

# Recursion

---



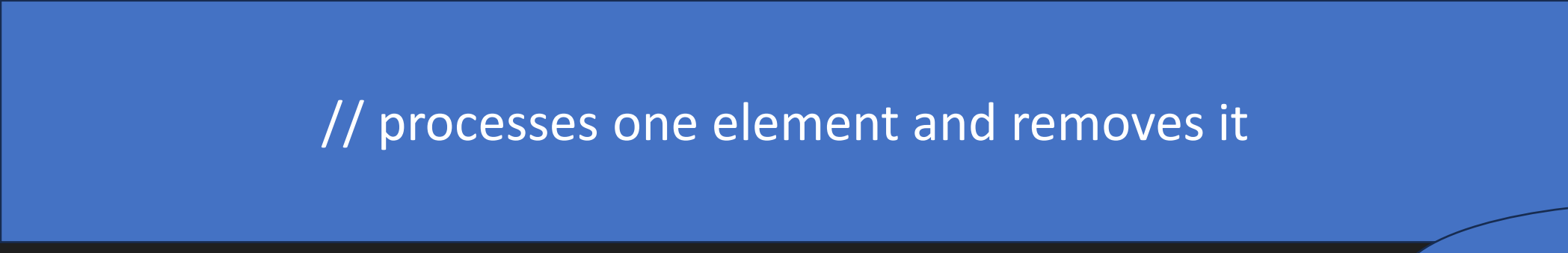
06016415 Functional Programming



- The Need for Recursion
- Recursive Algorithm
  - Three Key Principles
- @tailrec : Tail Recursion
- Recursion on Functional List

- **Loops** คือ การเขียน code ให้ทำซ้ำ ซึ่งไม่สอดคล้องกับแนวทางแบบ FP ที่เน้น Pure function
- ใน FP จึงมีกลไกแบบซ้อนกัน (nest expression) เพื่อทดแทน Loops คือ **Recursion**
- **Recursion** มีบทบาทสำคัญต่อ FP โดยทำให้สร้าง Pure function แต่ยังสามารถเขียน code ให้ทำซ้ำ โดยไม่ต้องมีตัวแปรสำหรับการทำซ้ำ (mutable loop counters)
- ดังนั้น การสร้าง Recursive Algorithm ให้สามารถการเขียนโปรแกรมเรียกซ้ำโดยตรง ซึ่งใช้กับโครงสร้างแบบเรียกซ้ำ เช่น list tree เป็นต้น เมื่อเปรียบเทียบกับ การเขียนโปรแกรมแบบทำซ้ำแบบเดิม (loop-based)
- Tail recursive functions จึงถูกนำมาใช้กับ FP เพื่อให้มีประสิทธิภาพมากขึ้นด้วย

## The Need for Recursion

- พิจารณาตัวอย่าง

```
def processOne[A](collection: ):  =  
      
    // processes one element and removes it
```

```
def processCollection[A](collection: ):  =  
    while collection.nonEmpty do  
        println(processOne(collection))
```

```
def processOne[A](collection:           ): Unit =  
  // processes one element and removes it
```

```
def processCollection2[A](collection: List[A]): Unit =  
  var remaining = collection  
  while remaining.nonEmpty do  
    println(remaining)  
    remaining = processOne(remaining)
```

```
def processOne[A](collection:           ): Unit =  
  // processes one element and removes it
```

```
def processCollection3[A](collection: List[A]): Unit =  
  println(collection)  
  if collection.nonEmpty then processCollection3(processOne(collection))
```

$$F(n) = \begin{cases} F(n-1) \cdot n & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases}$$

---

**Algorithm** Factorial( $n$ )

---

1: Input: A non-negative integer  $n$

```
2: |  
3: | def factorial2(n: Int): Int =  
4: |   if n == 0 then 1 else n * factorial2(n - 1)
```

