เอกสารประกอบการสอนรายวิชา 305234
การวิเคราะห์และการออกแบบวิธี

Algorithm Analysis and Design

จัดทำโดย

อาจารย์จิราพร พุกสุข

ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

สำหรับใช้ประกอบการสอน รายวิชา 305233 และ 305234 การวิเคราะห์และออกแบบขั้นตอนวิธี
ภาคเรียนปลาย ปีการศึกษา 2562

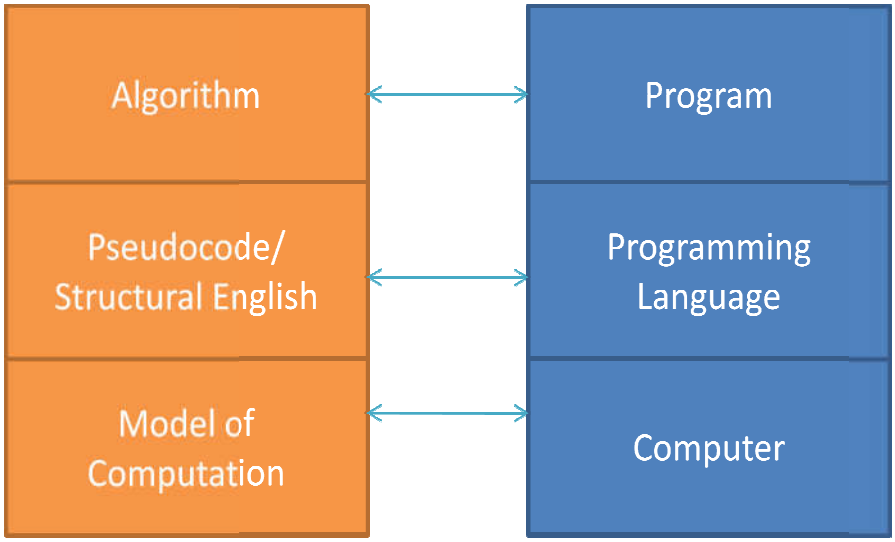
สารบัญ

แนะนำรายวิชา

อัลกอริทึมคือ ลำดับขั้นตอนการคำนวณที่ได้กำหนดไว้เพื่อในการแก้ปัญหาอย่างใดอย่างหนึ่ง ซึ่งจะมีการรับค่าอินพุตและแสดงค่าเอ๊าพุตสำหรับการแก้ปัญหานั้นๆ

อัลกอริทึมใดๆ จะถือว่าถูกต้อง ถ้าหากว่าไม่ว่าจะใส่อินพุตใดๆให้กับอัลกอริทึมนั้น มันจะต้องหยุดทำงานพร้อมกับให้เอ๊าพุตที่ถูกต้อง

<u>แผนภาพที่ 1</u> แสดงความสัมพันธ์ระหว่างแนวคิดทางทฤษฎี กับการทำแนวคิดนั้นให้สำเร็จจริง



ภาพที่ 1

ใน**การวัดประสิทธิภาพของอัลกอริทึม**นั้น มีการวัดผลหลักๆอยู่สองด้านคือ

1. ความถูกต้องของอัลกอริทึม

    ซึ่งจะต้องตรวจสอบว่าสามารถให้คำตอบได้ถูกต้องไม่ว่าจะเป็นอินพุตใดๆ

2. เวลาที่ใช้ในการประมวลผลอัลกอริทึม

    ซึ่งจะต้องตรวจสอบว่าใช้เวลาน้อยที่สุดในการที่จะทำการประมวลผลให้เสร็จ

ตัวอย่างที่ 1 การวัดประสิทธิภาพของอัลกอริทึม

กำหนดให้มีอัลกอริทึมอยู่ 2 อัลกอริทึมดังนี้

1. insertion sort ใช้เวลาในการคำนวณประมาณ xn^2

2. merge sort ใช้เวลาในการคำนวณประมาณ y nlgn ($lg=log_2$)

เมื่อ x,y คือค่าคงที่ใดๆ และ n คือขนาดของอินพุต

และกำหนดให้ มีคอมพิวเตอร์ 2 เครื่องซึ่งมีความเร็วในการรันชุดคำสั่งแตกต่างกันดังนี้

