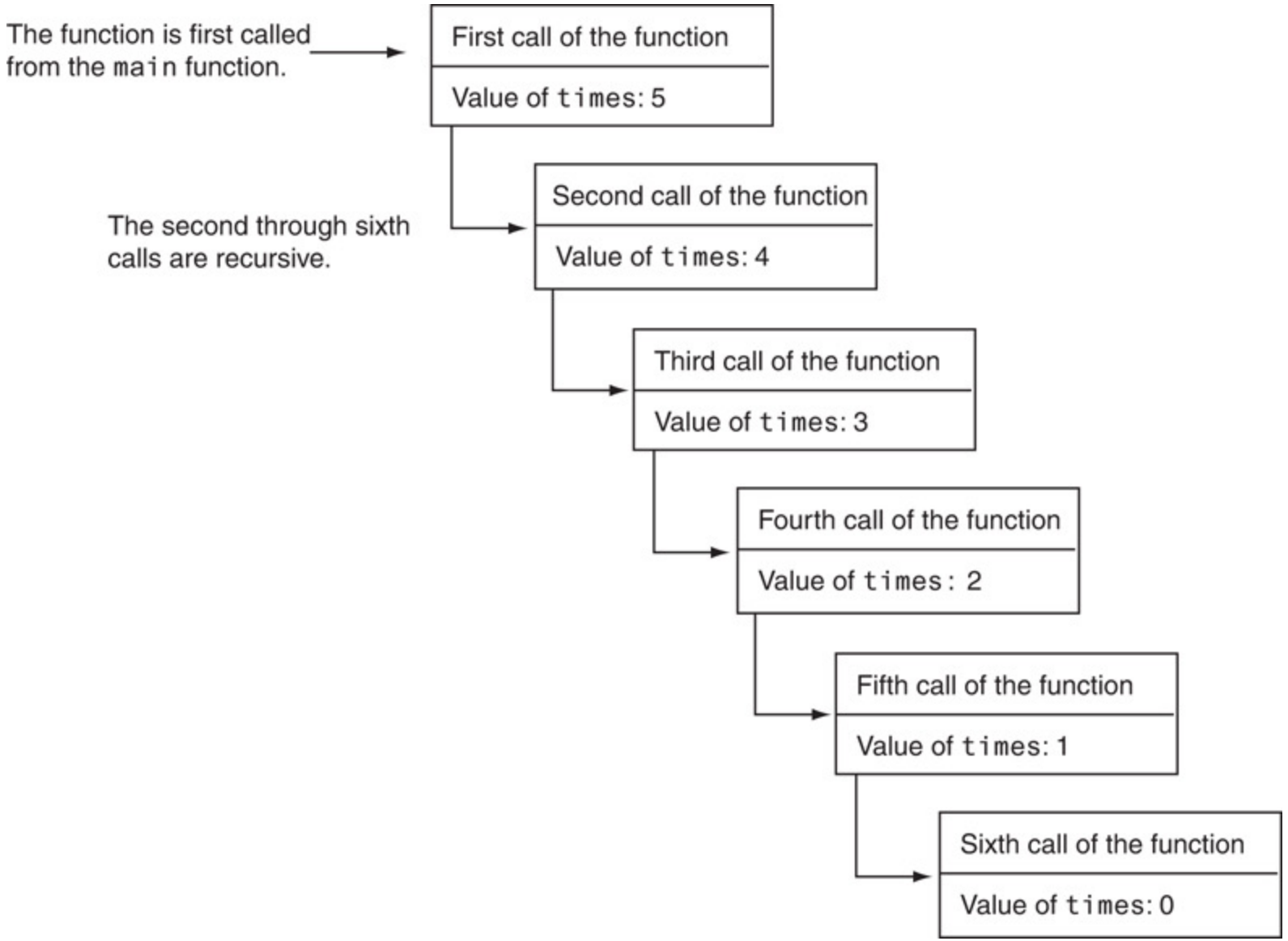
- 재귀 함수(Recursive Function)는 함수가 자기 자신을 호출하여 문제를 해결하는 프로그래밍 기법
- 반복문처럼 동일한 작업을 수행하는 코드를 간결하게 작성
- 일부 문제를 해결하기 위해 더 작은 부분 문제로 분할하고, 이들을 해결하여 전체 문제를 해결
- 함수가 자신을 호출하면서 호출 횟수를 제한하는 조건문 필요
- 호출 횟수 제한하는 조건문 없을 시 무한히 재귀 호출 발생

Recursion

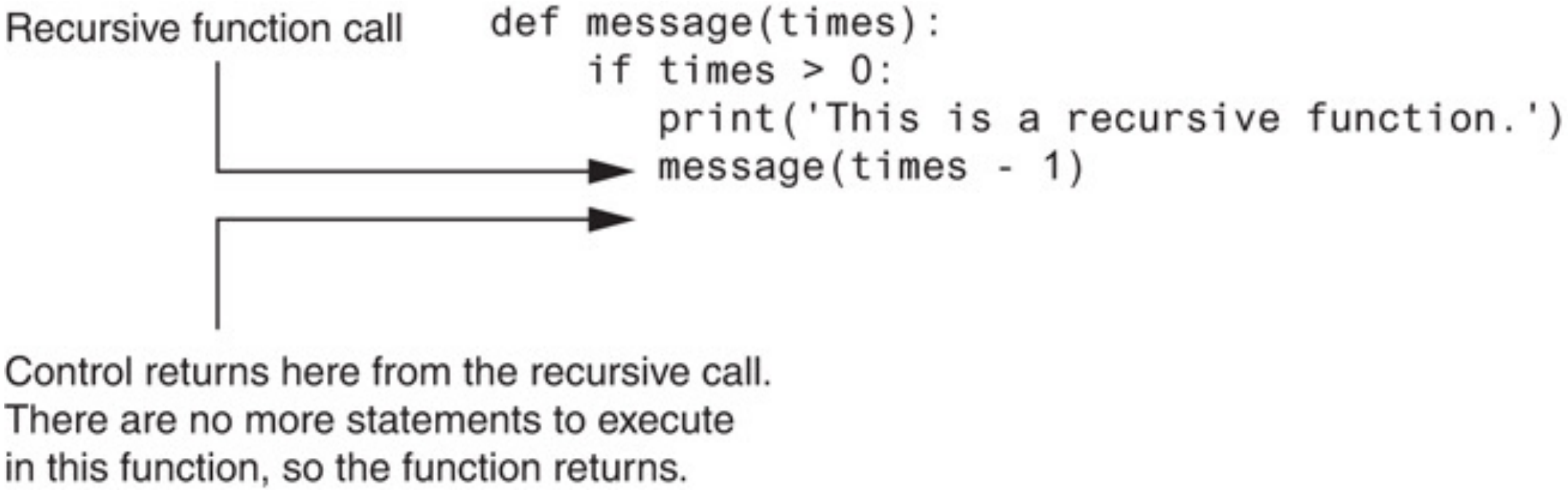
- 장점
- 코드의 간결성: 재귀 함수를 이용하면 반복문을 사용하는 것보다 간결한 코드를 작성 가능
- 문제 해결의 용이성: 몇몇 문제에서는 재귀 함수를 사용하는 것이 문제를 해결하기 용이한 경우 있음
  - 트리 구조나 그래프를 탐색하는 문제 등
- 단점
- 메모리 사용량 증가: 재귀 함수는 자기 자신을 호출하는 과정에서 스택 메모리를 사용하게 되므로, 호출 횟수가 많을 경우 메모리 사용량 증가
- 실행 속도 저하: 재귀 함수는 반복문보다 함수 호출과 반환에 대한 오버헤드가 크므로, 실행 속도가 느려질 수 있음

- 재귀 함수를 사용할 때에는 해결할 문제의 특성 및 메모리 사용량과 실행 속도에 대한 고려가 필요

Example(message)



Example(message)



Example(message)

```
def message(times):
      print("times : ", times)
      if times > 0 :
            print("This is a recursive function.")
            message(times - 1)


num = 5
message(num)
```

```
times :  5
This is a recursive function.
times :  4
This is a recursive function.
times :  3
This is a recursive function.
times :  2
This is a recursive function.
times :  1
This is a recursive function.
times :  0
```

Example(recursive_sum)

1부터 n까지의 합을 구하는 함수를 재귀적으로 구현

```
def recursive_sum(n):
    print("n : ", n)
    if n == 1:
        return 1
    else:
        return n + recursive_sum(n-1)

num=5
print(recursive_sum(num))
```

```
n :  5
n :  4
n :  3
n :  2
n :  1
15
```

Example(recursive_sum)

- recursive_sum은 매개변수 n이 1인 경우에는 1을 반환하고, 그렇지 않은 경우에는 n과 recursive_sum(n-1)의 합을 반환
- recursive_sum(n-1)은 n보다 작은 정수들의 합을 구하는 재귀적인 호출을 수행

- n이 5인 경우에는 다음과 같이 동작
  - recursive_sum(5)를 호출
  - n이 1이 아니므로, n + recursive_sum(n-1)을 반환
  - recursive_sum(4)를 호출
  - ...
  - recursive_sum(1)을 호출
  - n이 1이므로, 1을 반환

  - recursive_sum(2)에서는 2 + recursive_sum(1) = 3을 반환
  - recursive_sum(3)에서는 3 + recursive_sum(2) = 6을 반환
  - recursive_sum(4)에서는 4 + recursive_sum(3) = 10을 반환
  - recursive_sum(5)에서는 5 + recursive_sum(4) = 15를 반환

Example(Factorial)

