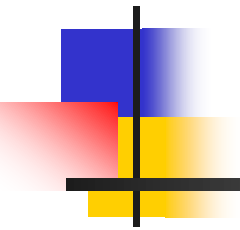


# Chapter 0

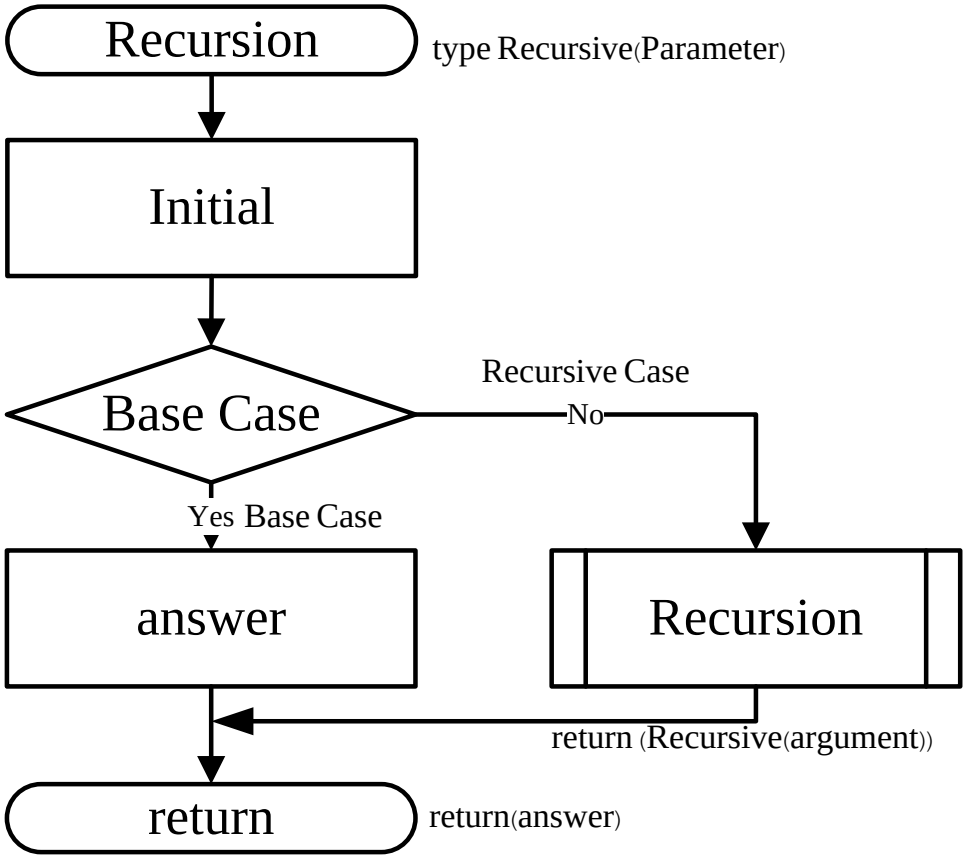
# Recursion Algorithms

# Iteration V.S. Recursion

- การวนซ้ำ (Iteration) เป็นการแก้ปัญหาโดยการวนรอบ(loop) ทำซ้ำหลายๆ ครั้งจนกว่าจะครบเงื่อนไขตามที่ต้องการ
  - ใช้คำสั่งประเภท for (), while() , do()
  - ต้องมีเงื่อนไขที่จะหลุดออกจากการวนรอบ
  - ต้องมีการเปลี่ยนแปลงตัวแปรที่ใช้เป็นเงื่อนไขเพื่อหลุดจากวนรอบ
- การเรียกซ้ำ (Recursion) เป็นการแก้ปัญหาโดยการแตกปัญหาออกเป็นปัญหาย่อยๆ ที่เกี่ยวข้อง ซึ่งแต่ละปัญหาจะมีลักษณะการแก้ปัญหาเหมือนปัญหาเดิม จนกว่าจะหาคำตอบของปัญหาย่อยได้ จึงย้อนกลับไปแก้ปัญหาเดิมที่ใหญ่กว่า
  - ใช้คำสั่งประเภท if () else ... เพื่อตรวจสอบคำตอบและตัดสินใจเรียกซ้ำ
  - ต้องสร้างเป็นฟังก์ชันที่มีพารามิเตอร์ และมีการเรียกใช้ตัวเอง
  - ต้องมีการเปลี่ยนแปลงค่าพารามิเตอร์ในการเรียกซ้ำ
  - ต้องมีคำตอบของการเรียกซ้ำในบางค่าของพารามิเตอร์
  - ในการเรียกซ้ำแต่ละครั้ง จะมีการเก็บสถานะของการทำงานที่ค้างอยู่ก่อนการเรียกซ้ำ (ฟังก์ชันยังทำงานไม่เสร็จ) ไว้ในหน่วยความจำแสต็ก(stack) แล้วเรียกซ้ำ ไปยังฟังก์ชันเดิมจนกว่าจะทำงานเสร็จ(return) แล้วจึงเรียกสถานะเดิมที่ทำค้างอยู่จากแสต็ก เพื่อทำงานเดิมต่อจนเสร็จ
  - ทำให้ประสิทธิภาพของโปรแกรมลดลง เมื่อเทียบกับการวนรอบธรรมดา
  - เหมาะสำหรับปัญหาการวนรอบที่ซับซ้อนหลายชั้น หรือไม่สามารถสร้างเงื่อนไขเพื่อควบคุมการหาคำตอบด้วยการวนรอบได้

# Recursion Problem

- ปัญหาที่สามารถแก้ด้วยการเรียกซ้ำ จะต้องมีองค์ประกอบอย่างน้อย 2 ส่วน
  1. **Base case** เป็นส่วนที่สรุปคำตอบได้
  2. **Recursive case** เป็นส่วนที่ยังเป็นปัญหาลักษณะเดิมแต่เปลี่ยนขอบเขต
- อาจมี **Base case** หรือ **Recursive case** หลายๆ ชุดได้




```
..... recursion (.....)
{ ....................
  if (Base Case)
    {  ................
       Set  answer & return ;
    }
  else /* recursive case */
    {  ..................
       Call recursion and return;
    }
}
```

