บทที่ 2

การแบ่งแยก และ การเอาชนะ(divide and conqure)

เทคนิคการออกแบบวิธีหนึ่งที่ใช้กันแพร่หลาย ได้แก่ ขั้นตอนวิธีการแบ่งแยกและการเอาชนะ ซึ่งขั้นตอนวิธีการแบ่งแยกและการเอาชนะ จะแบ่งปัญหาออกเป็นปัญหาย่อย ๆที่ไม่ซ้อนทับกัน โดยจะแบ่งเป็น 2 ส่วนได้แก่

1. ส่วนของการแบ่งแยก (divide) คือ การแบ่งแยกปัญหาออกเป็นปัญหาย่อยที่มีลักษณะเช่นเดิม ส่วนนี้จะใช้การเรียกซ้ำ (recursive) ในการแก้ปัญหา

2. ส่วนของการเอาชนะ (conquer) คือ การหาผลลัพธ์ของปัญหาเดิมซึ่งได้จากผลลัพธ์ของปัญหาย่อย

2.1 นิยามการเรียกซ้ำ

การหาคำตอบสำหรับปัญหาการเรียกซ้ำ จะใช้วิธีแบ่งปัญหาเดิมให้เป็นปัญหาที่เล็กลง, เพื่อที่จะหาคำตอบได้ง่ายขึ้น โดยปัญหาที่แบ่งออกมาจะมีลักษณะคล้ายๆ กัน จากนั้นค่อยๆ แก้ปัญหาเล็กๆ ทีละขั้นๆ ก็จะได้คำตอบของปัญหาเดิม

ขั้นตอนการเรียกซ้ำ คือ กระบวนการที่มีการเรียกเมธตอดของตัวมันเอง เมื่อเราแบ่งเป็นปัญหาที่เล็กลงเหมือนๆกัน ก็คือทำขั้นตอนเดิมซ้ำๆ เป็นการวนเรียกเมธตอดตัวมันเองเพื่อแก้ปัญหาเดิม

<u>ตัวอย่าง</u> 2.1 การหาค่าของ แฟคทอเรียล

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ nx\ (n-1)\,x \ldots .\ x1 & \text{if } n \geq 1 \end{cases}$$

เราสามารถเขียนใหม่เป็น

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

จะเห็นได้ว่าแฟคทอเรียล n จะมีการใช้วิธีการแฟคทอเรียลในตัวมันเอง คือเรียก (n-1)! ซึ่งจะเขียนเป็นเมธตอดได้ดังนี้

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

ถ้า พารามิเตอร์ n เป็น 0 ก็จะคืนค่าเป็น 1 แต่ถ้าไม่เท่ากับ 0 ก็จะคืนค่าผลคูณของ n กับ factorial (n-1)

เพื่อให้เข้าใจการทำงานของการเรียกซ้ำ ว่าเมื่อเมธตอดเรียกตัวมันเองจะเกิดอะไรขึ้น โดยให้เมธตอด main( ) เรียกใช้ factorial( ) ตามตัวอย่างต่อไปนี้

```
public static void main (String[] args) throws IOException {
    BufferedReader stdin = new BufferedReader(
                            New InputStreamReader(System.in));
    System.out.print("Enter n:");
    int n = Integer.parseInt(stdin.readLine());
    System.out.println( n+"!  = "+ factorial(n));
}
```

โปรแกรมจะแสดงข้อความพร้อมรับค่าจำนวนเต็มบวก สำหรับค่า n ในกรณีให้ n = 3 จากนั้นโปรแกรมก็จะพิมพ์ 3! = แล้วหาค่า 3! โดยเรียกเมธตอด factorial( ) แล้วส่งค่าพารามิเตอร์เป็น 3 โดยมีลำดับการหาค่า 3! (รูป 9.1) ดังนี้