- Factorial: 어떤 양의 정수 n에 대해서, 1부터 n까지의 모든 자연수를 곱한 값

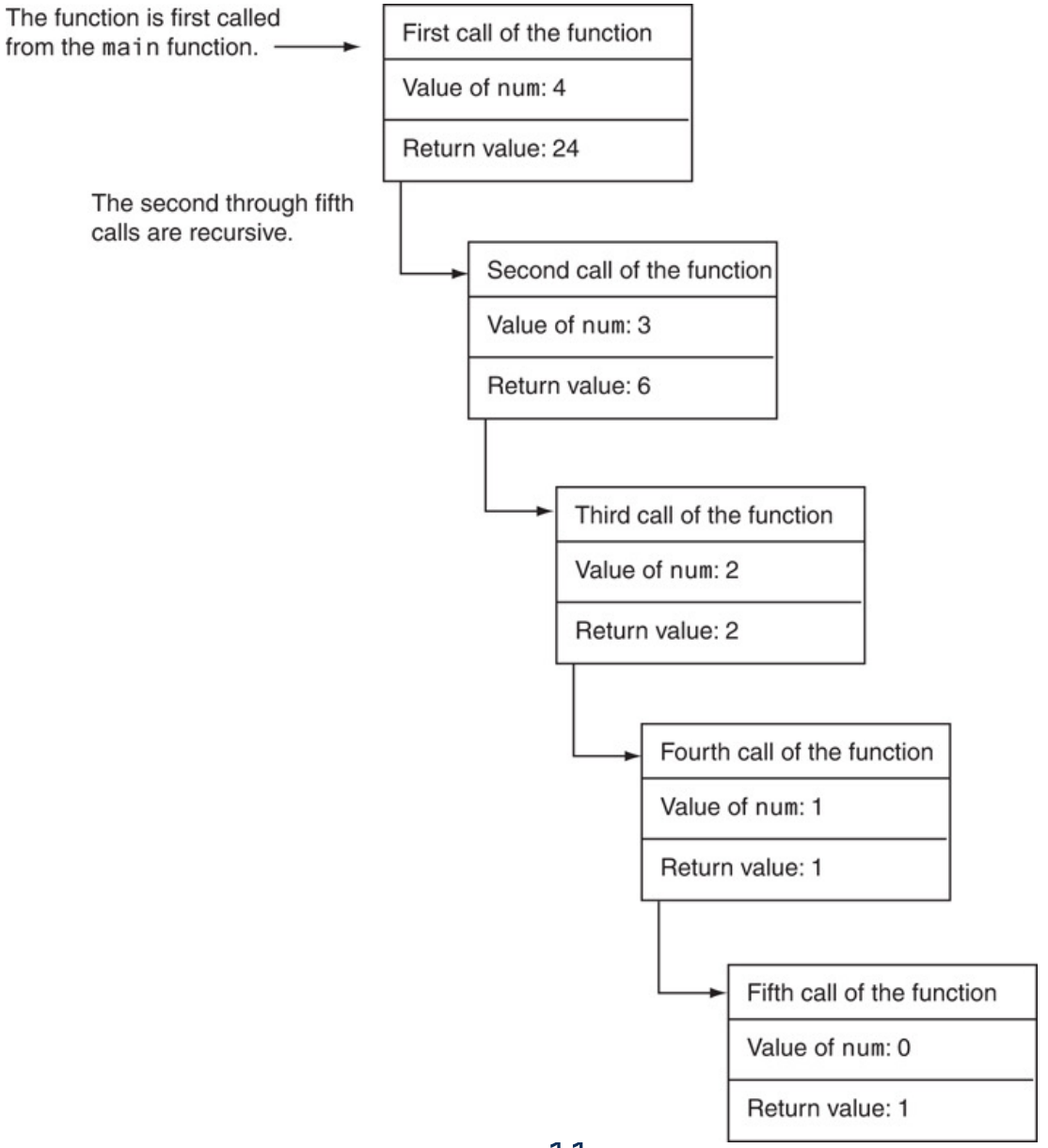
- n!으로 표현
- n! = n x (n-1) x (n-2) x ... x 2 x 1
- 4!는 4 x 3 x 2 x 1 = 24

Example(Factorial - recursion)

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

n = 4
result = factorial(n)
print(f"{n}! = {result}")
```

```
4! = 24
```

Example(Factorial - recursion)



Example(Factorial - recursion)

- 입력으로 받은 n이 1일 때까지 자신을 호출하여 n부터 1까지의 자연수를 차례대로 곱하여 팩토리얼을 구함

- factorial(5)를 호출하면 함수는 다음과 같은 순서로 실행

  1) factorial(4) 호출
  2) 4 * factorial(3) 호출
  3) 4 * 3 * factorial(2) 호출
  4) 4 * 3 * 2 * factorial(1) 호출
  5) 4 * 3 * 2 * 1 = 24 반환

- 재귀 함수를 이용한 팩토리얼 구현 방법은 코드가 간결하고 이해하기 쉽다는 장점이 있으나 함수를 반복 호출하면서 스택에 저장되는 메모리 공간이 증가하므로, n이 커질수록 재귀 함수의 호출 횟수와 메모리 사용량이 증가하여 실행 시간이 길어지는 단점이 있음

Example(Factorial - for)

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

n = 5
result = factorial(n)
print(f"{n}! = {result}")
```

```
5! = 120
```

Example(Factorial - for)

- 1부터 n까지의 자연수를 차례대로 곱하는 방식으로 팩토리얼을 구함
- 곱셈 연산을 반복문 안에서 수행하므로, n이 커질수록 계산 시간이 증가
- 5!을 구하는 경우 for 루프에서 i가 1부터 5까지 차례대로 변하며, result 변수는 1부터 5까지 차례대로 곱해져 최종적으로 120이 반환
- 파이썬의 반복문은 내부적으로 C 언어로 작성되어 있으므로, 재귀 함수보다 실행 속도가 빠르고 메모리 사용량도 적음
- 큰 수의 팩토리얼을 계산하는 경우에는 반복문을 이용한 방법을 사용하는 것이 좋음

Example(Fibonacci numbers)

- Fibonacci: 각 숫자가 이전 두 수의 합인 수열
- 0과 1로 시작하고, 이전의 두 수를 더해 나가는 수열
- (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...)과 같이 이어지는 수열

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) (n > 1)

Example(Fibonacci - recursion)

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

for i in range(10):
    print(fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
```

Example(Fibonacci - for)

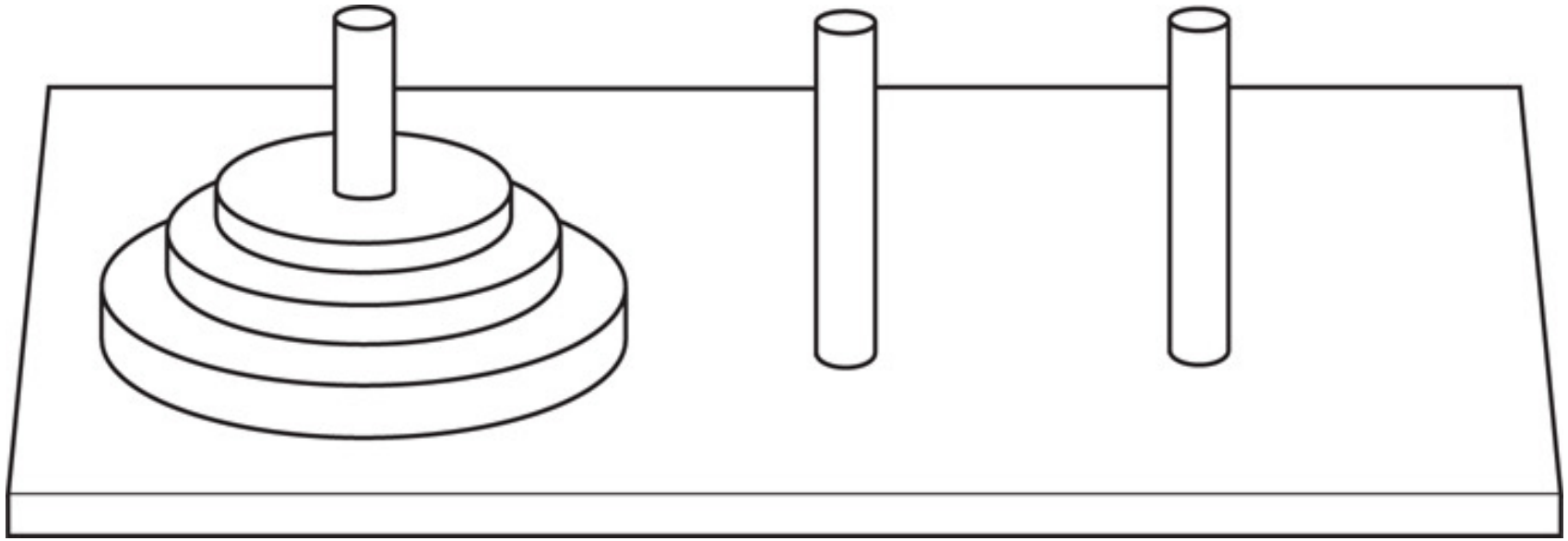
```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        a, b = 0, 1
        for i in range(n-1):
            a, b = b, a + b
        return b

