บทที่ 7

การเขียนโปรแกรมเรียกตัวเอง

วัตถุประสงค์

1. เพื่อให้นักศึกษาทราบถึงวิธีการเขียนโปรแกรมเรียกตัวเอง
2. เพื่อให้นักศึกษาทราบถึงความแตกต่างระหว่างการเขียนโปรแกรมทำงานซ้ำกับการเขียนโปรแกรมเรียกตัวเอง
3. เพื่อให้นักศึกษาสามารถเขียนโปรแกรมเรียกตัวเองเพื่อแก้ปัญหาโปรแกรมได้

Recursive หรือ Recursion เป็นการเขียนคำสั่งโปรแกรมที่มีการปฏิบัติงานเรียกตัวเอง โดยมีการเรียกการทำงาน ในลำดับก่อนมาทำงานในลำดับปัจจุบัน โดยจะเรียกลำดับก่อนวนซ้ำไปเรื่อยๆ จนถึงข้อมูลที่กำหนดการสิ้นสุดในการเรียกตัวเอง การทำงานจะเขียนในรูปแบบของทางเลือกในการทำงาน การเขียนโปรแกรมเรียกตัวเองจะสร้างในรูปแบบของฟังก์ชันโดยการปฏิบัติงานในฟังก์ชันต้องมี

a. ค่าเริ่มต้นการทำงานวนรอบ โดยทั่วไปเป็น formal parameter ที่มีการส่งผ่านค่ามาจากจุดเรียกใช้
b. ค่าสิ้นสุดการทำงานเป็นค่าของข้อมูลสุดท้าย ที่ผ่านการเรียกตัวเองโดยลดค่าลงอย่างเป็นลำดับ ในการเรียกฟังก์ชันแต่ละครั้ง
c. คำสั่งในการปฏิบัติงาน เรียกตัวเองเป็นการประมวลผลที่นำผลลัพธ์จากการทำงานในรอบที่แล้วมาปฏิบัติงานในรอบปัจจุบัน เพื่อส่งผลลัพธ์จากการทำงานไปยังจุดเรียกใช้

ภาษาโปรแกรมที่สามารถเขียนโปรแกรมเรียกตัวเองได้ เช่น Pascal,C, C++, C# แต่ก็มีบางภาษาที่เขียนโปรแกรมเรียกตัวเองไม่ได้ เช่น Basic ,Fortran และอื่นๆ โดยทั่วไปการเรียกตัวเองเปรียบเสมือนการทำงานที่กระทำซ้ำๆกันในรูปของคำสั่ง while, for แต่เราสามารถนำมาเขียนในรูปแบบที่เรียกตัวเองโดยแทนด้วยคำสั่งเงื่อนไข การกระทำซ้ำจะแทนด้วยการเรียกฟังก์ชันในลำดับก่อนทำงาน ซึ่งการเขียนโปรแกรมแนวนี้มีทั้งข้อดีและข้อเสียขึ้นอยู่กับงานประยุกต์ที่จะแก้ปัญหา ภาษา C++ นั้นอนุญาตให้ฟังก์ชันสามารถมีการเรียกใช้ตัวเองได้ โดยปรับเปลี่ยนการทำงานที่มีลักษณะเป็นการปฏิบัติงานที่ซ้ำๆหรือเหมือนๆกันเป็นคำสั่งเงื่อนไขที่มีการเรียกใช้ตัวเอง ซึ่งมีทางเลือกในการทำงานที่สามารถทำให้การเรียกตัวเองนี้สิ้นสุดได้ รูปแบบของคำสั่งเป็นดังนี้

```
if the stopping case is reached
    Return a value for the stopping case
else
    Return a value computed by calling the function again with
    different arguments.
```

ตัวอย่างที่ 7.1 ฟังก์ชันหาค่าแฟคตอเรียล

ฟังก์ชันนี้เป็นตัวอย่างในการหาค่าแฟคตอเรียล ใดๆ โดยค่าแฟคตอเรียลเป็นการคำนวณหาผลคูณของตัวเลขที่มีการลดตัวคูณอย่างเป็นลำดับจนกระทั่งมีค่าเท่ากับ 1 ดังนี้

10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1

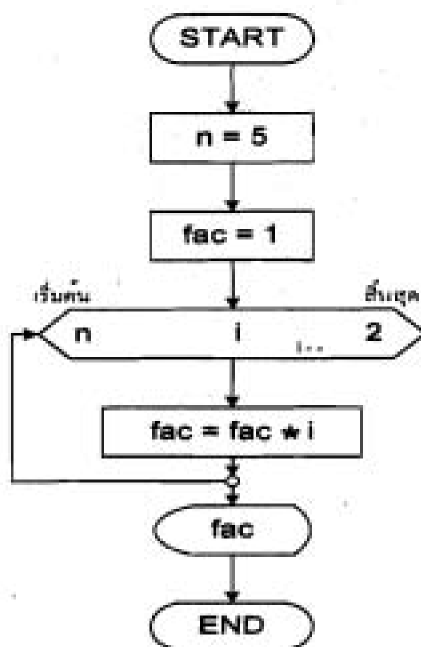
5! = 5 * 4 * 3 * 2 * 1

4! = 4 * 3 * 2 * 1

1! = 1

0! = 1

การหาค่าแฟคตอเรียลนี้เราสามารถนำคำสั่งการทำงานวนรอบเช่น for มาใช้ในการลดค่าตัวคูณลงจนกระทั่งมีค่าเท่ากับ 1 จึงหยุดได้ โดยออกแบบโดยใช้ผังโปรแกรมได้ดังนี้



n = 5

fac =1

n	fac
5	5 * 1
4	4 * 5
3	3 * 20
2	2 * 60

เราสามารถออกแบบฟังก์ชันการทำงานชื่อ fac โดยรับค่าที่ส่งผ่านมาเก็บในตัวแปร n ได้ดังนี้

```
int fac (int n)
{
        int sum, i;
        sum = 1;
        for (i = n; i > 1;  i - -)
                sum = sum * i;
        return sum;
}
```

การเรียกใช้

```
        output = fac (5);
```

ผลของการทำงานมีค่าเท่ากับ 120

จากตัวอย่างนี้เราจะมาเขียนฟังก์ชันแบบเรียกตัวเอง โดยใช้คำสั่ง if แทนคำสั่งวนรอบ for โดยดัดแปลงการทำงานให้อยู่ในรูปแบบของเงื่อนไข ให้สามารถเรียกฟังก์ชันซ้ำในลักษณะของการเรียกตัวเองได้ โดยหาให้ได้ว่าจุดเริ่มต้นอยู่ที่ไหน จุดสุดท้ายของการกระทำซ้ำอยู่ที่ใด และการปฏิบัติงานเพื่อเรียกการทำงานในลำดับที่แล้วเป็นอย่างไร เพื่อให้เข้าใจได้ดีขึ้นพิจารณาการคำนวณหาค่าแฟคตอเรียลดังนี้

5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!