1. คอมพิวเตอร์ A สามารถรันได้ 10^9 คำสั่งต่อวินาที

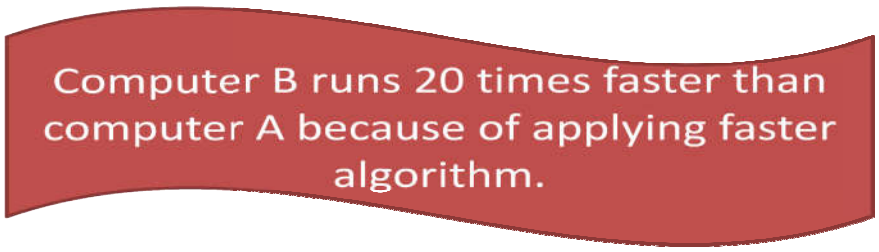
2. คอมพิวเตอร์ B สามารถรันได้ 10^7 คำสั่งต่อวินาที

ในการทดสอบ หากกำหนดให้ x = 2 , y = 50, n= 10^6 และให้เครื่องคอมพิวเตอร์ A รันอัลกอริทึม insertion sort ส่วน B รัน merge sort จะพบว่า

คอมพิวเตอร์ A ใช้เวลาในการประมวลผล insertion sort = $2*(10^6)^2 / 10^9 =$ 2000 วินาที

คอมพิวเตอร์ B ใช้เวลาในการประมวลผล merge sort = $50*10^6 \lg 10^6 / 10^7 =$ 100 วินาที

สามารถที่จะสรุปได้ว่า ถึงแม้เครื่องคอมพิวเตอร์ A จะมีความเร็วในการประมวลผลมากกว่าเครื่องคอมพิวเตอร์ B แต่หากนำเครื่อง A มารันอัลกอริทึม insertion sort ที่ต้องการเวลาในการคำนวณมากกว่า ก็จะทำให้สุดท้ายแล้วเครื่อง A ทำงานช้ากว่า เครื่อง B ที่รันอัลกอริทึม merge sort ที่ต้องการเวลาในการคำนวณน้อยกว่า



จากตัวอย่างที่ 1 จะเห็นว่าการเลือกอัลกอริทึมที่ดีในการทำงานนั้นมีผลต่อประสิทธิภาพของระบบอย่างมาก

ดังนั้นในการจะ**เลือกว่าอัลกอริทึมไหนดีหรือไม่ดีอย่างไร สามารถวิเคราะห์ได้จาก** สองด้านหลักๆ ตาม ประสิทธิภาพของอัลกอริทึม คือ

1. ความถูกต้องของอัลกอริทึม

    ซึ่งสามารถวิเคราะห์ได้ โดยการใช้เทคนิคการไม่แปรเปลี่ยนของการวนซ้ำ (loop invariants) ในกรณีที่อัลกอริทึมมีขั้นตอนการทำซ้ำ

2. เวลาที่ใช้ในการประมวลผลอัลกอริทึม

    ซึ่งสามารถวิเคราะห์ได้ จากการหาค่าขอบเขตบน ของเวลาที่ใช้ในการรันอัลกอริทึมนั้นๆ ซึ่งสามารถคำนวณคร่าวๆได้จากการดูอัตราการเติบโตของฟังก์ชัน (growth of function)