for i in range(10):
    print(fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
```

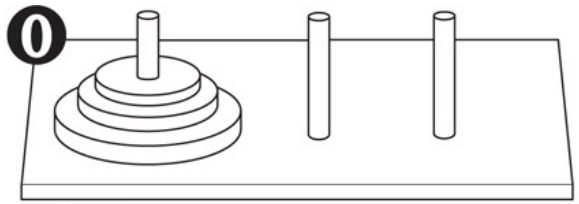
Example(Hanoi Tower)

- 하노이탑(Hanoi Tower) : 3개의 기둥과 그 위에 크기가 다른 원판들이 있을 때, 한 번에 한 개의 원판을 다른 기둥으로 옮김
- 목표는 첫 번째 기둥에 있는 모든 원판을 세 번째 기둥으로 옮기는 것

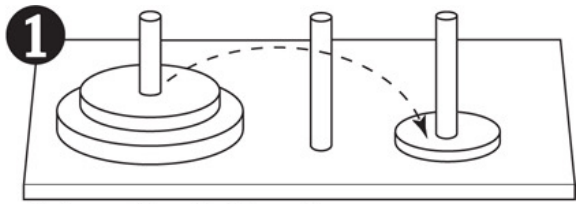
1. 한 번에 한 개의 원판만 옮길 수 있다.
2. 작은 원판 위에 큰 원판을 올릴 수 없다.



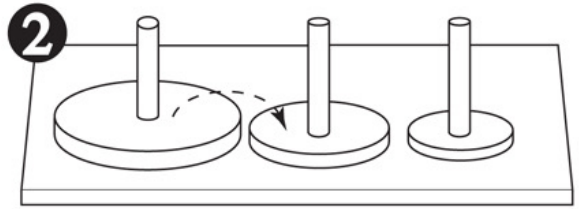
Example(Fibonacci numbers)



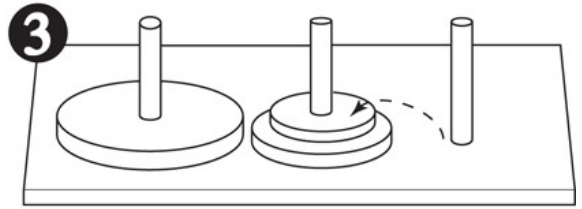
Original setup.



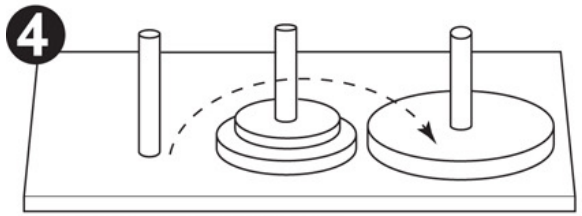
