Recursive In C++

คำนิยาม Recursion คือเทคนิคในการเขียนโปรแกรมที่ฟังก์ชันจะเรียกใช้ตัวเอง เพื่อแก้ปัญหา ซึ่งการใช้เทคนิคนี้ทำให้ปัญหาที่มีโครงสร้างซ้ำซ้อนสามารถแก้ไขได้ด้วยวิธีที่ดูเรียบง่ายและตรงไปตรงมา เช่น การคำนวณค่าแฟกทอเรียลหรือหาลำดับฟีโบนัชชี

ทำไมต้อง Recursion

1 **แก้ปัญหาซับซ้อนได้อย่างมีประสิทธิภาพ**
การเรียกฟังก์ชันตัวเองเหมาะสำหรับปัญหาที่มีลักษณะซ้ำซ้อนและสามารถแบ่งออกเป็นปัญหาย่อยๆ ได้

2 **ช่วยให้โค้ดกระชับและเข้าใจง่าย**
การเรียกฟังก์ชันตัวเองช่วยให้สามารถแยกปัญหาออกเป็นส่วนย่อยๆ ได้อย่างเป็นระบบ ทำให้โค้ดอ่านและบำรุงรักษาง่ายขึ้น

3 **เพิ่มความยืดหยุ่นในการแก้ปัญหา**
การเรียกฟังก์ชันตัวเองช่วยให้สามารถปรับเปลี่ยนหรือขยายขอบเขตของปัญหาได้โดยไม่ต้องเปลี่ยนแปลงโครงสร้างพื้นฐาน

การเรียกฟังก์ชันตัวเองอย่างง่ายในภาษา C++

1 **กำหนดจุดจบ**

คือการตรวจสอบเงื่อนไขที่จะสิ้นสุดการเรียกฟังก์ชันตัวเอง

2 **การเรียกฟังก์ชันตัวเอง**

เมื่อเงื่อนไขยังไม่เป็นจริง ฟังก์ชันจะเรียกตัวเองต่อไป

3 **การประมวลผลย้อนกลับ**

หลังจากการเรียกฟังก์ชันเสร็จสิ้น จะมีการประมวลผลจากด้านล่างขึ้นไป

ประเภทของปัญหาที่เหมาะกับการใช้การเรียกฟังก์ชันตัวเอง

1 **ปัญหาที่มีลักษณะซ้ำๆ**

เช่น การคำนวณลำดับฟิโบนัชชี การหาค่าแฟกทอเรียล การเปลี่ยนรูปแบบ

2 **ปัญหาที่เกี่ยวข้องกับการแบ่งและแยก**

เช่น การค้นหาแฟ้มใน directory ต่างระดับ การแบ่งปัญหาออกเป็นส่วนย่อย

3 **ปัญหาที่มีความสลับซับซ้อน**

เช่น การแก้ไขปริศนา หรือ ตัวอักษร การหาเส้นทางที่ดีที่สุด

ข้อดีและข้อเสียของการใช้การเรียกฟังก์ชันตัวเอง (Recursive)

ข้อดี

ง่ายในการเขียนและอ่านโค้ด, เหมาะสำหรับแก้ปัญหาที่มีลักษณะซ้ำซ้อน, สามารถปรับแต่งได้ง่าย

