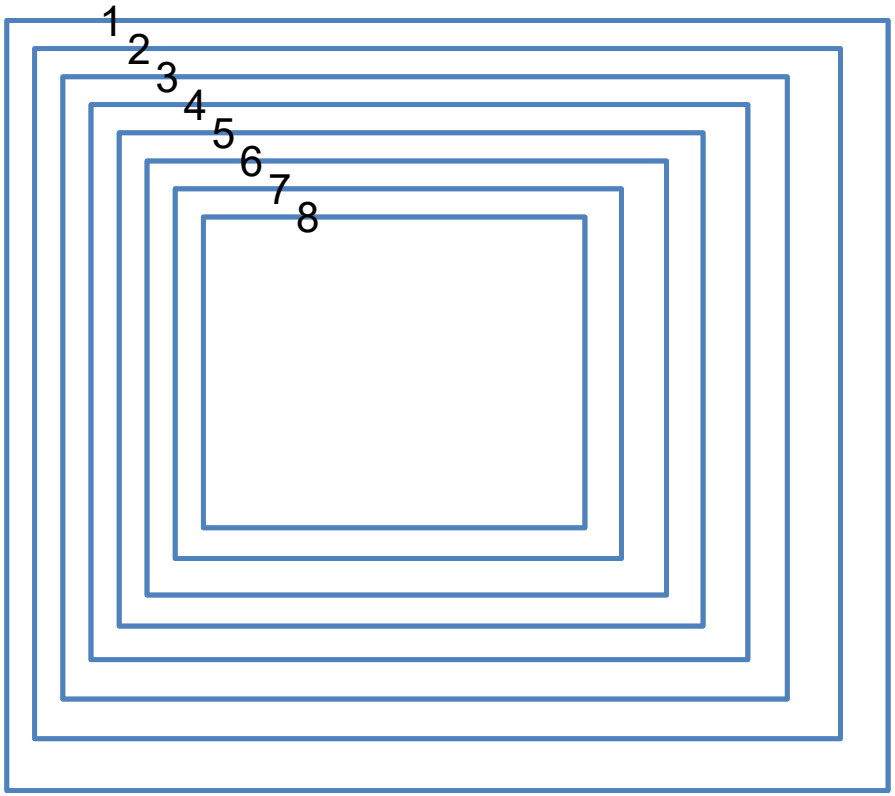

JAVA
자 료 구 조
Recursive

## 3. Recursive




### Recursive란....

어떤 사건이 자기 자신을 포함하고 다시 자기 자신을 사용하여 정의 될 때 재귀적이라고 한다.

문제를 해결하기 위해 원래 범위의 문제에서 더 작은 범위의 하위 문제를 먼저 해결함으로써 원래 문제를 해결해 나가는 방식.

예를 들어, 오른쪽 1번째 네모 칸이 안쪽으로 반복되어 나타나는데, 이를 재귀의 예.

# 3. Recursive

```java
public class multiple {

    Run | Debug
    public static void main(String[] args) {
        Recursive();
    }

    public static void Recursive() {
        System.out.println("Recursive!");
        Recursive();
        System.out.println("Recursive!!");
    }
}
```

