

ค่ายอบรมโอลิมปิกวิชาการ 2



โครงสร้างข้อมูล : ฟังก์ชันเรียกตัวเอง

Data Structure : Recursive function

รัชดาพร คณาวงษ์

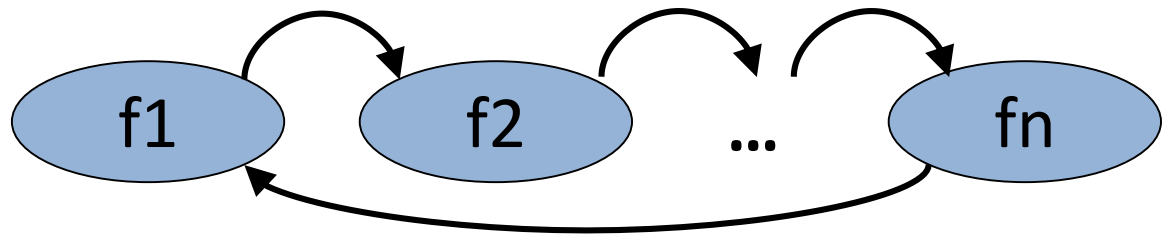
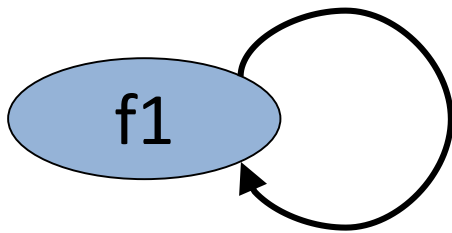
13 มีนาคม 2562

ศูนย์มหาวิทยาลัยศิลปากร



ฟังก์ชันเรียกตัวเอง (Recursive Function)

- เป็นฟังก์ชันชนิดที่มีคำสั่งเรียกตัวเอง (ทั้งทางตรงและทางอ้อม)
- ทำให้เกิดการวนการทำงานของชุดคำสั่งเมื่อมีการเรียกฟังก์ชันตัวเองใช้งาน
- ฟังก์ชันนี้จะต้องมีการส่งผ่านค่าพารามิเตอร์เพื่อให้เกิดการหยุดการทำงาน



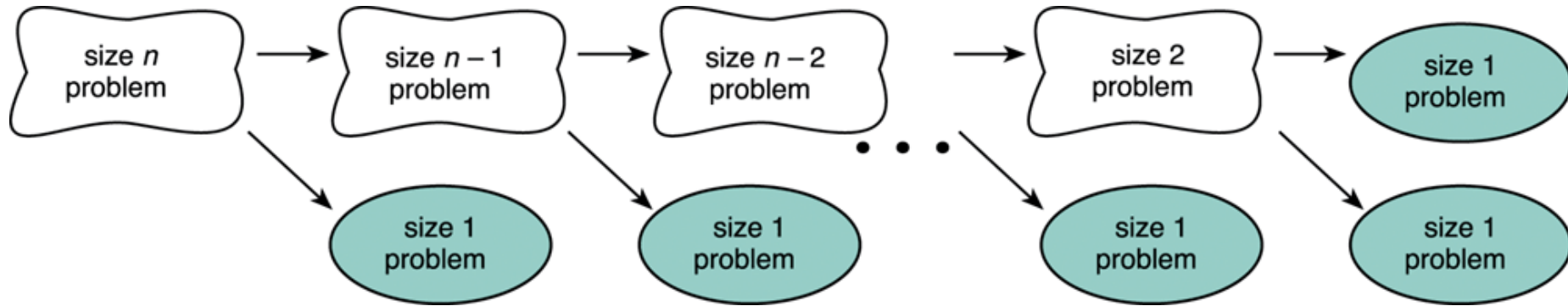


ปัญหาที่เหมาะสมกับฟังก์ชันเรียกตัวเอง

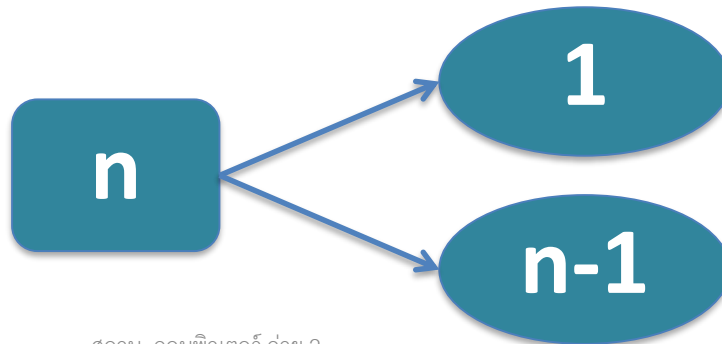
1. แบ่งเป็นกรณีพื้นฐาน (base case) ได้หนึ่งหรือมากกว่าที่สามารถหาค่าได้ตรงไปตรงมา
2. ปัญหาสามารถดำเนินการด้วยกรณีในข้อหนึ่งหรือการเรียกตัวเอง
if this is a simple case
 solve it
else
 redefine the problem using recursion



การแบ่งปัญหาให้เป็นปัญหาย่อย



- สมมติว่าปัญหาที่มีขนาด 1 (size 1) สามารถแก้ไขได้ง่ายๆ
- เราสามารถทำการแยกปัญหาเป็นปัญหาที่เป็นขนาด 1 กับปัญหาที่เป็นขนาด $n-1$