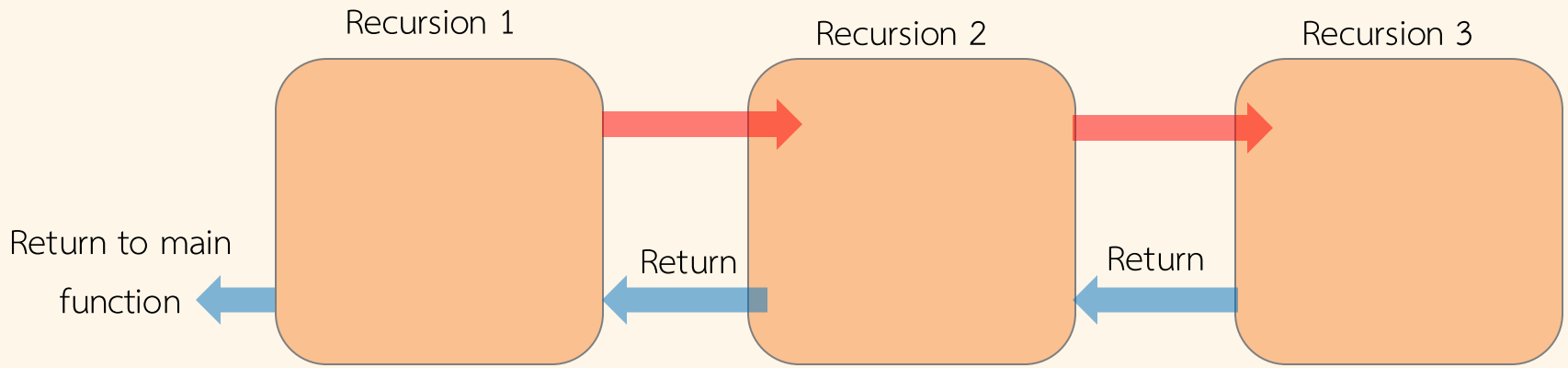
ข้อเสีย

อาจมีปัญหาด้านประสิทธิภาพและการใช้หน่วยความจำ, ความเสี่ยงของการสร้างกระบวนการไม่จบสิ้น

การเรียกตัวฟังก์ชัน

1. **เริ่มต้น**
   ฟังก์ชันถูกเรียกใช้ครั้งแรก
2. **เรียกฟังก์ชันใหม่**
   ฟังก์ชันเรียกตัวเองซ้ำเพื่อแก้ปัญหาย่อย
3. **ทำงานย้อนกลับ**
   เมื่อแก้ปัญหาย่อยเสร็จ จะประมวลผลย้อนกลับ
4. **จบการทำงาน**
   เมื่อถึงเงื่อนไขการจบ ฟังก์ชันหลักจะได้ผลลัพธ์ที่ต้องการ

ในการทำงานของ Recursion ทุกครั้งที่ฟังก์ชันถูกเรียกซ้ำ จะถูกเก็บไว้ในโครงสร้างข้อมูลที่เรียกว่า **Stack** ซึ่งเป็นหน่วยความจำแบบ LIFO (Last In, First Out) ซึ่งหมายความว่าฟังก์ชันที่ถูกเรียกเข้ามาล่าสุดจะถูกคืนค่าก่อน



เมื่อฟังก์ชัน Recursive ถูกเรียก จะมีการสร้าง **stack frame** ใหม่ขึ้นมาในหน่วยความจำ โดย stack frame นี้จะเก็บข้อมูลที่จำเป็นต่อการทำงานของฟังก์ชัน เช่น ค่าของพารามิเตอร์ ตัวแปรท้องถิ่น และตำแหน่งที่ต้องกลับมาทำงานหลังจากการเรียกฟังก์ชันเสร็จสิ้น

ตัวอย่างของการใช้ Stack ใน Recursion คือการคำนวณ **แฟกทอเรียล (Factorial)**

Factorial Formula

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

$$1! = 1$$
$$2! = 2 \times 1 = 2$$
$$3! = 3 \times 2 \times 1 = 6$$
$$4! = 4 \times 3 \times 2 \times 1 = 24$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

ตัวอย่างเพื่อทำความเข้าใจการใช้ Recursive

ค่าที่ส่งเข้า	ขั้นตอนการคำนวณ	ค่าที่ได้เป็นผลลัพธ์
5	factorial(5) = 5 x factorial(4)	120
4	factorial(4) = 4 x factorial(3)	24
3	factorial(3) = 3 x factorial(2)	6
2	factorial(2) = 2 x factorial(1)	2
1	factorial(1) = 1	1