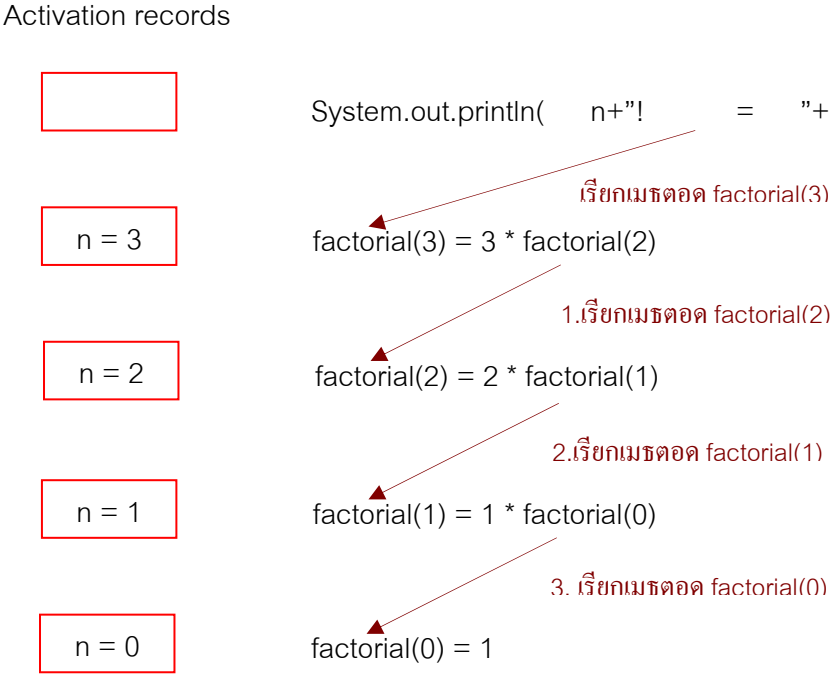
1. เมธตอด factorial(3) ตรวจค่า n ไม่เท่ากับ 0 จึงเรียกเมธตอด factorial(2)
2. เมธตอด factorial(2) ตรวจค่า n ไม่เท่ากับ 0 จึงเรียกเมธตอด factorial(1)
3. เมธตอด factorial(1) ตรวจค่า n ไม่เท่ากับ 0 จึงเรียกเมธตอด factorial(0)
4. เมธตอด factorial(0) ตรวจค่า n =0 ได้ค่าเป็น 1 จากนั้นคืนค่า กลับไปที่ factorial(1)
5. เมธตอด factorial(1) เมื่อรับค่า factorial(0) มาก็จะคำนวณ factorial(1) = 1*factorial(0) = 1*1 = 1 แล้วคืนค่าให้กับ factorial(2)
6. เมธตอด factorial(2) เมื่อรับค่า factorial(1) มาก็จะคำนวณ factorial(2) = 2*factorial(1) = 2*1 = 2 แล้วคืนค่าให้กับ factorial(3)
7. เมธตอด factorial(3) เมื่อรับค่า factorial(2) มาก็จะคำนวณ factorial(3) = 3*factorial(2) = 3*2 = 6 แล้วคืนค่าให้เมธตอด main( ) เพื่อแสดงผล

```
Enter a positive integer : 3
3! =  6
```

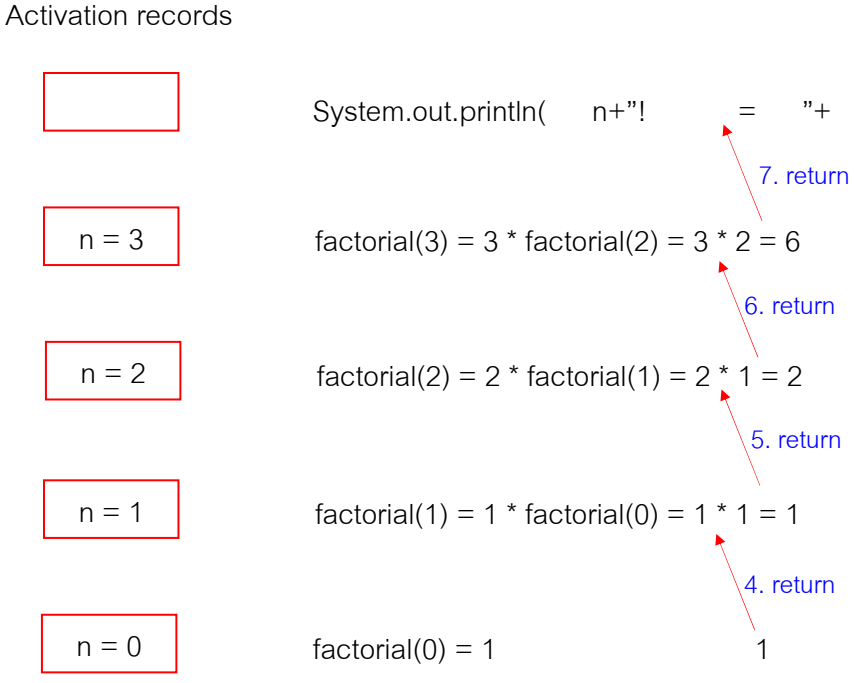
เมธตอดการเรียกซ้ำ จะประกอบด้วย 2 ส่วน คือ