4! = 4 * 3 * 2 * 1 = 4 * 3!

3! = 3 * 2 * 1 = 3 * 2!

2! = 2 * 1 = 2 * 1!

1! = 1

0! = 1

จะเห็นได้ว่า การหาค่าแฟคตอเรียลของลำดับปัจจุบันจะนำผลลัพธ์จากการทำงานในลำดับที่แล้วมาคูณกัน โดยสิ้นสุดเมื่อค่ามีค่าเท่ากับ 0 หรือ 1 เราสามารถสรุปการทำงานได้ดังนี้

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

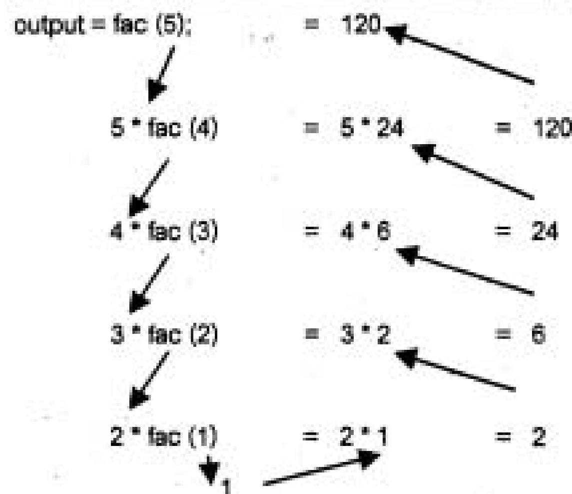
$$n! = n * (n-1)!$$

เราสามารถเขียนฟังก์ชันเรียกตัวเองได้ดังนี้

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

การทำงานในแต่ละรอบ จะมีการนำค่าปัจจุบันคูณด้วยผลลัพธ์ของการทำงานในรอบที่แล้วเห็นจากการลดค่าของตัวแปรลง 1 โดยการปฏิบัติงานแต่ละรอบเหมือนกับการวนรอบนำค่าของ n มาคูณกันจนกระทั่งค่าสุดท้ายเป็น 1 นั่นเอง

การเรียกใช้ และการทำงาน



ผลการทำงาน output มีค่าเท่ากับ 120

ตัวอย่างที่ 7.2 ฟังก์ชันหาค่ายกกำลัง

ฟังก์ชันนี้เป็นตัวอย่างในการหาค่ายกกำลังที่มีการเรียกตัวเองที่มีการส่งผ่านค่าอาร์กิวเมนต์ 2 ค่า เพื่อปฏิบัติงาน โดยเขียนเป็นฟังก์ชันเรียกตัวเองในการหาค่า x^y ใดๆ กำหนดให้ ตัวแปร x มีชนิดของมูลเป็น Double และ y เป็นชนิดข้อมูล int

วิธีการนั้นต้องสมมุติค่า x และ y ใดๆขึ้นมาพิจารณา

กรณีที่ x=3 และ y=3

$3^3 = 3 * 3 * 3$

$3^2 = 3 * 3$

$3^1 = 3$

$3^0 = 1$

กรณีที่ x=3 และ y=-3

$3^{-3} = \frac{1}{3} * \frac{1}{3} * \frac{1}{3}$

$3^{-2} = \frac{1}{3} * \frac{1}{3}$

$3^{-1} = \frac{1}{3}$

$3^0 = 1$

จะเห็นได้ว่าการเรียกตัวเองเมื่อ y เป็นจำนวนเต็มบวกมีการลดค่า y ครั้งละ 1 อย่างเป็นลำดับ ในการเรียกตัวเองแต่ละรอบจนกระทั่งค่า y=0 จึงหยุดการเรียกตัวเอง แต่กรณีที่ y มีค่าน้อยกว่า 0 การเรียกตัวเองแต่ละรอบจะมีการเพิ่มค่า y ขึ้น 1 จนกระทั่งค่า y เป็น 0 จึงหยุดการทำงานเช่นกัน ดังนั้นการเรียกตัวเองของฟังก์ชันนี้จะมีเงื่อนไขการทำงาน 3 ทางด้วยกัน คือเมื่อ y มีค่าเป็นจำนวนเต็มบวก , จำนวนเต็มลบ และ จำนวนเต็มศูนย์

y เป็นจำนวนเต็มบวก

$x^y = x * x^{y-1}$

$3^3 = 3 * 3^2$

y เป็นจำนวนเต็มลบ

$x^y = \frac{1}{x} * x^{y+1}$

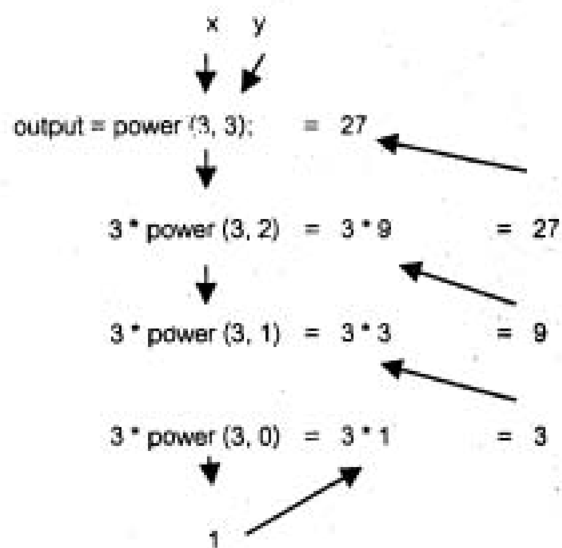
$3^{-3} = \frac{1}{3} * 3^{-2}$

สามารถเขียนเป็นฟังก์ชันชื่อ power ได้ดังนี้

```
double power (double x, int y)
{
        if (y == 0)
                return 1;
        else if (y > 0)
                return x * power (x, y – 1);
        else
                return 1/x * power (x, y + 1);
}
```