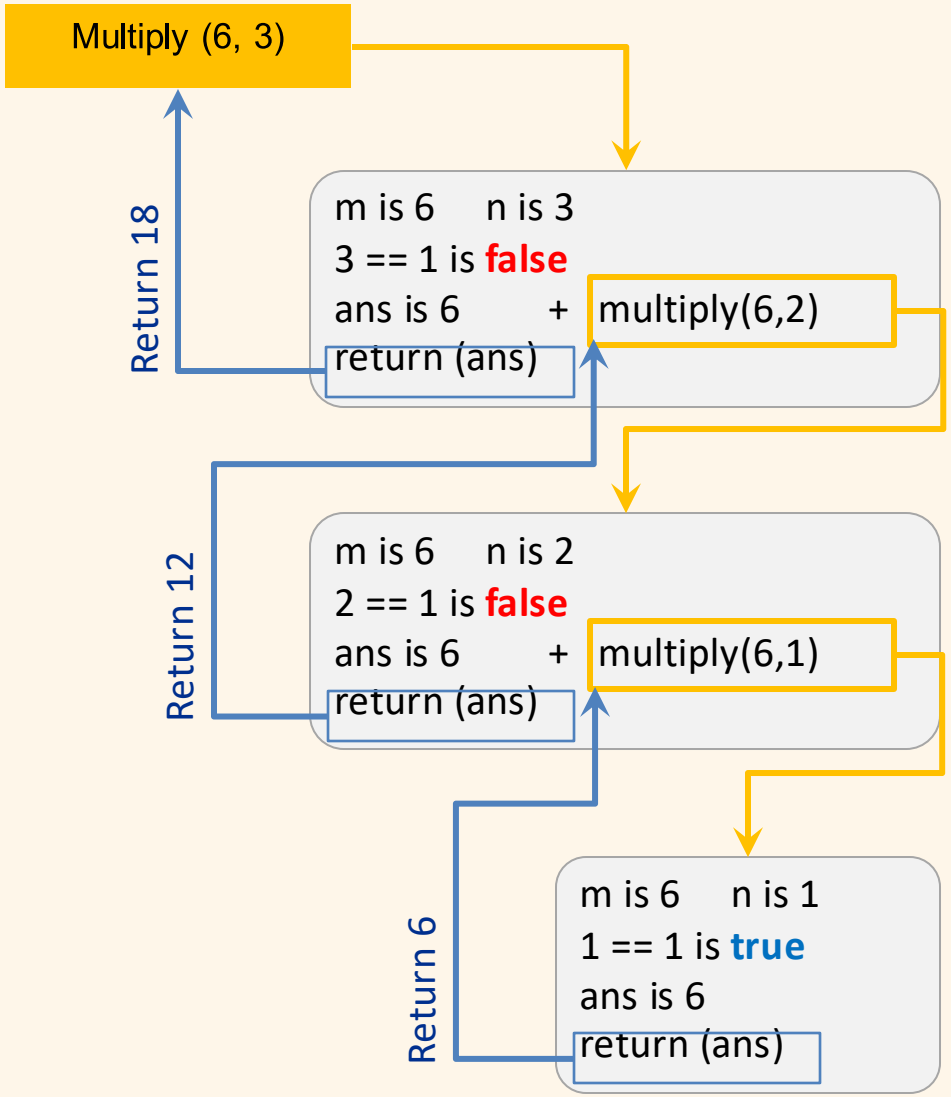
ตัวอย่างการใช้ Recursive ในการหาผลรวมตั้งแต่ 1 – 10

```cpp
#include <iostream>

int sum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sum(n - 1);
    }
}

int main(){
    int result = sum(10);
    std::cout << "Sum 1-10 total: " << result << std::endl;
    return 0;
}
```

ตัวอย่างการใช้ Recursive ในการคูณเลข




```
int multiply(int m, int n) {
        int ans;
        if (n == 1){
                ans = m;
        } else {
                ans = m + multiply(m, n - 1);
        }
        return ans;
}
```

Terminating Condition

- Recursive Function จะต้องมีเงื่อนไขหยุดทำงานอย่างน้อย 1 เงื่อนไขเสมอ โดยปกติจะเป็นเงื่อนไขของกรณีพื้นฐาน Simple Case

  Ex.
  ```
  if (n == 1)
      return m;
  ```

- หากไม่มี Terminating Condition ฟังก์ชันอาจทำงานไม่มีที่สิ้นสุด

ตัวอย่างการใช้ Recursive ในการคูณเลข

```cpp
#include <iostream>

int multiply(int m, int n) {
    int ans;
    std::cout << "Entering multiply with m = " << m << ", n = " << n << "\n";
    if (n == 1) {
        ans = m;
    } else {
        ans = m + multiply(m, n - 1);
    }
    return ans;
}

int main() {
    Int m, n;
    std::cout << "Enter two numbers to find their product: ";
    std::cin >> m >> n;

    Int result = multiply(m, n);
    std::cout << "(" << m << " x " << n << ") = " << result << "\n";
    return 0;
}
```

```
Enter two numbers to find their product: 2 4
Entering multiply with m = 2, n = 4
Entering multiply with m = 2, n = 3
Entering multiply with m = 2, n = 2
Entering multiply with m = 2, n = 1
(2 x 4) = 8
```

Recursive Function : Factorial

- คือการเขียนฟังก์ชันหาผลรวมจากการคูณกันของจำนวนเต็มทุกตัวตั้งแต่ 1 ถึง n

  Ex. 5! = 5 x 4 x 3 x 2 x 1 หรือ 5 x (4)!