First move: Move disc 1 to peg 3.



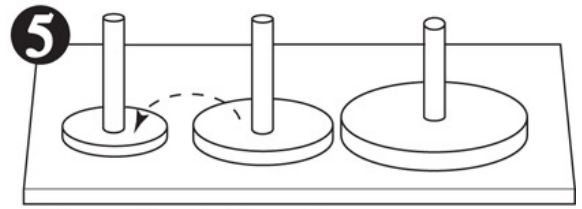
Second move: Move disc 2 to peg 2.



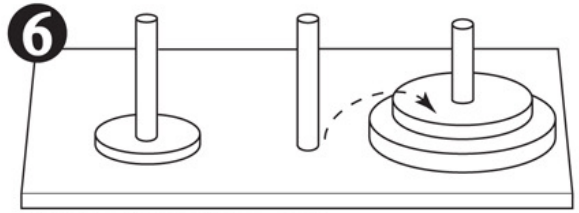
Third move: Move disc 1 to peg 2.



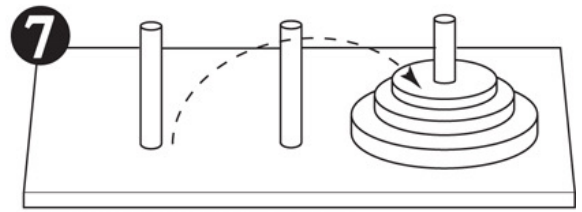
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



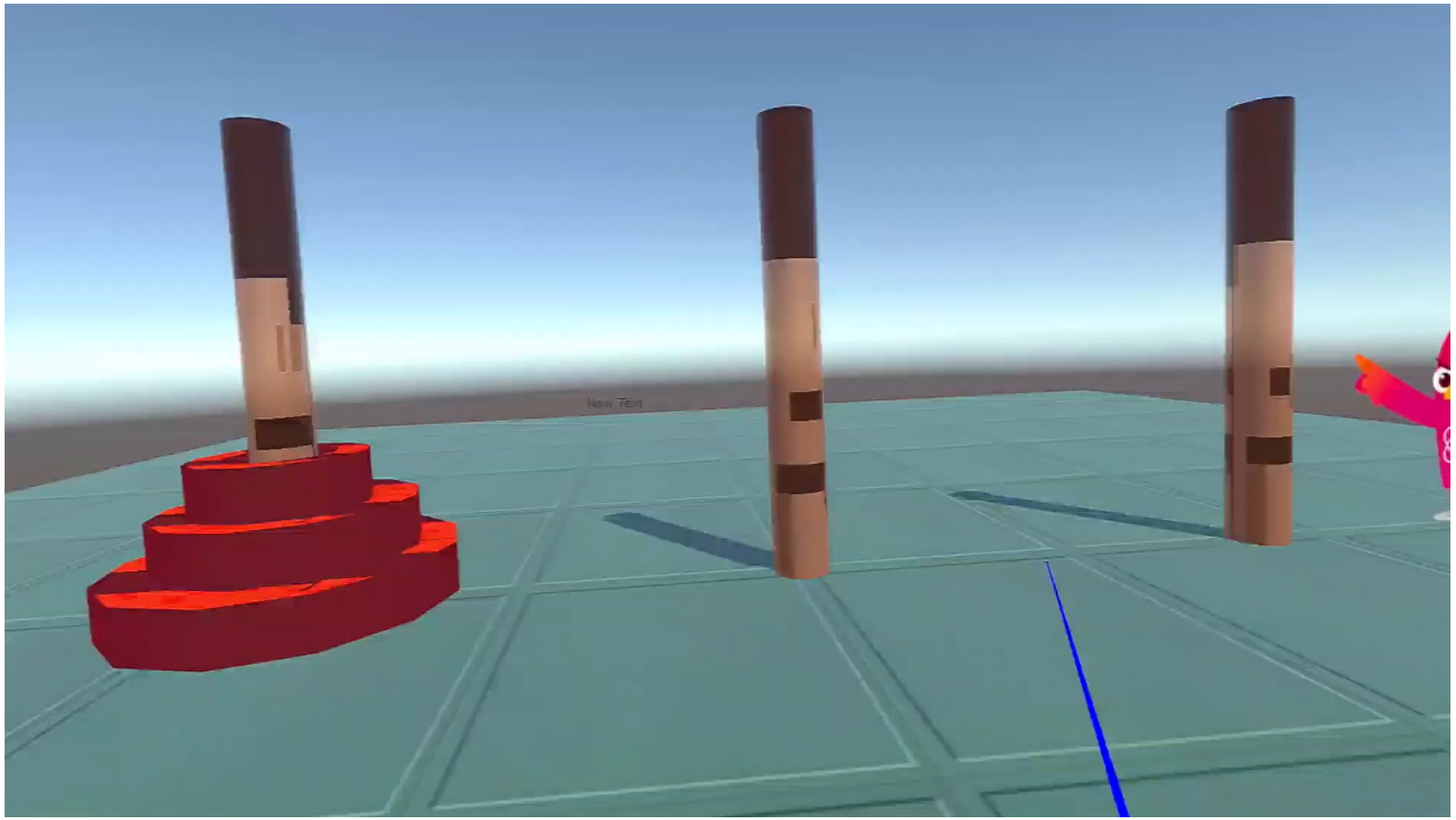
Sixth move: Move disc 2 to peg 3.



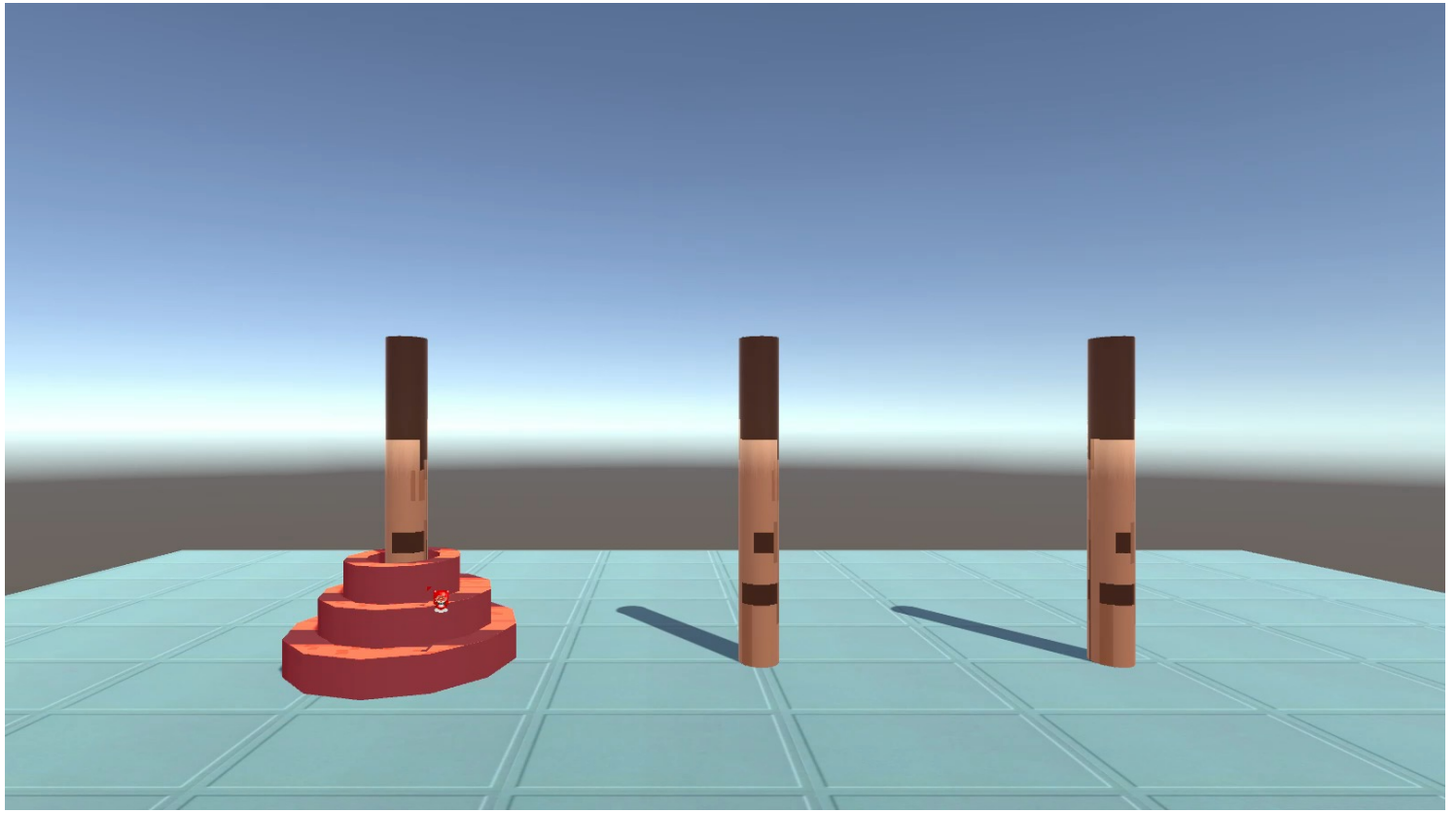
Seventh move: Move disc 1 to peg 3.

Hanoi_Intro

Hanoi_VR



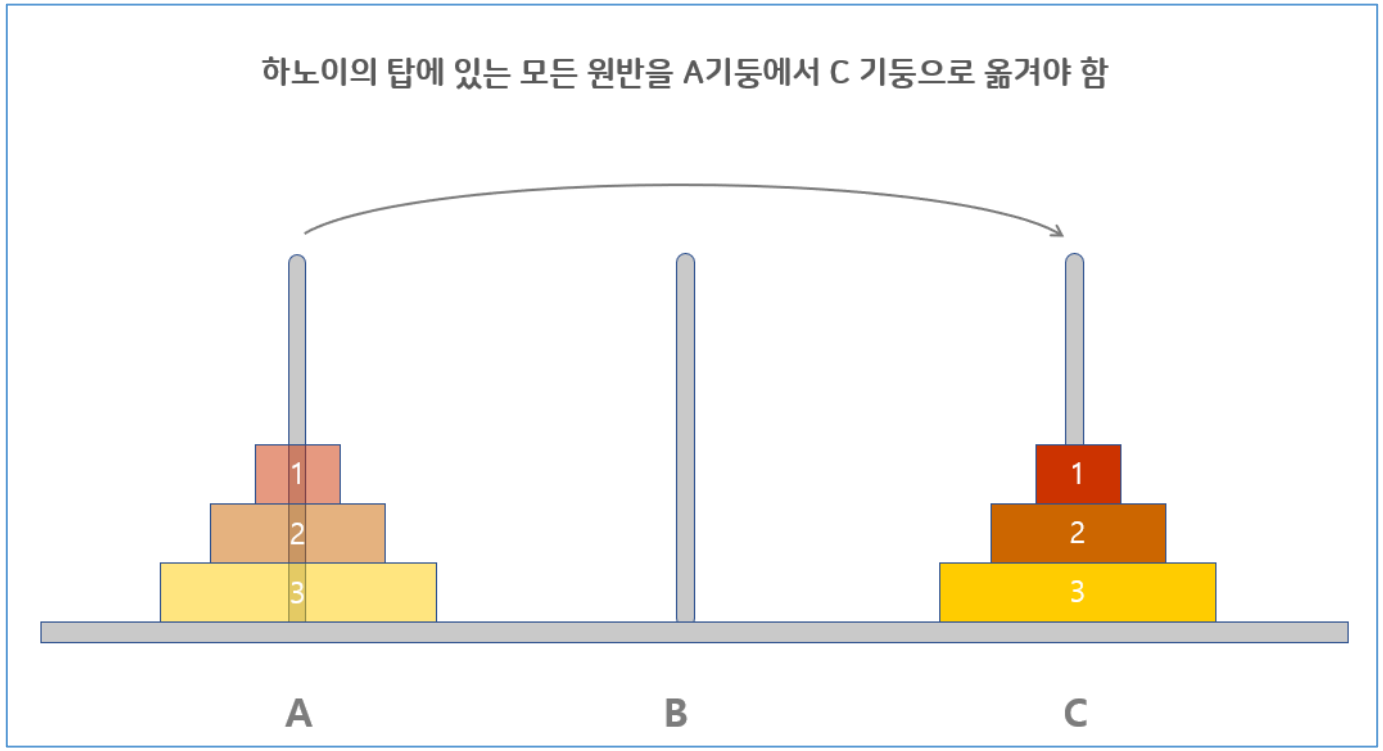
Hanoi_2D



Hanoi_2D 게임 실행하며 재귀적 특징 이해하기