กรณีที่ y มีค่าเป็นเลขจำนวนเต็มบวก

การเรียกใช้ และการทำงาน



ผลของการทำงาน output มีค่าเท่ากับ 27.0

กรณีที่ y มีค่าเป็นเลขจำนวนเต็มลบ

output = power (3, -3); $= \frac{1}{27}$

$\frac{1}{3}$ * power (3, -2) $= \frac{1}{3} \cdot \frac{1}{9} = \frac{1}{27}$

$\frac{1}{3}$ * power (3, -1) $= \frac{1}{3} \cdot \frac{1}{3} = \frac{1}{9}$

$\frac{1}{3}$ * power (3, 0) = $\frac{1}{3}$ * 1 $= \frac{1}{3}$

1

ผลของการทำงาน output มีค่าเท่ากับ $\frac{1}{27}$

ตัวอย่างที่ 7.3 ฟังก์ชันหาค่า Fibonacci

ฟังก์ชันนี้เป็นการหาค่า Fibonacci ของลำดับที่ต้องการ โดยเขียนแบบเรียกตัวเอง

การหาค่า Fibonacci นั้นเราต้องกำหนดลำดับที่ต้องการ และต้องทราบค่าของลำดับ 1 และ ลำดับที่ 2 จึงจะหาค่าของลำดับอื่นๆได้ ในที่นี้ถ้าเรากำหนดให้ค่าของลำดับที่ 1 และค่าของลำดับที่ 2 มีค่าเท่ากับ 1 ดังนั้นข้อมูลในลำดับต่อๆไปมีค่าดังนี้

1 1 2 3 5 8 13 21 34..............

จะเห็นได้ว่าการทำงานจะเริ่มจากลำดับที่ 3 เป็นต้นไป โดยค่าของลำดับที่ 3 จะเกิดจากนำค่าของลำดับที่ 1 บวกกับค่าในลำดับที่ 2 และทำนองเดียวกัน ค่าในลำดับที่ 4 จะเกิดจากค่าในลำดับที่ 2 บวกกับค่าในลำดับที่ 3 กล่าวกันง่ายๆคือ นำค่าของลำดับก่อนหน้า 2 ลำดับมาบวกกันนั่นเอง

ในที่นี้ ถ้า n = 7

$$f(7) = f(5) + f(6)$$

$$13 = 5 + 8$$

จากการคำนวณในลักษณะนี้นี่เองเราสามารถเขียนเป็นฟังก์ชันการทำงานโดยสรุปได้ดังนี้

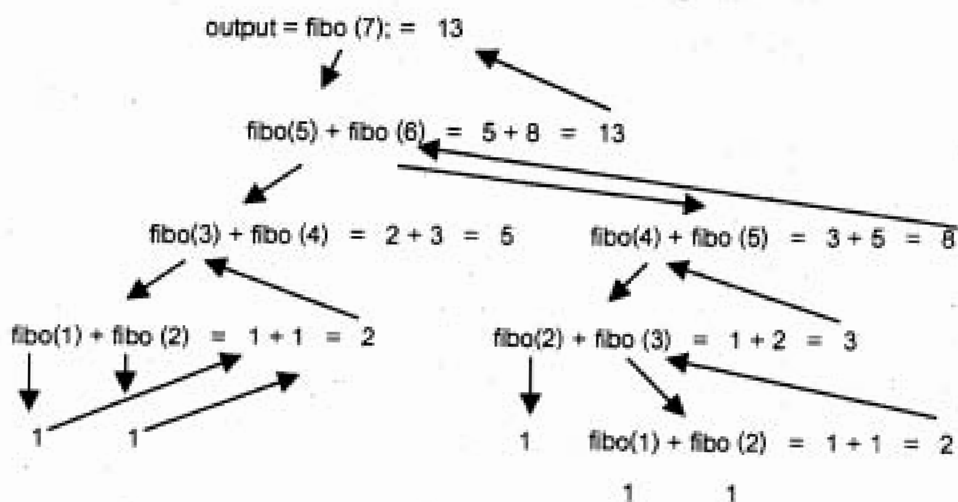
f(n) = 1 ถ้า n = 1 หรือ n = 2 // n คือ ลำดับ

= f(n – 2) + f(n – 1) ถ้า n > 2

และนำมาเขียนเป็นฟังก์ชันเรียกตัวเองได้ดังนี้

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

การเรียกใช้ และการทำงาน



จะเห็นได้ว่า

fibo (7) = fibo (5) + fibo (6) = 13

fibo (6) = fibo (4) + fibo (5) = 8

fibo (5) = fibo (3) + fibo (4) = 5

fibo (4) = fibo (2) + fibo (3) = 3

fibo (3) = fibo (1) + fibo (2) = 2

fibo (2) = 1

fibo (1) = 1

ผลของการทำงาน output มีค่าเท่ากับ 13

ตัวอย่างที่ 7.4 ฟังก์ชันหาค่า Binary search

การค้นหาข้อมูลที่ต้องการจากตัวแปรอาเรย์ 1 มิติที่มีการเก็บต่อๆกันในหน่วยความจำนั้น มีอยู่ด้วยกันหลายวิธี เช่น Sequential search เป็นการค้นหาเป็นลำดับเริ่มจากข้อมูลที่เก็บในช่องแรกจนกระทั่งถึงตัวสุดท้าย การค้นหาวิธีนี้จะใช้เวลาน้อยมากถ้าข้อมูลที่ต้องการค้นหาอยู่ในช่องแรกๆ แต่ถ้าข้อมูลที่ต้องการค้นหาอยู่ท้ายๆกลุ่มหรือไม่พบจะเสียเวลามาก ทำให้เวลาเฉลี่ยในการค้นหาข้อมูลแต่ละตัวใช้เวลาแตกต่างกันมาก สำหรับวิธี Binary search นั้นเป็นการค้นหาที่แก้ปัญหาทำให้เราสามารถใช้เวลาเฉลี่ยในการค้นหาข้อมูลแต่ละตัวเท่าๆกัน ถึงแม้ว่าข้อมูลจะอยู่ในตำแหน่งใดก็ตามจะพบหรือไม่พบก็ตาม