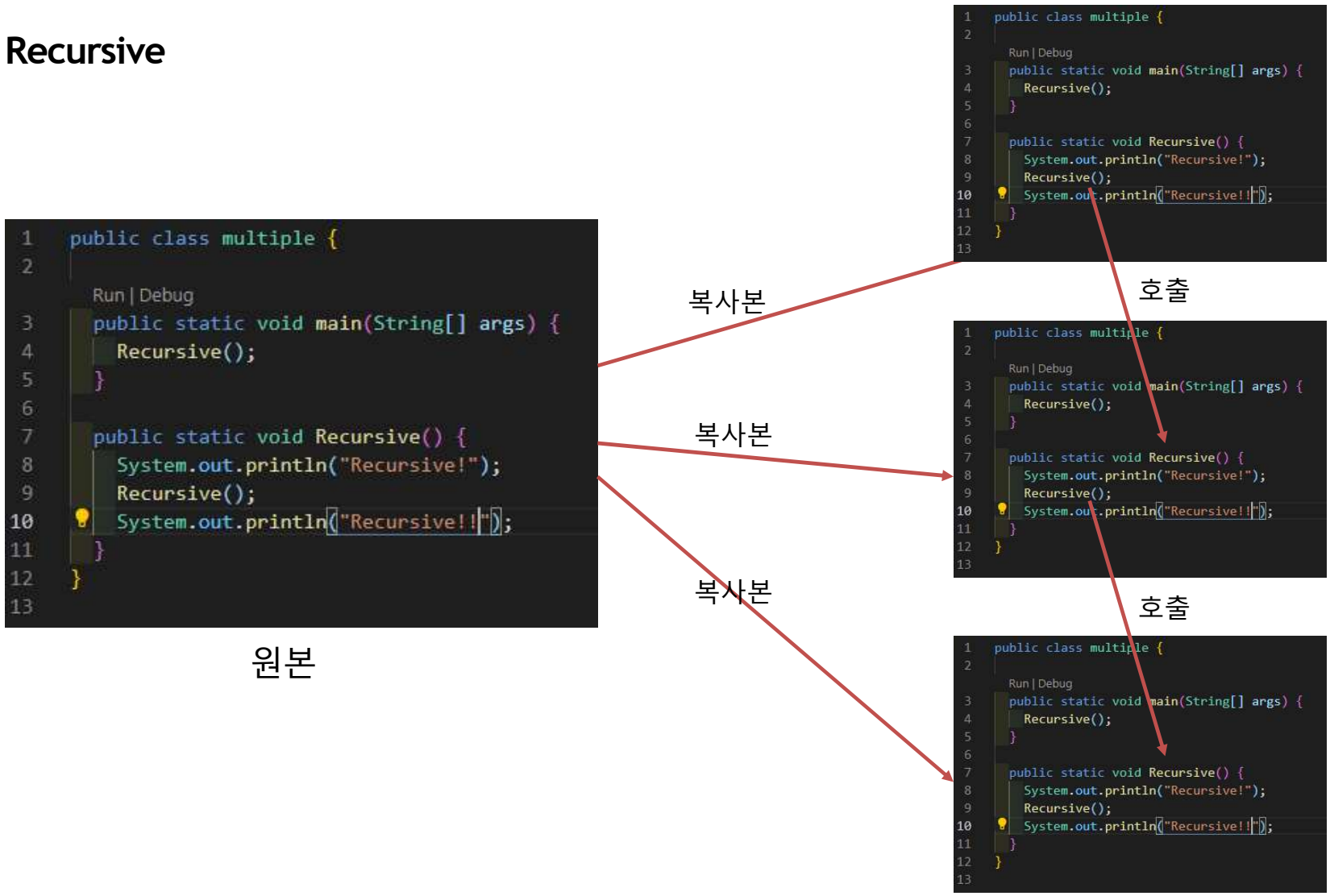
Recursive 함수의 이해

완료되지 않은 함수를 다시 호출하는것이 가능 ? -> 가능하다.

Recursive 함수 내에서 다시 Recursive 함수를 호출하고 있으므로 아래 System.out.println("Recursive!!") 함수를 거치지 않고 다시 실행.

사실 왼쪽 예제처럼 함수가 계속 반복이 된다고 생각하면 재귀를 이해하기 어렵다. 따라서 Recursive 함수의 복사본이 만들어져서 실행되는 구조로 생각하는 것이 이해하기 좋다.

# 3. Recursive


public class multiple {
Run | Debug
public static void main(String[] args) {
Recursive();
}
public static void Recursive() {
System.out.println("Recursive!");
Recursive();
System.out.println("Recursive!!");
}
}
원본
복사본
복사본
복사본
호출
호출

## 3. Recursive

위의 코드는 한 가지 문제가 있는데,
Recursive 함수에 재귀 탈출조건이 없다는 것.

만약 재귀 탈출조건을 명시해주지 않는다면, 시스템은 스택 영역을 계속 추가적으로 사용하게 되면서 일정 메모리 한계를 넘어서 스택 오버 플로우가 발생할 수 있다.