ตัวอย่างของฟังก์ชันเรียกตัวเอง

- ลองดูตัวอย่างการคูณด้วยการบวก

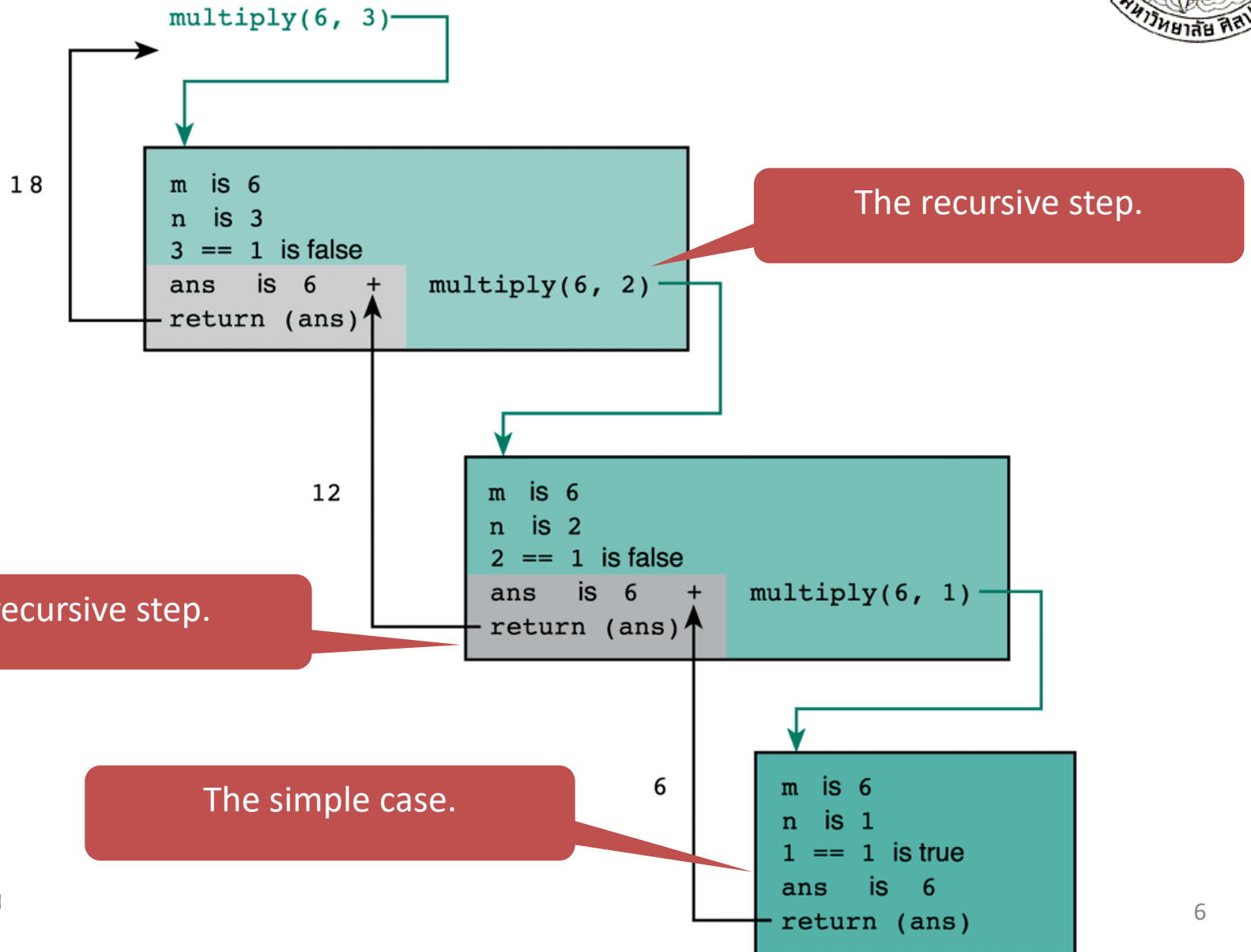
```
1.  /*
2.   * Performs integer multiplication using + operator.
3.   * Pre:  m and n are defined and n > 0
4.   * Post: returns m * n
5.   */
6.  int
7.  multiply(int m, int n)
8.  {
9.      int ans;
10.
11.      if (n == 1)
12.          ans = m;      /* simple case */
13.      else
14.          ans = m + multiply(m, n - 1); /* recursive step */
15.
16.      return (ans);
17. }
```

The simple case is " $m * 1 = m$."

The recursive step uses the following equation:
" $m * n = m + m * (n - 1)$."



ถ้าเราเรียก multiply (6,3) จะเกิดอะไรขึ้น





เราจะเขียนฟังก์ชันเรียกตัวเองได้อย่างไร

- กำหนดพารามิเตอร์ที่มีผลกับการดำเนินการ
- กำหนดกรณีพื้นฐาน (base case) คือกรณีที่มีการคำนวณหรือดำเนินการตรงไปตรงมา
- กำหนดกรณีทั่วไป (general case) คือกรณีที่มีการเรียกใช้ฟังก์ชันตัวเอง
- ตรวจสอบอัลกอริทึม โดยใช้วิธีถามสามคำถาม (Three-Question-Method)



Three-Question Method สำหรับตรวจสอบ

1. The Base-Case Question: มีทางที่จะออกจากฟังก์ชันโดยไม่ต้องเรียกตัวเองหรือไม่
2. The Smaller-Caller Question: ในการเรียกฟังก์ชันเป็นกรณีที่เล็กลงของปัญหาและนำไปสู่กรณีพื้นฐาน (base case) หรือไม่
3. The General-Case Question: ถ้าการเรียกตัวเองทำงานถูกต้อง ฟังก์ชันทั้งหมดทำงานถูกต้องด้วยหรือไม่