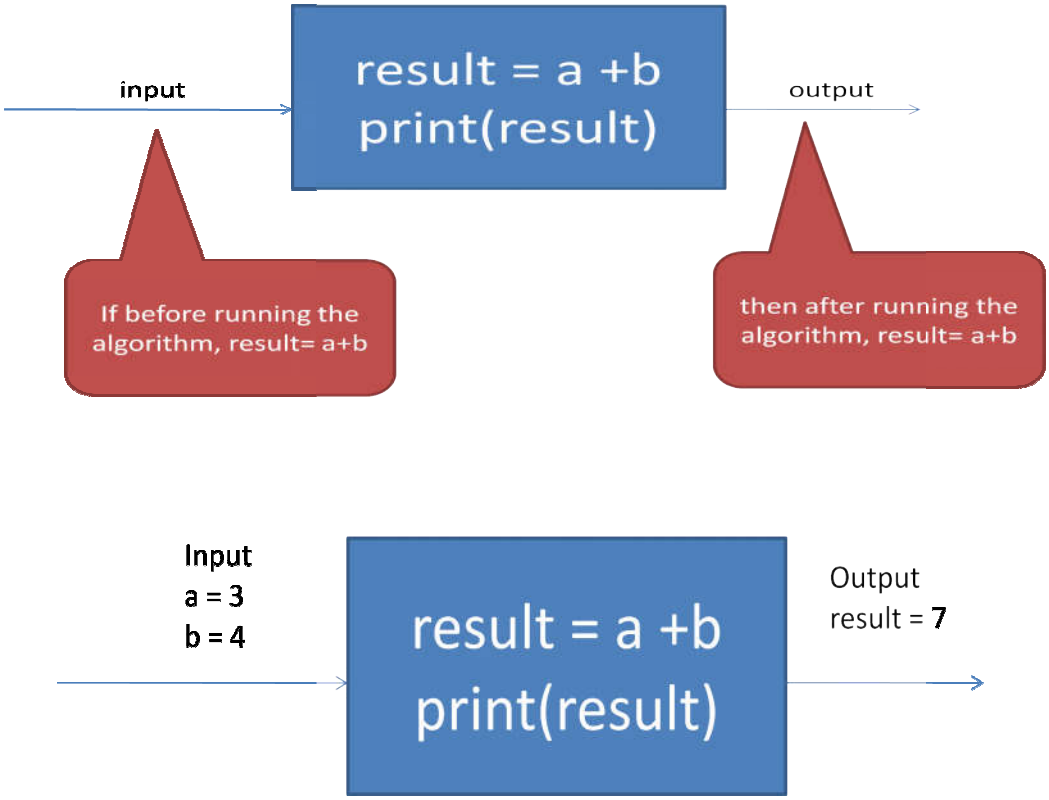
บทที่ ๑ การวิเคราะห์ความถูกต้องของอัลกอริทึม
กับ การไม่แปรเปลี่ยนของการวนซ้ำ (loop invariants)

การวิเคราะห์ความถูกต้องของอัลกอริทึม ก็คือการตรวจสอบว่า ไม่ว่าจะใส่อินพุตใดๆให้กับอัลกอริทึมนั้นๆ มันจะให้คำตอบที่ถูกต้องเสมอ

ดังแสดงตัวอย่าง ใน**ภาพที่ 2** หากกำหนดให้ กรอบสีน้ำเงินคืออัลกอริทึมที่เราต้องการวิเคราะห์

จะเห็นว่า ก่อนรันอัลกอริทึม ต้องทำการตรวจสอบว่า ค่าของ ตัวแปรresult จะต้องคือค่า อินพุต a + b แต่เนื่องจาก ก่อนรันอัลกอริทึม ค่าของ a,b ยังไม่มีค่า ดังนั้น result=0

ต่อมาทำการตรวจสอบว่า หลังรันอัลกอริทึม ค่าของ result= a+b หรือไม่ สมมติกำหนดให้ a = 3, b=4 ก็จะต้องพบว่า result = 7 เท่านั้น จึงจะถือว่า อัลกอริทึมนี้ถูกต้อง



ภาพที่ 2

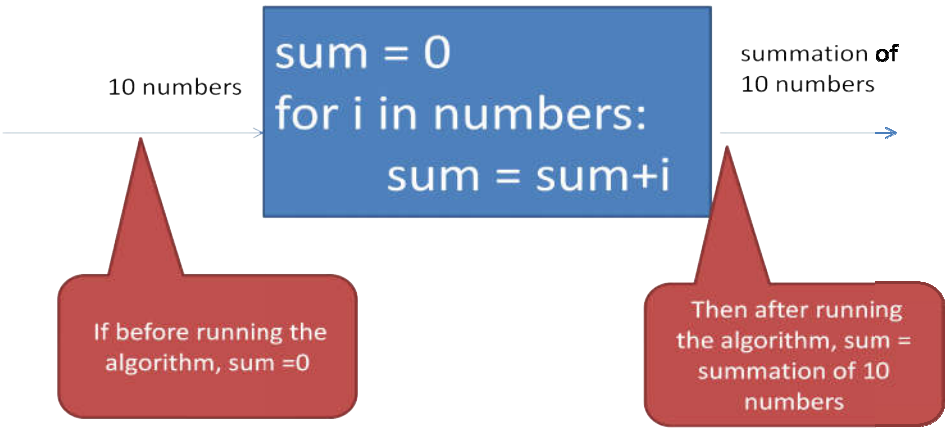
แต่กรณีที่อัลกอริทึมมีขั้นตอนส่วนของการทำซ้ำ จะไม่สามารถใช้เทคนิคการตรวจสอบแบบข้างต้นได้ เนื่องจากจะต้องทำการตรวจสอบว่า ในทุกๆรอบของการทำซ้ำนั้น ค่าของข้อมูลที่ต้องการใช้เพื่อเป็นคำตอบของการแก้โจทย์ปัญหานั้นไม่มีการเปลี่ยนแปลงค่า