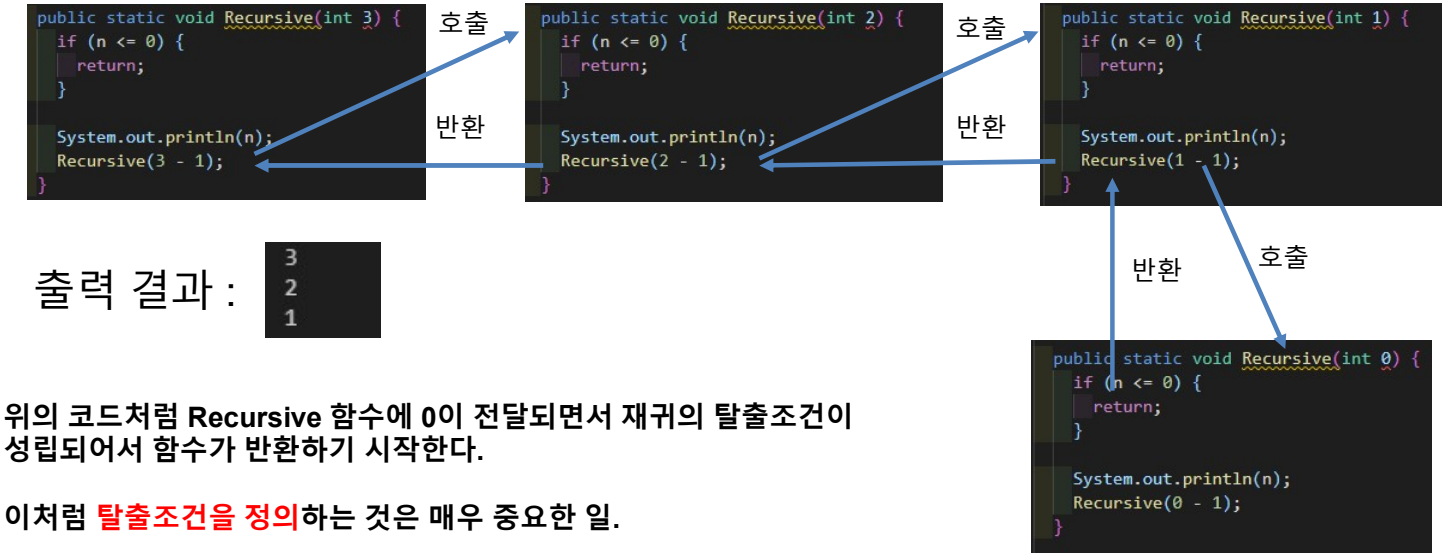
```java
public class Recursive {
    Run | Debug
    public static void main(String[] args) {
        Recursive(3);
    }

    public static void Recursive(int n) {
        if (n <= 0) {
            return;
        }

        System.out.println(n);
        Recursive(n - 1);
    }

}
```

시작




출력 결과 :

```
3
2
1
```

**위의 코드처럼 Recursive 함수에 0이 전달되면서 재귀의 탈출조건이 성립되어서 함수가 반환하기 시작한다.**

**이처럼 탈출조건을 정의하는 것은 매우 중요한 일.**

# 3. Recursive

재귀 사용의 이유

=> 재귀 알고리즘을 이용하면 복잡한 문제들도 간단하게 해결이 가능하다!

Ex) n을 입력 받아 재귀를 이용해서 Factorial 함수를 구성해라.

```java
import java.util.Scanner;

public class Recursive {

    public static int Factorial(int n) {

        해당 부분을 채워볼 것.

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;

        n = sc.nextInt();

        System.out.println("n의 Factorial은 : " + Factorial(n) + "입니다.");

    }

}
```

```
5
n의 Factorial은 : 120입니다.
PS C:\Users\7zjat\OneDrive\바탕 화면\AISL\AISL_JAVA\src>
```

# 3. Recursive

유클리드 호제법

유클리드 호제법이란, 간단히 말해서 최대공약수를 구하는 알고리즘의 하나.

호제법이라는 것이 두 개의 수를 서로 상대방 수로 나누어 결국 원하는 수를 얻는 알고리즘을 나타내는데,
자연수 a, b에 대해서 a를 b로 나눈 나머지를 r이라고 하면, a와 b의 최대공약수는 b와 r의 최대공약수와 같다.
이 성질에 따라서 b 를 r 로 나눈 나머지 r`을 구하고 다시 r 을 r` 으로 나눈 나머지를 구하는 과정을 반복해서 나머지 0이 되었을 때 나누는 수가 a와 b의 최대공약수다.