เงื่อนไขการจบ (Terminating Condition)



- ฟังก์ชันเรียกตัวเองต้องมีเงื่อนไขการจบ โดยจะมีหนึ่งหรือมากกว่าหนึ่งก็ได้
- เงื่อนไขการจบคือเงื่อนไขที่จะดำเนินการใดๆ ที่ไม่เรียกใช้ตัวเอง
- ถ้าไม่มีเงื่อนไขการจบ ฟังก์ชันเรียกตัวเองจะทำงานไม่สิ้นสุด
- จากตัวอย่างก่อนหน้านี้ เงื่อนไขการจบคือ $\text{if } (n==1) \text{ ans} = m$



ตัวอย่างการนับตัวอักษรในสตริง

- การนับจำนวนตัวอักษรที่ปรากฏในสตริง เช่น จำนวนของตัวอักษร 's' ในสตริง “Mississippi” คือ 4

```
1.  /*
2.   * Count the number of occurrences of character ch in string str
3.   */
4.  int
5.  count(char ch, const char *str)
6.  {
7.
8.      int ans;
9.
10.     if (str[0] == '\0')                /* simple case */
11.         ans = 0;
12.     else                                /* redefine problem using recursion */
13.         if (ch == str[0])               /* first character must be counted */
14.             ans = 1 + count(ch, &str[1]);
15.         else                            /* first character is not counted */
16.             ans = count(ch, &str[1]);
17.
18.     return (ans);
19. }
```

The terminating condition.



ตัวอย่างการกลับคำ (Reverse Words)

- ฟังก์ชันการเรียกตัวเองสามารถใช้กับการกลับสตริง

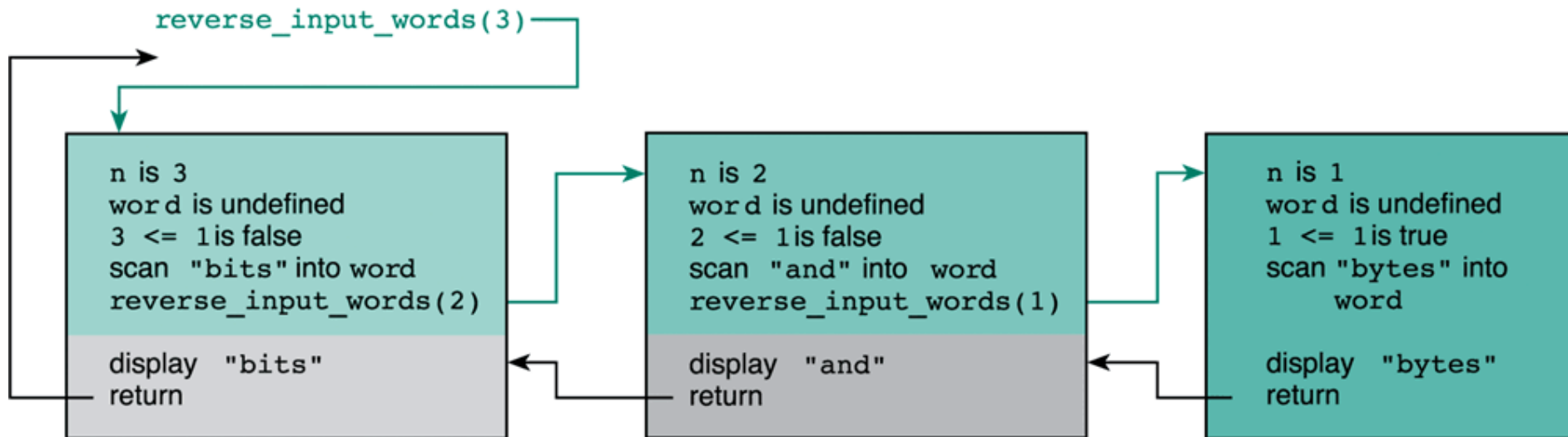
```
1.  /*
2.   * Take n words as input and print them in reverse order on separate lines.
3.   * Pre: n > 0
4.   */
5.  void
6.  reverse_input_words(int n)
7.  {
8.      char word[WORDSIZ]; /* local variable for storing one word */
9.
10.     if (n <= 1) { /* simple case: just one word to get and print */
11.
12.         scanf("%s", word);
13.         printf("%s\n", word);
14.
15.     } else { /* get this word;
16.             reverse order;
17.
18.             scanf("%s", word);
19.             reverse_input_words(n - 1);
20.             printf("%s\n", word);
21.         }
22. }
```

The scanned word will not be printed until the recursion finishes.

The first scanned word is last printed.