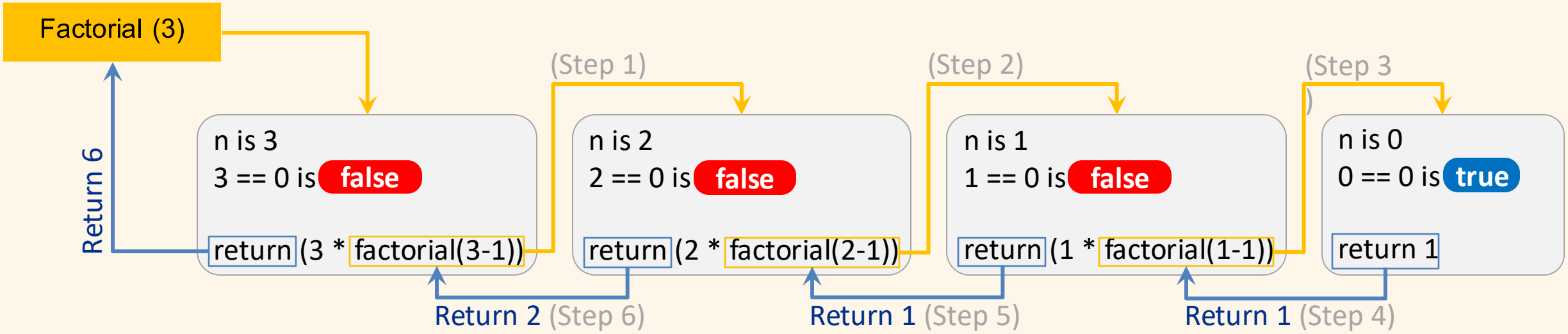
- Simple Case คือ 0! = 1

```
if (n == 0)
    return 1;
```

- สามารถเขียนในรูปแบบความสัมพันธ์เวียนเกิดได้ดังนี้ n! = n * (n-1)!

Recursive Function : Factorial

```
int factorial(int n) {
        if (n == 0)
                return 1;
        else
                return (n * factorial(n-1));
}
```



Recursive Function : Fibonacci

- คือลำดับ จำนวนต่างๆ ที่อยู่ในลำดับจำนวนเต็มดังนี้

ลำดับ	0	1	2	3	4	5	6	7	8	9	10	n
Fibonacci	0	1	1	2	3	5	8	13	21	31	55	…

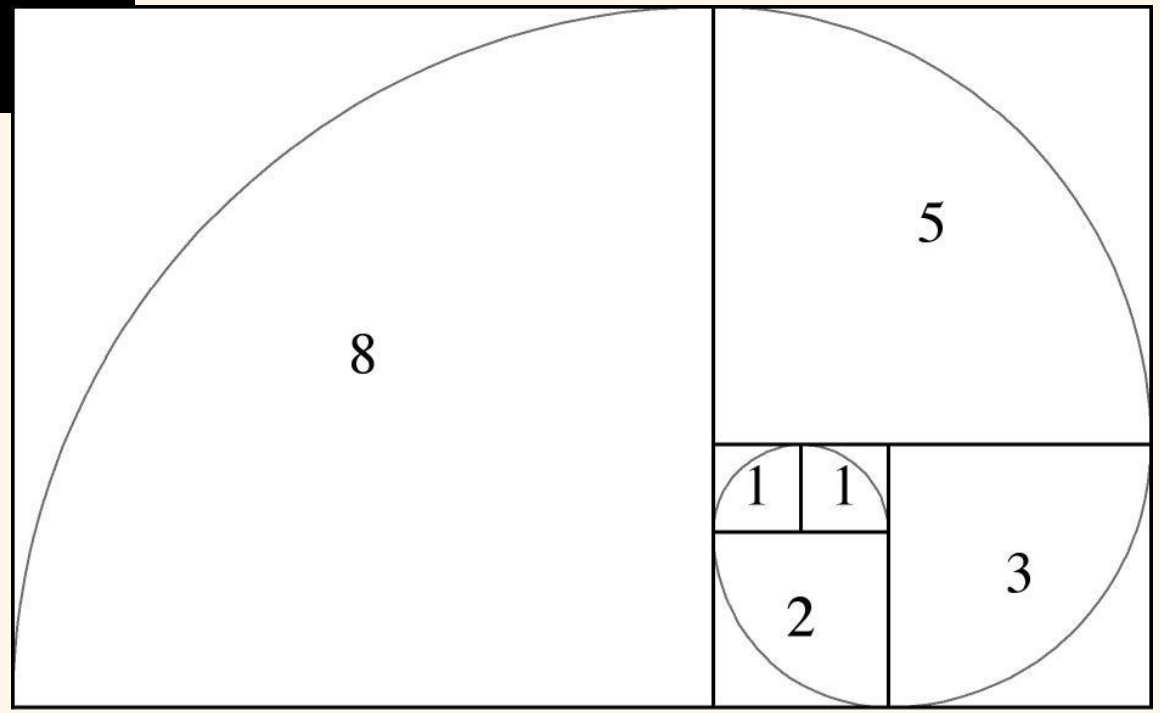
- Simple Case คือ $F_0 = 0$; $F_1 = 1$

```
if (n == 0 || n == 1)
    return n;
```