## 3. Recursive

위에서 읽은 것처럼 최대공약수를 구하기 위해서 a와 b를 나누고 나머지 r, 다시 b와 r을 나누고 나머지 r` 반복 후, 나머지 0이 되었을 때 나누는 수가 a와 b의 최대공약수입니다.
이 부분을 보면 재귀의 특성인 자기 자신을 호출해서 반복하는 것을 알 수 있다.

Ex) 100과 15의 최대공약수를 구해라.

```
1   public class Recursive {
        Run | Debug
2       public static void main(String[] args) {
3           System.out.println(gcd(100, 15));
4       }
5
6       public static int gcd(int n1, int n2) {
```

이 부분을 채워볼 것.

```
12
13  }
```

1. a = 100, b = 15
2. a / b
3. a % b => r = 10
4. b / r
5. b % r => r` = 5
6. r / r`
7. r % r` => r`` = 0
8. 나머지가 0이 되고, 그 때 나누는 수가 a와 b의 최대공약수이므로 답은 r` = 5이다.

## 3. Recursive

재귀 알고리즘의 분석

재귀 알고리즘을 좀 더 이해하기 위해서 더 많은 연습을 필요로 한다. 그렇기에 간단하게 피보나치 수열에 대해 공부를 해보자.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

흔히 하는 말로 앞에 2개의 수를 더해서 현재의 수를 만들어가는 수열이다.
첫 번째 0, 두 번째 1을 더해서 세번 째 1이 결정되고, 두 번째 1과 세 번째 1이 더해져서 네 번째 2가 결정된다.

수열의 n번째 값 = 수열의 n-1 번째 값 + 수열의 n-2번째 값

# 3. Recursive

```java
public class Recursive {

  public static int Fibo(int n) {
    if (n == 1)
      return 0;

    else if (n == 2)
      return 1;

    else
      return Fibo(n - 1) + Fibo(n - 2);

  }

  Run | Debug
  public static void main(String[] args) {

    for (int i = 1; i < 15; i++) {
      System.out.printf("%d ", Fibo(i));
    }

  }
}
```

출력결과

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

아직 우리는 흐름에 대해서 익숙하지 않기 때문에, 왼쪽의 코드를 약간 변경해서 실행 순서를 나타내 본다.

```
순서 : 7
순서 : 6
순서 : 5
순서 : 4
순서 : 3
순서 : 2
순서 : 1
순서 : 2
순서 : 3
순서 : 2
순서 : 1
순서 : 4
순서 : 3
순서 : 2
순서 : 1
순서 : 2
순서 : 5
순서 : 4
순서 : 3
순서 : 2
순서 : 1
순서 : 2
순서 : 3
순서 : 2
순서 : 1
```

왼쪽의 출력결과처럼 재귀함수는 매우 많은 함수호출을 동반한다.
그렇기에 성능상의 불리함은 존재한다.