하노이의 탑 풀이

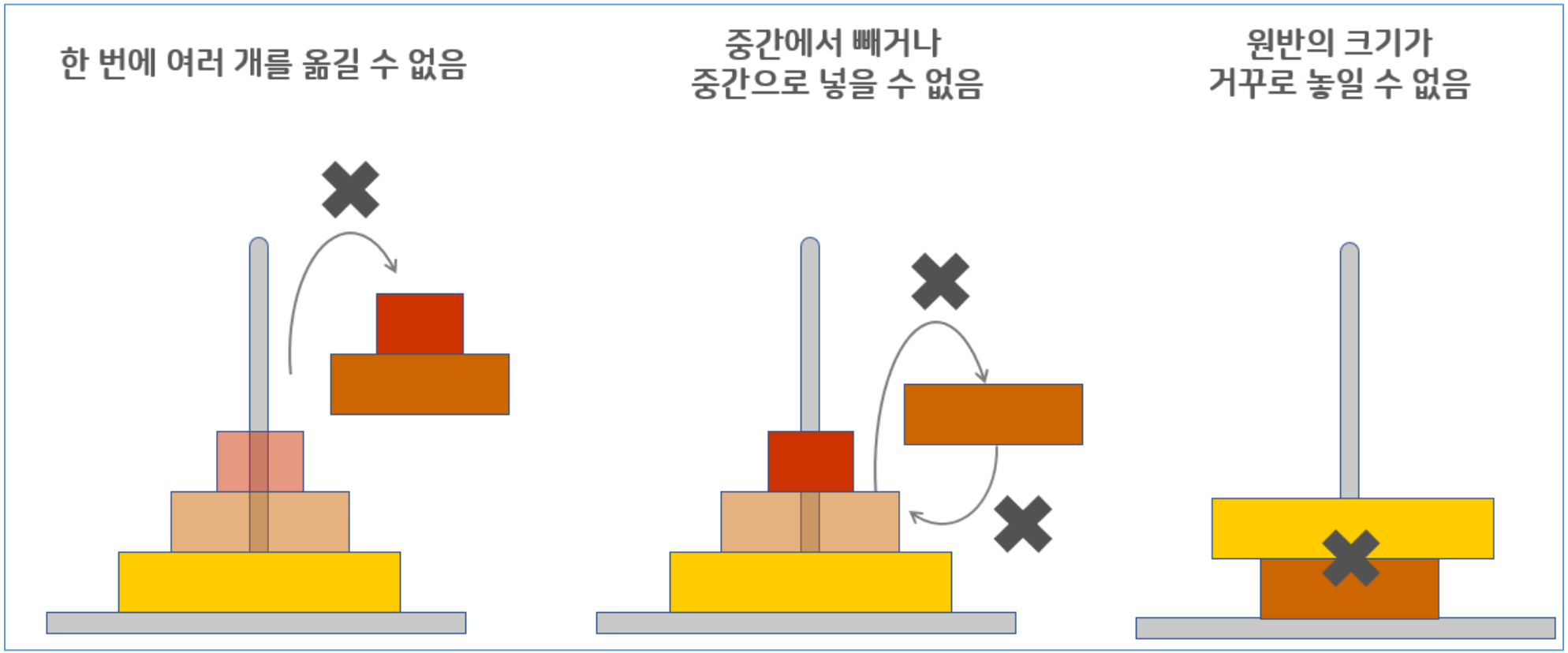
Example(Hanoi Tower)



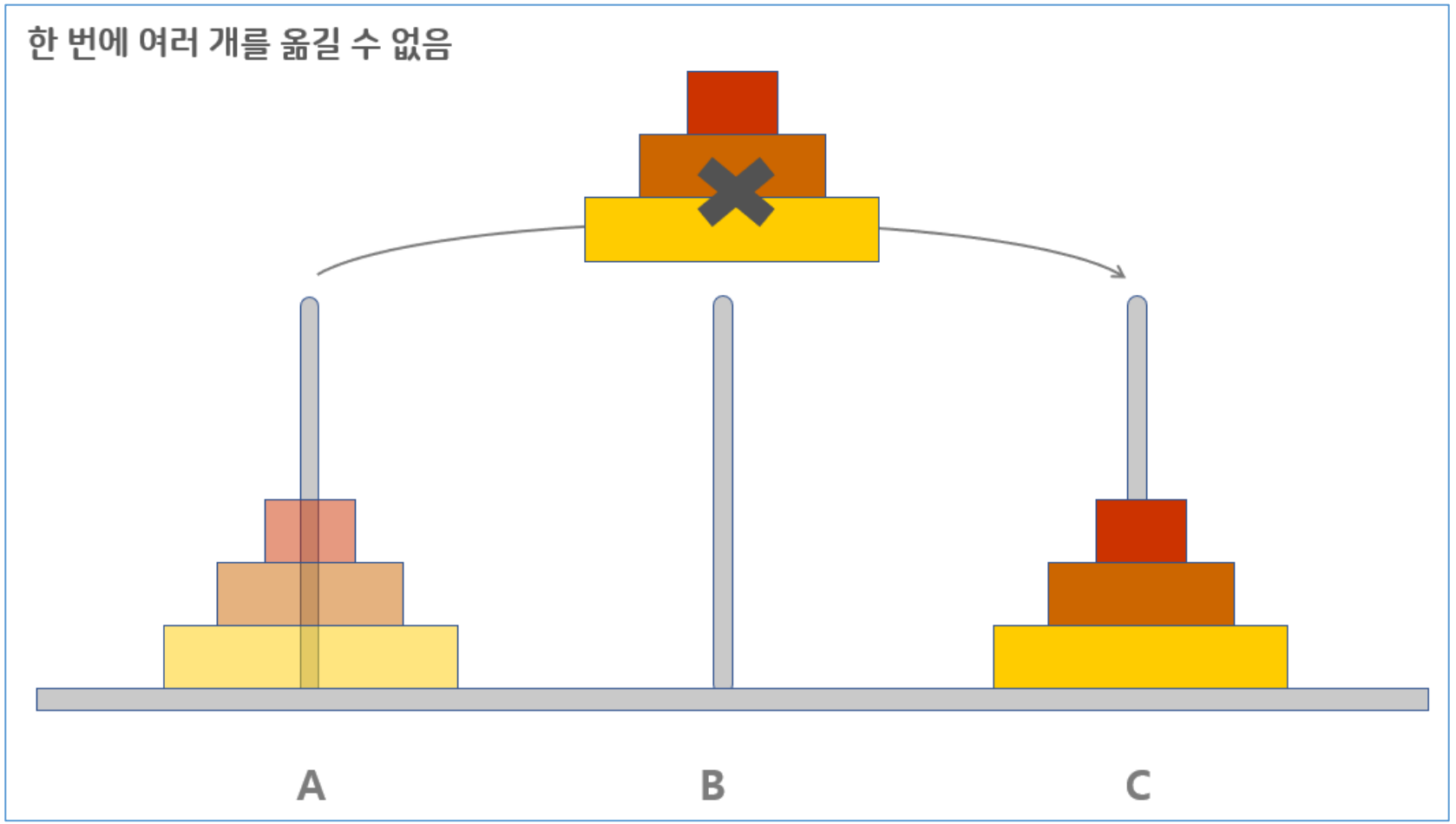
- 하노이탑(Hanoi Tower) : 3개의 기둥과 그 위에 크기가 다른 원판들이 있을 때, 한 번에 한 개의 원판을 다른 기둥으로 옮김
- 목표는 첫 번째 A기둥에 있는 모든 원판을 세 번째 C기둥으로 옮기는 것

1. 한 번에 한 개의 원판만 옮길 수 있다.
2. 작은 원판 위에 큰 원판을 올릴 수 없다.

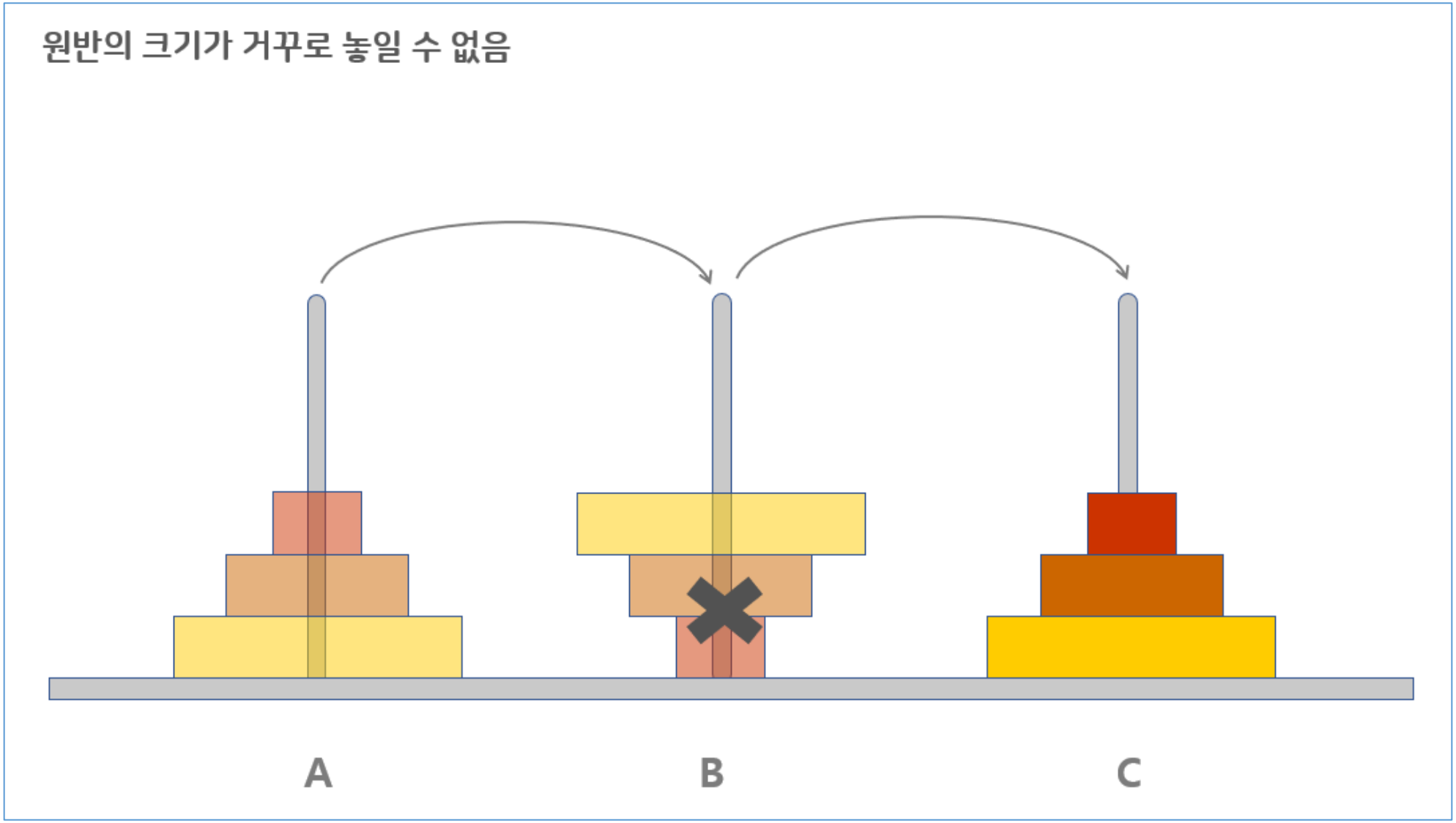
Example(Hanoi Tower)



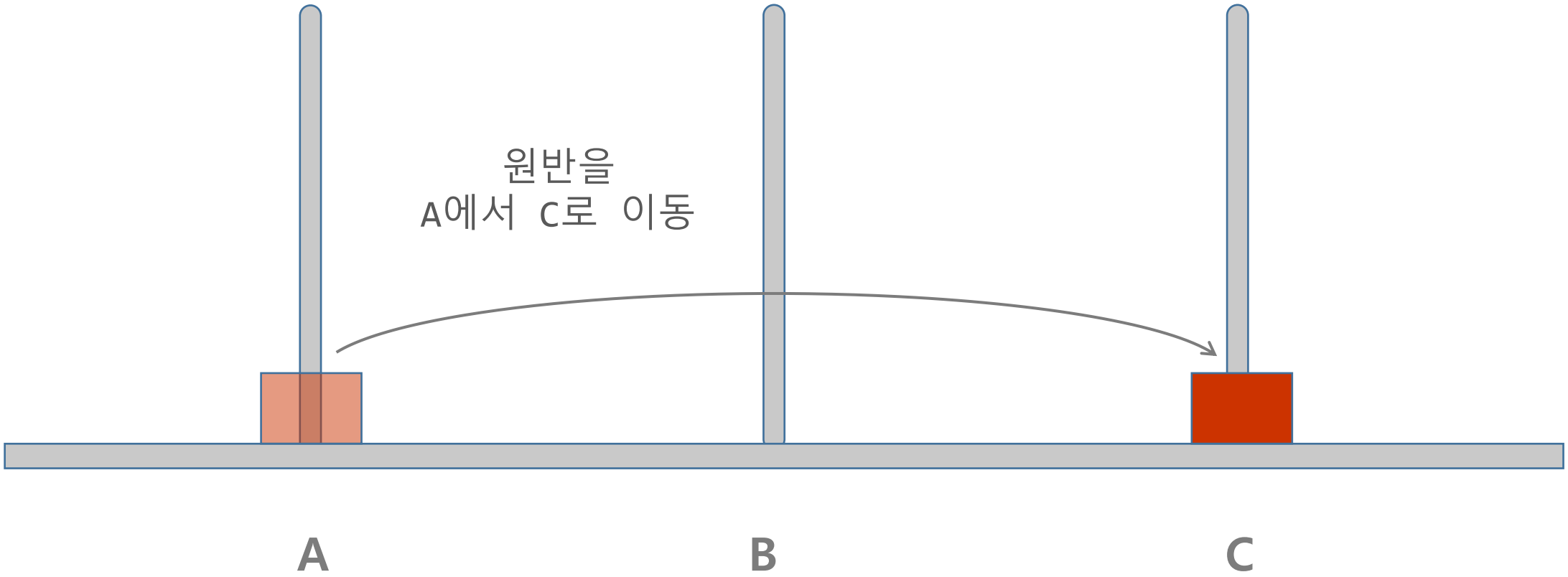
Example(Hanoi Tower)



Example(Hanoi Tower)



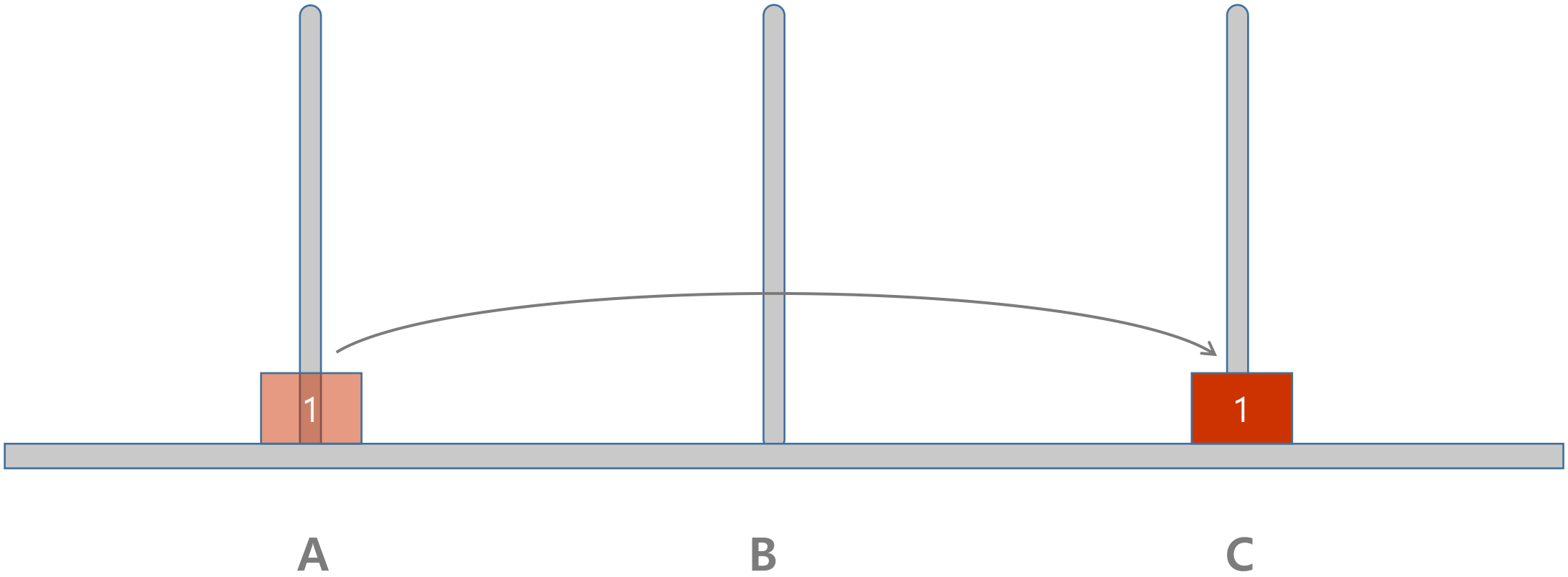
원반이 한 개 일 때



원반이 한 개일 때 (1)

n=1