ตัวอย่างการกลับคำ (2/2)

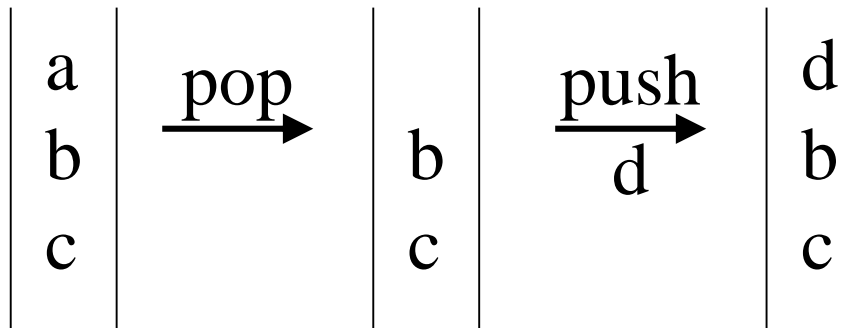


- หมายเหตุ ฟังก์ชันการเรียกตัวเองเป็นแค่วิธีการหนึ่งในการแก้ปัญหา ปัญหานี้สามารถแก้ไขโดยไม่ใช้ฟังก์ชันเรียกตัวเองได้



ภาษาซีมีขั้นตอนในการจัดการฟังก์ชันอย่างไร

- ภาษาซีจะเก็บและใช้งานค่าในตัวแปรโดยใช้โครงสร้างสแตก
 - สแตกเป็นโครงสร้างข้อมูลที่ข้อมูลตัวสุดท้ายจะถูกเพิ่มเข้าไปเป็นข้อมูลตัวสุดท้ายและเป็นตัวแรกที่ถูกดึงออกมาใช้งาน
 - มีตัวดำเนินการข้อมูลในสแตก 2 อย่างคือ push และ pop





ภาษาซีมีขั้นตอนในการจัดการฟังก์ชันอย่างไร

- ทุกครั้งที่ฟังก์ชันถูกเรียก สถานะปัจจุบัน (execution state) ของฟังก์ชันเรียก (caller function) ซึ่งสถานะอาจจะเป็นพารามิเตอร์ ตัวแปรท้องถิ่น (local variables) และตำแหน่งหน่วยความจำ จะถูกจัดเก็บในสแตก (pushed onto the stack)
- เมื่อเอนจินตีวฟังก์ชันที่ถูกเรียกเสร็จ (called function is finished) ก็ จะทำการดึงข้อมูลของฟังก์ชันออกมาจากสแตก (popping up the execution state from stack)
- กระบวนการนี้เพียงพอที่จะทำให้ฟังก์ชันเรียกตัวเองทำงานได้อย่างมีประสิทธิภาพ



การดีบั๊กฟังก์ชันเรียกตัวเอง

- ฟังก์ชันเรียกตัวเองจะติดตามการทำงานและดีบั๊กการทำงานได้ยาก เพราะค่าของตัวแปรในฟังก์ชันก่อนการเรียกตัวเองจะถูกเก็บในสแต็ก และตัวดีบั๊กไม่สามารถแสดงตรงส่วนนี้ได้

```
1. /*
2.  * *** Includes calls to printf to trace execution ***
3.  * Performs integer multiplication using + operator.
4.  * Pre:   m and n are defined and n > 0
5.  * Post:  returns m * n
6.  */
7. int
8. multiply(int m, int n)
9. {
10.     int ans;
11.
12.     printf("Entering multiply with m = %d, n = %d\n", m, n);
13.
14.     if (n == 1)
15.         ans = m;      /* simple case */
16.     else
17.         ans = m + multiply(m, n - 1); /* recursive step */
```

Watch the input arguments passed into each recursive step.



การหาค่าแฟคทอเรียล

- ปัญหาหลายอย่างสามารถแก้ได้ด้วยฟังก์ชันเรียกตัวเอง
ตัวอย่างเช่น n factorial (n!)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\dots*(n-1)*n & \text{if } n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)!*n & \text{if } n > 0 \end{cases} \quad (\text{recursive solution})$$



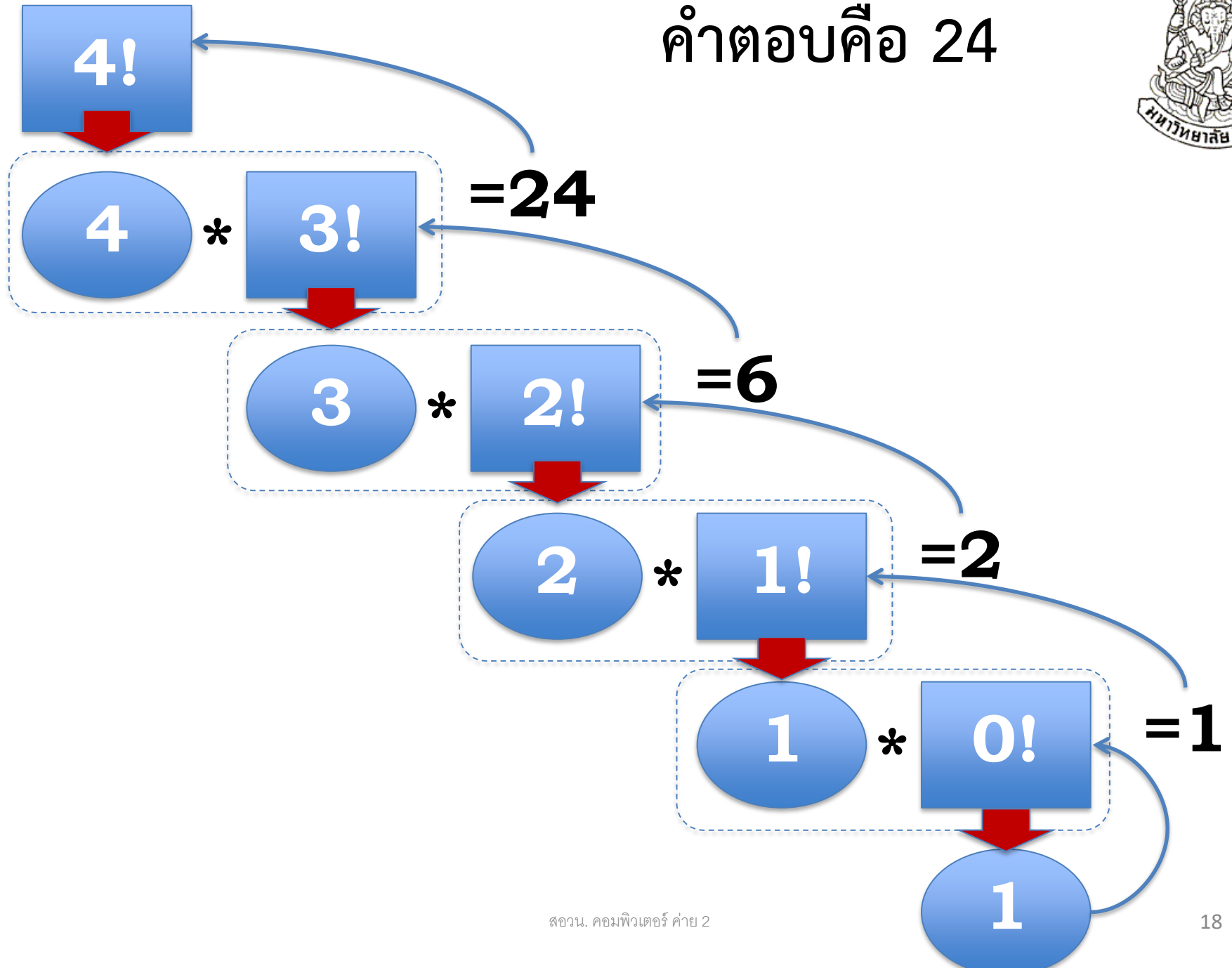
การหาค่าแฟคทอเรียลด้วยฟังก์ชันเรียกตัวเอง