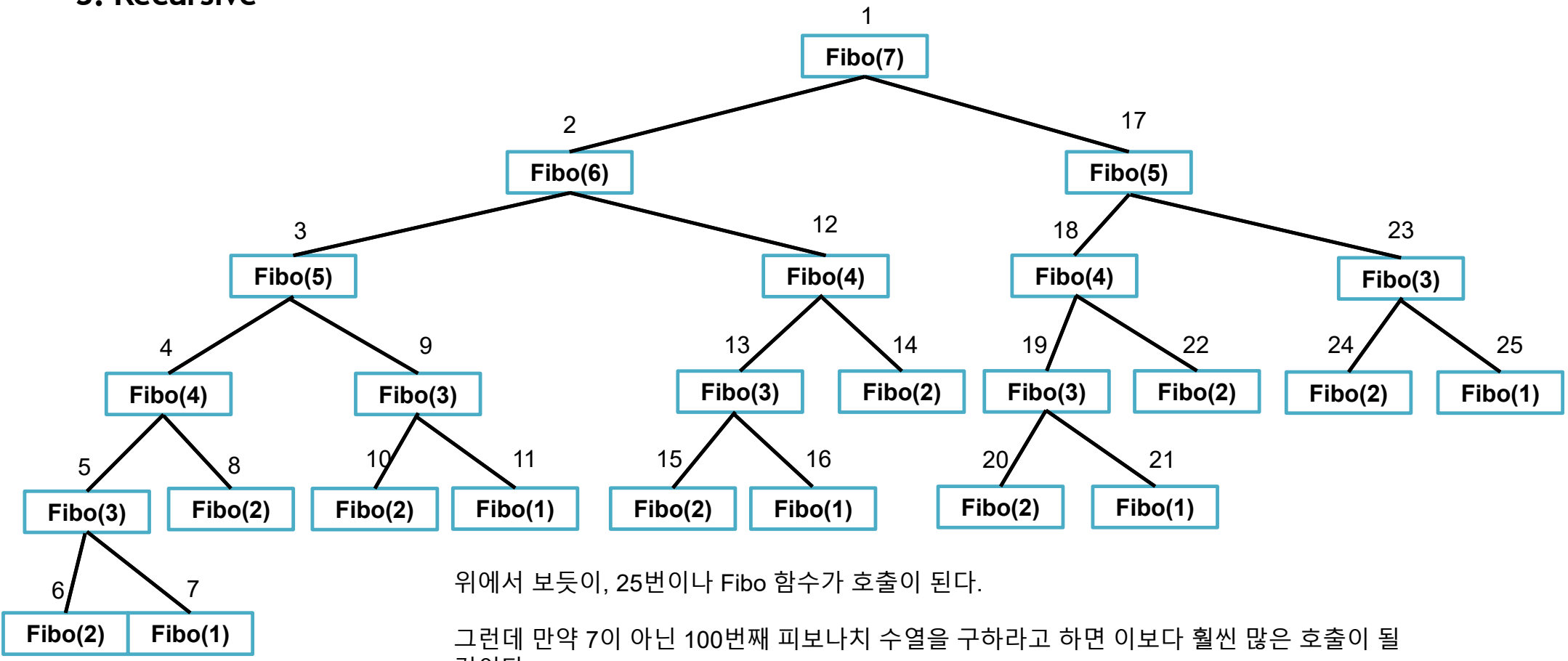
우선 함수의 호출이 어떻게 진행이 되는지 정리를 해보면, Fibo()함수의 7이 전달되고 이어서 Fibo 함수내에 다음 문장이 실행된다.

return Fibo(6) + Fibo(5);

즉 다시 2개의 함수가 다시 호출되는데, '+' 연산자의 왼쪽에 있는 Fibo(6) 함수호출이 완료되어야 비로소 오른쪽 Fibo(5) 함수가 호출이 진행된다.

그림으로 표현하면 다음페이지와 같다.

# 3. Recursive




위에서 보듯이, 25번이나 Fibo 함수가 호출이 된다.

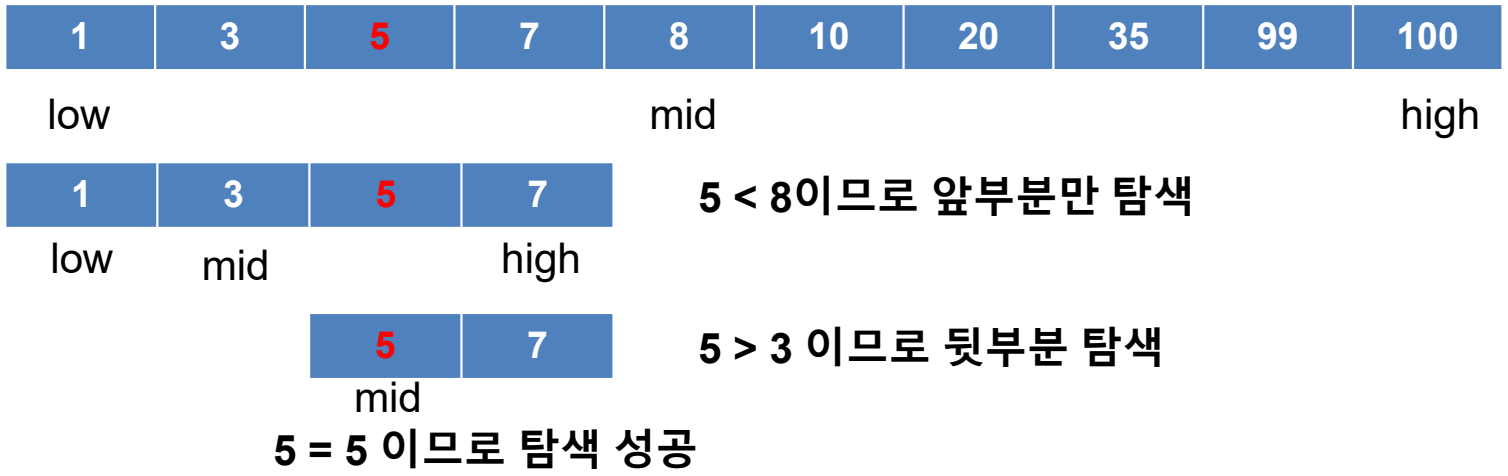
그런데 만약 7이 아닌 100번째 피보나치 수열을 구하라고 하면 이보다 훨씬 많은 호출이 될 것이다.
이는 호출 순서를 하나하나 보는 것이 의미가 없으므로 재귀 함수 자체를 이해하는 것이 중요하다는 것이다.