- กรณีมูลฐาน(Base case) คือ ส่วนที่มีคำตอบสำหรับปัญหาพื้นฐาน และเป็นการหยุดการเรียกเมธตอดตัวเอง
- กรณีเรียกซ้ำ(Recursive case) คือ ส่วนที่เรียกเมธตอดเดียวกัน

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



รูป 2.1 ก. ประโยคการเรียกซ้ำเมธตอด factorial



รูป 2.1 ข. ประโยคการคืนค่ากลับจากเมธตอด factorial

<u>ตัวอย่าง 2.2</u> เมธตอดเรียกซ้ำ เพื่อ คำนวณ x^n โดยที่ x,n เป็นเลขจำนวนเต็มบวก

เรารู้ว่า x^0 เป็น 1 สำหรับทุกค่าของ x เราจะใช้ความจริงนี้เป็นกรณีมูลฐาน และเราก็รู้ว่า $x^n = x*x^{n-1}$ เมื่อ n>0 ซึ่งเราจะใช้ส่วนนี้เป็นกรณีเรียกซ้ำ สังเกตุว่าค่า n จะลดลงเรื่อยๆ ทำให้มั่นใจได้ว่าการเรียกซ้ำนี้สามารถจบได้แน่

เขียนเมธตอดหาค่า x^n ใน แบบของการเรียกซ้ำ ได้ดังนี้

```
public static int power(int x, int n) {
    if (n==0)
        {return 1; }
    else
        { return x * power(x,n-1);}
}
```

2.2 Fibonacci numbers

เราเคยเขียนโปรแกรมเพื่อหาค่า fibonacci number โดยใช้วิธีการวนซ้ำ มาในบทนี้เราจะใช้เมธตอดการเรียกซ้ำ ถ้าลำดับของ fibonacci number:1,1,2,3,5,8, หลังกำหนดค่า 2 พจน์แรกเป็น 1, 1 แล้ว พจน์ถัดมาจะเท่ากับผลบวกของ 2 พจน์ก่อนหน้ามัน ซึ่งจะเขียนเป็นนิยามทางคณิตศาสตร์ได้ดังนี้

$$f_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ f_{n-1} + f_{n-2} & \text{if } n > 2 \end{cases}$$

<u>พิจารณา</u>

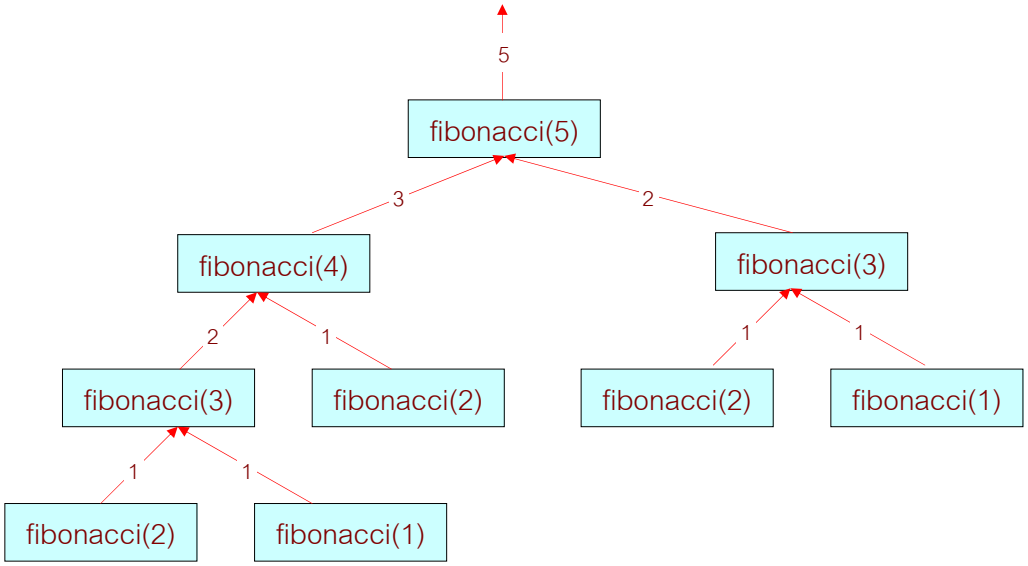
กรณีมูลฐาน ➔ กรณี n =1 หรือ n=2 จะได้คำตอบเป็น 1, 1