- ฟังก์ชันทางคณิตศาสตร์หลายฟังก์ชันสามารถกำหนดและแก้ปัญหาด้วยการเรียกตัวเอง

```
1.  /*
2.   *  Compute n! using a recursive definition
3.   *  Pre:  n >= 0
4.   */
5.  int
6.  factorial(int n)
7.  {
8.      int ans;
9.
10.     if (n == 0)
11.         ans = 1;
12.     else
13.         ans = n * factorial(n - 1);
14.
15.     return (ans);
16. }
```



คำตอบคือ 24





การหาค่าแฟคทอเรียลด้วยการวนซ้ำ

- การหาค่าแฟคทอเรียลด้วยการวนซ้ำ (loop) ด้วยคำสั่ง for

```
1. /*
2.  * Computes n!
3.  * Pre: n is greater than or equal to zero
4.  */
5. int
6. factorial(int n)
7. {
8.     int i,          /* local variables */
9.     product = 1;
10.
11.     /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
12.     for (i = n; i > 1; --i) {
13.         product = product * i;
14.     }
15.
16.     /* Return function result */
17.     return (product);
18. }
```

การเขียนโปรแกรมด้วยการวนซ้ำ (iterative) จะมีประสิทธิภาพมากกว่าการใช้ฟังก์ชันเรียกตัวเอง (recursive)



เลขลำดับฟีโบนัชชี (fibonacci sequence)

เลขลำดับฟีโบนัชชีเป็นเลขลำดับที่มีชื่อเสียงมาก ซึ่งมีนิยามดังนี้

$$F(1) = F(2) = 1 \text{ และ } F(n) = F(n-1) + F(n-2) \text{ เมื่อ } n=3,4,\dots$$

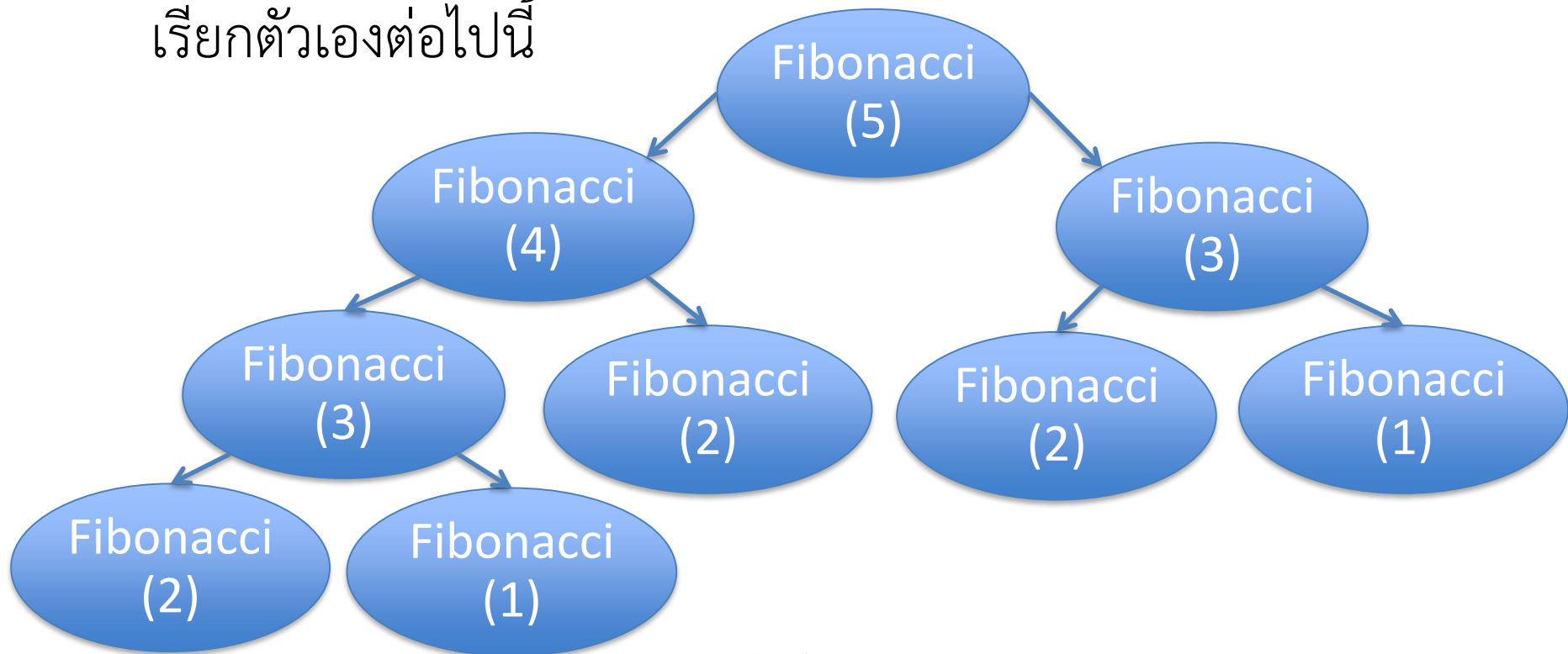
วิธีการแก้ปัญหานี้ เราอาจใช้ฟังก์ชันเรียกตัวเอง

```
1.  /*
2.   * Computes the nth Fibonacci number
3.   * Pre: n > 0
4.   */
5.  int
6.  fibonacci(int n)
7.  {
8.      int ans;
9.
10.     if (n == 1 || n == 2)
11.         ans = 1;
12.     else
13.         ans = fibonacci(n - 2) + fibonacci(n - 1);
14.
15.     return (ans);
16. }
```