วิธีการนั้นสมมุติว่ามีข้อมูลที่เก็บในหน่วยความจำเรียงต่อๆกันดังรูป

0	1	2	3	4	5	6	7
31	63	82	25	64	77	50	10

1. นำข้อมูลมาเรียงลำดับจากน้อยไปมาก หรือจากมากไปน้อยก็ได้

0	1	2	3	4	5	6	7
10	25	31	50	63	64	77	82

2. หาข้อมูลตำแหน่งกลางของกลุ่ม

first = 0

last = 7

$$mid = \frac{0+7}{2} = 3$$

3. แบ่งกลุ่มออกเป็น 2 กลุ่ม นำค่าที่ต้องการค้นหา เปรียบเทียบกับตำแหน่งตรงกลาง ถ้าใช่ตำแหน่ง mid จะ return mid และหยุดค้นหา แต่ถ้าค่าน้อยกว่า จะพิจารณาเฉพาะกลุ่มทางซ้าย โดยกำหนดค่า first = 0 และให้ last = mid – 1 แต่ถ้าค่าที่ต้องการค้นหามากกว่าตำแหน่ง mid จะพิจารณา เฉพาะกลุ่มทางขวา โดยกำหนดให้ค่า first = mid + 1 และค่า last = 7 เป็นการกำหนดกลุ่มของข้อมูลใหม่มีขนาดเล็กลงไปเรื่อยๆ แต่ถ้าแบ่งข้อมูลจนกระทั่งไม่มีข้อมูลที่ต้องการค้นหาแล้วแสดงว่า ไม่พบข้อมูลที่ต้องการจะส่งให้ค่า first > last เราจะส่งค่า –1 กลับเพื่อให้ผู้ใช้ทราบว่าไม่พบข้อมูลนั่นเอง การทำงานจะเป็นการเรียกตัวเองโดยกระทำซ้ำข้อ 2-3 จนพบข้อมูล หรือไม่พบข้อมูล

จากตัวอย่างนี้เรากำหนดตัวแปรต่างๆในการปฏิบัติงานดังนี้

table เป็น array 1 มิติ เก็บข้อมูลเป็นเลขจำนวนเต็มใดๆ

target ค่าที่ต้องการค้นหา

first เก็บตำแหน่งแรกของกลุ่ม

last เก็บตำแหน่งสุดท้ายของกลุ่ม

mid เก็บตำแหน่งกลาง

<u>การเรียกใช้</u>

```
loc = binSearch (table, 25, 0, 7);
```

0	1	2	3	4	5	6	7
10	25	31	50	63	64	77	82

โดย table คือ โครงสร้างของอาเรย์ 1 มิติที่มีข้อมูลเป็นเลขจำนวนเต็มเก็บอยู่

25 คือ ค่าที่ต้องการค้นหา

0 คือ ตำแหน่งแรก

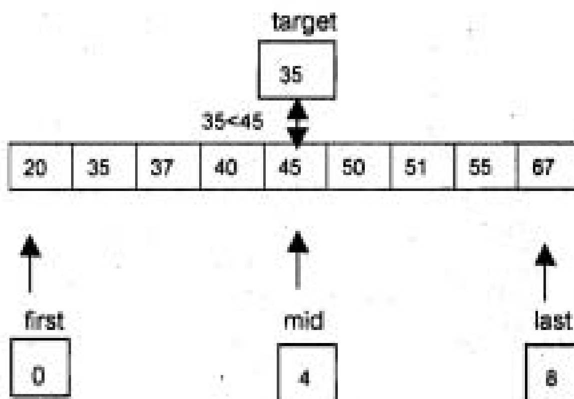
7 คือ ตำแหน่งสุดท้าย

เราสามารถเขียนฟังก์ชัน binSearch ได้ดังนี้

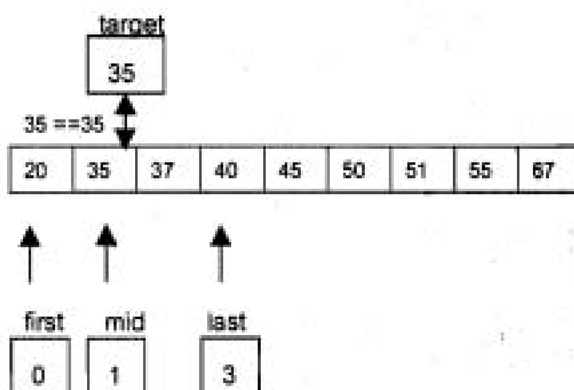
```
// Searches for target in elements first through last of array
//      Precondition : The elements of table are sorted & first and last are defined.
//      Postcondition: IF target is in the array, return its position; otherwise, returns -1.

int binSearch (int table[], int target, int first, int last)
{
        int mid;

        mid = (first + last)/2;
        if (first > last)           //หาไม่พบ
                return -1;
        else if (target = = table[mid])   //ค้นพบในตำแหน่งกลาง
                return mid;
        else if (target < table[mid])
                return binSearch (table, target, first, mid - 1);
        else
                return binSearch (table, target, mid + 1, last);
}
```

สำหรับการทำงานจะมีเรียกตัวเองโดยมีการเปลี่ยนแปลงค่าของ first และ last ในการทำงานแต่ละรอบโดยแบ่งกลุ่มของข้อมูลออกเป็นสองส่วน จนกระทั่งพบค่าที่ต้องการอยู่ในตำแหน่งกลางของกลุ่มในที่นี้คือตำแหน่งที่ 1 ฟังก์ชันจะหยุดการทำงานและส่งผ่านค่า 1 กลับมาให้แก่ตัวแปร loc

การทำงานของฟังก์ชันแสดงได้จากรูปภาพต่อไปนี้ โดยการทำงานในรอบแรกจะพิจารณาข้อมูลทั้งหมดก่อน



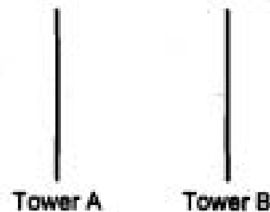
การทำงานในรอบต่อๆไปจะมีการปรับเปลี่ยนกลุ่มของข้อมูลโดยแบ่งครึ่งจากข้อมูลเดิมและกระทำในลักษณะเดียวกัน



ถ้าข้อมูลที่ต้องการค้นหาอยู่ในตำแหน่ง mid จะหยุดการกระทำซ้ำและส่งตำแหน่งของข้อมูลที่ต้องการค้นหากลับมายังจุดเรียกใช้

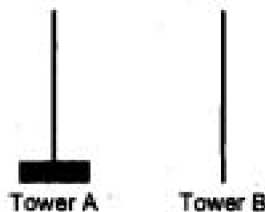
ตัวอย่างที่ 7.5 ฟังก์ชันหอคอยฮานอย

The Tower of Hanoi หรือ หอคอยฮานอย เป็นตัวอย่างของการปฏิบัติงานในลักษณะของการเรียกตัวเองที่ไม่มีการคำนวณหรือส่งผ่านค่าใดๆกลับ แต่เป็นการปฏิบัติงานที่กระทำในลักษณะที่เป็นขั้นตอนที่คล้ายๆกันเป็นการย้ายสิ่งของจากหอหนึ่งไปยังอีกหอหนึ่ง



