การเขียนโปรแกรมคอมพิวเตอร์ 2

Computer Programming II

การเรียกซ้ำ

Recursion

ภิญโญ แท้ประสาทสิทธิ์

ภาควิชาคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร

(taeprasartsit_p at silpakorn dot edu, pinyotae at gmail dot com)

Web site: ???

Facebook group: ComputerProgramming@CPSU

สัปดาห์ที่ 9

เราจะเรียนอะไรในวันนี้

- รู้จักกับการเรียกเมธอดซ้ำ (คือเมธอดเรียกตัวเอง)
- การหาค่าแฟคทอเรียลแบบเรียกซ้ำ
- กลไกการทำงานในการคำนวณแบบเรียกซ้ำ
- ตัวอย่างต่าง ๆ
- กรณีพื้นฐานและกรณีเรียกซ้ำ
- ซ้อมทำโจทย์
- ตัวอย่างที่ซับซ้อนขึ้น

การเรียกซ้ำ (Recursion)

- โดยปรกติแล้ว เวลาที่เราเรียกเมธอดสักอัน เมธอดนั้นก็จะทำงานจนเสร็จ หรือเรียกเมธอดอื่น ๆ มาช่วยกันทำหน้าที่
  - แต่อันที่จริงเมธอดที่ถูกเรียก สามารถเรียกตัวเองซ้ำก็ได้
  - ซึ่งการเรียกตัวเองซ้ำ (ไม่ว่าจะโดยทางตรงหรือทางอ้อม) เป็นเทคนิคในการเขียนโปรแกรมแบบหนึ่งที่เราเรียกว่า *การเรียกซ้ำ* (*Recursion*)
- ขอเริ่มจากตัวอย่างที่ง่าย (แต่ยังไม่ค่อยมีประโยชน์) คือการหาค่า factorial
  - ในทางคณิตศาสตร์ 5! = 5 x 4 x 3 x 2 x 1 = 120
  - ซึ่งเราอาจจะเขียนลูปวนตามโค้ดทางด้านล่าง

```
int result = 1;
for(int i = 2; i <= n; ++i) {
  result = result * i;
}
```

หาค่าแฟคทอเรียลแบบเรียกซ้ำ

- ถ้าจะหาค่าแบบเรียกซ้ำ เราก็ต้องแปลงการคำนวณให้เป็นเมธอดก่อน
  - ในที่นี้ตั้งชื่อเมธอดว่า fact และรับเลขจำนวนเต็ม n เข้ามา
  - เช่น ถ้า n = 5 เราก็จะหาค่าของ n! = 5! ออกมาเป็นผลลัพธ์สุดท้าย
  - นอกจากนี้เราต้องคำนึงถึงนิยามว่า 0! = 1 ด้วย
- เมื่อพิจาณาปัจจัยเงื่อนไขต่าง ๆ แล้ว เมธอดแบบเรียกซ้ำเพื่อคำนวณค่าแฟคทอเรียลที่ได้ก็คือ

```
1  int fact(int n) {
2    if(n == 0)
3      return 1;
4    return n * fact(n - 1);
5  }
```

อธิบายเมธอดหาค่าแฟคทอเรียลแบบเรียกซ้ำ

- พิจารณากรณีที่ n มีค่าเท่ากับ 5 เป็นตัวอย่าง
  - เมื่อเราต้องการหาค่าของ 5! เก็บไว้ใน result เราจะเขียนว่า

```
int result = fact(5);
```

- ซึ่งเมธอด fact จะถูกเรียก โดยมีค่า n เป็น 5
  - เมื่อพบ if ในบรรทัดที่ 2 โปรแกรมก็จะข้ามไปบรรทัดที่ 4
  - ตรงบรรทัดที่ 4 นี่แหละที่เกิดการเรียกซ้ำ

```
return n * fact(n - 1);
```

  - สังเกตด้วยว่าในการเรียกซ้ำ ค่าพารามิเตอร์ที่ส่งไปจะลดลงไปหนึ่ง ดังนั้นในตัวอย่างนี้ การเรียกซ้ำจะมีค่าเป็น fact(4)
- คำถามจากจุดนี้ก็คือว่า แล้วผลลัพธ์สุดท้ายจะออกมาได้อย่างไร