# Iteration V.S. Recursion

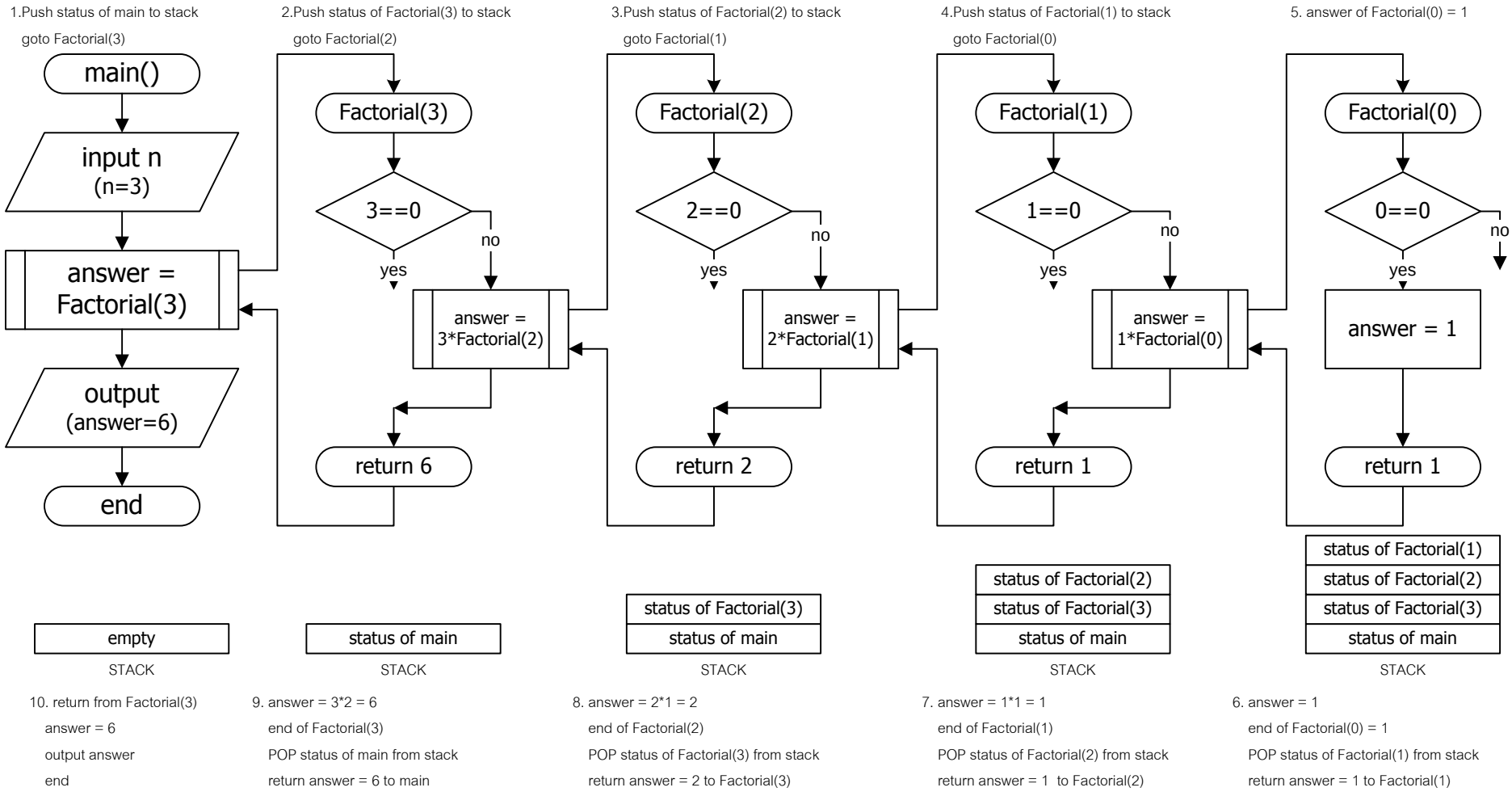
- ตัวอย่างการแก้ปัญหาของการวนรอบ และ เรียกซ้ำ
  - หาคำตอบของ n! = n*n-1*n-2*n-3*....*2*1 //มองแบบวนรอบ
  - n! = n * (n-1)! //มองแบบรีเคอร์ชัน

  เมื่อ n เป็นจำนวนเต็มบวก และ 0! = 1

| Iteration Problem | Recursion Problem |
|---|---|
| **แก้ปัญหาโดยใช้วนรอบ**<br>`long Factorial (int n)`<br>`{ long ans;`<br>`  int i;`<br>`  for (ans = 1, i = n; i>0; i--)`<br>`  ans = ans *i ;`<br>`  return ans;`<br>`}`<br>`int main()`<br>`{ long answer;`<br>`  int i, n;`<br>`  printf("Enter n ");`<br>`  scanf("%d",&n);`<br>`  answer = Factorial(n);`<br>`  printf("%d! = %d\n",n,answer);`<br>`  return 0;`<br>`}` | **มองปัญหาแบบการเรียกซ้ำ**<br>n! = n*(n-1)! ถ้ารู้ค่า (n-1)! จะหาค่า n! ได้<br>(n-1)! ยังเป็นปัญหาที่มีวิธีการคิดในรูปแบบ n! เหมือนเดิม<br>(n-1)! = (n-1)! *(n-2)! จะหาค่า (n-1)! ก็ต้องหา (n-2)! ให้ได้ก่อน<br>ถ้าซ้อนปัญหาจนกระทั่ง n = 0 จะได้ว่า 0! = 1 ตามนิยามที่กำหนดให้<br>ถ้ารู้คำตอบของ 0! = 1 แล้ว จะสามารถย้อนกลับไปแทนค่าหาคำตอบของ 1! = 1*0! = 1 ได้<br>ถ้าแทนค่าคำตอบย้อนกลับไปเรื่อยๆ จะทำให้สามารถย้อนคำตอบกลับไปแทนค่าจนถึงปัญหาของ n! ที่ต้องการได้<br><br>`long Factorial (int n)`<br>`{ if (n == 1)`<br>`  return 1; // คำตอบที่รู้ค่า`<br>`  else`<br>`  return n*Factorial(n-1); //คำตอบที่ต้องเรียกซ้ำ`<br>`}`<br>จุดที่เรียกซ้ำ<br>เมื่อได้คำตอบของ (n-1)! ต้องนำกลับมาคูณกับ n ก่อนจึงจะได้คำตอบของ (n)! |

# Recursion & Stack

- การเรียกตัวเอง จะเก็บสถานะการทำงานที่ทำค้างอยู่ไว้ใน **stack** แล้วกระโดดไปทำงานใหม่ซ้อนไปเรื่อยๆ จนเสร็จ แล้วจึงเรียกสถานะเดิมก่อนกระโดดไปทำงานจาก **stack** เพื่อกลับมาทำงานที่ค้างไว้ต่อเนื่องจากจุดเดิมจนสำเร็จ




10. return from Factorial(3)
    answer = 6
    output answer
    end

9. answer = 3*2 = 6
   end of Factorial(3)
   POP status of main from stack
   return answer = 6 to main

8. answer = 2*1 = 2
   end of Factorial(2)
   POP status of Factorial(3) from stack
   return answer = 2 to Factorial(3)

7. answer = 1*1 = 1
   end of Factorial(1)
   POP status of Factorial(2) from stack
   return answer = 1 to Factorial(2)

6. answer = 1
   end of Factorial(0) = 1
   POP status of Factorial(1) from stack
   return answer = 1 to Factorial(1)

# Greated Common Divisor (GCD)

- ตัวหารร่วมมากของจำนวนเต็มสองจำนวน ซึ่งไม่เป็นศูนย์พร้อมกัน คือจำนวนเต็มที่มากที่สุดที่หารทั้งสองจำนวนลงตัว
- กำหนดให้ a > b
- Solution
  - คิดแบบวนรอบ gcd(a,b) : a>b
    - เริ่มต้นให้ i = b
    - วนรอบหา i ที่หาร a และ b ลงตัวพร้อมกัน
    - ถ้าไม่เจอลดค่า i ลงทีละ 1 จนกว่าจะเจอ
  - คิดแบบ recursion gcd ( a, b, i) : a>b, i = b
    - ถ้าทั้ง a และ b หาร i ลงตัวพร้อมกัน คำตอบ = i
    - ถ้าหารไม่ลงตัวพร้อมกัน ให้ลดค่า i ลง 1 แล้วเรียกซ้ำใหม่
- Base case

  if (a%i == 0&&b%i==0) answer = i
- Recursive case

  if (a%i != 0&&b%i !=0) answer = gcd(a, b, i-1)

```
int gcd1(int a, int b)
{int i;
    for(i=b;i>0&&(b%i!=0&&a%i!=0);i--);
 return  i;
}
```

```
int gcd(int a, int b, int i)
{ if (a%i==0&&b%i==0)
      return i ;
  else
      return  gcd(a,b,i-1);
}
```

# Greated Common Divisor (GCD)

- Euclidean Algorithm
  - คิดแบบวนรอบ gcd(a,b) : a>b
    - หาเศษที่เหลือจากการหาร(modulo) a%b
      - ทำซ้ำจนกว่าตัวหาร b จะกลายเป็น 0
        - หาเศษที่เหลือจากการหาร a%b
        - เปลี่ยน b ให้กลายเป็นตัวตั้ง(a = b)
        - เปลี่ยนเศษ a%b เป็นตัวหาร (b = a%b)
      - วนรอบจนกว่า b=0 คำตอบที่ได้คือ a
  - คิดแบบเรียกซ้ำ gcd(a,b) : a>b
    - ถ้า b เท่ากับ 0 คำตอบ = a
    - ถ้า b ไม่เท่ากับ 0 คำตอบ = gcd(b, a%b)

```
int  gcd(int a, int b)
{  int mod;
   while (b != 0)
   { mod = a%b;
     a = b;
     b = mod;
    }
  return a;
}
```

- Base case

  if (b == 0) answer = a

- Recursive case

  if (b != 0) answer = gcd(b, a%b)

```
int  gcd (int a, int b)
{ if (b == 0) // Base Case
      return a ;
  else          // Recursive Case
      return gcd (b, a%b);
}
```

# 1. Factorial Function

- หาผลคูณของเลขจำนวนเต็มบวกทั้งหมดที่น้อยกว่าหรือเท่ากับ n
- Solution $n! = \prod_{i=1}^{n} i$ n! = 1 x 2 x 3 x 4 x 5 x ... x n
  - 1. n! = n x (n-1)!
    = n x (n-1) x (n-2)!
    = n x (n-1) x (n-2) x n-3) x ... x 1
  - 2. 0! = 1

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

- Base case
  if n == 0 answer 0! = 1
- Recursive case
  if n > 0 answer n! = n x (n-1)!