## 3. Recursive

이진 탐색 알고리즘의 재귀적 표현

이진탐색이란 탐색의 대상이 되는 자료들이 배열 안에 들어있다고 할 때, (정렬이 되어 있어야함)
arr[low]와 arr[high]값에 의거해 arr[mid(=(low+high)/2)] 값을 구하고 arr[mid]값과 key값을 비교한다.
low > high 가 될때까지 위의 과정을 반복하면서 구하고자 하는 값을 찾는다.

**5를 탐색하는 경우**



## 3. Recursive

그러면, 위의 이진 탐색을 재귀로 표현하기 위해 이진 탐색 알고리즘의 반복 패턴을 정리해보면,

1. 탐색 범위의 중앙에 목표 값이 저장되었는지
2. 저장되지 않았다면 탐색 범위를 반으로 줄여서 다시 탐색 시작
3. 탈출조건은 탐색 범위의 시작위치를 의미하는 first가 탐색 범위 끝을 의미하는 last보다 커지는 경우

코드로 표현하면 다음과 같다.

```java
public class Recursive {

  public static int BSearhRecur(int arr[], int first, int last, int target) {

    int mid;

    if (first > last)
      return -1;

    mid = (first + last) / 2;

    if (arr[mid] == target)
      return mid;

    else if (target < arr[mid])
      return BSearhRecur(arr, first, mid - 1, target);

    else
      return BSearhRecur(arr, mid + 1, last, target);

  }

```

```java
  public static void main(String[] args) {
    int[] arr = { 1, 3, 5, 7, 9 };
    int i;

    i = BSearhRecur(arr, 0, arr.length, 7);

    if (i == -1)
      System.out.println("탐색 실패");
    else
      System.out.println("타겟 저장 인덱스 " + i);

    i = BSearhRecur(arr, 0, arr.length, 4);

    if (i == -1)
      System.out.println("탐색 실패");
    else
      System.out.println("타겟 저장 인덱스 " + i);

  }
}
```

출력결과

```
타겟 저장 인덱스 3
탐색 실패
```

# 3. Recursive

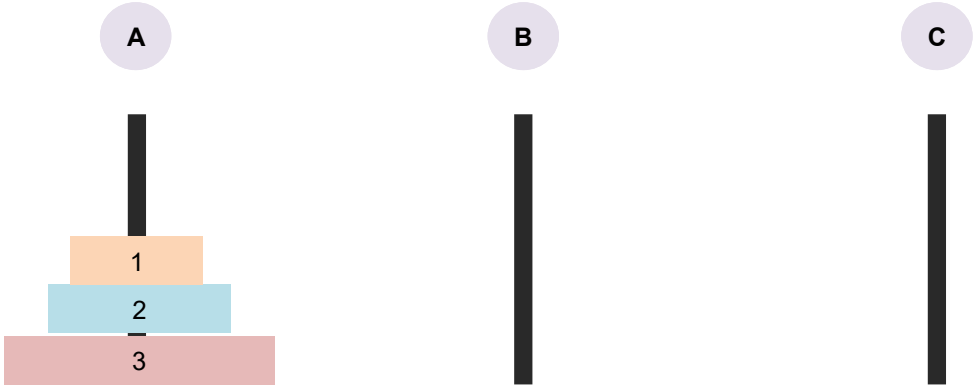
## 하노이탑

위에서 잠깐 언급한 것처럼 하노이 타워 문제는 재귀함수의 대표적인 예시입니다. 하노이 타워는 재귀함수를 쓰지 않고 접근을 한다면 쉽지 않은 문제가 될 것입니다.