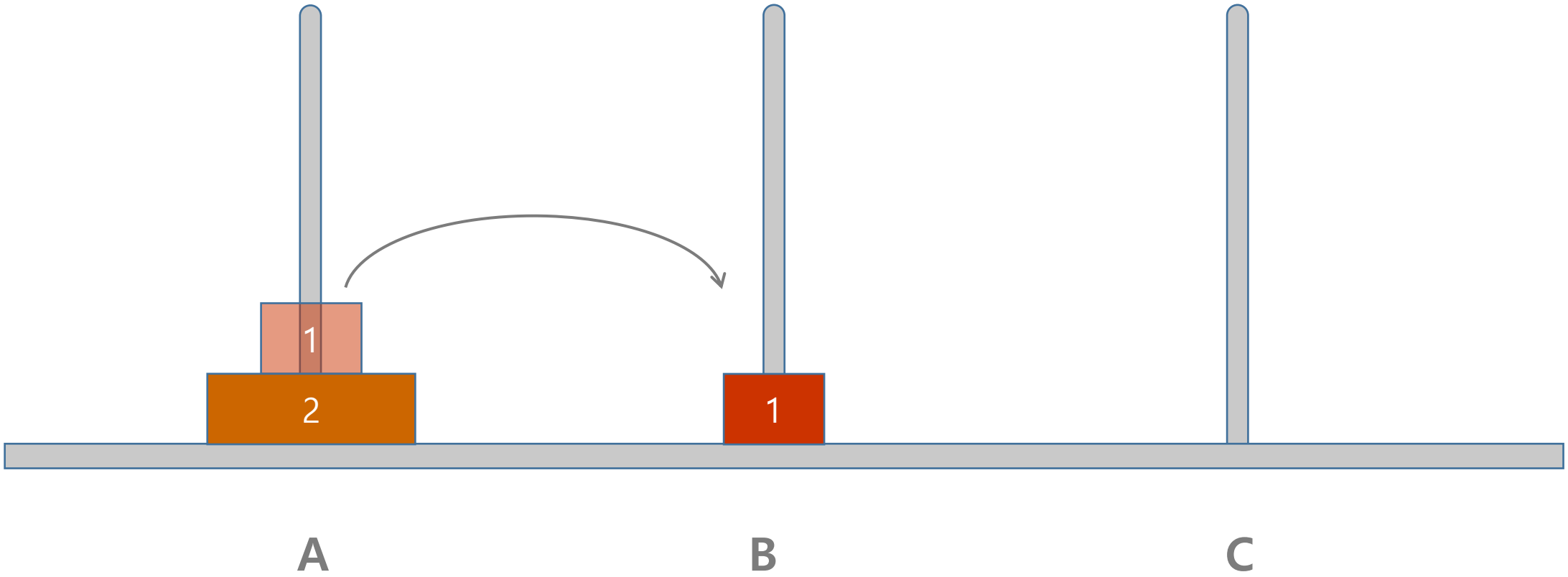
1 원반을 A에서 C로 이동



원반이 두 개일 때 (1)

n=2

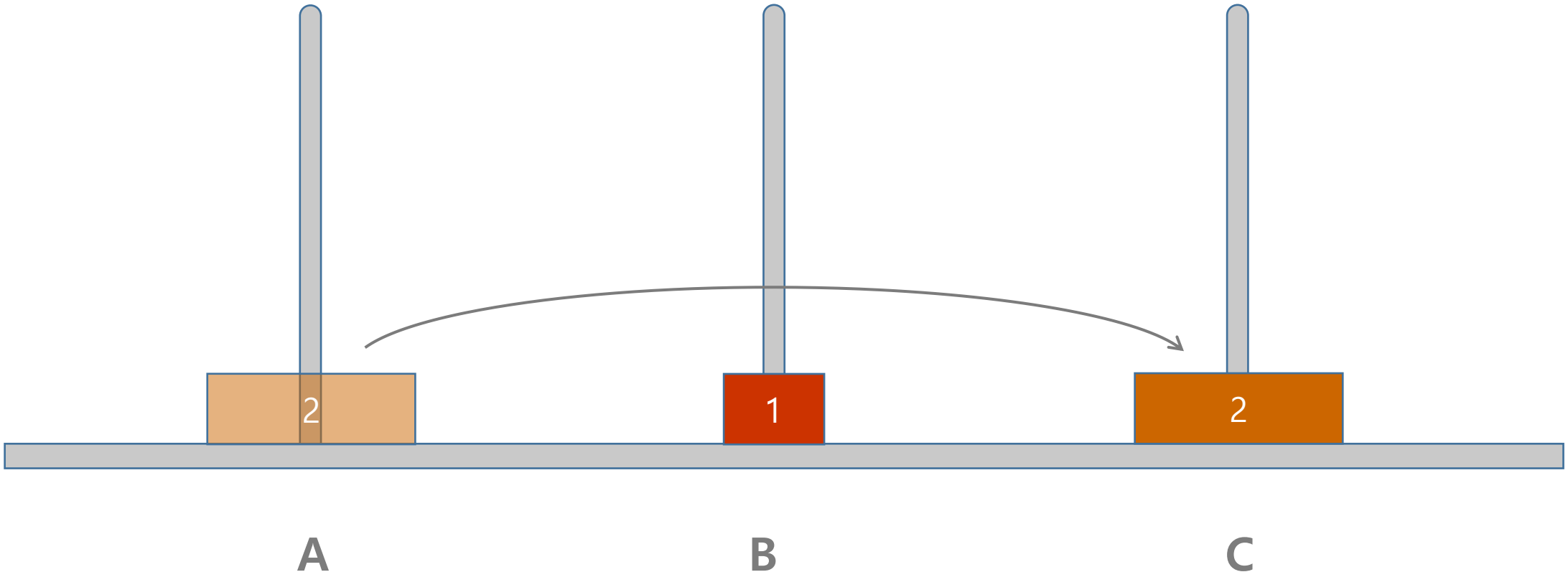
1 번 원반을 A에서 B로 이동



원반이 두 개일 때 (2)

n=2

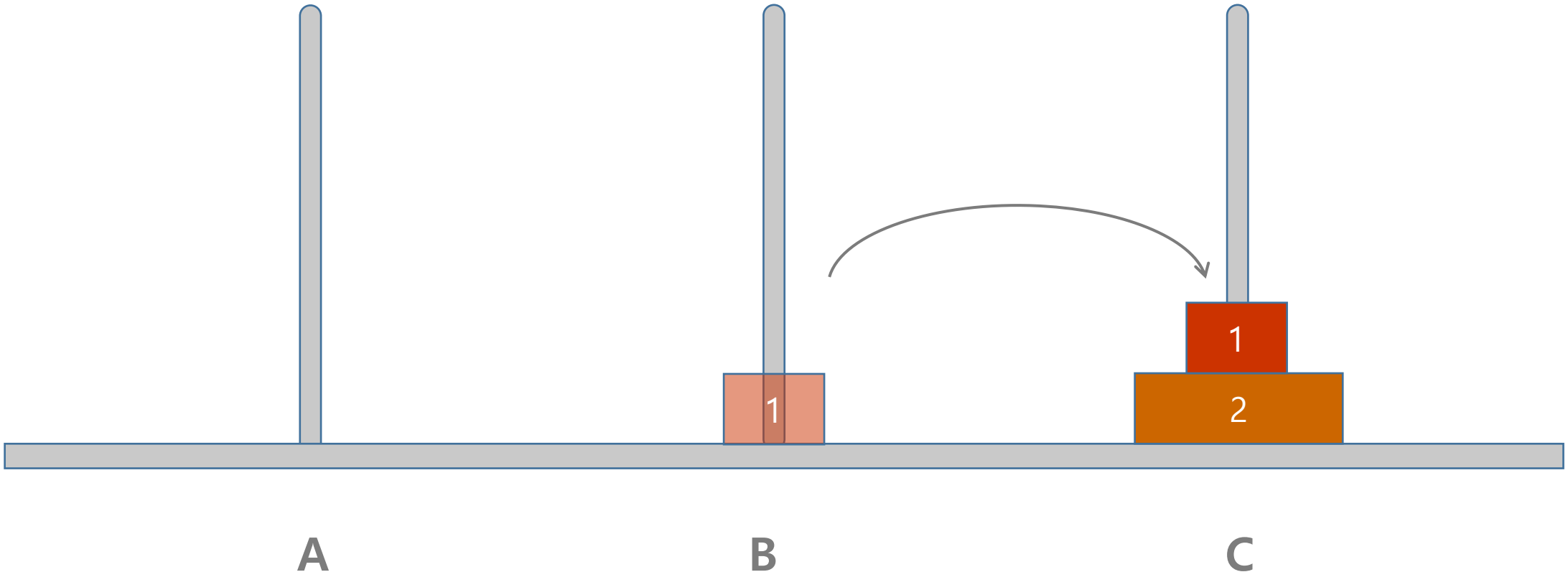
2 번 원반을 A에서 C로 이동



원반이 두 개일 때 (3)

n=2

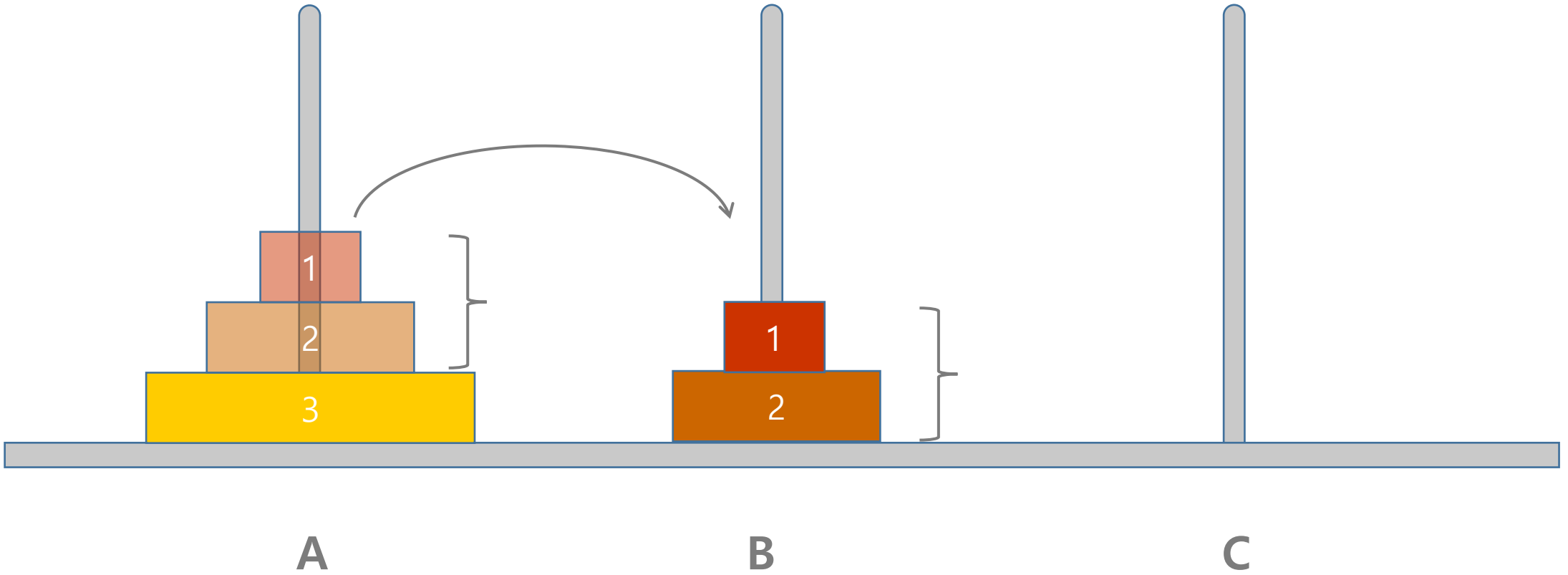
1 번 원반을 B에서 C로 이동



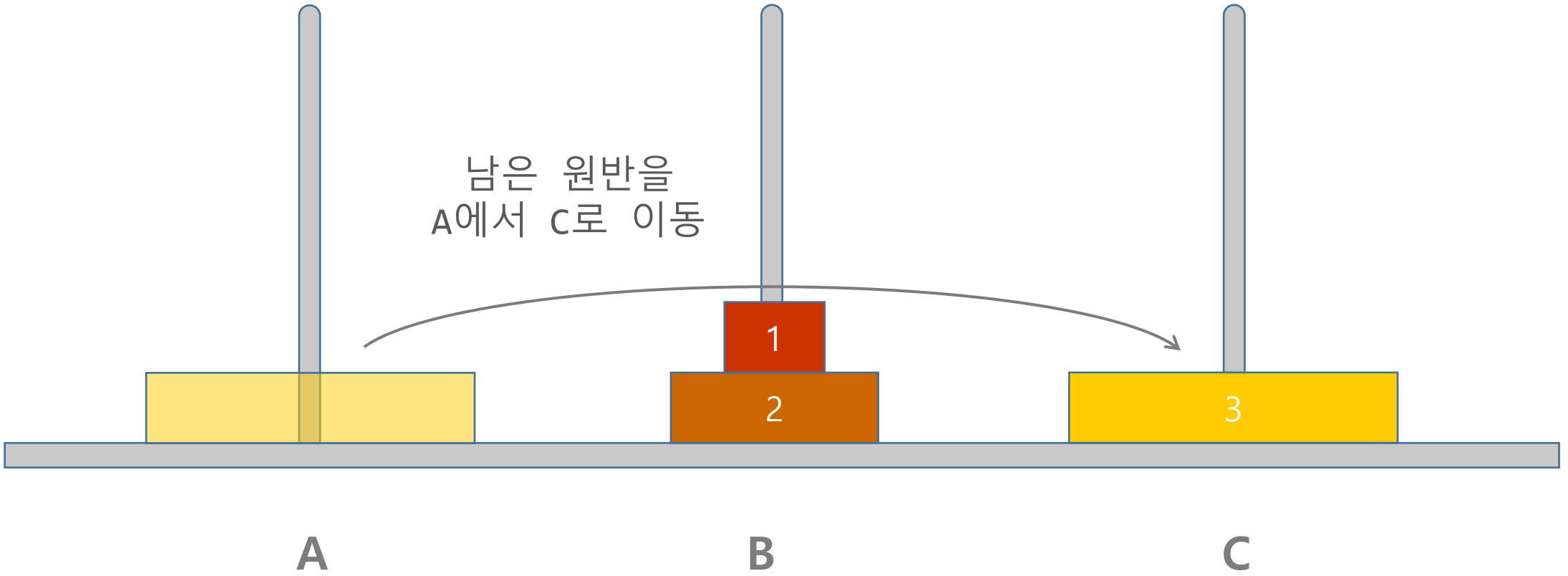
원반이 세 개일 때 (1)

n=3

원반 두 개를
A에서 B로 이동
(A->C, A->B, C->B)

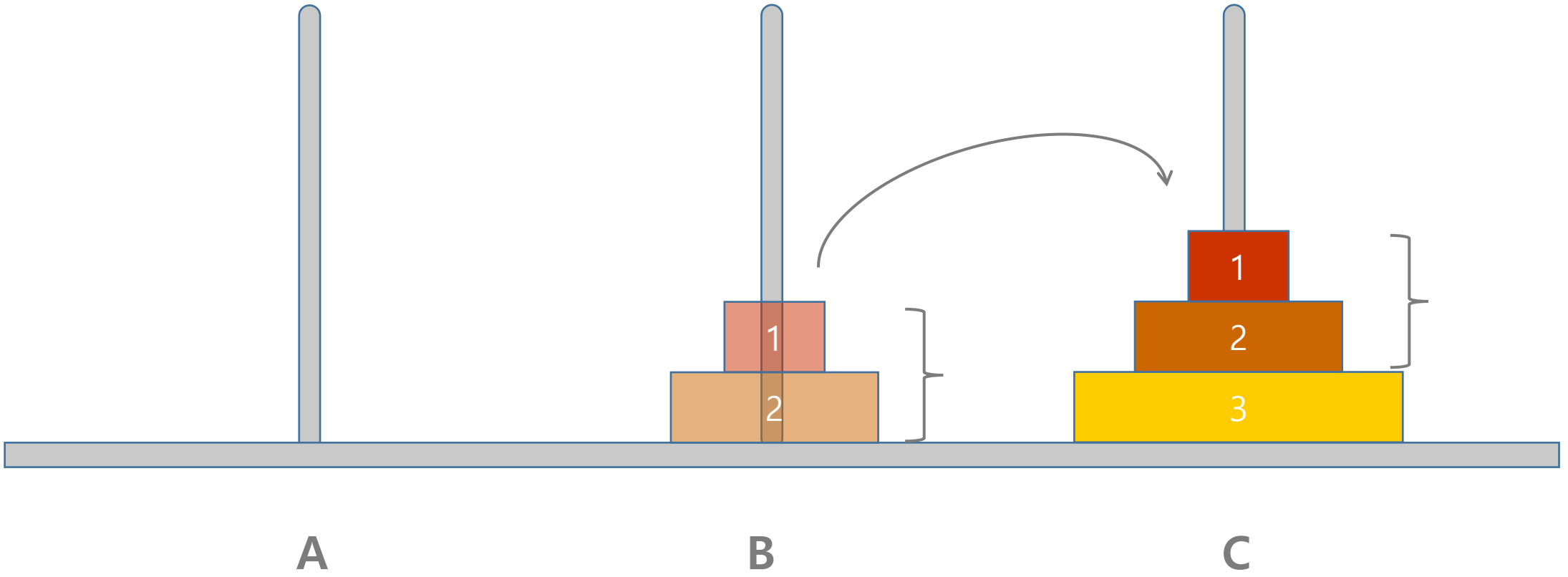


원반이 세 개일 때 (2)

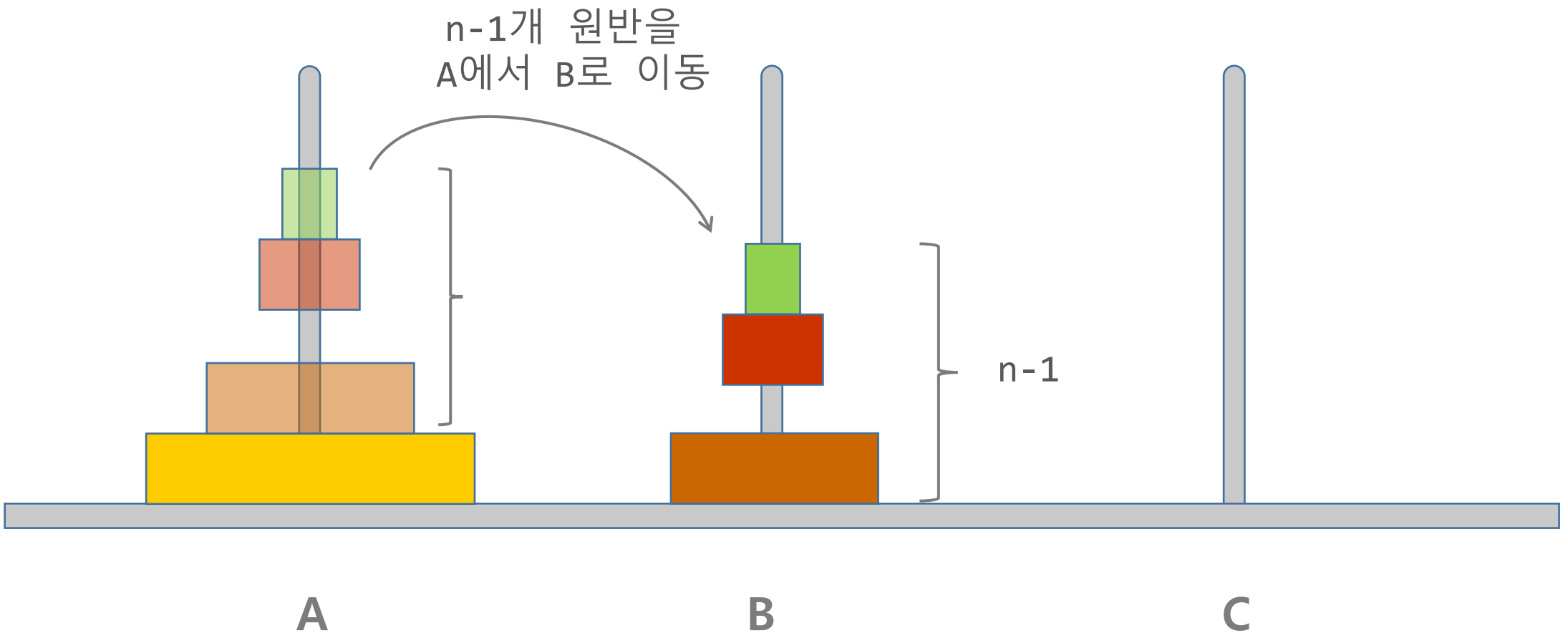


원반이 세 개일 때 (3)

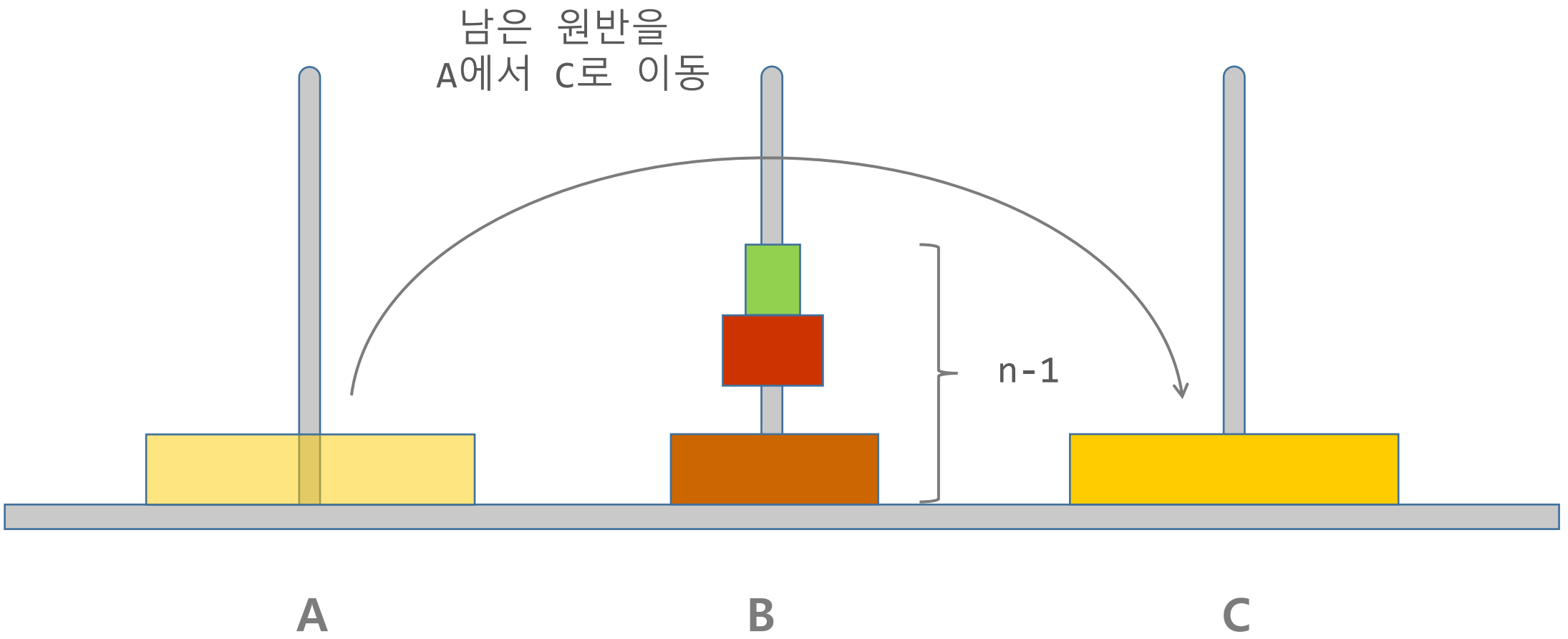