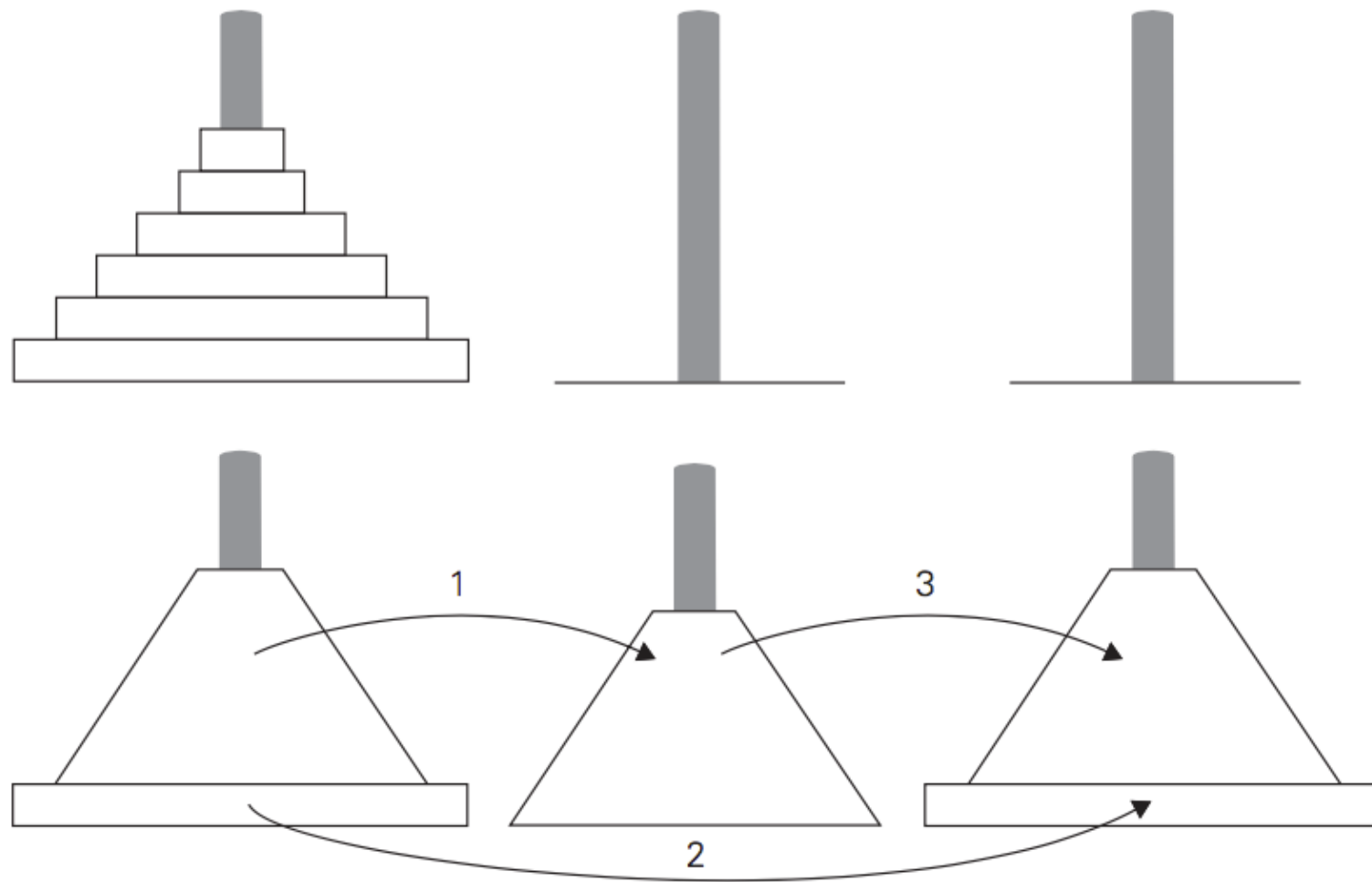
5: | return 1

6: | else

7: | return Factorial( $n - 1$ )\* $n$

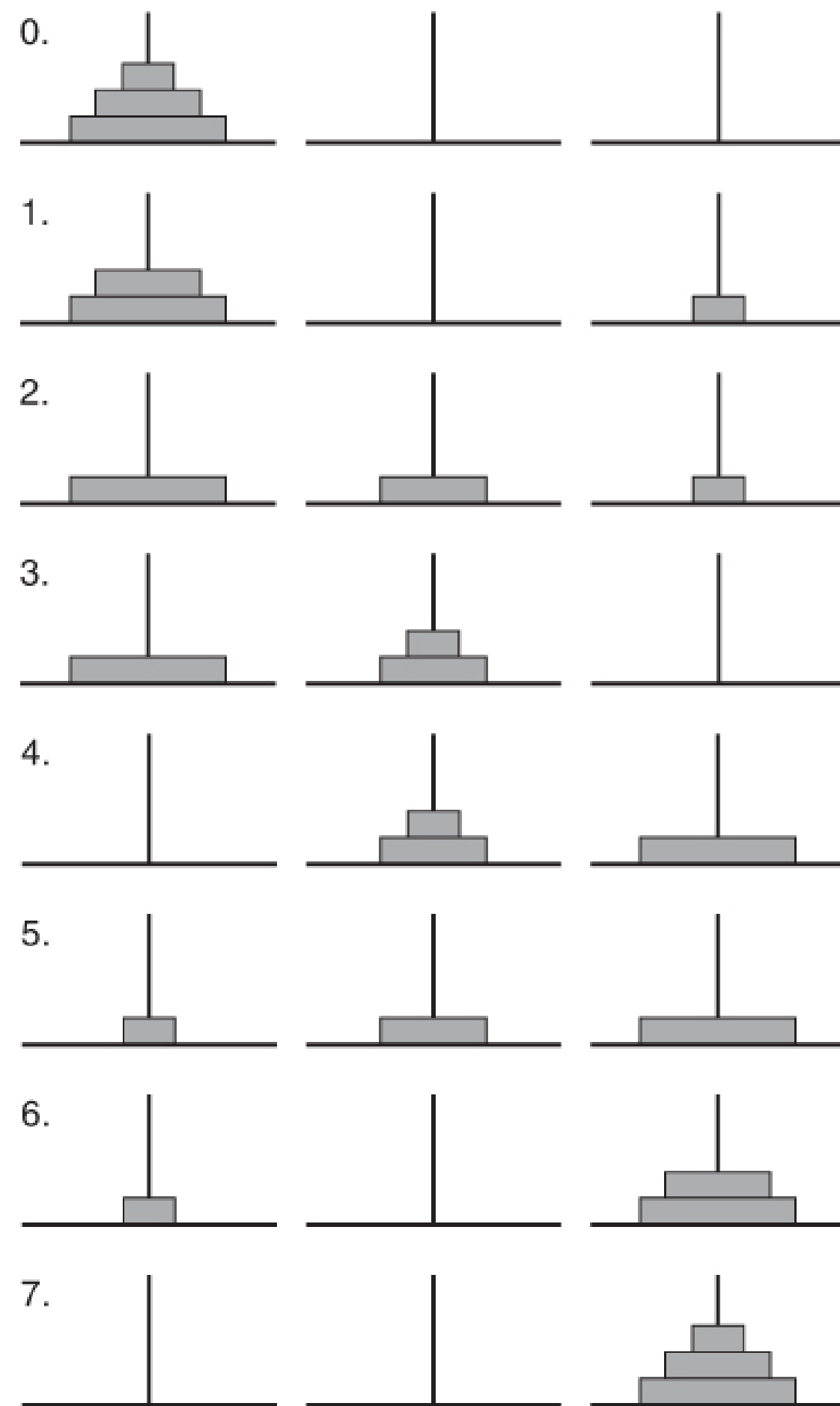
---

## EX: Tower of Hanoi Puzzle



- Move all disks from left peg to right peg one-by-one
  - Middle peg used as auxiliary
  - Forbidden to place a larger disk on a smaller one.





```
def hanoi[A](n: Int, from: A, middle: A, to: A): Unit =
  if n > 0 then
    hanoi(n - 1, from, to, middle)
    println(s"$from -> $to")
    hanoi(n - 1, middle, from, to)
```