การเรียกซ้ำและจุดยุติ

- ถ้าเมธอดจะถูกเรียกซ้ำไปเรื่อย ๆ แล้วผลลัพธ์จะออกมาอย่างไร
  - เป็นไปได้อย่างไรที่สุดท้ายเราจะได้ค่าของ fact(5) ออกมาเป็นผลลัพธ์ ดูเหมือนเมธอดจะถูกเรียกซ้ำไปเรื่อย ๆ
  - เช่นในตัวอย่างนี้ เมื่อเรียก fact(4) จะได้ว่าค่า n ไม่เป็น 0 และโปรแกรมจะข้ามไปบรรทัดที่ 4 และคำนวณว่า return 4 * fact(4 – 1);
  - นั่นค่อ ต่อไปโปรแกรมจะเรียกเมธอดพร้อมพารามิเตอร์ fact(3)
  - และจะเรียกซ้ำต่อกันแบบนี้ไปเรื่อย ๆ จนกว่าจะถึงจุดยุติ
- เรื่องจุดยุตินี่แหละที่เป็นหนึ่งในประเด็นหลักของการเรียกซ้ำ เพราะถ้ามันเรียกตัวเองซ้ำไปเรื่อย ๆ ไม่มีจุดยุติ โปรแกรมก็ไม่มีวันทำงานเสร็จ
- แล้วจุดยุติของเมธอด fact อยู่ที่ใด

จุดยุติและการคืนผลลัพธ์

- จุดยุติของการเรียกซ้ำคือ กรณีที่เมธอดไม่เรียกซ้ำและคืนค่าผลลัพธ์กลับไป
  - ซึ่งของแฟคทอเรียลก็คือบรรทัดที่ 2 และ 3

```
1  int fact(int n) {
2    if(n == 0)
3      return 1;
4    return n * fact(n - 1);
5  }
```

  - ถ้า n มีค่าเป็น 0 เมธอดจะไม่เรียกตัวเองซ้ำ แต่จะคืนค่า 1 กลับไป
  - เมื่อคืนค่ากลับไปให้ผู้เรียกได้ ผู้เรียกก็จะคืนค่าผลลัพธ์อื่น ๆ สืบคืนต่อไปได้
- ก่อนที่ fact(0) จะถูกเรียก fact(1) ก็ถูกเรียกก่อน
  - และค่าที่ fact(1) จะคืนก็มาจาก return n * fact(n – 1); เมื่อ n == 1 ซึ่งก็คือ 1 * fact(1 – 1); และได้ผลลัพธ์จาก fact(0) มาคำนวณสืบต่อ

ผลลัพธ์ที่ถูกคืนส่งมาเป็นทอด ๆ

- เมื่อ `fact(1)` ได้ค่ามาเป็น `1 * fact(0) = 1 * 1 = 1`
  คำสั่ง `return 1 * fact(1 - 1);` ก็จะคืนค่า 1 กลับไปหาผู้เรียก
- เนื่องจากผู้เรียกก็คือ `fact(2)` ค่าที่จะคืนกลับไปก็คือ
  `2 * fact(2 - 1) = 2 * fact(1) = 2 * 1 = 2`
- ในทำนองเดียวกัน ผู้เรียก `fact(2)` ก็คือ `fact(3)` ค่าที่จะคืนกลับไปก็คือ `3 * fact(3 - 1) = 3 * 2 = 6`
- ต่อมาเราก็คำนวณค่า `fact(4)` ได้เป็น
  `4 * fact(4 - 1) = 4 * fact(3) = 4 * 6 = 24`
- สุดท้าย `fact(5) = 5 * fact(5 - 1) = 5 * fact(4)`
  `= 5 * 24 = 120`
- นั่นคือ ผลลัพธ์ที่ส่งคืนกลับมาเป็นทอด ๆ ช่วยให้คำนวณค่าที่ต้องการได้

ข้อสังเกต

- การเรียกซ้ำ ทำให้โปรแกรมสามารถคำนวณค่าได้โดยที่ไม่ต้องมีลูป
- เทคนิคในการเรียกซ้ำจึงเป็นสิ่งที่จำเป็นในภาษาบางประเภทที่ไม่มีลูป เช่น ภาษากลุ่ม functional programming
  - มี Haskell และ Scheme เป็นต้น
- การเรียกซ้ำดูเข้าใจยากในตอนแรก และยากกว่าการใช้ลูป
  - แต่แท้จริงจะยากกว่าการใช้ลูปหรือเปล่า มันขึ้นอยู่กับว่างานที่เราต้องคำนวณเป็นแบบไหน
  - ถ้างานที่ต้องคำนวณมีโครงสร้างแบบซ้อนไปเรื่อย ๆ อย่างต้นไม้ที่เก็บข้อมูล (เรียนในวิชาโครงสร้างข้อมูล) การเรียกซ้ำอาจจะง่ายกว่าลูป

ดูอีกตัวอย่าง: ทำความรู้จักกับการเรียกซ้ำให้มากขึ้น

- เมื่อกำหนดค่า N ซึ่งเป็นจำนวนเต็มบวกใด ๆ มาให้ จงเขียนโปรแกรมหาผลบวกของ 1 + 2 + 3 + ... + N โดยไม่ใช้ลูป แต่ใช้การเรียกซ้ำ
- เริ่มกันที่นิยามของเมธอดแบบเรียกซ้ำตามแบบข้างล่าง

```
int sum(int N) { ... }
```