สมมุติว่าต้องการย้ายสิ่งของมีด้วยกันหลายขนาดจากหอ A ไปยังหอ B โดยสิ่งของมีหลายขนาดแตกต่างกัน เรานำสิ่งของมาซ้อนทับกันโดยสิ่งของขนาดเล็กต้องอยู่บนขนาดใหญ่เท่านั้น สมมุติว่าสิ่งของทุกชิ้นมีช่องตรงกลาง ให้ผู้ย้ายสามารถใส่หรือนำออกในหอต่างๆได้ การย้ายสิ่งของกำหนดให้ย้ายได้ครั้งละ 1 ชิ้น เท่านั้น การย้ายสิ่งของนั้นวิธีการขึ้นอยู่กับจำนวนชิ้นที่มีอยู่ในหอ

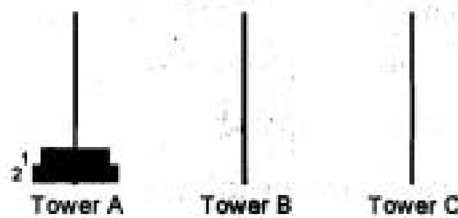
กรณีที่ 1 มีสิ่งของ 1 ชิ้นที่หอ A ต้องการย้ายไปที่ หอ B



เราสามารถนำสิ่งของจากหอ A ย้ายไปที่หอ B ได้ทันที

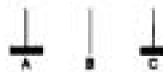
```
n = 1 ;
cout <<"Move 1 from A to B";
```

กรณีที่ 2 มีสิ่งของ 2 ชิ้น ขนาดไม่เท่ากัน ต้องการย้ายจากหอ A ไปที่ หอ B

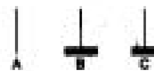


การย้ายสิ่งของไม่สามารถกระทำได้โดยตรงเพราะขนาดของสิ่งของไม่เท่ากัน และข้อกำหนด ตกลงไว้ว่าสิ่งของชิ้นเล็กต้องอยู่บนชิ้นใหญ่เสมอ ดังนั้นเราต้องกำหนดหอ C มาช่วยในการย้าย และมีกระบวนการย้ายแบ่งเป็น 3 ขั้นตอนดังนี้

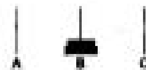
Move 1 from A to C



Move 2 from A to B

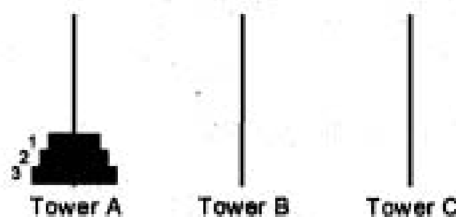


Move 1 from C to B



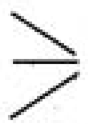
การทำงานจะทำการย้ายสิ่งของชิ้นเล็กสุดก่อนโดยไปฝากไว้ที่หอ C และย้ายสิ่งของที่ใหญ่ที่สุดไปไว้ที่หอ B และย้ายสิ่งของชิ้นเล็กกลับมาไว้ที่หอ B ดังรูป

กรณีที่ 3 มีสิ่งของ 3 ชิ้นขนาดไม่เท่ากัน ต้องการย้ายจากหอ A ไปไว้ที่หอ B

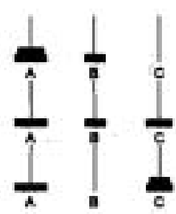


เราจะแบ่งขั้นตอนในการย้ายสิ่งของออกเป็น 3 ขั้นตอนใหญ่แต่เป็น 7 ขั้นตอนย่อยด้วยกัน วิธีการย้ายจะใช้ลักษณะการย้ายเหมือนกับกรณีที่ 2 มาใช้ โดยจะนำสิ่งของที่มีขนาดใหญ่ที่สุดมาไว้ที่หอ B แต่การจะนำสิ่งของที่ใหญ่ที่สุดมาไว้ได้ต้องนำสิ่งของที่มีขนาดเล็กกว่ามาฝากไว้ที่หอ C เสียก่อนซึ่งในที่นี้มีอยู่ 2 ชิ้นดังนั้นจึงเป็นการย้ายสิ่งของ 2 ชิ้นจากหอ A มาไว้ที่หอ C ซึ่งจากกรณีที่ 2 ต้องกระทำ 3 ขั้นตอนแต่กรณีนี้หอ B กลายเป็นหอที่รับฝากสิ่งของ และทำนองเดียวกันเมื่อย้ายสิ่งของที่ใหญ่ที่สุดมาไว้ที่หอ B แล้วก็ต้องนำเอาสิ่งของที่ฝากไว้ที่หอ C ย้ายกลับมาที่หอ B ก็ต้องกระทำอีก 3 ขั้นตอนเพราะไม่ได้มีสิ่งของชิ้นเดียวมีถึง 2 ชิ้นต้องนำสิ่งของไปฝากไว้ที่หอ A และกระทำการย้ายอีก 3 ขั้นตอน สรุปได้ว่าการย้ายสิ่งของ 3 ชิ้นจากหอ A ไป หอ B ต้องกระทำทั้งสิ้น 7 ขั้นตอนด้วยกัน ดังนี้

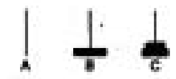
Move 1 from A to B
Move 2 from A to C
Move 1 from B to C



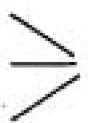
Move 2 from A to C



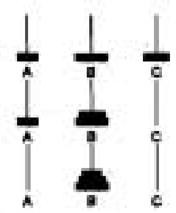
Move 3 from A to B



Move 1 from C to A
Move 2 from C to B
Move 1 from A to B



Move 2 from C to B



เราสามารถนำมาเขียนเป็นฟังก์ชันชื่อ tower ได้ดังนี้

```
void tower (char from, char too, char temp, int n)
{
        if (n = = 1)
```

```
            cout << "Move Object 1 from tower" << from
                  << "to tower" << too << endl;
        else
        {
            tower (from, temp, too, n – 1);
            cout <<"Move Object" <<n <<"from tower" <<from
                  <<"to tower" <<too <<endl;
            tower (temp, too, from, n – 1);
        }
}
```