**hanoi(3, 'L', 'M', 'R')**

1. At least one branch of the computation should not be recursive.
2. All the subproblems solved recursively must be smaller than the original problem in some way.
3. Your focus as a programmer should be on using the solutions to the smaller problems, not on how these smaller problems are themselves solved by the recursive calls

At least one branch should not be recursive

```
// DON'T DO THIS!  
def last[A](list: List[A]): A = list match  
  case Nil          => throw new Exception("last(empty)")  
  case _ :: tail    => last(tail)
```

All the subproblems must be smaller than the original

---

**Algorithm** MergeSort( $A$ )

---

- 1: Input: An array of  $A[l \dots r]$  of orderable elements
  - 2: Output: Array  $A[l \dots r]$  sorted in non-decreasing order
  - 3:
  - 4: **if**  $A$  has more than one element **then**
  - 5:      $m \leftarrow \lfloor (l + r) / 2 \rfloor$
  - 6:     MergeSort( $A[l \dots m]$ )                      $\triangleright$  Sort 1<sup>st</sup> half of  $A$
  - 7:     MergeSort( $A[m + 1 \dots r]$ )                $\triangleright$  Sort 2<sup>nd</sup> half of  $A$
  - 8:     Merge( $A, l, m, r$ )                          $\triangleright$  Merge two halves
- 

```
// DON'T DO THIS!
def mergeSort1(list: List[Int]): List[Int] =
  if list.isEmpty then list
  else
    val (left, right) = list.splitAt(list.length / 2) // split in the middle
    merge(mergeSort1(left), mergeSort1(right))
```

```
//pseudo code  
function f(x,y,z) {  
    <code>  
    return g(t,u,v)  
}
```

```
@tailrec def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)
```

```
import scala.util.control.TailCalls.*  
def isEven(xs: Seq[Int]): TailRec[Boolean] =  
    if xs.isEmpty then done(true) else tailcall(isOdd(xs.tail))  
  
def isOdd(xs: Seq[Int]): TailRec[Boolean] =  
    if xs.isEmpty then done(false) else tailcall(isEven(xs.tail))
```

```
// DON'T DO THIS!  
def last[A](list: List[A]): A = list match  
  | case Nil          => throw new Exception("last(empty)")  
  | case _ :: tail    => last(tail)  
  
@tailrec  
✓ def last2[A](list: List[A]): A = list match  
  | case Nil          => throw new Exception("last(empty)")
```

Check Tail is Empty or Not

```
//Recursion on Multiple/Nested Lists
def concat[A](list1: List[A], list2: List[A]): List[A] = list1 match
  case Nil => list2
  case head1 :: tail1 => head1 :: concat(tail1, list2)

def append[A](list: List[A], value: A): List[A] = ???

def flatten[A](list: List[List[A]]): List[A] = list match
  case Nil => Nil
  case head :: tail => ???
```

```

//Recursion on Sublists Other Than the Tail
// DON'T DO THIS!
def group1[A](list: List[A], k: Int): List[List[A]] =
  take(list, k) :: group1(drop(list, k), k)

def isEmpty[A](list: List[A]): Boolean = list match
  | _      ???

✓ def group[A](list: List[A], k: Int): List[List[A]] =
  if isEmpty(list) then Nil
  else
    val (first, more) = splitAt(list, k)
    first :: ???

```



- Loops are a programming language mechanism that implements repetition.
- A strict functional programming style relies on composition of pure functions, making loops useless.
- Recursive functions are functions that trigger calls to themselves.
- Recursive functions fit algorithms that recursively decompose a given problem into smaller problems of the same type.
- A recursive algorithm combines a strategy for using solutions to smaller instances of the same problem with one or more trivial (non-recursive) cases.