กรณีเรียกซ้ำ ➔ กรณี n>2 จะได้คำตอบเป็น $f_{n-1} + f_{n-2}$

เราจะได้เมธตอด fibonacci เป็น

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

จะเห็นความแตกต่างจากตัวอย่างก่อนๆ ที่ในคราวนี้เราเรียกซ้ำ 2 อัน ซึ่งจะทำการคืนค่าเป็นคู่ๆ กัน เราจะหาค่าที่จะคืนได้ เพราะค่าของพารามิเตอร์ตัวที่คืนจะน้อยกว่าตัวที่ทำการเรียก ตามรูป 2.2



รูป 2.2 การคำนวณ fibonacci พจน์ที่ 5

จากรูปที่ 2.2 จะเห็นว่าเมท็อดที่ได้ไม่มีประสิทธิภาพ เนื่องจากมีการเรียกเมท็อด fibonacci ด้วยค่าอากิวเมนต์เดียวกันซ้ำหลายครั้ง ตัวอย่างเช่น ในการคำนวณ fibonacci(5) จะได้จากผลบวกของการเรียกเมท็อด fibonacci (4) และ fibonacci (3) การคำนวณ fibonacci (4) จะได้จากผลบวกของการเรียกเมท็อด fibonacci (3) (เป็นการเรียกครั้งที่ 2) และ fibonacci (2) การคำนวณ fibonacci (3) จะได้จากผลบวกของการเรียกเมท็อด fibonacci (2) และ fibonacci (1) ซึ่งจะเห็นว่ามีการเรียกเมท็อด fibonacci (3) 2 ครั้ง เมท็อด fibonacci (2) 3 ครั้ง และเมท็อด fibonacci (1) 2 ครั้ง

เนื่องจากมีการเรียกเมท็อดซ้ำซ้อน เวลาที่ใช้ในการคำนวณ fibonacci (n) จึงเป็นแบบเอ็กซ์โปแนนเชียล* $O(2^n)$

$$T(n) = \begin{cases} O(1) & n < 2 \\ T(n-1) + T(n-2) + O(1) & n \geq 2 \end{cases}$$

หรือ $$T(n) = \begin{cases} 1 & n < 2 \\ T(n-1) + T(n-2) + 1 & n \geq 2 \end{cases}$$

* http://www.ics.uci.edu/~eppstein/161/960109.html

จะได้คร่าวๆว่า

$$T(n) > 2xT(n-2)$$
$$> 2x2xT(n-4)$$
$$> 2x2x2xT(n-6)$$
$$\vdots$$
$$> 2x2x2x2x2.....x2xT(0)$$
$$n/2 \ terms$$
$$2^{\frac{n}{2}} = 1.414^n$$

ใช้ วีธี generating functions ในการหาคำตอบ Fibonacci recurrence

นิยาม function

$$F(z) = \sum_{i=0}^{\infty} F_i z^i$$
$$=0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 3z^4 + 3z^4 + 3z^4$$

ถ้าเราเขียนโปรแกรมfibonacci2.java เป็นแบบ iteration loop จะมีประสิทธิภาพดีกว่า จะได้ complexity เป็น O(n)

```
public static int fibonacci2 (int n)
 {   int fib[] = new int[50];
     fib[1]=1;
     fib[2]=1;
      for (i=3; i<=n;++i )
          { fib[i]=fib[i-1]+fib[i-2];
          }
     Return fib[n];
} // fibonacci2
```