- ต่อมาเราจะหาค่าของการบวกของค่า N ใหญ่ ๆ จากค่า N ที่เล็กกว่า
  - เราสามารถใช้วิธีแบบเดียวกับการหาค่าแฟคทอเรียล และได้โค้ดเป็น

```
int sum(int N) {
  if(N == 1)
    return 1;
  return N + sum(N - 1);
}
```

เรียกซ้ำได้มากกว่า 1 จุดในเมธอดเดียว

ตัวอย่าง Fibonacci Number

เลขฟิบอนนาชชี เป็นค่าที่ถูกนิยามไว้ด้วยฟังก์ชันคณิตศาสตร์ Fib ว่า

Fib(n) = Fib(n – 1) + Fib(n – 2) เมื่อ Fib(0) = 0 และ Fib(1) = 1

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

แล้วแบบนี้จะเกิดอะไรขึ้น

- จากตัวอย่างเรื่องจำนวนฟิบอนนาชชี ตรงจังหวะการคำนวณ

```
return fib(n - 1) + fib(n - 2);
```

- เครื่องจะคำนวณค่า `fib(n-1)` ก่อน แล้วจึงคำนวณ `fib(n-2)`
  - หมายความว่า ถ้าหากเราเรียกเมธอด `fib(4)` เครื่องจะหาคำตอบของ `fib(3)` ให้ได้ก่อน จากนั้นจึงไปหาค่า `fib(2)`
  - เมื่อได้ค่า `fib(3)` และ `fib(2)` แล้วจึงนำค่าทั้งสองนั้นมาบวกกัน เป็นคำตอบของค่า `fib(4) = fib(3) + fib(2)`
- ในทำนองเดียวกันตอนที่เครื่องจะหาค่า `fib(3)` มันก็จะพยายามหาค่า `fib(2)` ก่อน
  - เสร็จแล้วก็จะคำนวณค่า `fib(1)` ซึ่งมีค่าเป็น 1 และได้คำตอบเป็น `fib(3) = fib(2) + 1`

ลำดับการคำนวณเพื่อให้ได้คำตอบสุดท้ายเป็นอย่างไร

- [ดูการคำนวณจากอาจารย์บนกระดานหรือวิชวลไลเซอร์]

เราจะเรียนอะไรในวันนี้

- รู้จักกับการเรียกเมธอดซ้ำ (คือเมธอดเรียกตัวเอง)
- การหาค่าแฟคทอเรียลแบบเรียกซ้ำ
- กลไกการทำงานในการคำนวณแบบเรียกซ้ำ
- ตัวอย่างต่าง ๆ
- **กรณีพื้นฐานและกรณีเรียกซ้ำ**
- ซ้อมทำโจทย์
- ตัวอย่างที่ซับซ้อนขึ้น

กรณีพื้นฐานและกรณีเรียกซ้ำ

- จากตัวอย่างที่ผ่านมา เราเรียกเงื่อนไขตรง if ที่ทำให้หยุดการวนซ้ำว่า *กรณีพื้นฐาน* (*base case*)
- และเรียกเหตุการณ์ที่มีการเรียกเมธอดซ้ำว่า *กรณีเรียกซ้ำ* (*recursive case*)
- เวลาที่เราจะเขียนเมธอดที่มีการเรียกซ้ำ (recursive method) เราต้องคิดถึงกรณีทั้งสองนี้ควบคู่กันไปเสมอ
  - ถ้าไม่มีกรณีพื้นฐาน การเรียกซ้ำจะไม่มีวันหยุด คล้ายกับโปรแกรมติดลูปไม่รู้จบ
  - ส่วนถ้าไม่มีกรณีเรียกซ้ำ เราก็อาจจะไม่ได้สิทธิประโยชน์จากความสามารถนี้ในภาษาคอมพิวเตอร์
- ในตัวอย่างเรื่องจำนวนฟิบอนนาชชี เราจะเห็นได้ว่าการเรียกซ้ำทำให้เราเขียนโปรแกรมคำนวณค่าฟิบอนนาชชีเสร็จเร็วมาก ง่ายกว่าเขียนลูปเอง