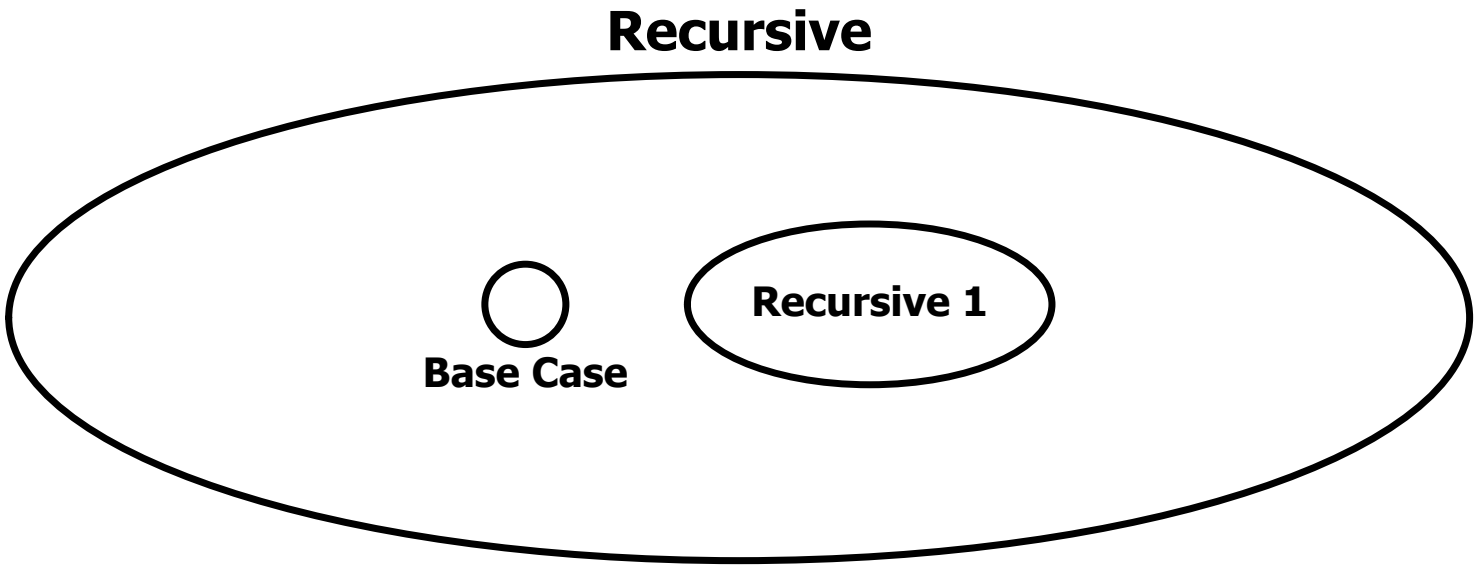
# Single/Linear Recursion

```
long long factorial (int n)
{     if   (n == 0)                 // Base Case
           return(1);
      else                          // Recursive Case
           return( n * factorial(n-1));
}
```

Java: [public] static long factorial (int n)

ตัวแปรประเภท integer ในจาวา
byte = 8 bit
short = 16 bit
int = 32 bit
long = 64 bit (C ใช้ long long คู่กับ %llu)

ค่าที่ return กลับมาจากการรีเคอร์ชัน มีการนำไปใช้คำนวณต่อ ก่อนจบฟังก์ชัน



# Step of recursion

factorial(5) = 5 * factorial(4)

= 5 * (4 * factorial(3))

= 5 * (4 * (3 * factorial(2)))

= 5 * (4 * (3 * (2 * factorial(1))))

= 5 * (4 * (3 * (2 * (1 * factorial(0)))))

= 5 * (4 * (3 * (2 * (1 * 1))))

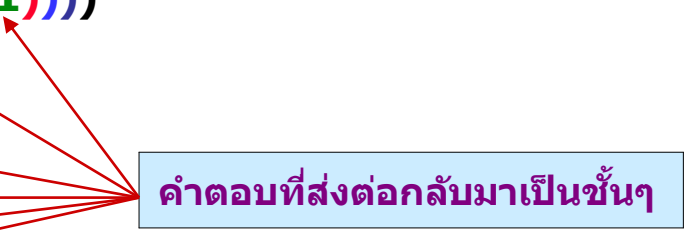
= 5 * (4 * (3 * (2 * 1)))

= 5 * (4 * (3 * 2))

= 5 * (4 * 6)

= 5 * 24

= 120



# 2. Fibonacci Numbers

- ค่าในเทอมปัจจุบัน เกิดจากค่าในลำดับที่อยู่ก่อนหน้า 2 ตัว รวมกัน
  - Fibonacci series = $F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8, F_9, \ldots$
    = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Condition n >= 0
- Base case
  1. $F_0 = 0$ {for n = 0}
  2. $F_1 = 1$ {for n = 1}
- Recursive case
  3. $F_n = F_{n-1} + F_{n-2}$ {for $n \geq 2$ }

มีค่าเริ่มต้น 2 ตัว

ตัวถัดไปมีค่าเท่ากับ 2 ตัวหน้าบวกกัน

ถ้าจะเขียนด้วยวนรอบ ต้องมีการจำค่าในรอบที่แล้ว และค่าในรอบก่อนหน้า

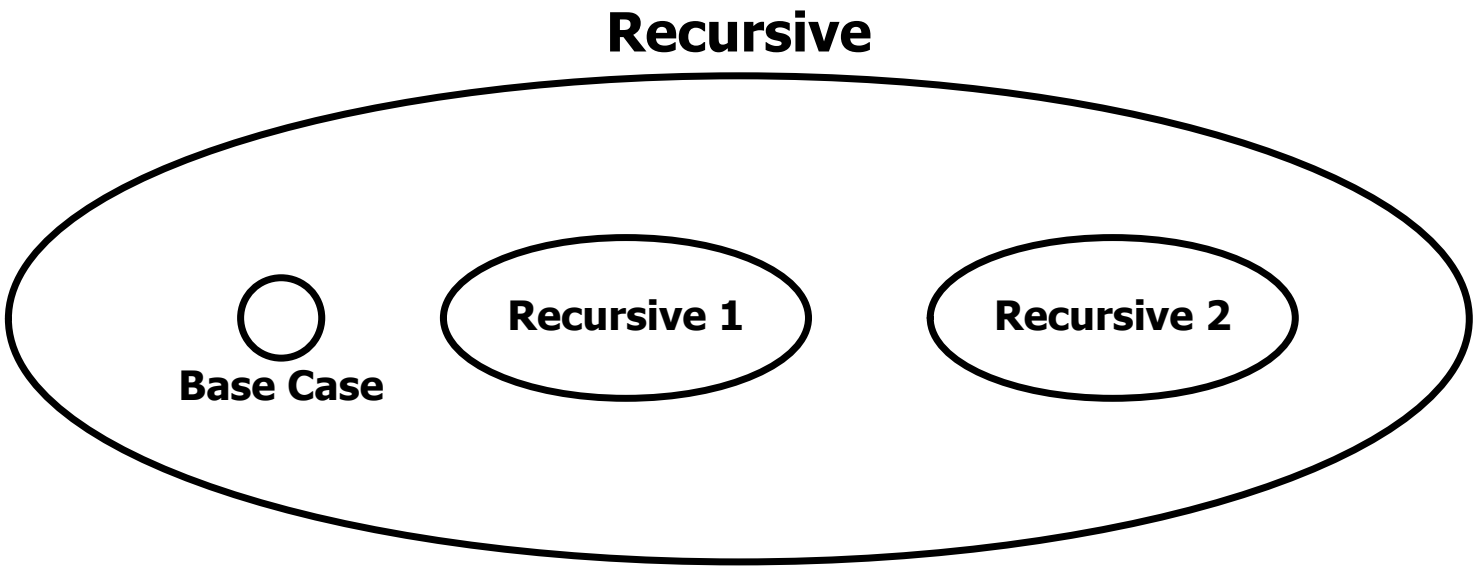
```
for (fn1=1,fn2=0,i=2;i<=n;i++)
    { fn  = fn1 + fn2; //ค่าที่คำนวณในรอบนี้
      fn2 = fn1;       //เตรียมค่าที่จะใช้คำนวณรอบหน้า
      fn1 = fn;        //เตรียมค่าที่จะใช้คำนวณรอบหน้า
     }
```