ในที่นี้ n คือจำนวนสิ่งของที่ต้องการย้าย

from คือหอที่มีสิ่งของอยู่จำนวน n ชิ้นโดยชิ้นเล็กซ้อนทับชิ้นใหญ่

too คือหอที่ต้องการย้ายสิ่งของไปเก็บไว้

temp คือหอที่ใช้สำหรับรับฝากสินค้าไว้เพื่อขนย้าย

ในกรณีที่ n = 1

```
     from  too  temp  n
tower ('A', 'B', 'C', 1);
```

ผล Move Object 1 from tower A to tower B

ในกรณีที่ n = 2

```
tower ('A', 'B', 'C', 2);
```

จะกระทำ 3 คำสั่งดังนี้

1. tower (from, temp, too, n – 1); ①

   tower ('A', 'C', 'B', 1);

   ผล Move Object 1 from tower A to tower C

2. พิมพ์ ②

   tower ('A', 'B', 'C', 2);

   ผล Move Object 2 from tower A to tower B

3. tower ('C', 'B', 'A', 1); ③

   ผล Move Object 1 from tower C to tower B

ในกรณีที่ n = 3

```
tower ('A', 'B', 'C', 3);
```

จะกระทำ 3 ขั้นตอนหลักเหมือนกัน แต่มีการเรียกตัวเองเพื่อปฏิบัติงานในลักษณะซ้ำกัน ผลของการทำงาน จะปฏิบัติงานทั้งหมด 7 ขั้นตอนด้วยกัน ดังนี้

ผลของการทำงาน

Move disk 1 from tower A to tower C

Move disk 2 from tower A to tower B

Move disk 1 from tower C to tower B

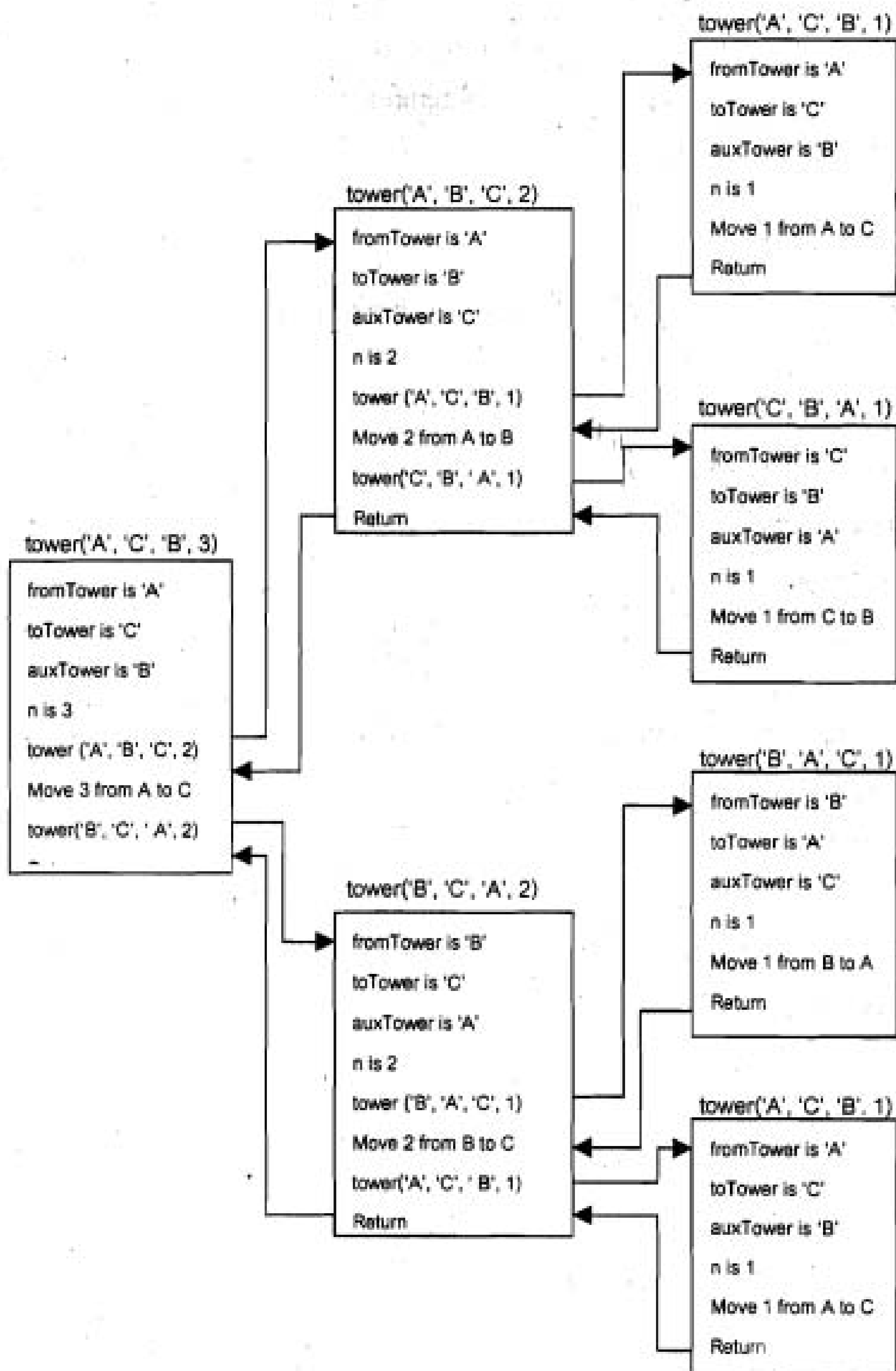
Move disk 3 from tower A to tower C

Move disk 1 from tower B to tower A

Move disk 2 from tower B to tower C

Move disk 1 from tower A to tower C

เราสามารถติดตามการทำงานได้จากแผนภาพดังต่อไปนี้


tower('A', 'C', 'B', 3)
fromTower is 'A'
toTower is 'C'
auxTower is 'B'
n is 3
tower ('A', 'B', 'C', 2)
Move 3 from A to C
tower('B', 'C', ' A', 2)
tower('A', 'B', 'C', 2)
fromTower is 'A'
toTower is 'B'
auxTower is 'C'
n is 2
tower ('A', 'C', 'B', 1)
Move 2 from A to B
tower('C', 'B', ' A', 1)
Return
tower('A', 'C', 'B', 1)
fromTower is 'A'
toTower is 'C'
auxTower is 'B'
n is 1
Move 1 from A to C
Return
tower('C', 'B', 'A', 1)
fromTower is 'C'
toTower is 'B'
auxTower is 'A'
n is 1
Move 1 from C to B
Return
tower('B', 'C', 'A', 2)
fromTower is 'B'
toTower is 'C'
auxTower is 'A'
n is 2
tower ('B', 'A', 'C', 1)
Move 2 from B to C
tower('A', 'C', ' B', 1)
Return
tower('B', 'A', 'C', 1)
fromTower is 'B'
toTower is 'A'
auxTower is 'C'
n is 1
Move 1 from B to A
Return
tower('A', 'C', 'B', 1)
fromTower is 'A'
toTower is 'C'
auxTower is 'B'
n is 1
Move 1 from A to C
Return