ตัวอย่าง 2.3 การเรียกซ้ำเพื่อสืบค้นแบบไบนารี(Binary Search)

เราได้ศึกษาขั้นตอนของการสืบค้นแบบไบนารี ในการหาตัวเลข key ว่าอยู่ตำแหน่งใดของเซ็ท มาตอนนี้เราจะใช้การเรียกซ้ำในการสืบค้นแบบไบนารี ในการหาชื่อ เบอร์โทรศัพท์ จากสมุดจด

<u>วิเคราะห์และออกแบบ</u>

เราจะออกแบบคลาส RecursiveBinarySearch ให้มีเมธตอดเพื่อทำหน้าที่ค้นหาชื่อในวัตถุ AddressEntry ซึ่งจะประกอบด้วย ชื่อและเบอร์โทรศัพท์ และการกำหนดคลาส ดังนี้

```
public class AddressEntry{
    private String personName;
    private String telephoneNumber;
    // …… constructor
    public AddressEntry(String name, String number) {
        personName = name;
        telephoneNumber = number;
    }

    public String getName() {
        return personName;
    }
    public String getNumber() {
        return telephoneNumber;
    }

public void setName(String Name) {
        personName = Name;
    }
    public void setTelephoneNumber(String number) {
        telephoneNumber = number;
    }
}
```

<u>ขั้นตอนแรก</u> ในการพัฒนาเมธตอด binary search ก็คือ พิจารณาว่าอาร์กิวเมนต์อะไรที่จะรับค่าเข้ามาในเมธตอด และ เมธตอดจะคืนค่าอะไรกลับไป จะได้ว่าพารามิเตอร์ของเมธตอด ควรจะเป็น address book ที่ต้องการค้น และ ชื่อที่ต้องการค้น จากนั้นเมธตอดควรจะคืนค่า AddressEntry ที่ค้นได้ โดยกำหนดเมธตอด recSearch ดังนี้

```
public static AddressEntry recSearch(AddressEntry[ ]
    addressBook, String name)
```

<u>ต่อมา</u> พิจารณาถึงขั้นตอนการเรียกซ้ำในการสืบค้นที่จะต้องใช้อีก 2 พารามิเตอร์คือ first, last ที่จะชี้ย่อย (sublist) ลงไปว่าให้สืบค้นจากรายการที่เท่าไรไปถึงรายการใด การเรียกซ้ำให้ได้ผลจะต้องทำให้รายการที่ต้องสืบค้นน้อยลงเรื่อยๆ

```
public static AddressEntry recSearch(AddressEntry[ ]
addressBook, String name, int first, int last)
```

คราวนี้มาพิจารณา

<u>กรณีมูลฐาน</u> : มี 2 กรณีคือ กรณีที่ค้นเจอแล้ว กับกรณีไม่มีข้อมูลในรายการ (length เป็น 0) ซึ่งจะทำให้หยุดกระบวนการเรียกซ้ำ

<u>กรณีเรียกซ้ำ</u> : มี 2 กรณีคือ กรณีสมาชิกตัวที่ต้องการอยู่ในครึ่งซ้ายของรายการย่อย(sublist) กับกรณีสมาชิกตัวที่ต้องการอยู่ในครึ่งขวา

ซึ่งจะมีขั้นตอนวิธี ดังนี้

ขั้น 1: ถ้า รายการ ไม่มีสมาชิกอยู่เลย

return null

ขั้น 2: คำนวณตำแหน่งกลาง (midpoint) ของ รายการ

ขั้น 3: ถ้า keyName = name ในตำแหน่งกลาง

return สมาชิกในตำแหน่ง midpoint

ขั้น 4: ถ้า keyName < name ในตำแหน่งกลาง

เรียก recursive search โดยส่ง sublist จนถึง midpoint –1

ขั้น 5: ถ้า keyName > name ในตำแหน่งกลาง

เรียก recursive search โดยส่ง sublist จากตำแหน่ง midpoint +1

จากขั้นตอนวิธีข้างต้น สามารถเขียนโปรแกรมในส่วนเมธตอด recSearch( ) ได้ตามโปรแกรม 2.1 RecursiveBinarySearch.java และเมธตอด main( ) จะทำการทดสอบชื่อแรก, ชื่อตรงกลาง, ชื่อคนสุดท้าย และในกรณีไม่ชื่อในรายการ