# Multiple Recursion

```
long long fibonacci(int n)
{  if ((n== 0)||(n==1))  // Base Case
      return (n) ;
   else    // Recursive Case
      return fibonacci(n-1) + fibonacci(n-2);
}
```

Java: [public] static long fibonacci(int n)

แตกเป็น 2 Recursion



# Step of recursion

```
F(4) =   F(3)              +              F(2)
         F(3) = F(2)       +       F(1)        //recursion to F(3)
                F(2) = F(1) + F(0)            //recursion to F(2)
                       F(1) = 1               //recursion to F(1)   *
                            = 1  + F(0)       //return to F(2)
                                   F(0) = 0   //recursion to F(0)   *
                F(2) =  1  + 0    = 1         //return to F(2)      *
         F(3) =  1         +       F(1)        //return to F(3)
                                   F(1) =  1  //recursion to F(1)   *
         F(3) =  1         +       1    =  2  //return to F(3)      *
    =    2                +              F(2) //return to F(4)
         2                +              F(2) = F(1) + F(0)  //recursion to F(2)
                                                F(1) = 1      //recursion to F(1)  *
                                             = 1  +  F(0)     //return to F(2)
                                                     F(0)=0 //recursion to F(0) *
                                             F(2) =  1  +  0 = 1  //return to F(2)  *
F(4) =   2                +              1  =  3               //return to F(4)  *
```

# 3. Ackermann's Function

- เป็นปัญหาที่ต้องแก้แบบ backtracking คือต้องสำรวจทุกทางเลือกที่เป็นไปได้จนถึงที่สุดก่อน จึงย้อนกลับมา(ตัดสินใจเลือกเส้นทาง)
- Condition  n and m  >= 0
- Solution

  1. A(0, n)  =  n + 1  if  m = 0;  n ≥ 0;
  2. A(m, 0)  =  A(m-1, 1)  if  m > 0;  n = 0;
  3. A(m, n)  =  A(m-1, A(m, n-1))  if  m > 0;  n > 0;

- Base Case

  A(0, n)  =  n + 1  if  m = 0;  n ≥ 0;

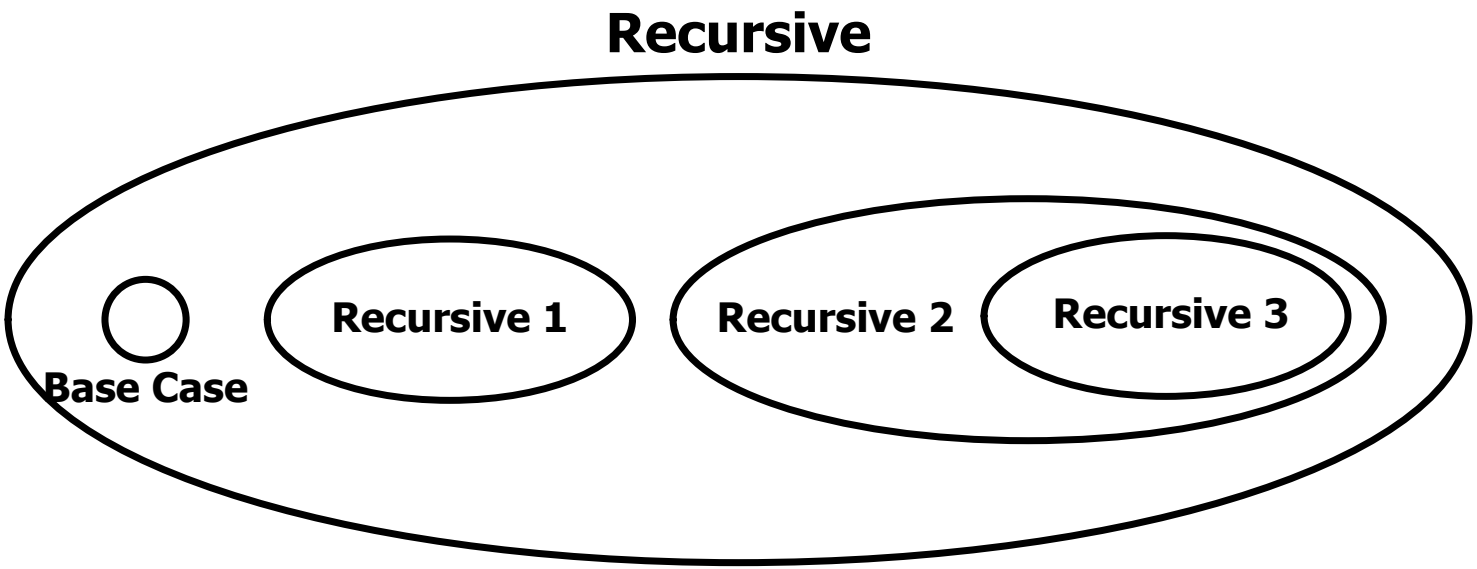
- Recursive Case
  - A(m, 0)  =  A(m-1, 1)  if  m > 0;  n = 0;
  - A(m, n)  =  A(m-1, A(m, n-1))  if  m > 0;  n > 0;

# Nested Recursion

```
int ackermann( int m, int n)
{       if (m==0)
            return (n + 1) ;
        else if ( (m != 0) && (n==0) )
            return (ackermann (m-1, 1)) ;
        else
            return (ackermann(m-1, ackermann(m, n-1)));
}
```

Java: [public] static int fibonacci(int n)

มี Recursion ซ้อนอยู่ในพารามิเตอร์ของอีก recursion



# Step of recursion

A(1,3) = A(0, A(1,2))

A(1,2) = A(0,A(1,1)) //recursion to A(1,1)

A(1,1) = A(0, A(1,0))

A(1,0) = A(0,1) //recursion to A(1,0)

A(0,1)=1+1= 2

A(1,0) = 2 //return to A(1,0)

A(1,1) = A(0,2)

A(0,2) = 2+1 = 3

A(1,1) = 3 //return to A(1,1)

A(1,2) = A(0,3) //return to A(1,2)