แล้วอะไรเป็นแนวปฏิบัติในการเขียนเมธอดแบบเรียกซ้ำ

- เราควรมองหาความสัมพันธ์ก่อนว่าปัญหาขนาดใหญ่ แก้ได้จากปัญหาขนาดเล็กที่มีลักษณะคล้ายกันหรือไม่
  - เช่น เราหาค่าแฟคทอเรียล n จากแฟคทอเรียล n – 1 ได้หรือไม่
  - เราหาจำนวนฟิบอนนาชชี n จากจำนวนฟิบอนนาชชี n – 1 ได้หรือไม่
  - ถ้าได้ก็แสดงว่าธรรมชาติของปัญหาสามารถคิดแบบเรียกซ้ำได้
- เมื่อเห็นว่าคิดแบบเรียกซ้ำได้ ก็ต้องคำนึงถึงกรณีพื้นฐาน ซึ่งเป็นเหมือนต้นขั้วของค่าผลลัพธ์ว่ามีอะไรบ้าง
  - จากนั้นยกกรณีพื้นฐานพวกนี้ไว้ทางตอนต้นของเมธอด คือเอาไว้ก่อนที่จะมีการเรียกซ้ำ
- ส่วนกรณีเรียกซ้ำจะตามหลังกรณีพื้นฐาน คือจะเรียกซ้ำก็ต่อเมื่อค่าที่ได้มาตรงพารามิเตอร์ไม่ตรงกับกรณีพื้นฐานเท่านั้น

มาทดสอบความเข้าใจ คุณคิดข้อนี้ออกหรือไม่

- จงเขียนเมธอดเพื่อหาผลคูณของ a และ b โดยที่
  - กำหนดให้เมธอดมีรูปแบบเป็น `int multiply(int a, int b)`
  - a และ b เป็นจำนวนเต็มบวก
  - ห้ามใช้เครื่องหมายคูณ ใช้ได้เฉพาะ + (หรือจะใช้ – ด้วยก็ตามใจ)
  - ห้ามใช้ลูปด้วย
- เอ้า ให้เวลาคิดสักพัก แล้วค่อยมาดูเฉลย
  - อย่าลืมนะว่าเราต้องคิดหาความสัมพันธ์ว่าปัญหาใหญ่ คิดออกมาจากปัญหาเล็กได้หรือเปล่า
  - เช่น เราจะหาค่า a x b ได้เมื่อ a หรือ b มีค่าน้อยลงหรือไม่
  - และก็อย่าลืมกรณีพื้นฐานด้วย

เฉลย recursive multiply

```
int multiply(int a, int b) {
  if(b == 1)
    return a;
  return a + multiply(a, b - 1);
}
```

หรือ

```
int multiply2(int a, int b) {
  if(a == 1)
    return b;
  return b + multiply2(a - 1, b);
}
```

ลองอีกข้อ เผื่อใครอยากแก้มือ

- กำหนดอาเรย์ที่ข้างในมีเลขจำนวนเต็มอยู่ N ตัว
  - ต้องการนับว่าข้างในมีเลข 5 อยู่กี่ตัว
  - ให้เมธอดมีรูปแบบว่า `int count5(int[] A, int n)`
  - กำหนดให้อาเรย์มีตัวเลขอยู่อย่างน้อยหนึ่งตัว
  - ห้ามใช้ลูป ให้ใช้เฉพาะการเรียกซ้ำ
- เอ้า ให้เวลาคิดสักพัก แล้วค่อยมาดูเฉลย
  - เช่นเดิม อย่าลืมคิดว่าเราแก้ปัญหาขนาดใหญ่ได้จากปัญหาที่มีขนาดเล็กลงหรือไม่
  - ว่าแต่กรณีพื้นฐานที่จะสรุปค่าได้มันควรจะเป็นอย่างไร

เฉลยข้อ count5

```
int count5(int[] A, int n) {
  int count = 0;
  if(A[n - 1] == 5)
    count = 1;

  if(n == 1)
    return count;
  return count + count5(A, n - 1);
}
```

เอาอีก ๆ อยากได้แบบที่เกี่ยวกับสตริงมั่ง มีมั้ย

- สมมติว่ามีวัตถุสตริง `str` ความยาว `N`
  - ต้องการหาว่าในสตริงนี้มี `a` หรือ `A` อยู่กี่ตัว
  - กำหนดเมธอดในรูปแบบ `int countA(String str, int n)`
  - กำหนดให้ `str` เป็นข้อความที่มีตัวอักขระอยู่อย่างน้อยหนึ่งตัว
  - ห้ามใช้ลูป ให้ใช้เฉพาะการเรียกซ้ำ
- เหมือนเดิม ให้เวลาคิดสักพัก แล้วค่อยมาดูเฉลย

เฉลยข้อ countA

```
int countA(String str, int n) {
  int count = 0;
  char c = str.charAt(n - 1);
  if(c == 'a' || c == 'A')
    count = 1;

  if(n == 1)
    return count;
  return count + countA(str, n - 1);
}
```