- สามารถเขียนในรูปแบบความสัมพันธ์เวียนเกิดได้ดังนี้ $F_n = F_{n-1} + F_{n-2}$

Recursive Function : Fibonacci

```
int fibo(int n){
        if(n == 0 || n == 1 ){
                return n;
        } else {
                return fibo(n-1) + fibo(n-2);
        }
}
```

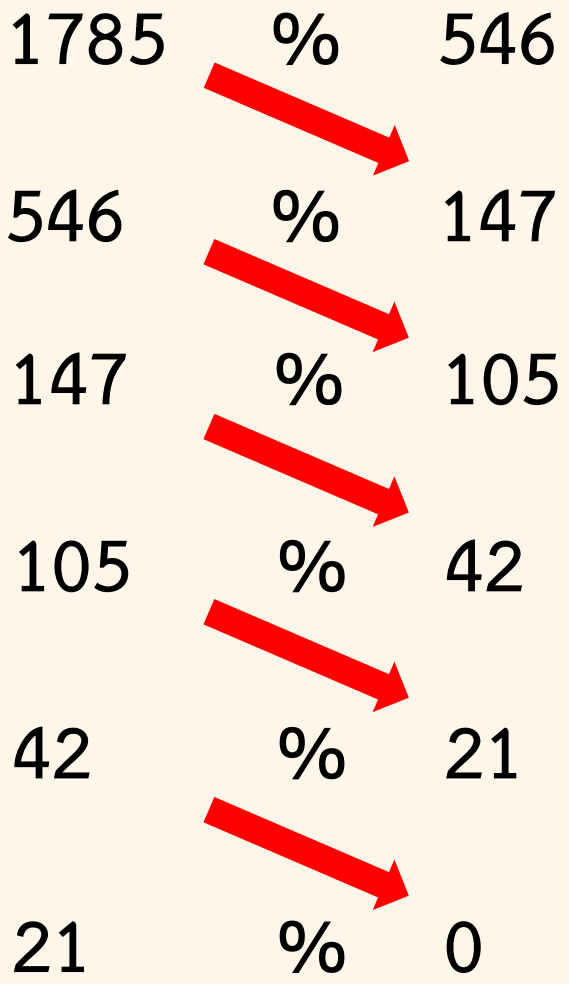


Recursive Function : GCD (Greatest Common Divisor)

- คือการหาจำนวนเต็มที่มากที่สุดที่สามารถนำไปหารจำนวนตั้งแต่ 2 จำนวนขึ้นไปได้ลงตัว หรือเรียกว่า หารร่วมมาก (ห.ร.ม.)
- ขั้นตอนการคำนวณแบบยูคลิด (Euclid) คือขั้นตอนวิธีการหารร่วมมากของจำนวนนับสองจำนวนที่มีค่ามากได้อย่างรวดเร็ว ตัวอย่างการหา ห.ร.ม. ของ 1785, 546

```
int gcd(int a, int b) {
        if (b == 0) {
                return a;
        }
        return gcd(b, a % b);
}
```

ขั้นตอนการหา ค.ร.น.