A(0,3) = 3+1 = 4

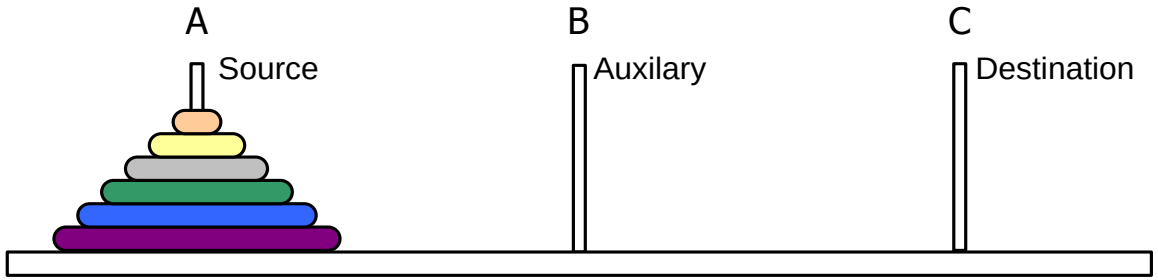
A(1,2) = 4

A(1,3) = A(0, 4) = 4+1 = 5

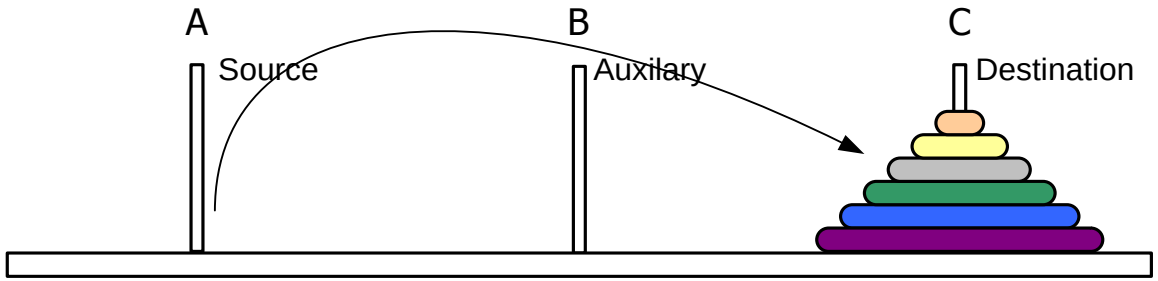
A(1,3) = 5

# 4. Tower of Hanoi Ploblem

- เกมส์ทางคณิตศาสตร์ ประกอบด้วยหมุด 3 แท่งคือ ต้นทาง หมุดพัก และปลายทาง ตำแหน่งต้นทางมีจานขนาดต่างๆ เรียงซ้อนตามลำดับโดยจานใบใหญ่อยู่ล่างสุด
  - ย้ายจานทั้งหมด จากหลักต้นทาง ไปยังหลักปลายทาง
  - ย้ายจานได้ทีละ 1 ใบ
  - ใบเล็กวางบนใบใหญ่ได้ แต่ใบใหญ่ทับบนใบเล็กไม่ได้
- **Initial State**

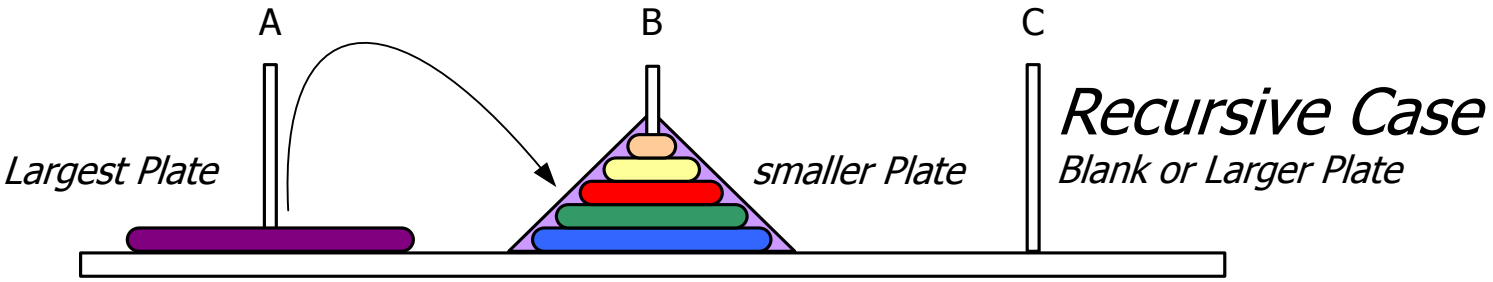



- **Final State**

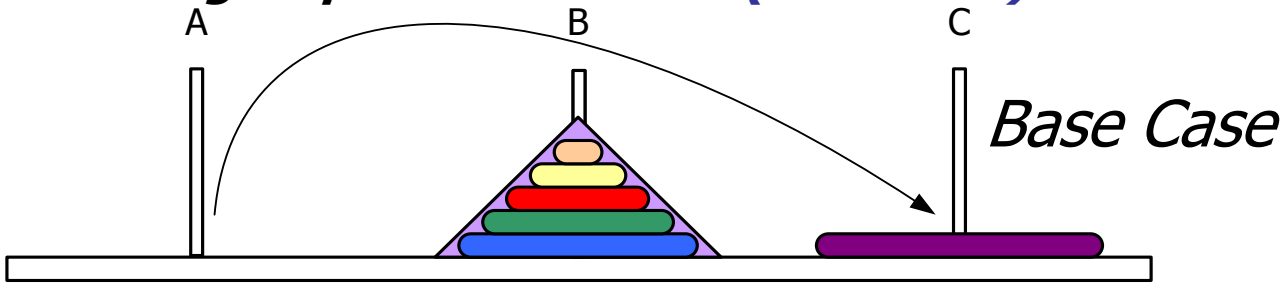


# Solution

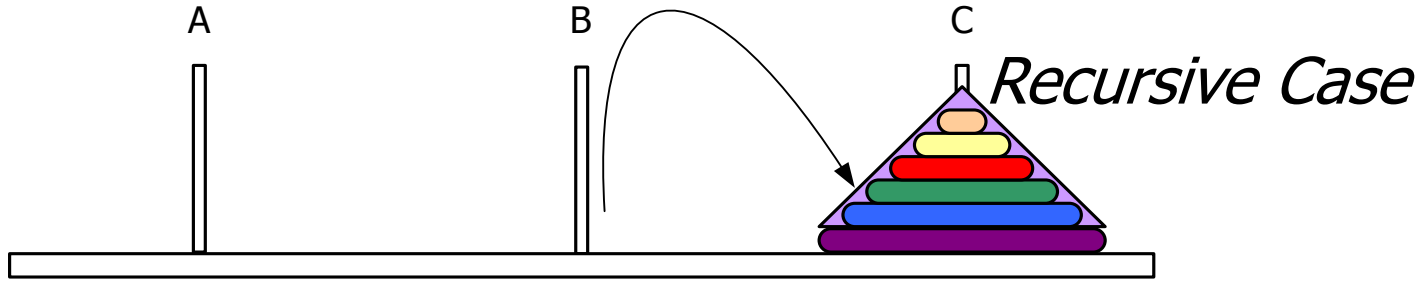
- แบ่งปัญหาออกเป็น 1 จานใหญ่ และ 1 กลุ่มของจานเล็ก(n-1)
  - ***move n-1 plate from A to B (Recursive Case)***




  - ***move Largest plate from A to C (Base Case)***




  - ***move n-1 plate from B to C (Recursive Case)***



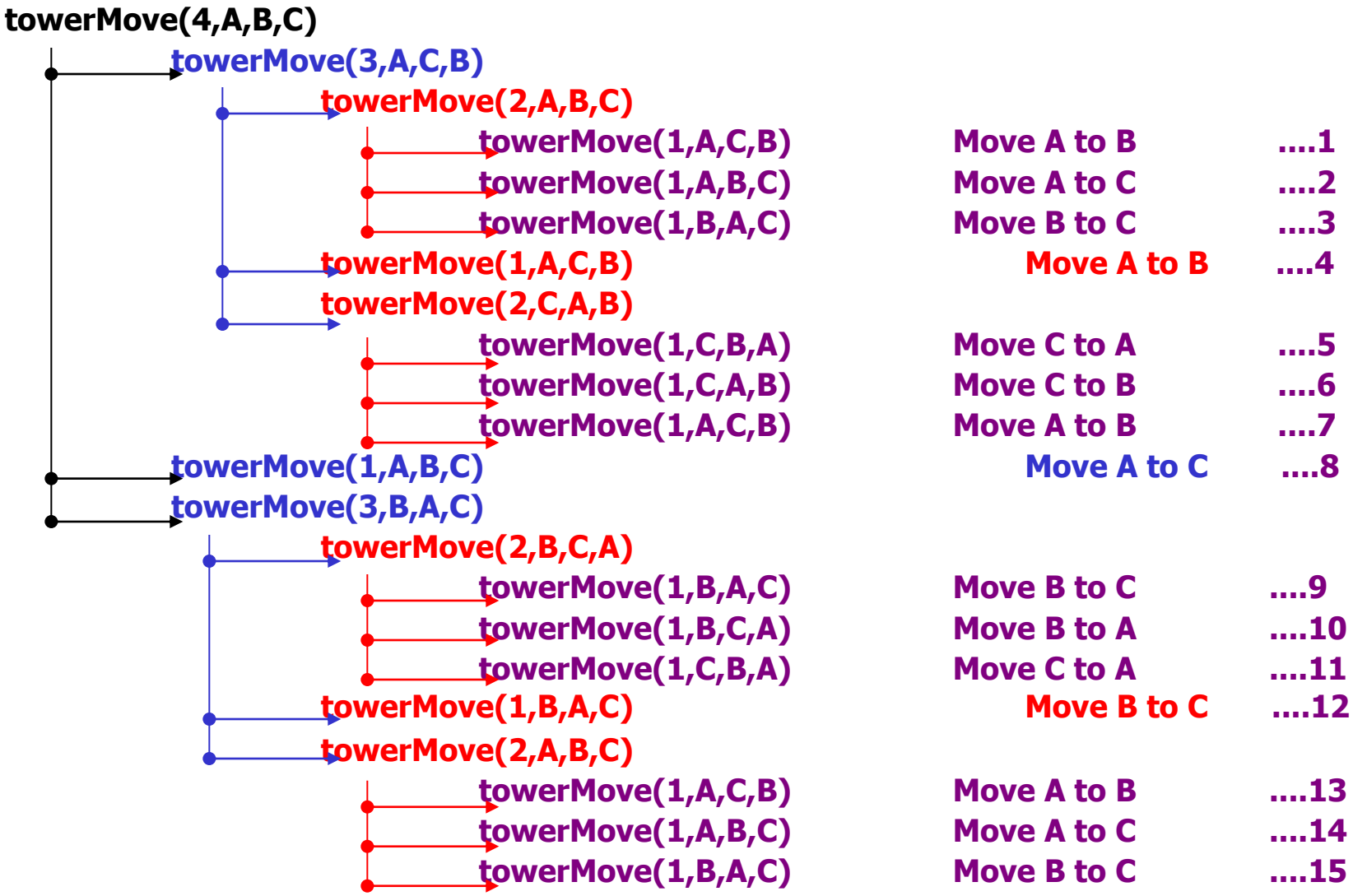
# Function/method

```
void  towerMove(int n, char a, char b, char c)
{
        /*  move n disks from a to c using b */
      if  (n  ==  1)
          printf("Move  %c  to  %c\n" , a , c);
       else  {    /* move the n-1 disks from a to b, using c */
                     towerMove(n-1, a, c, b);
                  /*  move the remaining  disk from a to c  */
                     towerMove(1, a, b, c);
                  /*  move the n-1 disks from b to c, using a */
                     towerMove(n-1, b, a, c);
             }
}
```