ตัวอย่างที่ 7.6 โปรแกรม reverse

โปรแกรมนี้เป็นตัวอย่างการสร้างฟังก์ชันชื่อ reverse ในการรับข้อความทางแป้นพิมพ์เป็นลำดับ แต่จะมีการแสดงผลโดยย้อนอักขระตัวสุดท้ายย้อนกลับมาจนถึงอักขระตัวแรก โดยเขียนในลักษณะของการเรียกตัวเอง

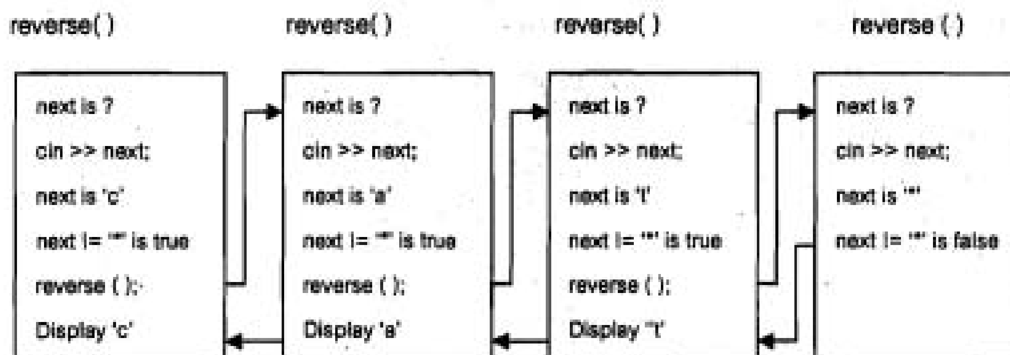
```
// File: reverseTest
// Tests a function which displays keyboard input in reverse

#include<iostream>
using namespace std;
// Function prototype
void reverse( );
int main ( )
{
        reverse( );        // Reverse the keyboard input
        cout << endl;
        return 0;
}
// Displays keyboard input in reverse
void reverse( )
{
        char next;
        cout << "Next character or * to stop: ";
        cin >> next;
        if (next != '*' )
        {
                reverse ( );        // recursive step
```

```
            cout << next;   // Display next after return
      }
}
```

เมื่อนำโปรแกรมไป run จะมีการแสดงหน้าจอเพื่อให้ผู้ใช้ป้อนข้อมูล ดังนี้

```
Next character or * to stop:  c
Next character or * to stop:  a
Next character or * to stop:  t
Next character or * to stop:  *
tac
```

โดยผู้ใช้จะป้อนข้อมูลครั้งละ 1 ตัวอักษรไปเรื่อยๆ ต้องการหยุดป้อนโดยกดเป็นเครื่องหมาย * การทำงานจะพิมพ์ข้อความย้อนกลับจากตัวสุดท้ายจนถึงตัวแรก



ตัวอย่างที่ 7.7 โปรแกรมหาค่าหารร่วมมาก

โปรแกรมต่อไปนี้เป็นอัลกอริทึม Euclid สำหรับการหาค่าตัวหารร่วมมาก

```
// FILE: gcdTest.cpp
// Program and recursive function to find greatest common divisor

#include<iostream>
using namespace std;
```

```
// Function prototype
int gcd(int, int);
int main( )
{
        int m, n;                   // the two input items
        cout << "Enter two positive integer: " ;
        cin >> m >> n;
        cout << endl << "Their greatest common divisor is "
             << gcd(m, n) << endl;
        return 0;
}
// Finds the greatest common divisor of two integers
// Pre:         m and n are defined and both are > 0.
// Post:        None
// Returns:     The greatest common divisor of m and n.
int gcd(int m, int n)
{
        if (m < n)
                return gcd(n, m);           // transpose arguments
        else if(m % n == 0)
                return n;                   // n is GCD
        else
                return gcd(n, m % n);  // recursive step
}
```

เมื่อนำโปรแกรมไปทำงานจะได้ผลลัพธ์ดังนี้

```
Enter two positive integer: 24 84
Their greatest common divisor is 12
```

ตัวอย่างที่ 7.8 ฟังก์ชันหาผลรวมของข้อมูล

ฟังก์ชันต่อไปนี้เป็นการหาผลรวมของเลข 1+2+3+..+10 โดยสมมุติว่าข้อมูลมีการเก็บในตัวแปรอาร์เรย์ 1 มิติชื่อ x ดังนี้

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

การหาค่าผลรวมเราจะสร้างฟังก์ชันชื่อ findSum โดยส่งผ่านค่าของกลุ่มข้อมูล x และขนาดของข้อมูล ไปยังฟังก์ชันเพื่อส่งค่าผลรวมของข้อมูลทั้งหมดกลับมายังจุดเรียกใช้ ซึ่งการทำงานจะเป็นไปในลักษณะเรียกตัวเอง

การทำงานจะมีการเรียกใช้จากคำสั่งต่อไปนี้

```
// Calculate sum two ways
sum1 = findSum(x, SIZE);
sum2 = (SIZE * (SIZE + 1)) / 2;
cout << "Recusive sum is " << sum1 << endl;
cout << " Calculate sum is " << sum2 << endl;
```

โดย ตัวแปร sum1 จะเป็นผลรวมของข้อมูลทั้งหมดที่เก็บในตัวแปรอาร์เรย์ 1 มิติชื่อ x

ตัวแปร sum2 จะเป็นผลรวมของข้อมูลเช่นเดียวกันแต่ใช้สูตรสำเร็จในการหาผลรวม

```
// Finds the sum of integers in an n-element array
int findSum(int x[ ], int n)
{
    if (n == 1)
        return x[0];                    // stopping case
```

```
    else
        return x[n-1] + findSum(x, n-1);        // recursive step
}
```