ประโยชน์จากการเขียนโปรแกรมแบบเรียกซ้ำ

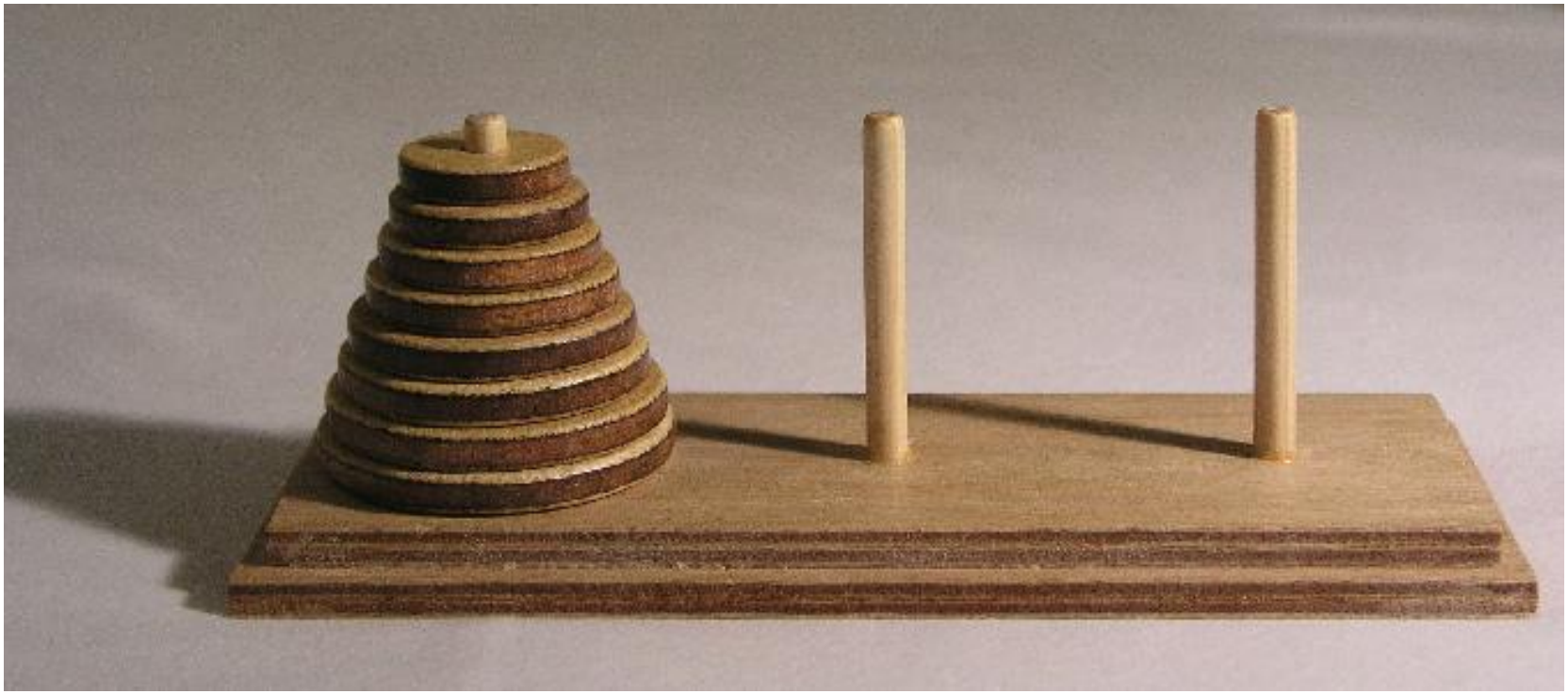
- เราลุยโจทย์ไปหลายข้อแล้ว บางทีเรารู้สึกว่าเขียนแบบลูปไปตลอดเลยจะง่ายกว่ามั้ย
  - ยกเว้นข้อฟิบอนนาชชี ที่เขียนแบบเรียกซ้ำแล้วง่ายกว่า เพราะธรรมชาติของมันเป็นแบบเรียกซ้ำ
  - แต่ไม่ว่าจะเป็นแบบไหน เราจะพบว่าคำตอบที่เขียนแบบเรียกซ้ำมันสั้นและสวยงามมาก ดูออกก็ง่ายว่าโปรแกรมถูกหรือผิด
  - และการที่ดูออกง่ายว่าคิดถูกหรือผิดนี้แหละที่เป็นประโยชน์ที่ยิ่งใหญ่อันหนึ่งของการเขียนโปรแกรมแบบเรียกซ้ำ
- คราวนี้เราจะมาเน้นตัวอย่างที่แสดงให้เห็นถึงประโยชน์ของการเขียนโปรแกรมแบบเรียกซ้ำมากขึ้น อันแรกที่เราจะพูดถึงคือหอคอยฮานอย

เราจะเรียนอะไรในวันนี้

- รู้จักกับการเรียกเมธอดซ้ำ (คือเมธอดเรียกตัวเอง)
- การหาค่าแฟคทอเรียลแบบเรียกซ้ำ
- กลไกการทำงานในการคำนวณแบบเรียกซ้ำ
- ตัวอย่างต่าง ๆ
- กรณีพื้นฐานและกรณีเรียกซ้ำ
- ซ้อมทำโจทย์
- **ตัวอย่างที่ซับซ้อนขึ้น**