바로 문제로 들어가보면 하노이 타워에는 세 개의 기둥과 이 기둥에 꽂을 수 있는 크기가 다양한 원반들이 있습니다.
한 기둥에 원반들이 작은 것이 위에 있도록 순서대로 쌓여 있는데, 2가지 조건을 만족시키면서 한 기둥에 꽂힌 원반들을 그 순서 그대로 다른 기둥으로 옮겨서 다시 쌓아야만 합니다.

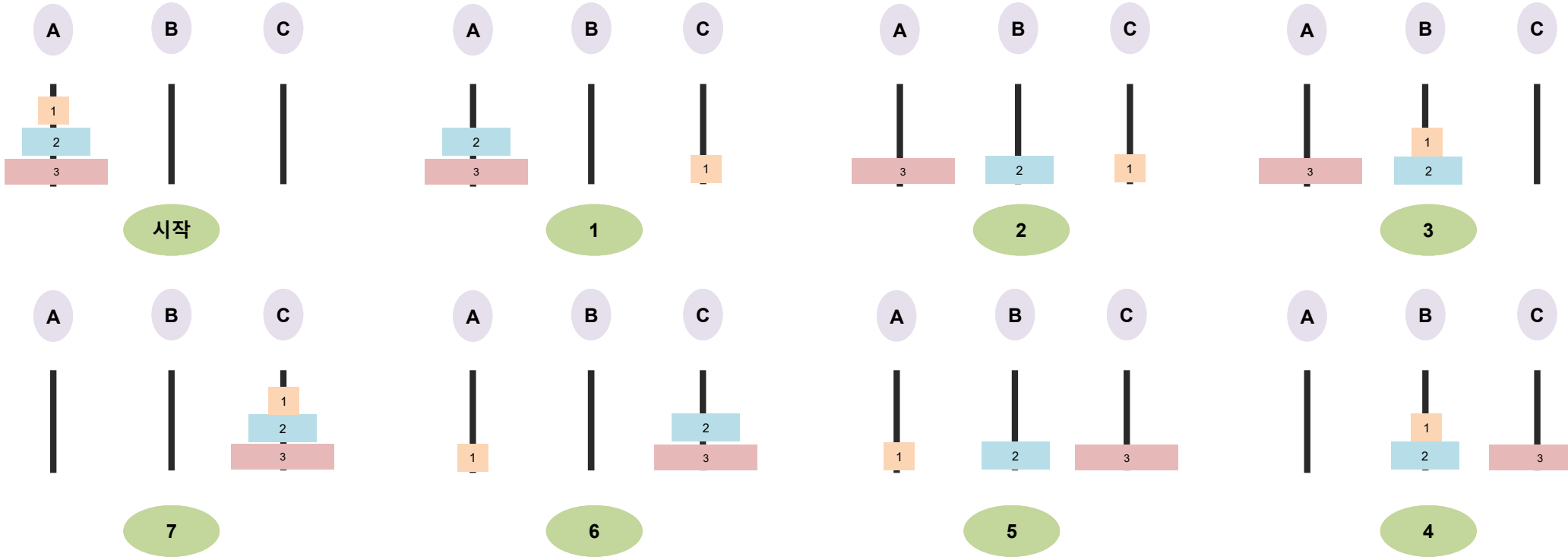
1. 한 번에 한 개의 원반만 옮길 수 있다.
2. 큰 원반이 작은 원반 위에 있어서는 안된다.

이 두가지 조건을 만족시켜야만 합니다.