Recursive Complexity จะเท่ากับ

$$T(n) = T(\frac{n}{2}) \quad + \quad 1$$

T(n/2) จำนวนการเปรียบเทียบใน recursive

1 จำนวนการเปรียบเทียบก่อนการเรียก recursive

$$recursive\,case: \quad T(n) = T(\frac{n}{2}) \quad + \quad 1 \quad for\, n > 1,\, n\ a\ power\ of\ 2$$

$$base\ case: \quad T(1) = 1$$

$$O(\log n)$$

<u>โปรแกรม 2.1</u> RecursiveBinarySearch.java

```
1.  public class RecursiveBinarySearch {
2.
3.          public static AddressEntry recSearch(AddressEntry[] addressBook,
4.                  String name, int first, int last) {
5.              // base case: if the array section is empty, not found
6.              if (first > last)
```

```
7.                      return null;
8.                  else {
9.                      int mid = (first + last) / 2;
10.                     // if we found the value, we're done
11.                     if (name.equalsIgnoreCase(addressBook[mid].getName()))
12.                         return addressBook[mid];
13.                     else if (name.compareToIgnoreCase(
14.                     addressBook[mid].getName()) < 0) {
15.                         // if value is there at all, it's in the left half
16.                         return recSearch(addressBook, name, first, mid-1);
17.                     }
18.                     else { // array[mid] < value
19.                         // if value is there at all, it's in the right half
20.                         return recSearch(addressBook, name, mid+1, last);
21.                     }
22.                 }
23.             }
24.
25.             public static void main(String[] args) {
26.                 // list must be in sorted order
27.                 AddressEntry addressBook[] = {
28.                     new AddressEntry("Audrey", "434-555-1215"),
29.                     new AddressEntry("Emily" , "434-555-1216"),
30.                     new AddressEntry("Jack"  , "434-555-1217"),
31.                     new AddressEntry("Jim"   , "434-555-2566"),
32.                     new AddressEntry("John"  , "434-555-2222"),
33.                     new AddressEntry("Lisa"  , "434-555-3415"),
34.                     new AddressEntry("Tom"   , "630-555-2121"),
35.                     new AddressEntry("Zach"  , "434-555-1218")
36.                 };
37.                 AddressEntry p;
38.                 // first element
39.                 p = recSearch(addressBook, "Audrey", 0, addressBook.length-1);
40.                 if (p != null) {
41.                     System.out.println("Audrey's telephone number is " +
42.                     p.getNumber());
43.                 }
44.                 else {
45.                     System.out.println("No entry for Audrey");
46.                 }
47.                 // middle element
48.                 p = recSearch(addressBook, "Jim", 0, addressBook.length-1);
49.                 if (p != null) {
50.                     System.out.println("Jim's telephone number is " +
51.                     p.getNumber());
52.                 }
53.                 else {
54.                     System.out.println("No entry for Jim");
55.                 }
```

```
56.             // last element
57.             p = recSearch(addressBook, "Zach", 0, addressBook.length-1);
58.             if (p != null) {
59.                 System.out.println("Zach's telephone number is " +
60.                 p.getNumber());
61.             }
62.             else {
63.                 System.out.println("No entry for Zach");
64.             }
65.             // non existent entry
66.             p = recSearch(addressBook, "Frank", 0, addressBook.length-1);
67.             if (p != null) {
68.                 System.out.println("Frank's telephone number is " +
69.                 p.getNumber());
70.             }
71.             else {
72.                 System.out.println("No entry for Frank");
73.             }
74.         }
75.
76. }
```

จะได้ผลลัพธ์ ดังนี้

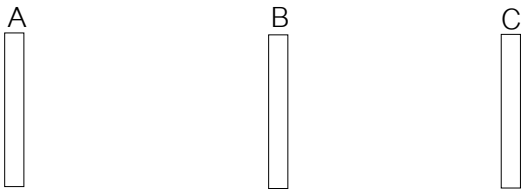
```
Audrey's telephone number is 434-555-1215
Jim's telephone number is 434-555-2566
Zach's telephone number is 434-555-1218
No entry for Frank
```