หอคอยฮานอย (Tower of Hanoi)

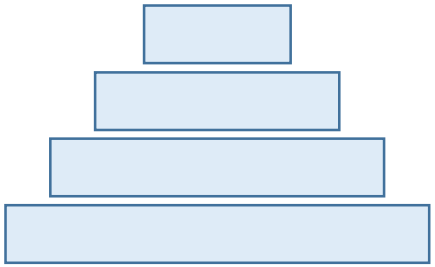
- จากภาพข้างล่าง เราต้องการย้ายแผ่นวงกลมไปไว้ที่อีกเสาหนึ่ง
  - ย้ายได้ทีละแผ่น
  - ห้ามวางแผ่นวงกลมไว้บนแผ่นที่เล็กว่า
  - ขั้นตอนการย้ายควรเป็นอย่างไร



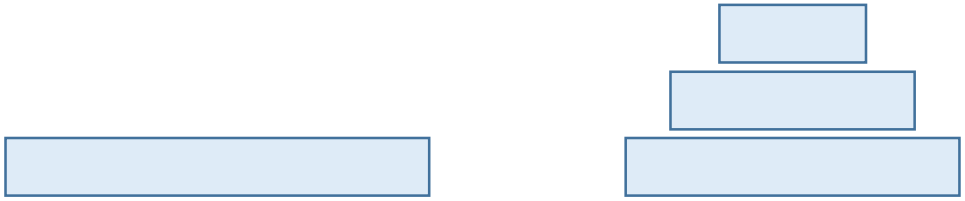
ภาพจาก Wikipedia [https://en.wikipedia.org/wiki/Tower_of_Hanoi]

มองแบบ Recursive

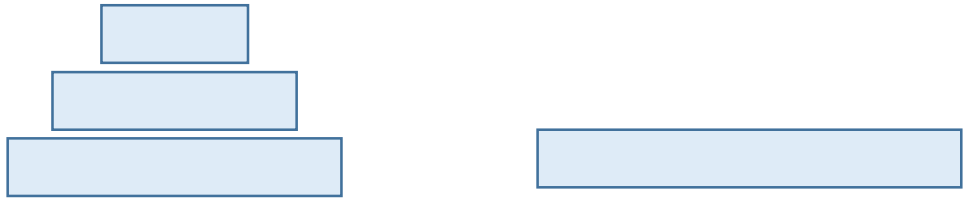
- สมมติว่าเรามีวงกลมสี่แผ่น



- เราจะคิดเหมือนกับว่า เราจะย้ายแค่ 3 แผ่นไปไว้ที่เสาอื่น

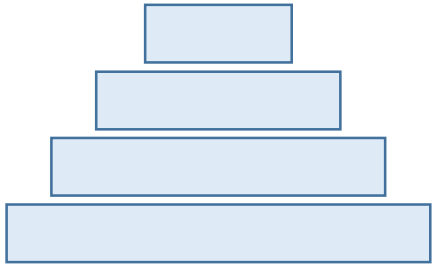


- ต่อจากนั้นก็ย้ายฐานไปไว้ที่อีกเสา



มองแบบ Recursive (2)