Towers of Hanoi

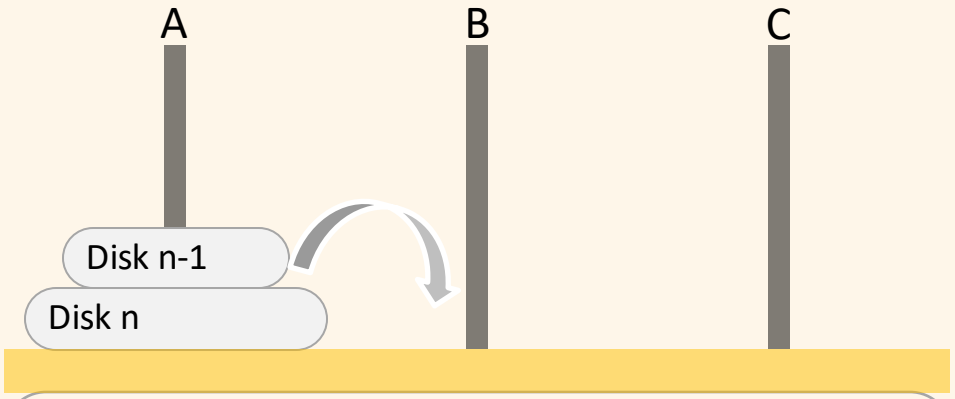
- ทาวเวอร์ออฟฮานอย เป็นเกมคณิตศาสตร์ ประกอบด้วยหมุด 3 แห่ง และจานกลมแบบขนาดต่างๆ ที่มีรูตรงกลางสำหรับใส่หมุด
- เป้าหมายของเกมคือ พยายามย้ายกองจานทั้งหมดจากหมุดเริ่มต้นไปยังหมุดปลายทาง โดยมีกติกา คือ **ไม่สามารถวางจานที่มีขนาดใหญ่ไว้บนจานที่มีขนาดเล็กกว่าได้**

Towers of Hanoi

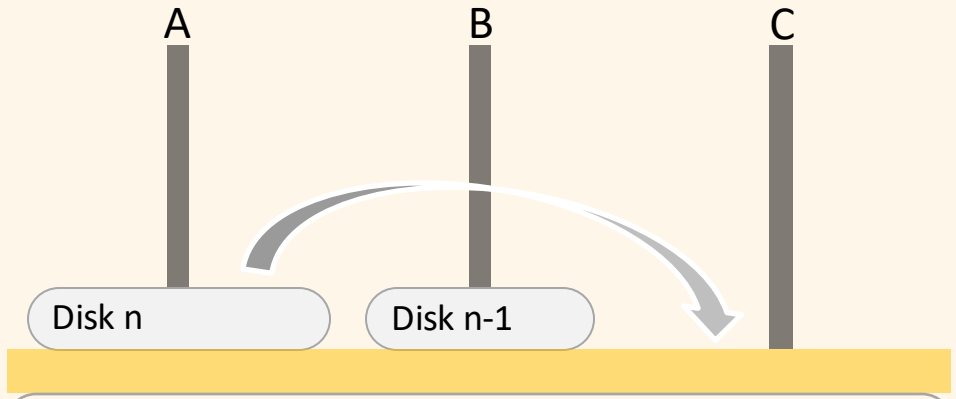
ขั้นตอน Recursive

- กำหนดชื่อหมุดทั้ง 3 เป็น A, B, C
- สมมุติมีจานทั้งหมด n ใบ
- กำหนดให้จานใบเล็กที่สุดคือ Disk 1 ไปจนถึงจานที่ใหญ่ที่สุดคือ Disk n
- ย้ายจาน Disk n-1 จาก A ไป B ก่อน จะเหลือจาน Disk n เพียงใบเดียวที่ A
- ย้ายจาน Disk n จาก A ไป C
- ย้ายจาน Disk n-1 จาก B ไป C