B번 기둥에 있는 원반 두 개를
A에서 C으로 이동
(B->A, B->C, A->C)



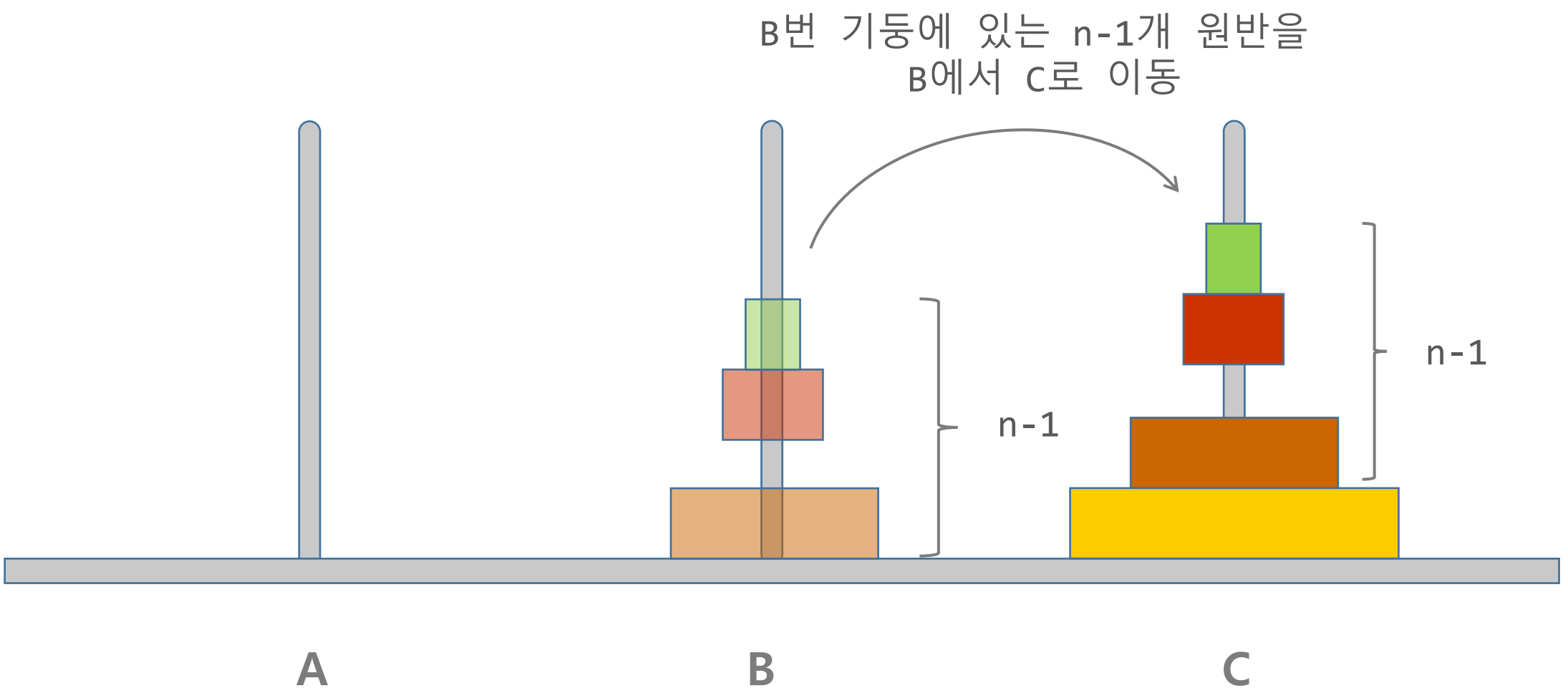
원반이 n개일 때 (1)



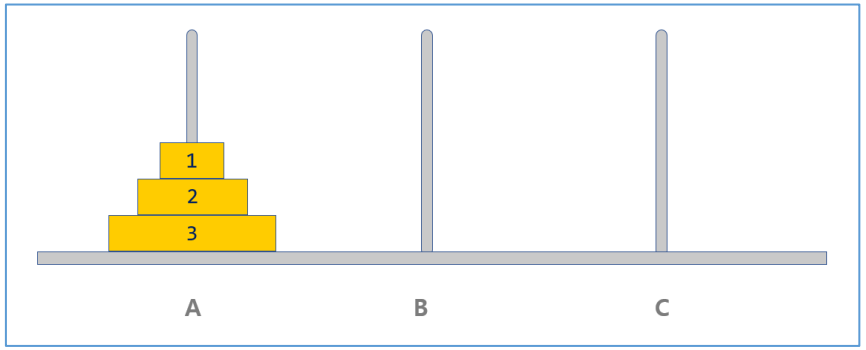
원반이 n개일 때 (2)



원반이 n개일 때 (3)



N=3



이동횟수 : f(n)=1+2f(n-1)
f(3)=1+2*3 -> 7회
f(4)=1+2*7 -> 15회
f(5)=1+2*15-> 31회

원반개수
짝수 : 1번을 임시기둥(B)로 먼저 이동
홀수 : 1번을 목표기둥(C)로 먼저 이동

원반 이동횟수

N=1 1 A -> C 1회

N=2
1 A -> B
2 A -> C
1 B -> C
3회

N=3
N-1 A -> B 3회
N A -> C 1회
N-1 B -> C 3회

3회
1 A -> C
2 A -> B
1 C -> B

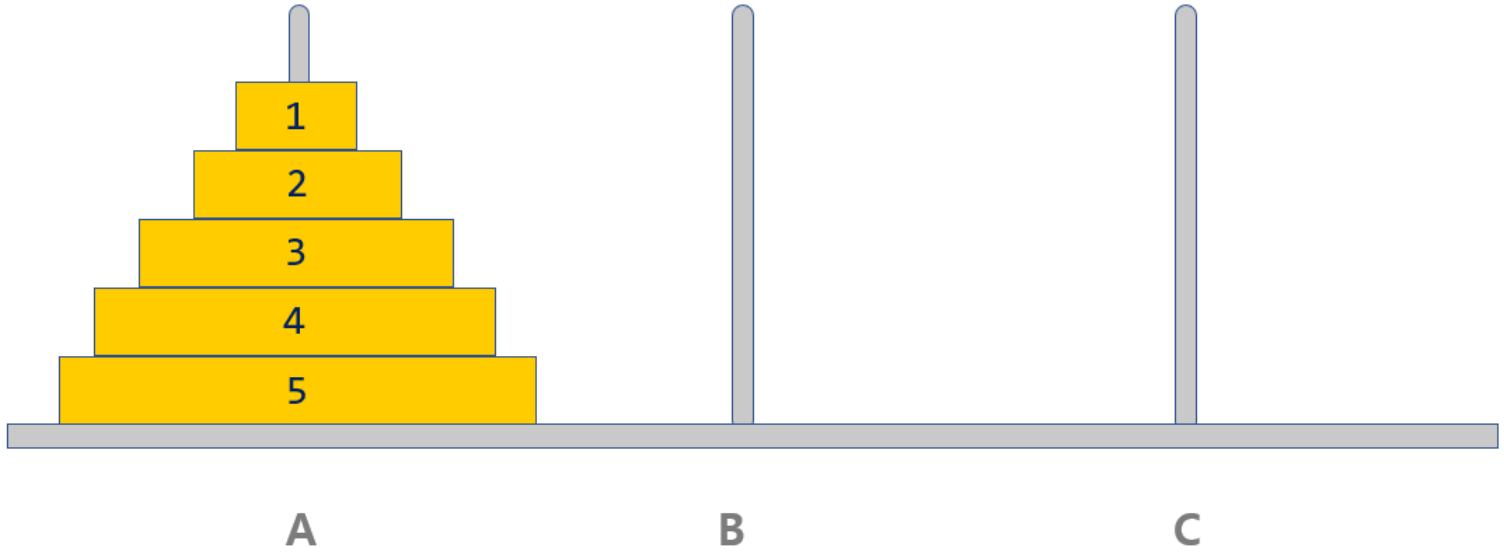
3
A -> C

3회
1 B -> A
2 B -> C
1 A -> C

7회