- ต่อจากนั้นก็ย้าย 3 แผ่นดังกล่าวไปไว้บนฐานที่เสาใหม่

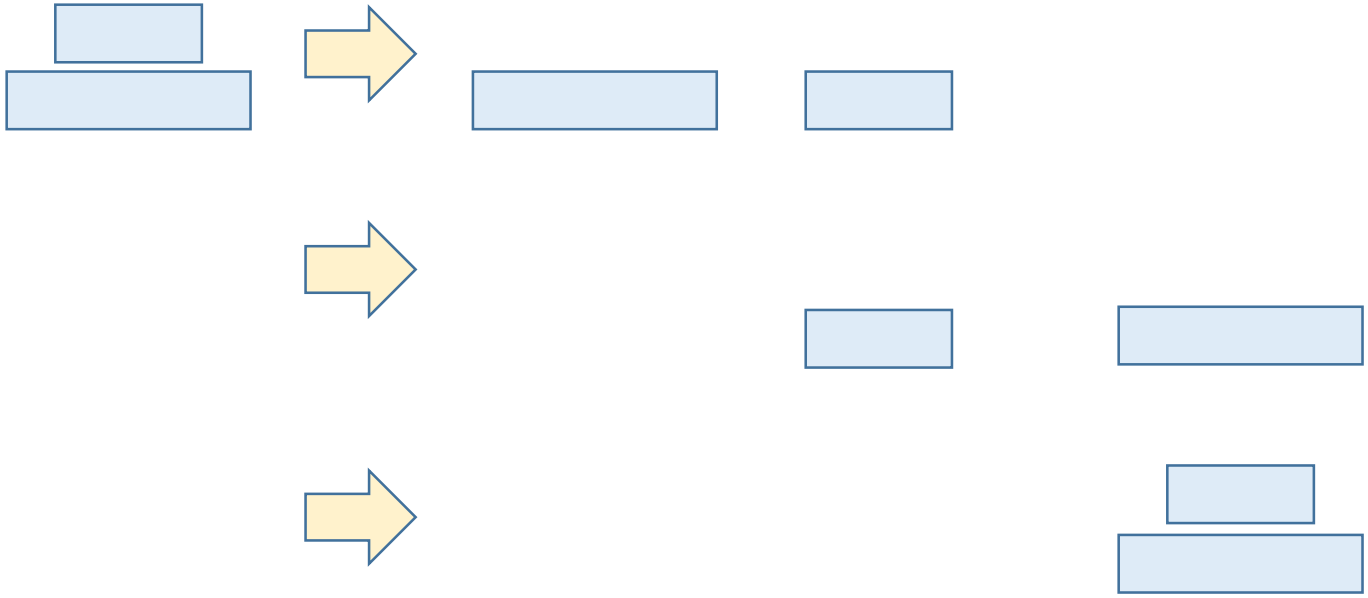


- เดี๋ยว ๆ เราย้ายได้ทีละแผ่นไม่ใช่เหรอ
  - ใช่แล้วล่ะ แต่เรากำลังมองแบบรีเคอร์ซีฟโดยย่อยปัญหาลงหนึ่งระดับก่อน
  - ซึ่งถ้าเราพบว่าการย่อยลงหนึ่งระดับช่วยในการแก้ปัญหาที่ใหญ่ขึ้นได้ ก็แสดงว่าปัญหานี้แก้ได้ด้วยวิธีแบบเรียกซ้ำ
  - ในที่นี้ก็คือ แทนที่เราจะคิดย้ายหอคอยแบบ 4 แผ่น เราคิดย้ายแบบ 3 แผ่นให้ได้ บวกกับการย้ายฐาน และย้ายแบบ 3 แผ่นอีกที
  - ตอนคิดจะย้ายแบบ 3 แผ่น เราก็มองแบบย่อยปัญหาไปเหลือ 2 แผ่นอีกที

ลองคิดย่อยปัญหาต่อไปอีก

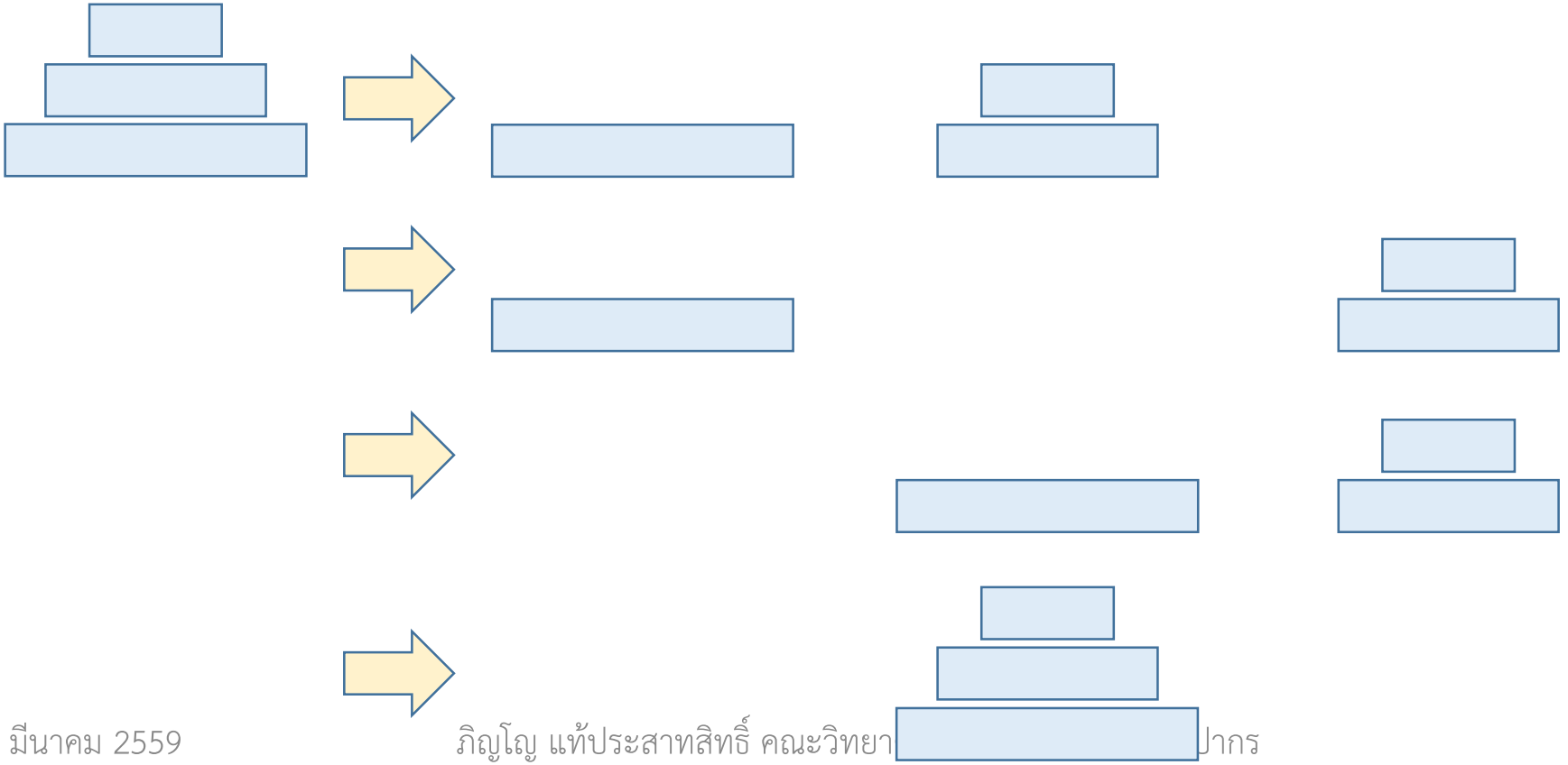
- สมมติว่าจะย้ายแบบ 3 แผ่นให้ได้ มันก็จะเป็นแบบนี้