เลขลำดับฟีโบนัชชี (fibonacci sequence)

- ซึ่งจะเห็นว่าเป็นวิธีที่ไม่ดีเลย เพราะค่าลำดับของฟีโบนัชชีค่าเดียวกันอาจจะถูกคำนวณมากกว่าหนึ่งครั้ง ลองดูโครงสร้างการเรียกตัวเองต่อไปนี้





การหาค่าหารร่วมมาก(gcd)

- ถ้าอัลกอริทึมที่ใช้แก้ปัญหามีกำหนดให้ดำเนินการเรียกตัวเอง เราก็ควรจะใช้ฟังก์ชันเรียกตัวเองในการแก้ปัญห
- ตัวอย่างเช่นการหาค่าหารร่วมมาก (the greatest common divisor:GCD) ของสองจำนวนเต็ม m และ n

อัลกอริทึมการหาค่าหารร่วมมาก

$\text{gcd}(m,n)$ มีค่าเป็น n เมื่อ n หาร m ลงตัว

$\text{gcd}(m,n)$ มีค่าเป็น $\text{gcd}(n, \text{เศษที่เหลือจากการหาร } m \text{ ด้วย } n)$



Recursive **gcd** function

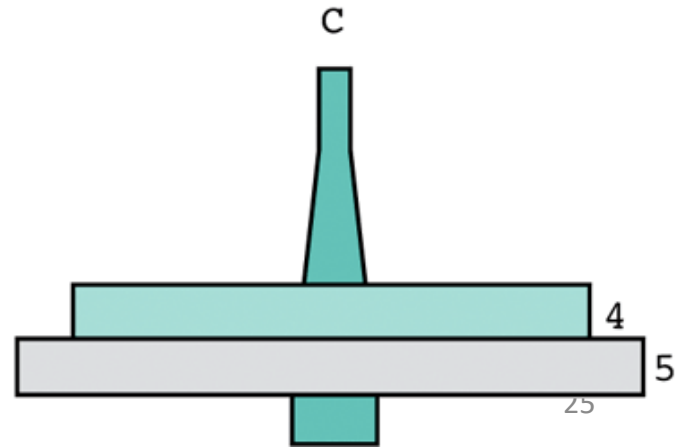
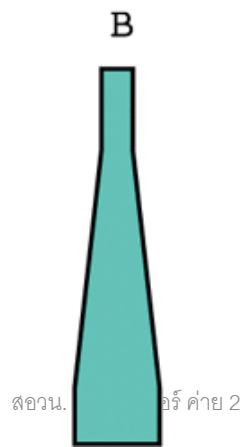
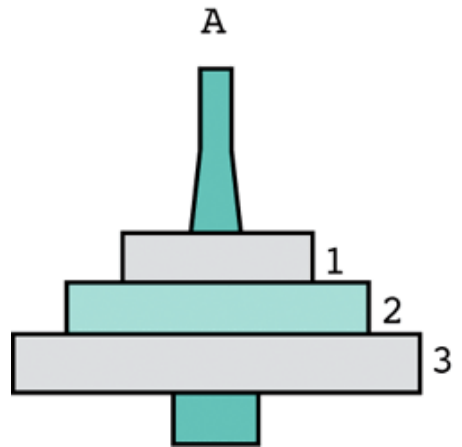
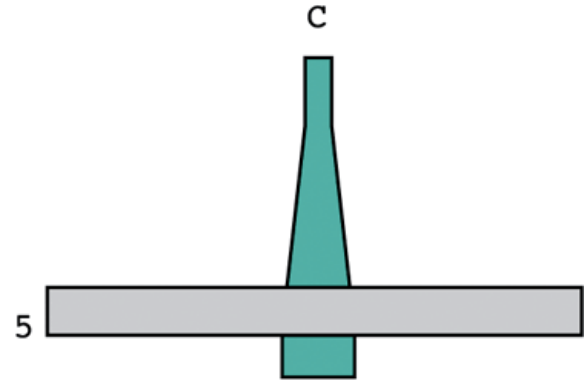
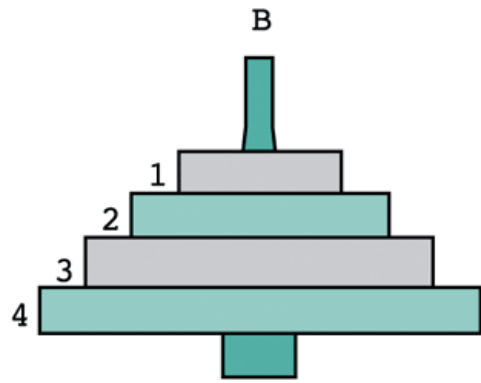
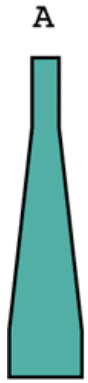
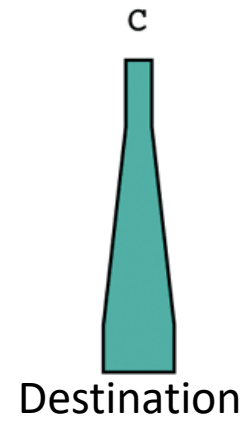
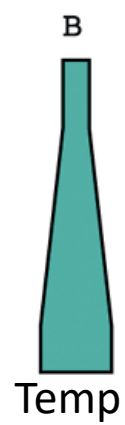
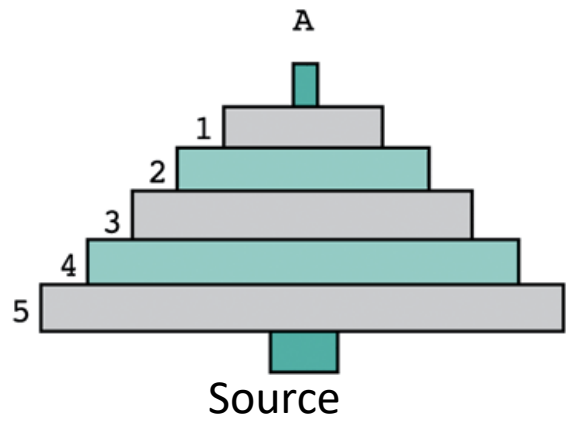
```
1.  /*
2.   *   Displays the greatest common divisor of two integers
3.   */
4.
5.  #include <stdio.h>
6.
7.  /*
8.   *   Finds the greatest common divisor of m and n
9.   *   Pre:  m and n are both > 0
10.  */
11. int
12. gcd(int m, int n)
13. {
14.     int ans;
15.
16.     if (m % n == 0)
17.         ans = n;
18.     else
19.         ans = gcd(n, m % n);
20.
21.     return (ans);
22. }
```

(continued)

A classical case : Towers of Hanoi



- ปัญหาหอคอยฮานอย (The towers of Hanoi) เป็นการเคลื่อนจำนวนของจาน (ที่มีขนาดต่างกัน) จากหอคอยหนึ่งไปอีกหอคอย
- ข้อจำกัดของปัญหาคือจากขนาดใหญ่จะไม่สามารถวางบนจานขนาดเล็กกว่าได้
- เราย้ายได้แค่ 1 จานต่อครั้งเท่านั้น
- ให้มี 3 หอคอยที่สามารถใช้ได้





A classical case : Towers of Hanoi

- ปัญหานี้แก้ได้ด้วยการเรียกตัวเอง