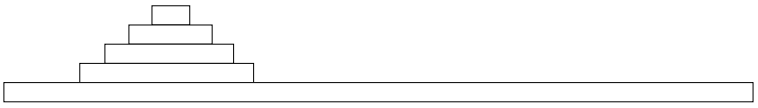
ตัวอย่าง2.4 หอคอยฮานอย (Towers of Hanoi)

ตัวอย่างที่ผ่านมาอาจใช้วิธีการวนซ้ำ (iteration) หรือ ใช้การเรียกซ้ำ(recursive) ก็ได้ แต่ปัญหาหอคอยฮานอยจะง่ายถ้าใช้วิธีการเรียกซ้ำ และจะยากมากถ้าไม่ใช้วิธีนี้

ปัญหาหอคอยฮานอย จะแสดงได้ตามรูป 2.3 ที่จะต้องย้ายจานจากเสา A ทางซ้าย ไปที่เสา C ทางขวา โดยมีกฎดังนี้

1. เมื่อจานถูกย้าย จะต้องวางไปบนเสาใดเสาหนึ่ง
2. การย้ายแต่ละครั้ง จะหยิบจานได้ทีละใบเท่านั้น และจะต้องวางจานไว้บนสุดของแต่ละเสา
3. ห้ามวางจานใบใหญ่กว่า บนจานใบที่เล็กกว่า





รูป 2.3 หอคอยฮานอย

พิจารณา ถ้าจำนวนจานทั้งหมด n ใบ

กรณีมูลฐาน : มี 2 กรณีคือ

กรณีที่มีจานใบเดียว ย้ายจากเสา A ไปเสา C หรือ

กรณีที่ย้ายใบเล็กสุดไปวางบนเสา C

ซึ่งจะทำให้หยุดกระบวนการเรียกซ้ำ

กรณีเรียกซ้ำ : คือ กรณี n > 1 หรือกรณีที่ยังย้ายไม่เสร็จ

ซึ่งจะมีขั้นตอนวิธี ดังนี้

ขั้น 1: ย้ายจานใบบนๆ n-1 ใบ จากเสา A ไปที่เสา B โดยใช้ เสา C ช่วย

ขั้น 2: ย้ายจานใบใหญ่สุดที่เหลือ(ใบที่ n) จากเสา A ไปเสา C

ขั้น 3: ย้ายจาน n-1 ใบ จากเสา B ไปที่เสา C โดยใช้ เสา A ช่วย

โปรแกรม 2.2 HanoiTower.java

```
1.  import java.io.*;
2.  public class HanoiTower {
3.      final static char PEG1 = 'A';
4.      final static char PEG2 = 'B';
5.      final static char PEG3 = 'C';
6.      public static void moveDisk
7.              (int n,char startPeg, char auxPeg, char endPeg) {
8.          if (n==1) {
9.              System.out.println("move disk from "+ startPeg + " to "+endPeg);
10.         }
11.         else{
12.             moveDisk (n-1, startPeg, endPeg,auxPeg);
13.             moveDisk(1, startPeg, ' ', endPeg);
14.             moveDisk(n-1, auxPeg, startPeg, endPeg);
15.         }
16.     }
17.     public static void main(String[] args)throws IOException {
18.         BufferedReader stdin = new BufferedReader(
19.                 new InputStreamReader(System.in));
20.         System.out.print ("enter number of disk:");
21.         int number = Integer.parseInt(stdin.readLine());
22.         moveDisk(number, PEG1, PEG2, PEG3);
```

```
23.       }
24. }
```

ผลลัพธ์

```
enter number of disk : 3
move disk from  A to C
move disk from  A to B
move disk from  C to B
move disk from  A to C
move disk from  B to A
move disk from  B to C
move disk from  A to C
```

เราสามารถคำนวณหาจำนวนครั้งของการเคลื่อนย้ายจาน T(n) ที่จำเป็นในการแก้ปัญหานี้ จากความสัมพันธ์เรียกซ้ำ (recurrence relation) โดยสังเกตุจาก