- ย่อยไปแบบซ้อน 2 แผ่นนี่แหละที่เราจะหาคำตอบได้ง่าย ดังนี้



ยังไม่เข้าใจว่ามันสำคัญอย่างไร

- จะเห็นได้ว่าตอนที่มันมี 4 แผ่น ที่จริงเรายังไม่ต้องมองไปที่แผ่นทั้ง 4
  - แต่มองย้อนไปเหมือนมีแค่สองแผ่นก่อน
  - เสร็จแล้วก็ย้ายสองแผ่นนั้นให้ได้
  - พอย้ายสองแผ่นได้ก็มองย้อนกลับมาที่ระดับ 3 แผ่น



สรุปได้ว่า แก้ปัญหาเล็ก ๆ ให้ได้แล้วค่อยแก้ปัญหาใหญ่

- ในปัญหานี้ ถ้าเราจะวางแผนแก้ปัญหาใหญ่โดยตรงจะเป็นมุมมองที่ยาก
  - เราจึงมองปัญหาย่อย และย่อยลงมาเรื่อย ๆ จนถึงจุดที่วิธีแก้มันชัดเจน
  - ซึ่งในที่นี้ก็คือตอนที่มันเหลือ 2 แผ่นจะชัดเจนมาก (แต่จะมองไปจนถึง 1 แผ่นเลยก็ได้เช่นกัน)
  - เมื่อย้ายแบบ 2 แผ่นไปที่เสาอื่นแล้ว ก็ย้ายแผ่นที่ 3 ไปไว้ที่เสาอื่น
  - จากนั้นก็ย้าย 2 แผ่นที่ย้ายไปตอนแรกไปไว้บนแผ่นที่ 3
  - ก็จะได้ว่าเราย้ายแบบ 3 แผ่นไปไว้ที่เสาอื่นได้
  - ต่อจากนั้นก็ย้ายฐานไปไว้อีกเสา
  - สุดท้ายก็ย้ายแบบ 3 แผ่นไปทับฐานที่ย้ายไปแล้ว

ยังมีปัญหาอื่น ๆ ที่มองแบบเรียกซ้ำแล้วจะง่ายกว่า

- เช่นการเรียงข้อมูลแบบ Quick Sort ซึ่งถือได้ว่าเป็นวิธีการที่เร็วที่สุดวิธีหนึ่ง การเขียนโปรแกรมแบบเรียกซ้ำนับว่าเป็นวิธีที่ดี
- การสร้างต้นไม้โครงสร้างของโฟลเดอร์
  - คือมองว่ามีโฟลเดอร์ย่อยในโฟลเดอร์ใหญ่ซ้อนไปเรื่อย ๆ
  - ทำให้สามารถลบไฟล์ที่โฟลเดอร์ใหญ่พร้อมกับโฟลเดอร์และไฟล์ที่ซ้อนอยู่ข้างในทั้งหมดได้ (ตอนลบโปรแกรมจะมองย้อนเข้าไปในโฟลเดอร์ย่อย)
- การนับจำนวนของรูปแบบสิ่งของทั้งหมดที่เป็นไปได้
- การดำเนินการบนกราฟต่าง ๆ ที่จะได้เรียนในวิชาอัลกอริทึม
  - เช่นการหาเส้นทางเดินออกจากเขาวงกต