อัลกอริทึม

ถ้า n เป็น 1

ย้าย disk 1 จาก source tower ไป destination tower

ถ้า n ไม่เป็น 1

1. ย้าย $n-1$ disks จาก source tower ไปที่ temp tower
2. ย้าย disk n จาก source tower ไปที่ destination

tower

3. ย้าย $n-1$ disks จาก temp tower ไปที่ source tower



A classical case : Towers of Hanoi

```
1.  /*
2.   * Displays instructions for moving n disks from from_peg to to_peg using
3.   * aux_peg as an auxiliary. Disks are numbered 1 to n (smallest to
4.   * largest). Instructions call for moving one disk at a time and never
5.   * require placing a larger disk on top of a smaller one.
6.   */
7.  void
8.  tower(char from_peg,    /* input - characters naming the three pegs */
9.        char to_peg,     /* the problem's goal */
10.        char aux_peg,    /* the auxiliary peg */
11.        int n)           /* input - number of disks to move */
12.  {
13.    if (n == 1) {
14.      printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
15.    } else {
16.      tower(from_peg, aux_peg, to_peg, n - 1);
17.      printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
18.      tower(aux_peg, to_peg, from_peg, n - 1);
19.    }
20.  }
```

The recursive step

The recursive step

A classical case : Towers of Hanoi



ผลลัพธ์เมื่อเรียก Tower('A', 'B', 'C', 3);

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C



จงเขียนโปรแกรมแก้ปัญหานี้

- หาค่าของฟังก์ชันต่อไปนี้

$$f(n) = 2 * f(n - 1) + 1 \text{ when } f(0) = 1$$



$$F(n) = 2 * F(n-1) + 1$$

```
int f(int x)
{
    int y;

    if (x==0)
        return 1;
    else {
        y = 2 * f(x-1);
        return y+1;
    }
}
```



push copy of f

x = 3
y = ?
call f(2)

push copy of f

x = 2
y = ?
call f(1)

push copy of f

x = 1
y = ?
call f(0)

push copy of f

x = 0
y = ?