กรณี n = 0 T(0) = 0

กรณี n = 1 T(1) = 1

กรณี n = 2 T(2) = 3

กรณี n = 3 T(3) = 7

คำตอบของการเรียกซ้ำ สัมพันธ์กับการย้ายจานจำนวน n-1 จานจากเสา A ไปมาระหว่างเสา A-B-C จากนั้นย้ายจานจำนวน n-1 จาน ไปยังเสา C แล้วค่อย ย้ายจานเล็กสุดวางบน รวมจำนวนครั้งเป็น[†]

$$\boxed{T(n) \le T(n-1)+1+T(n-1) = 2T(n-1)+1}$$

$$\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1 \\
&= 2(2(2T(n-3)+1)+1)+1 = 2^3T(n-2)+4+2+1 \\
&= \vdots \\
&= 2^{n-1}T(1)+2^{n-2}+2^{n-3}+\cdots+2+1 \\
&= 2^n-1 \qquad \text{เนื่องจาก } \sum_{i=0}^{n-1}2^i = 2^n-1 \\
&= O(2^n)
\end{aligned}$$

ควรหลีกเลี่ยงการใช้เทคนิค divide and conquer ในกรณี Fibonacci เพราะมีการแบ่งพจน์ที่ n เป็น 2 พจน์ คือ พจน์ที่ n-1 กับพจน์ที่ n-2 ทำให้จำนวนครั้งที่ต้องคำนวณพจน์ต่างๆ เป็น exponential in n ในขณะที่ วิธี การวนซ้ำ (iterative) มีจำนวนครั้งในการคำนวณเท่ากับ (n) แต่สำหรับปัญหา Tower of Hanoi จำเป็นต้องใช้ เทคนิค divide and conquer จะให้ประสิทธิภาพเหนือกว่าเทคนิคอื่น ถึงแม้ว่า complexity จะเป็น exponential ก็ตาม

[†] http://www.cut-the-knot.org/Curriculum/Combinatorics/TowerOfHanoi.shtml

แบบฝึกหัดบทที่ 2

1. จงหาผลลัพธ์ ที่ได้จากโปรแกรมต่อไปนี้

```
class SelfCheck1 {
    public static int  f (int n) {
        if (n<=1)
            return n;
        else
            return f(n-1) + f(n-2);
        }
    public static void main(String[ ] args) {
        System.out.println(f(4));
    }
}
```

2. จงเขียนโปรแกรม โดยใช้เมธตอดเรียกซ้ำ เพื่อหาค่า e ตามสูตร

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \text{.....}$$

3. จงเขียนเมธตอดเรียกซ้ำ เพื่อคำนวณค่า ห.ร.ม.(เลขที่มากที่สุดที่สามารถหารได้ลงตัว)ของเลขจำนวนเต็ม 2 จำนวน

$$\text{gcd}(m,n) = \begin{cases} n & \text{ถ้า n หาร m ลงตัว} \\ \text{gcd}(n, \text{เศษของ m หารด้วย n}) & \text{ถ้า n หาร m ไม่ลงตัว} \end{cases}$$

4. palindrome คือ ข้อความที่อ่านจากซ้ายไปขวา เหมือนกับที่อ่านจากขวาไปซ้าย โดยที่ไม่สนใจช่องว่าง, ตัวพิมพ์เล็กพิมพ์ใหญ่, เครื่องหมายวรรคตอน เช่น

   palindrome “level”, “Was it a rat I saw”, “A man, a plan, a canal : Panama”

   จงเขียนเมธตอด isPalindrome( ) เพื่อ return true ถ้าอาร์กิวเมนต์เป็น palindrome มิฉะนั้น return false โดยใช้ตัวอย่างในการทดสอบ

```
public static boolean isPalindrome (String phrase)
```

5. จงเขียนเมธตอดเรียกซ้ำ sum(int m, int n) เพื่อคำนวณค่าผลรวมของตัวเลขจำนวนเต็ม ตั้งแต่ m ถึง n ➔ m+(m+1)+(m+2)+…+(n-2)+(n-1)+n

http://www.cut-the-knot.org/Curriculum/Combinatorics/TowerOfHanoi.shtml

http://www.ics.uci.edu/~eppstein/161/960109.html