Java: [public] static void towerMove(int n, char a, char b, char c)

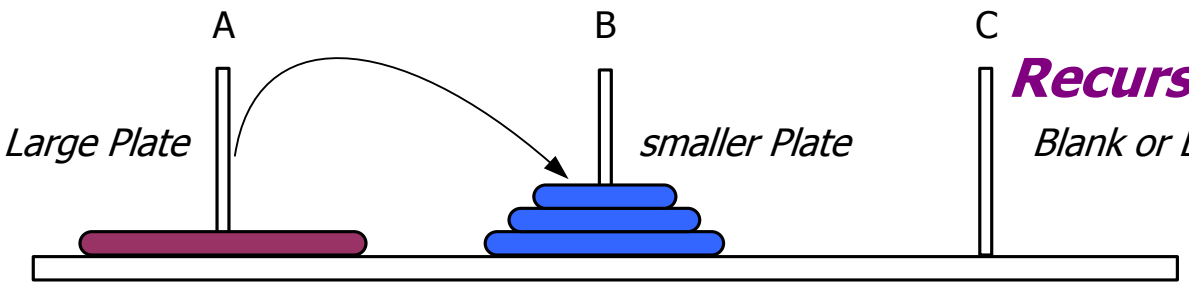
Java: System.out.printf ("Move %c to %c\n",a, c);

# move 4 disk from A to C

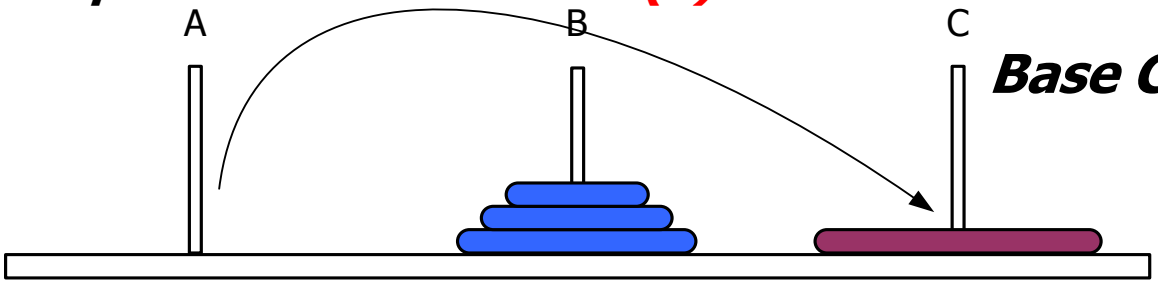


# move 4 plate from A to C

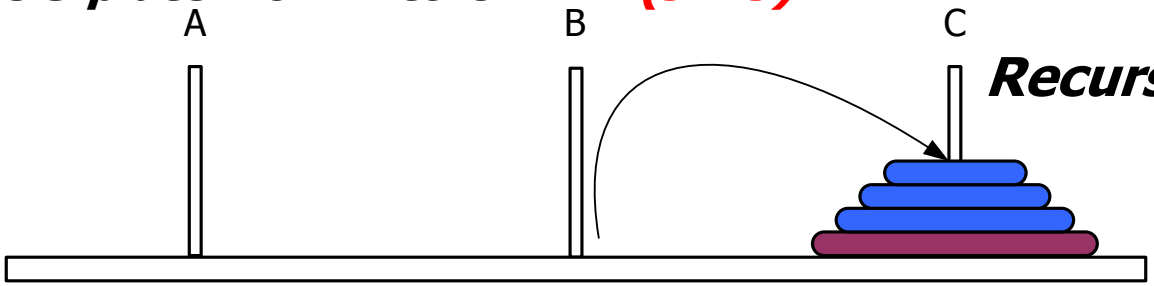
- ***move 3 plate from A to B*** ***...(1-7)***