อย่างเช่น อัลกอริทึมการหาค่าผลรวมของตัวเลขจำนวน 10 ตัว ที่จะต้องทำการตรวจสอบทุกรอบ(ทั้ง 10 รอบ) ว่า

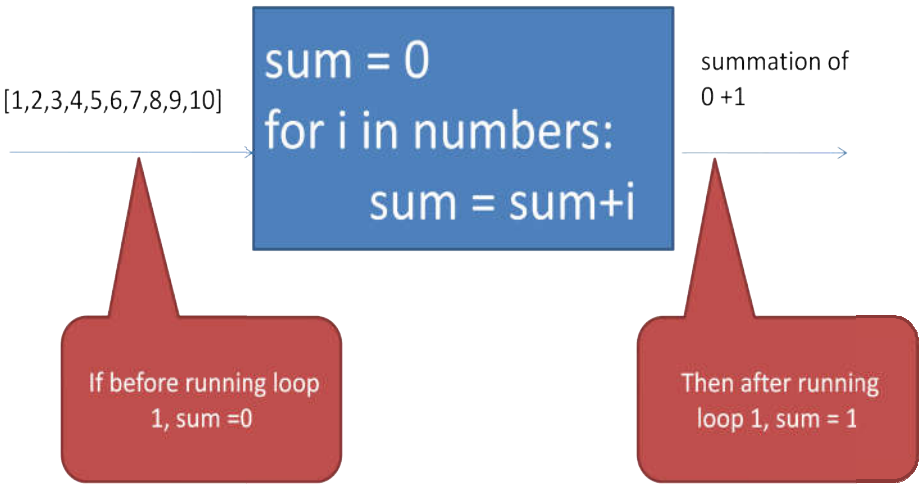
1. ค่าผลรวมนั้นถูกต้องก่อนที่จะรันลูป ในแต่ละรอบ และ