```
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```

N=5 최소이동횟수 : 31회




```
0: 원반 1을 A에서 C로 이동
1: 원반 2을 A에서 B로 이동
2: 원반 1을 C에서 B로 이동
3: 원반 3을 A에서 C로 이동
4: 원반 1을 B에서 A로 이동
5: 원반 2을 B에서 C로 이동
6: 원반 1을 A에서 C로 이동
7: 원반 4을 A에서 B로 이동
8: 원반 1을 C에서 B로 이동
9: 원반 2을 C에서 A로 이동
10: 원반 1을 B에서 A로 이동
11: 원반 3을 C에서 B로 이동
12: 원반 1을 A에서 C로 이동
13: 원반 2을 A에서 B로 이동
14: 원반 1을 C에서 B로 이동
15: 원반 5을 A에서 C로 이동
16: 원반 1을 B에서 A로 이동
17: 원반 2을 B에서 C로 이동
18: 원반 1을 A에서 C로 이동
19: 원반 3을 B에서 A로 이동
20: 원반 1을 C에서 B로 이동
21: 원반 2을 C에서 A로 이동
22: 원반 1을 B에서 A로 이동
23: 원반 4을 B에서 C로 이동
24: 원반 1을 A에서 C로 이동
25: 원반 2을 A에서 B로 이동
26: 원반 1을 C에서 B로 이동
27: 원반 3을 A에서 C로 이동
28: 원반 1을 B에서 A로 이동
29: 원반 2을 B에서 C로 이동
30: 원반 1을 A에서 C로 이동
총 이동 횟수: 31
```

Example(Hanoi_tower - recursion)

```
def hanoi(n, source, target, auxiliary):
    if n > 0:
        # 1단계: n-1개의 원반을 보조 기둥으로 이동
        hanoi(n-1, source, auxiliary, target)
        # 2단계: 가장 큰 원반을 목표 기둥으로 이동
        print(f"원반 {n}을 {source}에서 {target}로 이동")
        # 3단계: n-1개의 원반을 목표 기둥으로 이동
        hanoi(n-1, auxiliary, target, source)

# 예제 실행
hanoi(3, 'A', 'C', 'B')
```

```
원반 1을 A에서 C로 이동
원반 2을 A에서 B로 이동
원반 1을 C에서 B로 이동
원반 3을 A에서 C로 이동
원반 1을 B에서 A로 이동
원반 2을 B에서 C로 이동
원반 1을 A에서 C로 이동
```

Example(Hanoi_tower - recursion)

```
def hanoi(n, source, target, auxiliary):
    global move_count
    if n > 0:
        hanoi(n-1, source, auxiliary, target)
        print(f"{move_count}: 원반 {n}을 {source}에서 {target}로 이동")
        move_count += 1
        hanoi(n-1, auxiliary, target, source)

# 원반 이동 횟수를 추적하기 위한 전역 변수
move_count = 0

# 원반 5개로 예제 실행
hanoi(5, 'A', 'C', 'B')
print(f"총 이동 횟수: {move_count}")
```

```
0: 원반 1을 A에서 C로 이동
1: 원반 2을 A에서 B로 이동
2: 원반 1을 C에서 B로 이동
3: 원반 3을 A에서 C로 이동
4: 원반 1을 B에서 A로 이동
5: 원반 2을 B에서 C로 이동
6: 원반 1을 A에서 C로 이동
7: 원반 4을 A에서 B로 이동
8: 원반 1을 C에서 B로 이동
9: 원반 2을 C에서 A로 이동
10: 원반 1을 B에서 A로 이동
11: 원반 3을 C에서 B로 이동
12: 원반 1을 A에서 C로 이동
13: 원반 2을 A에서 B로 이동
14: 원반 1을 C에서 B로 이동
15: 원반 5을 A에서 C로 이동
16: 원반 1을 B에서 A로 이동
17: 원반 2을 B에서 C로 이동
18: 원반 1을 A에서 C로 이동
19: 원반 3을 B에서 A로 이동
20: 원반 1을 C에서 B로 이동
21: 원반 2을 C에서 A로 이동
22: 원반 1을 B에서 A로 이동
23: 원반 4을 B에서 C로 이동
24: 원반 1을 A에서 C로 이동
25: 원반 2을 A에서 B로 이동
26: 원반 1을 C에서 B로 이동
27: 원반 3을 A에서 C로 이동
28: 원반 1을 B에서 A로 이동
29: 원반 2을 B에서 C로 이동
30: 원반 1을 A에서 C로 이동
총 이동 횟수: 31
```

Example(Hanoi Tower)

- `n` (이동해야 할 원반의 수), `source` (출발 기둥), `target` (목표 기둥), `auxiliary` (보조 기둥).

  1. 기본 사례(Base Case):
     - `n > 0`이 아닐 경우, 즉 원반의 수가 0이면, 함수는 아무 작업도 수행하지 않고 반환, 재귀 호출의 종료 조건

  2. 첫 번째 단계 (n-1 원반을 보조 기둥으로 이동):
     - `n-1`개의 원반을 `source` 기둥에서 `auxiliary` 기둥으로 이동
     - `target` 기둥은 보조 기둥으로 사용

  3. 두 번째 단계 (가장 큰 원반을 목표 기둥으로 이동):
     - 가장 큰 원반 (즉, `n`번째 원반)을 `source` 기둥에서 `target` 기둥으로 이동
     - 이 단계는 단순한 출력으로, 실제 이동을 나타냄

  4. 세 번째 단계 (n-1 원반을 목표 기둥으로 이동):
     - `n-1`개의 원반을 `auxiliary` 기둥에서 `target` 기둥으로 이동
     - `source` 기둥은 보조 기둥으로 사용

  - 이 알고리즘의 핵심은 큰 문제 (n개의 원반을 이동)를 두 개의 더 작은 문제로 나누는 것
  - (1) `n-1`개의 원반을 보조 기둥으로 이동
  - (2) 가장 큰 원반을 목표 기둥으로 이동
  - (3) `n-1`개의 원반을 보조 기둥에서 목표 기둥으로 이동.
  - 이 과정은 재귀적으로 반복되어, 각 단계에서 문제의 크기가 점점 줄어듦