return ①

pop copy of f

y = 2 * 1 = 2

return y + 1 = ③

pop copy of f

y = 2 * 3 = 6

return y + 1 = ⑦

pop copy of f

y = 2 * 7 = 14

return y + 1 = ⑮

pop copy of f

value returned by call is 15



จงเขียนโปรแกรมแก้ปัญหาดังต่อไปนี้

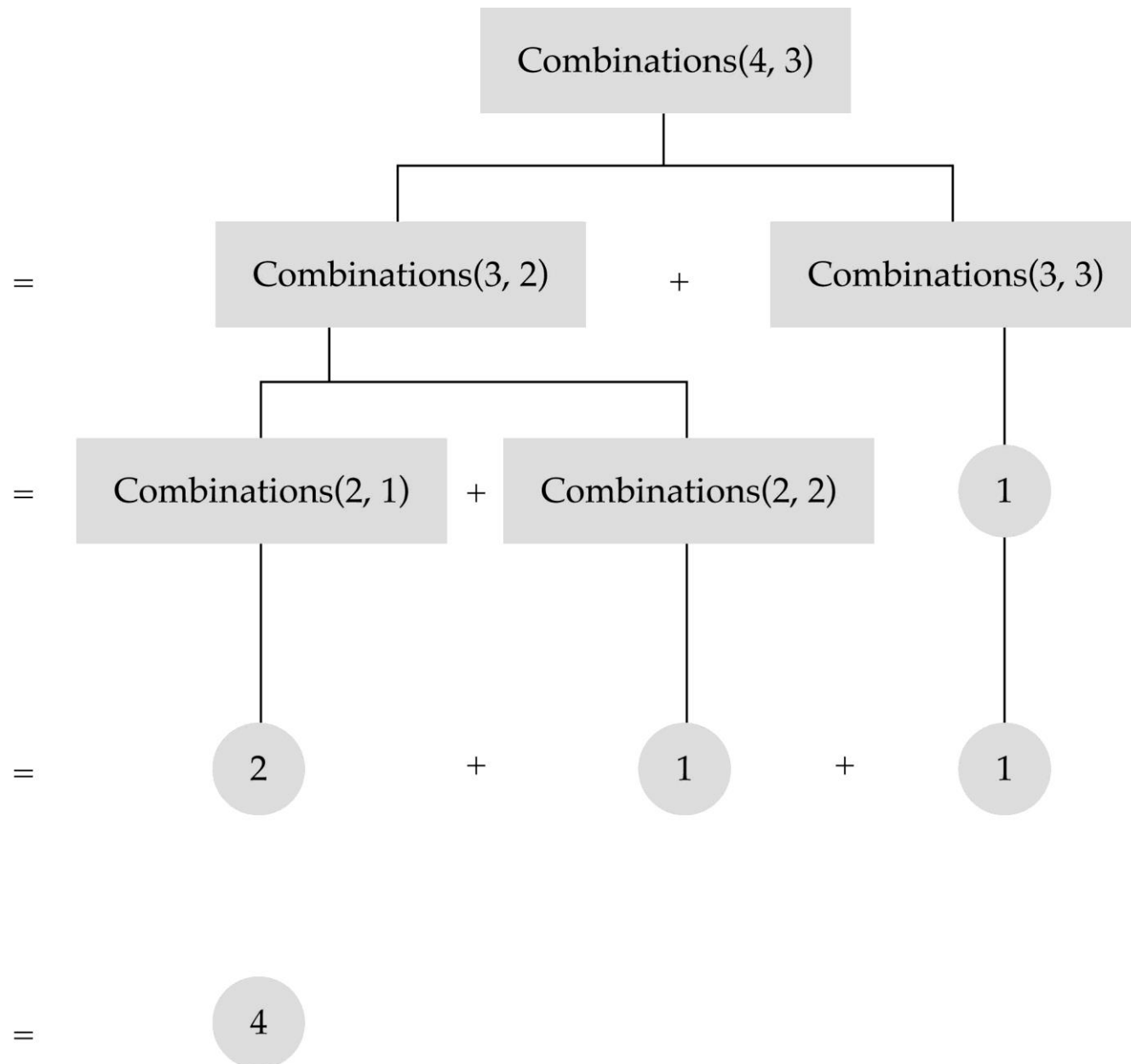
- n choose k Combinations
- ถ้ามี n สิ่ง จะมีวิธีที่แตกต่างกันในการเลือกสิ่งของ k ชิ้นกี่วิธี

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, 1 < k < n$$

$$\binom{n}{k} = \frac{n!}{k! (n-k)!}, 1 < k < n$$

โดย

$$\binom{n}{1} = n \text{ เมื่อ } k = 1, \binom{n}{n} = 1 \text{ เมื่อ } k = n$$





n choose k Combinations

```
int combinations(int n, int k)
{
    if(k == 1)    // base case 1
        return n;
    else if (n == k)    // base case 2
        return 1;
    else
        return (Combinations(n-1, k) +
                Combinations(n-1, k-1));
}
```



จงใช้ฟังก์ชันเรียกตัวเองเพื่อหาสิ่งต่อไปนี้

$$a_n = cn + (n - 1)a_{n-1}$$

เมื่อ c คือค่าคงที่

n คือตัวเลขจำนวนเต็มบวก

a_n คือค่าที่ n