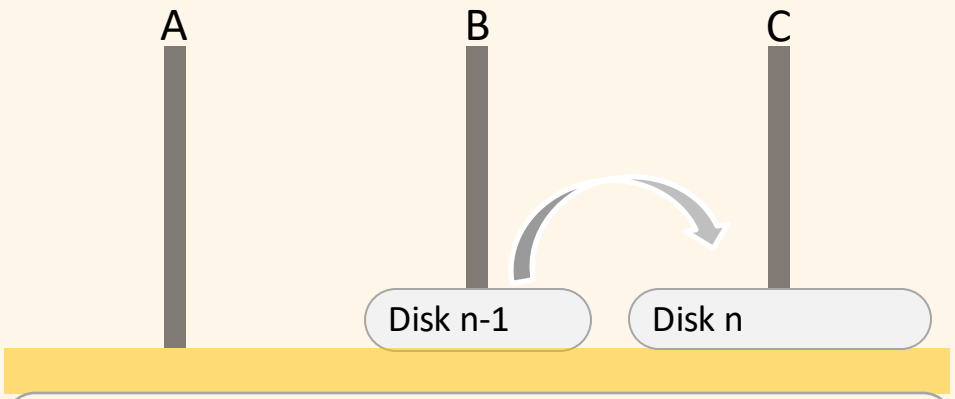
ตัวอย่าง Recursive Function : Towers of Hanoi



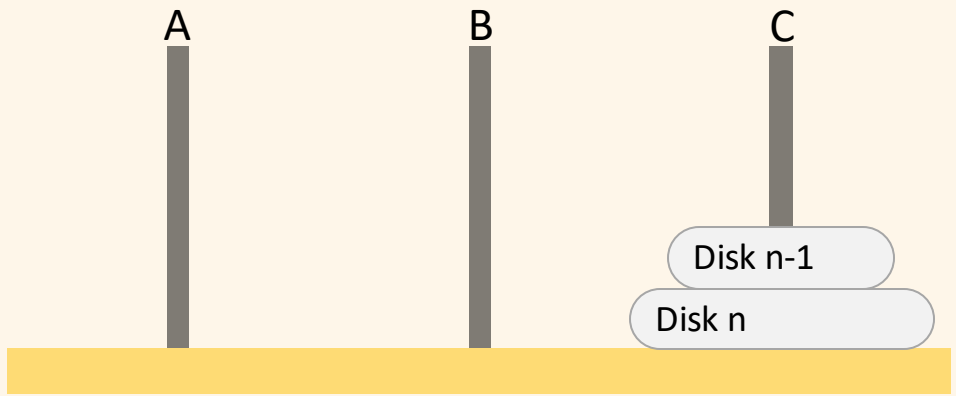
1) ย้ายจาน Disk n-1 จาก A ไป B ก่อน จะเหลือจาน Disk n เพียงใบเดียวที่ A