ผลของการเรียกใช้จะเป็นดังนี้

Recursive sum is 55

Calculated sum is 55

การทำงานของฟังก์ชันจะเห็นได้ว่ามีการเรียกตัวเองโดยมีการส่งผ่านกลุ่มของอาเรย์ในการทำงานโดยค่าของ n จะมีการลดค่าครั้งละ 1 จนกระทั่งมีค่าเท่ากับ 1 จึงหยุดการกระทำซ้ำ

สรุป

1. Recursive หรือ Recursion เป็นการเขียนคำสั่งโปรแกรมที่มีการปฏิบัติงานเรียกตัวเอง โดยมีการเรียกการทำงาน ในลำดับก่อนมาทำงานในลำดับปัจจุบัน โดยจะเรียกลำดับก่อนวนซ้ำไปเรื่อยๆ จนถึงข้อมูลที่กำหนดการสิ้นสุดในการเรียกตัวเอง
2. การเขียนโปรแกรมเรียกตัวเองจะสร้างในรูปแบบของฟังก์ชันโดยการปฏิบัติงานในฟังก์ชันต้องมี ค่าเริ่มต้นการทำงานวนรอบ โดยทั่วไปเป็น formal parameter ที่มีการส่งผ่านค่ามาจากจุดเรียกใช้ ค่าสิ้นสุดการทำงานเป็นค่าของข้อมูลสุดท้าย ที่ผ่านการเรียกตัวเองโดยลดค่าลงอย่างเป็นลำดับ ในการเรียกฟังก์ชันแต่ละครั้ง และคำสั่งในการปฏิบัติงาน เรียกตัวเองเป็นการประมวลผลที่นำผลลัพธ์จากการทำงานในรอบที่แล้วมาปฏิบัติงานในรอบปัจจุบัน เพื่อส่งผลลัพธ์จากการทำงาน ไปยังจุดเรียกใช้
3. ภาษาโปรแกรมที่สามารถเขียนโปรแกรมเรียกตัวเองได้ เช่น Pascal,C, C++, C#
4. โดยทั่วไปการเรียกตัวเองเปรียบเสมือนการทำงานที่กระทำซ้ำๆกันในรูปของคำสั่ง while, for แต่เราสามารถนำมาเขียนในรูปแบบที่เรียกตัวเองโดยแทนด้วยคำสั่งเงื่อนไข การกระทำซ้ำจะแทนด้วยการเรียกฟังก์ชันในลำดับก่อนทำงาน

แบบฝึกหัด

1. การเขียนโปรแกรมแบบเรียกตัวเองแตกต่างจากการเขียนโปรแกรมแบบธรรมดาอย่างไร จงอธิบายพอเข้าใจ
2. พิจารณาฟังก์ชันต่อไปนี้ จงแปลงให้เป็นฟังก์ชันที่มีการเรียกตัวเอง

2.1
```
int  test1(int x )
{
        int sum=0 ;
        while (x>=5)
        {
                sum = sum + x;
                x – ;
        }
        return sum ;
}
```

2.2
```
double test2(int  y)
{
        double  sum = 1.0 ;
        for (int i = 0 ; i < 10 ; i ++ )
                sum = sum * y  *  y ;
        return sum ;
}
```

2.3
```
int  test3( int x)
{
        int sum = 0 ;
        int  y = 50 ;
        do
```

```
    {
        sum = sum + y ;
        y = y -5 ;
    } while (y >10) ;
    return sum ;
}
```

3. จงเขียนโปรแกรมเรียกตัวเองในการคำนวณหา combination ของค่า n และ r ดังนี้

   C(n,r) = r! / (r!(n-r)!)

4. จงเขียนโปรแกรมเรียกตัวเองในการทำงานจากฟังก์ชันต่อไปนี้

   F(X,Y) = X –Y ถ้าค่า X หรือค่า Y มีค่าน้อยกว่า 0

   F(X,Y) = F(X-1,Y)+F(X,Y-1) ถ้ากรณีอื่นๆ

5. จงเขียนฟังก์ชันเรียกตัวเองเพื่อทำงานดังนี้ 1 + 1/1! + 1 / 2! + ... + 1/n!

6. พิจารณาสมการต่อไปนี้

   Y(X) = 1 + (3/2) + (5/3) + (7/4) + ... + (2X-1) / X โดย X>=1

   จงเขียนฟังกชันนี้แบบเรียกตัวเอง โดยใช้คำสั่ง if แทนคำสั่ง while

7. พิจารณาฟังก์ชันต่อไปนี้

```
void Hanoi(int N , char A , char B , char C)
{
    if (N>2)
    {
        Hanoi(N-2 , A , C , B);
        Writeln("Move No." , C , "from " , ,"to " , B );
        Hanoi(N-2 , B , A , C);
    }
    else
        cout << "Move No." << N << "from " << A << "to " << C );
}
```

ถ้ามีการเรียกใช้ Hanoi(4 , '1' , '2' , '3'); ผลการทำงานเป็นอย่างไร

8. พิจารณาสมการต่อไปนี้ Y(X) = 1/4 + 2/7 + 3/10 + 4/13 + ... + X / (3X+1)

   จงเขียนฟังก์ชั่นเรียกตัวเองเพื่อทำงานตามสมการข้างต้น

9. จงเขียนโปรแกรมเรียกตัวเองจากฟังก์ชันต่อไปนี้

```
int  Test(int n)
{
   int sum ;
   for(int  x=1 ; x<6 ; x++)
        sum = sum + x * x ;
   return sum ;
 }
```

10. พิจารณาฟังก์ชันต่อไปนี้

F(M,N) = N+1 ถ้า M=0 และ N <>0
= M+2 ถ้า N=0 และ M <> 0
= M+N ถ้า M =0 และ N = 0
= F(M-1,N) + F(M,N-1) ถ้ากรณีอื่นๆ

10.1 จงเขียนฟังก์ชั่นเพื่อสามารถทำงานได้ตามฟังก์ชั่นข้างต้น

10.2 ถ้ามีการเรียกใช้ F(2,1) ผลการทำงานมีค่าเป็นเท่าใด

11. กำหนดฟังก์ชั่น CV(N) เมื่อ N >= 1

CV(1) = 2
CV(2) = 5
CV(3) = CV(1) * 2 + 1 = 5
CV(4) = CV(2) * 3 - 5 = 10
CV(5) = CV(3) * 2 + 1 = 11
CV(6) = CV(4) * 3 – 5 = 25

จงเขียนโปรแกรมC++ โดยใช้ RECURSIVE SUBPROGRAM เพื่อทำการคำนวณหาค่า CV(N) เมื่อ N คือ INPUT DATA