- ***move 1 plate from A to C*** ***...(8)***



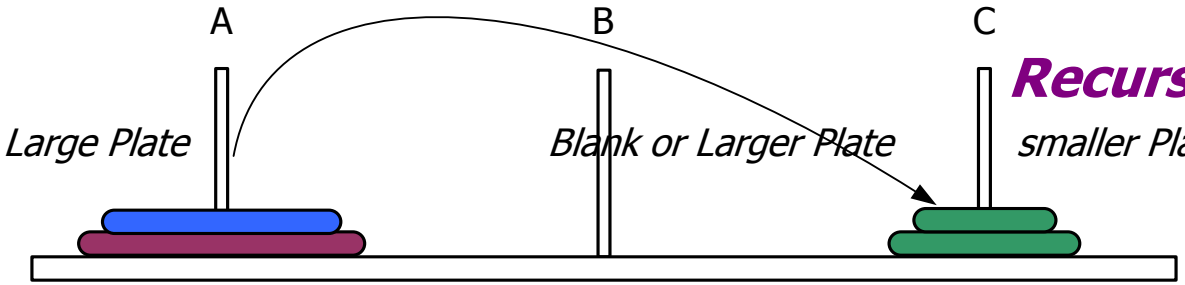

- ***move 3 plate from B to C*** ***...(9-15)***



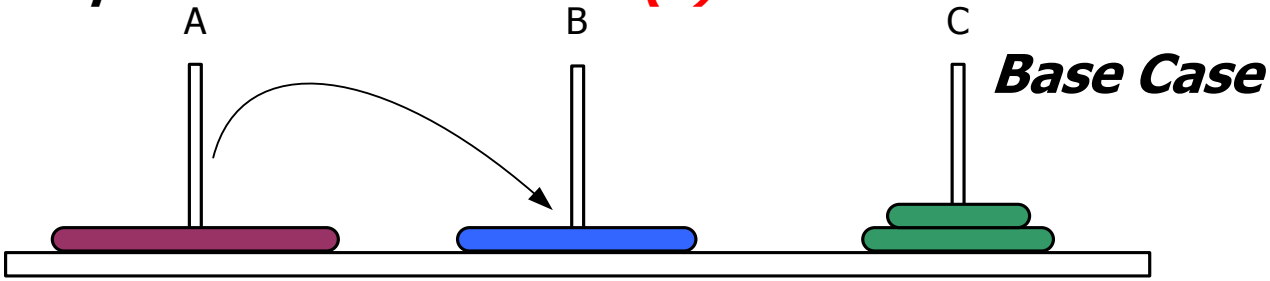

- **return Solution**

# move 3 plate from A to B

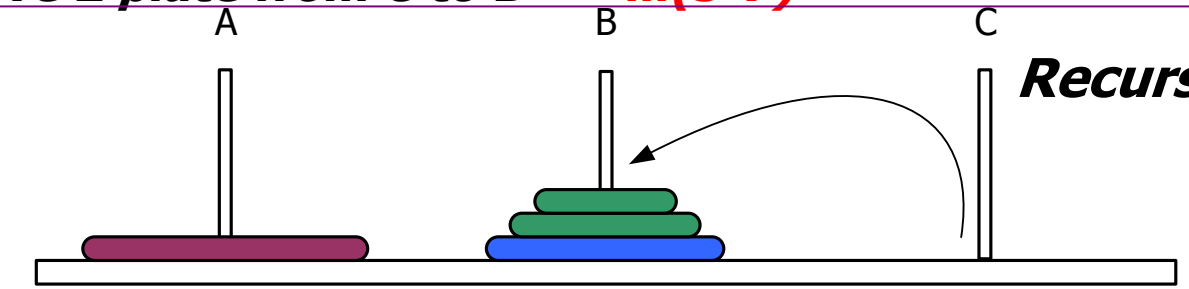
- ***move 2 plate from A to C*** ***...(1-3)***




- ***move 1 plate from A to B*** ***...(4)***



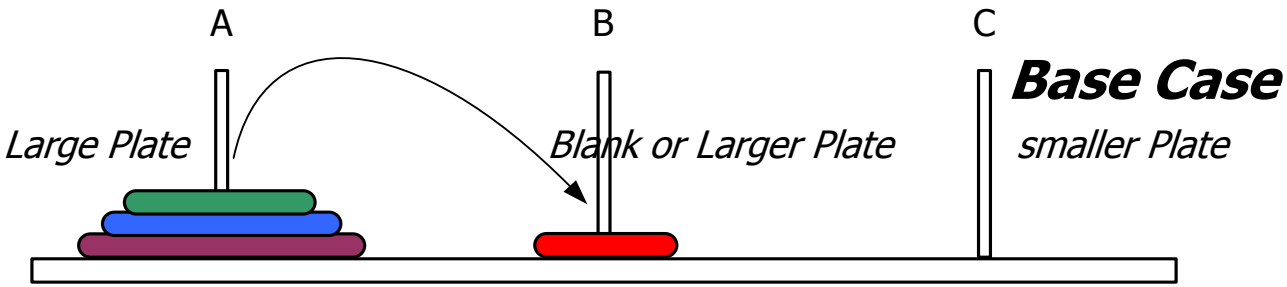

- ***move 2 plate from C to B*** ***...(5-7)***



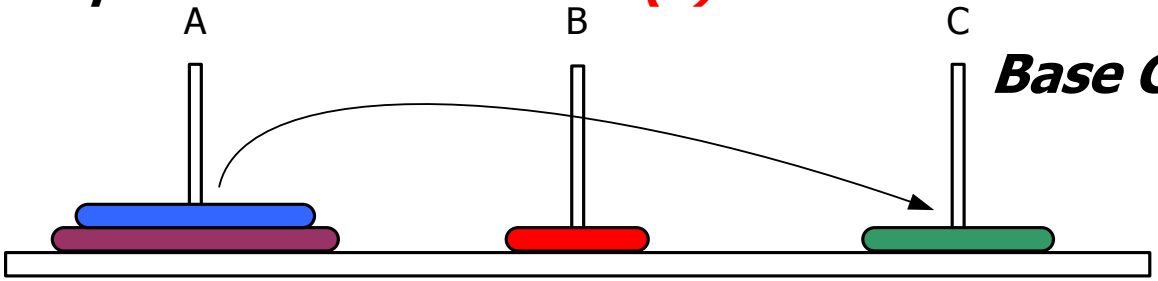

- **return to move 4 plate from A to C**

# move 2 plate from A to C

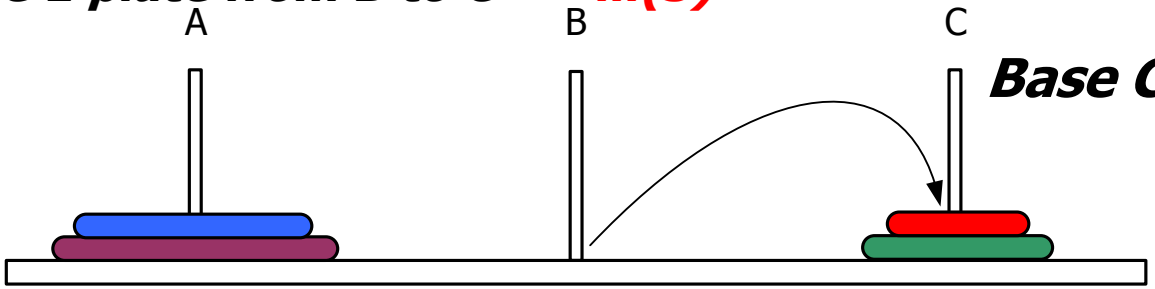
- ***move 1 plate from A to B*** ***...(1)***




- ***move 1 plate from A to C*** ***...(2)***



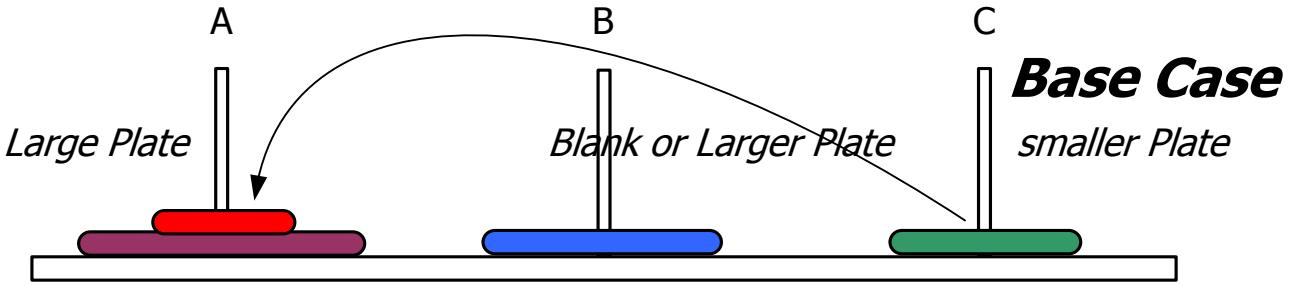

- ***move 1 plate from B to C*** ***...(3)***



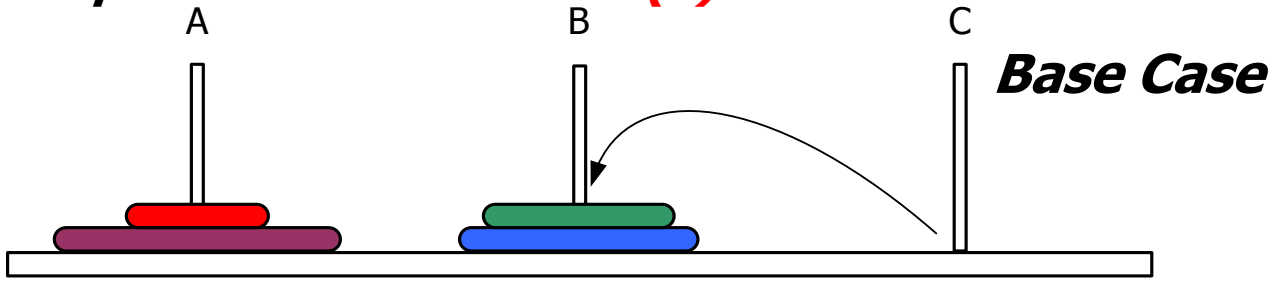

- **return to move 3 plate from A to C**

# move 2 plate from C to B

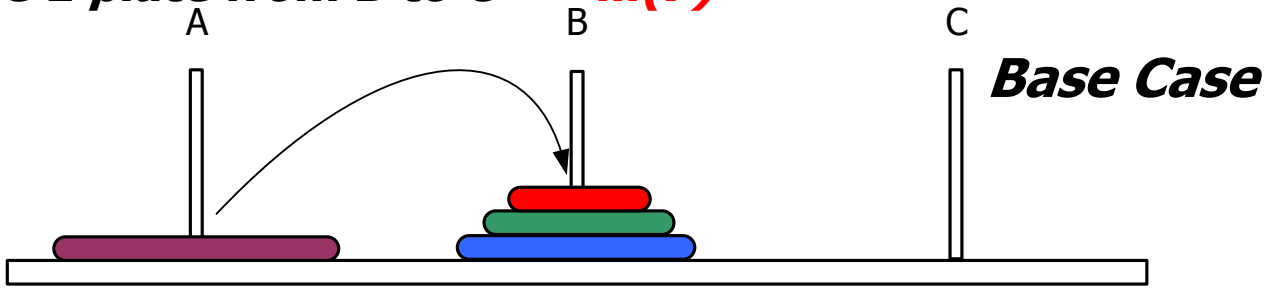
- ***move 1 plate from A to B*** ***...(5)***