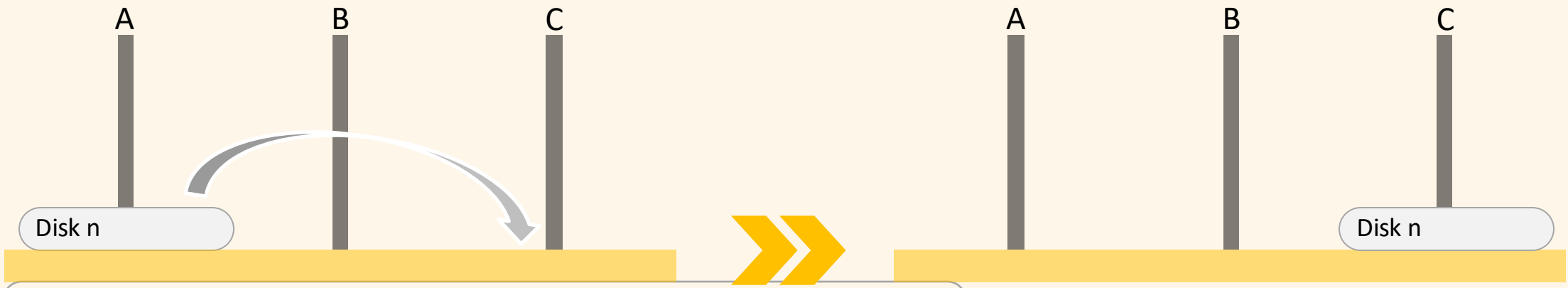
2) ย้ายจาน Disk n จาก A ไป C



3) ย้ายจาน Disk n-1 จาก B ไป C

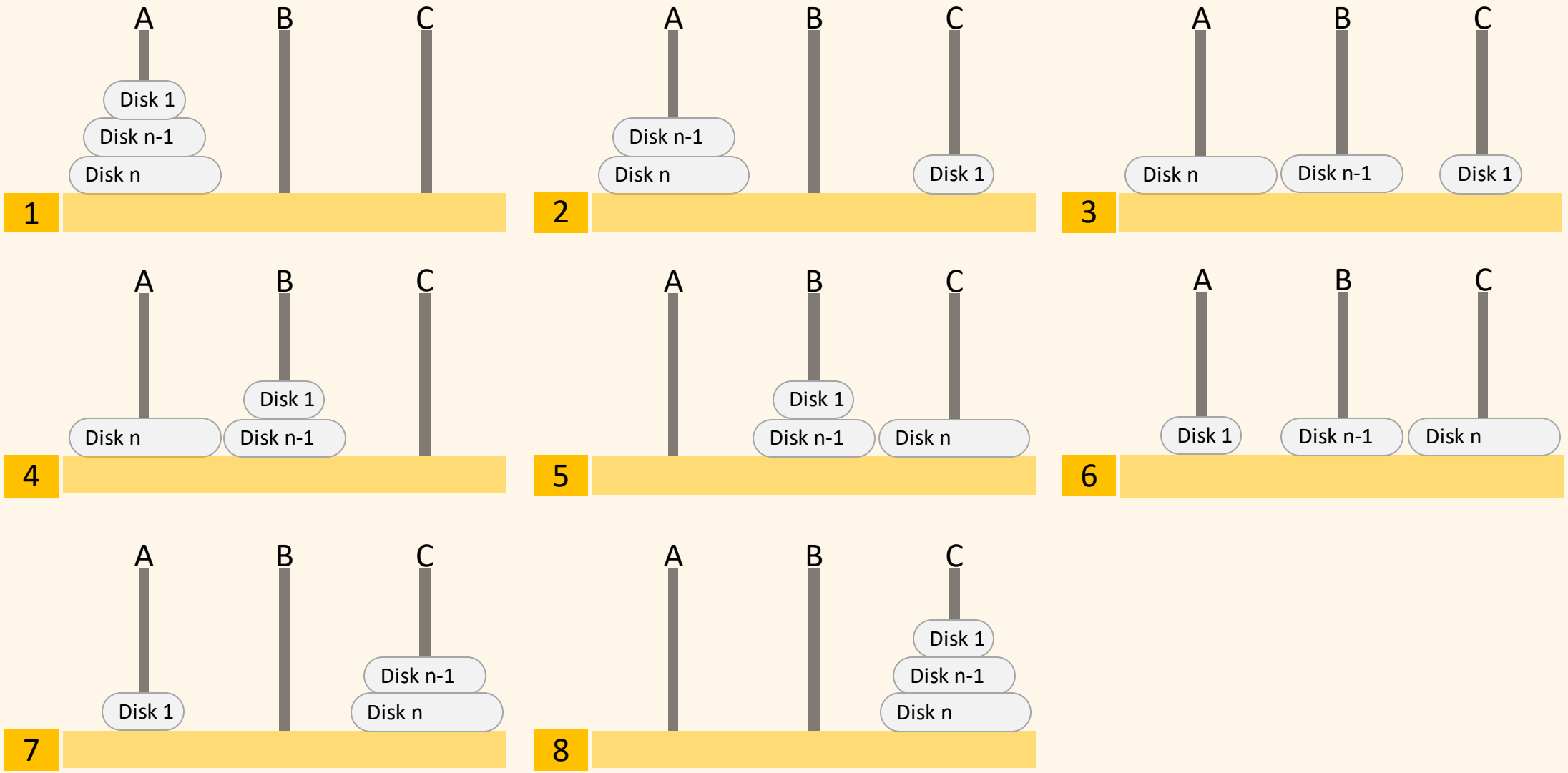


ตัวอย่าง Recursive Function : Towers of Hanoi



กรณีที่มีจานเพียง 1 ใบ สามารถย้ายจาน Disk n จาก A ไป C ได้เลย

ตัวอย่าง Recursive Function : Towers of Hanoi



จงสร้างโปรแกรมแสดงรูป * ตามจำนวน n แถว

จงสร้างโปรแกรมหาค่าผลคูณของ 1-n จนกว่าจะกด Enter

จงเขียนโปรแกรมหาค่าตัวเลขเฉลี่ย n ตัวใน array

จงสร้างโปรแกรมสำหรับหาค่า Factorial

จงสร้างโปรแกรมสำหรับหาค่า Fibonacci

จงสร้างโปรแกรมสำหรับเลขยกกำลัง

จงเขียนโปรแกรมหา ค.ร.น.

จงเขียนโปรแกรมแสดงวิธีการย้ายจานของ Tower of Hanoi