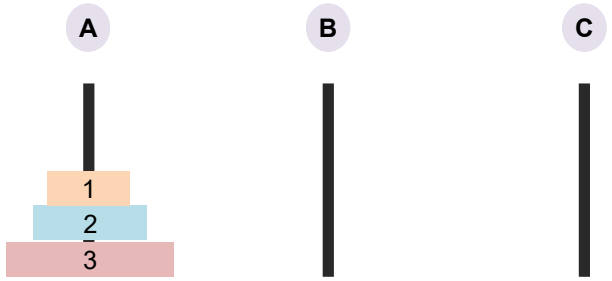


# 3. Recursive

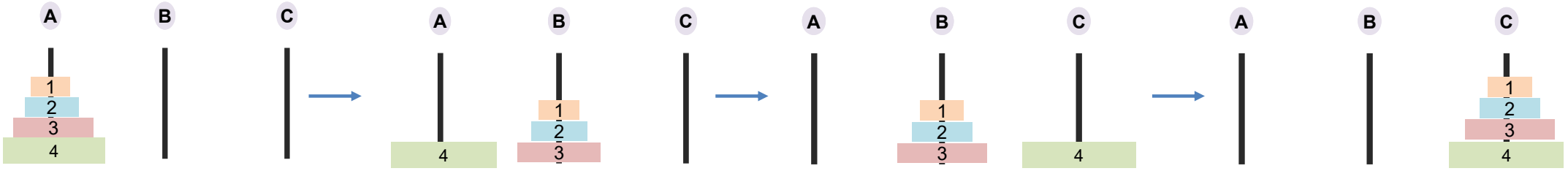
전 페이지에서 보듯 막대가 2개가 아닌 3개인 이유는 원반을 옮기는데 1, 2번 조건을 만족 해야 하기 때문입니다. 때문에 모든 원반을 막대 A에서 막대 C로 옮기기 위해서는 막대 B의 도움이 필요합니다.
그럼 문제 해결을 하기위한 아래 그림을 보면 3개밖에 없기 때문에 누구나 해결할 수 있지만, 원반이 7개 10개로 늘어나면 해결하기 까다롭습니다.
해결방법에는 차이가 없지만 과정을 더 많이 반복해야 할 뿐입니다.



# 3. Recursive




위의 그림에서 보듯이 3번 원반을 C에 가져다 놔야 하는데 위에 1번과 2번 원반이 있기 때문에 먼저 1번 2번 원반을 B에 가져다 놓으면 3번 원반을 C에 가져다 놓을 수 있다.
그러면 4개의 원반을 대상으로 생각해보자.




원반 전체를 막대C로 옮기기 위해서는 숫자 4가 적힌 원반을 막대C로 옮기는 것이 우선이니 1,2,3 원반을 막대 B로 옮겨야 한다.

그러면 원반을 옮기는 문제가 해결이 된다.
원반 1,2,3을 옮기는 과정은 전페이지에 나와 있다.

# 3. Recursive

다시 정리해보면,

1. 작은 원반 3개를(맨 아래를 제외한) A에서 B로 이동
2. 큰 원반(맨 아래의 원반) 1개를 A에서 C로 이동
3. 작은 원반(B로 옮긴 원반) 3개를 B에서 C로 이동

그러면 일반화 시켜서 막대 A에 꽂혀 있는 원반 n개를 막대 C로 옮긴다면

1. 작은 원반 n-1개를 A에서 B로 이동
2. 큰 원반 1개를 A에서 C로 이동
3. 작은 원반 n-1개를 B에서 C로 이동

이렇듯 원반 n개를 이동하는 문제는 원반 n-1개를 이동하는 문제로 세분화가 되고, 원반 n-1개를 옮기는 문제는 원반 n-2개를 이동하는 문제로 세분화가 됩니다.

그러면 백준의 11729 문제를 풀어보자.

## 3. Recursive

EX) 4_1

입력받은 두 양수를 x,n 변수에 할당하고 x의 n제곱을 재귀적으로 구현하라

## 3. Recursive

EX) 4_2

아래 삼각형 모양을 출력하는 메서드를 재귀적으로 구현하라

```
   *
  * *
 * * *
* * * *
```

## 3. Recursive

Homework

백준 사이트 10872, 10870, 2447, 11729 풀기.

10872. 팩토리얼

10870. 피보나치수 5

2447. 별 찍기 10

11729. 하노이탑 이동 순서