- ***move 1 plate from A to C*** ***...(6)***



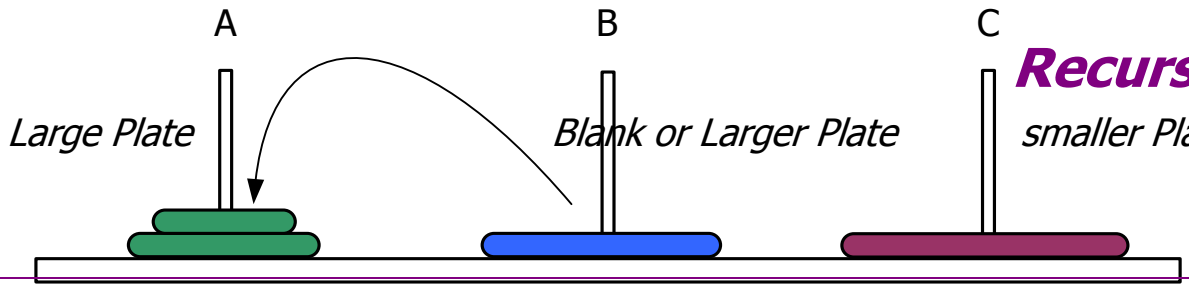

- ***move 1 plate from B to C*** ***...(7)***



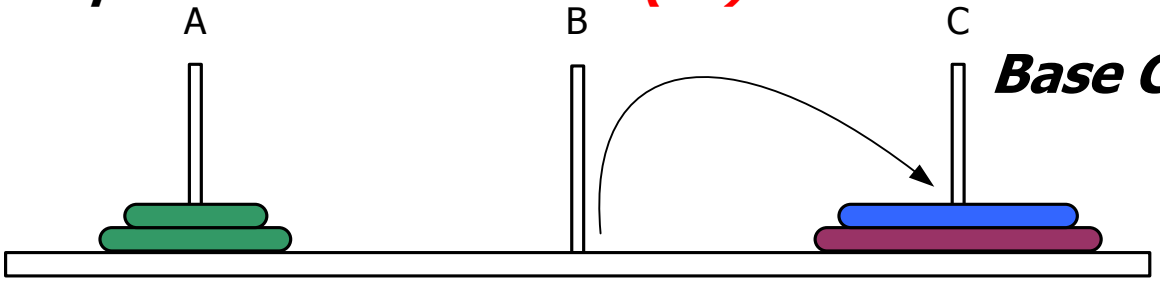

- **return to move 3 plate from A to C**

# move 3 plate from B to C

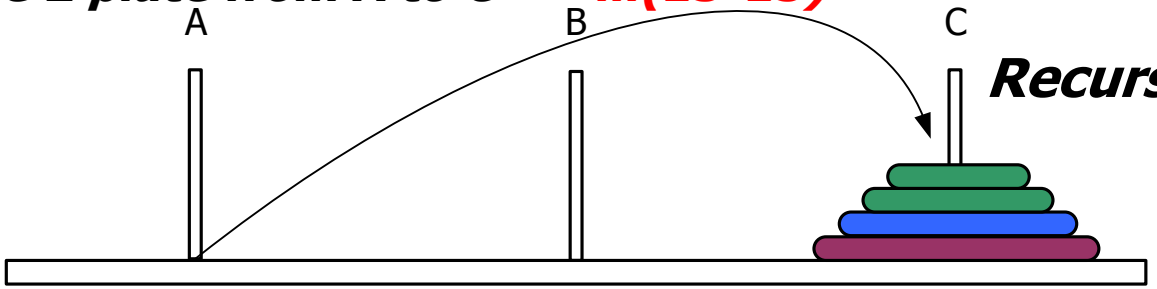
- ***move 2 plate from B to A*** ***...(9-11)***




- ***move 1 plate from B to C*** ***...(12)***




- ***move 2 plate from A to C*** ***...(13-15)***




- **return to move 4 plate from A to C**

# Tail Recursion

```
int  gcd (int a, int b)
{    if (b == 0) // Base Case
        return a ;
     else          // Recursive Case
       return gcd (b, a%b);
}
```

ค่าที่ return กลับมาจากการรีเคอร์ชันใช้เป็นคำตอบ แล้วจบฟังก์ชันได้เลย ไม่ได้นำไปทำงานต่อ

คำตอบที่ส่งต่อกลับมาเป็นชั้นๆ

gcd(180,48) = gcd(48,36)

= gcd(36,12)

= gcd(12,0)

gcd (12,0) = 12

gcd (36,12) = 12

gcd (48, 36) = 12

gcd(180,48) = 12

- Compiler สามารถแปลง code ประเภท tail recursion ให้เป็นคำสั่งภาษาเครื่องแบบวนรอบได้ (ทำให้ไม่ต้องจัดการเรื่อง stack)

# Factorial with tail recursion

```
long  factorial_tail (int n, long ans)
{  if   (n <= 1)      // Base Case
        return ans;
   else               // Recursive Case
        return factorial_tail (n-1, n*ans);
}
long  factorial (int n)
{     return factorial_tail (n, 1); // Call tail recursion
}
```

ส่งคำตอบกลับมาในพารามิเตอร์

คำตอบเริ่มต้นตอนเรียกใช้(Base case)

คำตอบที่ส่งต่อกลับมาเป็นชั้นๆ

```
factorial(5)        = factorial_tail(5,1)
                    =     factorial_tail(4,5)
                    =         factorial_tail(3,20)
                    =             factorial_tail(2,60)
                    =                 factorial_tail(1,120)
                    =                 120
                    =             120
                    =         120
                    =     120
                    = 120
```

# Programming Example

```
static int Read_Int(int min, int max) {
int a = 0;
boolean success = false;
    Scanner in = new Scanner(System.in);
    while (!success) {
      try {  a = in.nextInt();
          if (a >= min && a <= max)
              success = true;
          else
              System.out.printf("Please enter between %d - %d\n", min, max);
      } catch (Exception e) {
          System.out.println(e.getMessage());
          System.out.println("Error ! Please Enter number again.");
          in.nextLine();
      }
    }
    return a;
}
```

ทำซ้ำเมื่อยังอ่านไม่สำเร็จ

ตรวจสอบความผิดพลาด

ถ้าอ่านตัวเลขได้ถูกต้องให้หลุดจากวนรอบ

เตือนเมื่ออ่านตัวเลขไม่อยู่ในช่วง

ทำเมื่อเกิดความผิดพลาดใน try {}

ความผิดพลาดที่พบ

อ่านจนหมดบรรทัด

**C:**
```
int  Read_Int (int min, int max)
{ int  a = 0;
    while (scanf("%d",&a)==0||a<=min||a>=max)
    {  rewind(stdin);  // fflush(stdin); s = gets(s);
       printf("Please enter between %d - %d\n", min, max);
    }
    return a;
}
```

**Java:**
```
public static void main(String[] args) {
int n;
long ans;
    System.out.printf("Enter number ");
    n = Read_Int(0, 15);
    ans = factorial(n);
    System.out.printf ("%d! = %d\n", n, ans);
}
```

**C:**
```
int main() {
int n;
long long ans;
    printf("Enter number ");
    n = Read_Int(0, 15);
    ans = factorial(n);
    printf("%d! = %llu\n" ,n,ans);
    return 0;
}
```