2. ค่าผลรวมนั้นถูกต้องหลังจากที่รันลูปในแต่ละรอบ

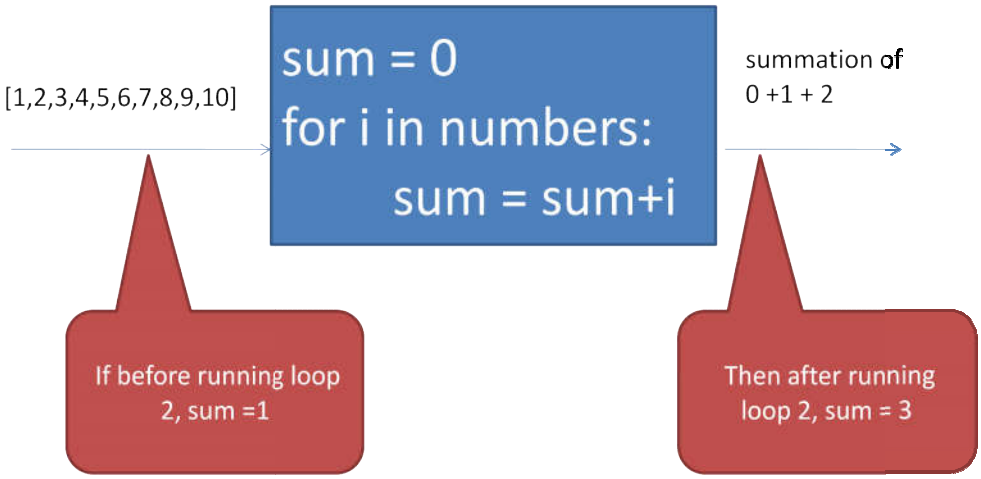
ดังแสดงตัวอย่างใน**ภาพที่ 3**



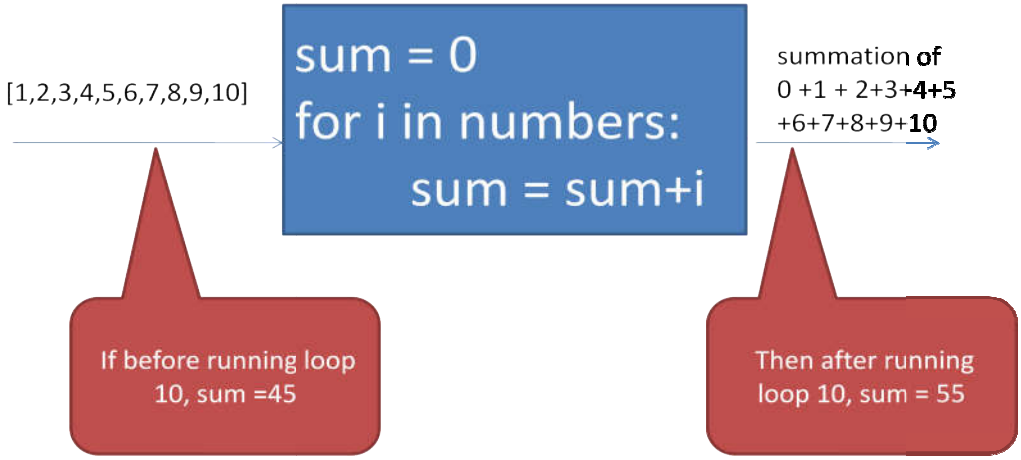
รอบที่ 1



รอบที่ 2



รอบต่อๆไป จนถึง รอบที่ 10



ภาพที่ 3

จากรูปตัวอย่างเราจะพบข้อสังเกตว่า

ก่อนที่จะรันรอบที่ n ใดๆ ค่าผลรวมของตัวเลขจะต้องเป็นผลรวมของตัวเลขตั้งแต่ตัวแรกจนถึงตัวที่ n

Loop invariant คืออะไร

การไม่แปรเปลี่ยนของการวนซ้ำ หรือ loop invariant คือ คำสั่งแสดงคุณลักษณะบางอย่างของตัวแปรในอัลกอริทึม ซึ่งจะต้องเป็นจริงทั้งก่อนและหลังในการรันลูปแต่ละรอบ

เทคนิคในการพิสูจน์การไม่แปรเปลี่ยนของการวนซ้ำ มีขั้นตอนหลักๆด้วยกันอยู่ 4 ขั้นตอนคือ

1. ขั้นตอนกำหนด loop invariant

    ทำการกำหนด คำสั่งแสดงคุณลักษณะของตัวแปรในอัลกอริทึม (loop invariant) ที่ต้องการจะตรวจสอบว่า คำสั่งนี้ จะต้องเป็นจริง ทั้งก่อนและหลัง ในทุกๆรอบของการทำซ้ำ

2. ขั้นตอน initialization

    ทำการตรวจสอบว่า คำสั่งในข้อ 1 นั้นเป็นจริง ก่อนที่จะเริ่มรันลูปแรก หรือไม่

3. ขั้นตอน maintenance

    ทำการตรวจสอบว่า **ถ้า**คำสั่งในข้อ 1 นั้นเป็นจริง ก่อนที่จะรันลูปรอบนั้นๆ (รอบที่ i) **แล้ว** คำสั่งนั้นจะต้องเป็นจริง ก่อนที่จะรันลูปรอบถัดไป (รอบที่ i+1)

4. ขั้นตอน termination

    ทำการตรวจสอบว่า เมื่อหยุดการทำซ้ำแล้ว (หมดลูป) คำสั่งในข้อที่ 1 (การที่คุณลักษณะนั้นไม่เปลี่ยนแปลง) ช่วยให้เห็นว่าอัลกอริทึมนั้นถูกต้อง

    คือเมื่อหยุดลูปแล้ว จะต้องเห็นว่าเอ้าพุตของอัลกอริทึมนั้นถูกต้อง

ข้อสังเกต จะเห็นว่าขั้นตอน initialization ของเทคนิค loop invariant จะคล้ายกับเทคนิคการอุปนัยเชิงคณิตศาสตร์ (mathematical induction) ที่จะต้องมีการพิสูจน์ base case และ ขั้นตอน maintenance จะคล้ายกับการพิสูจน์ inductive step

ตัวอย่างที่ 2 จงวิเคราะห์ว่าอัลกอริทึมในการหาผลรวมของตัวเลขทั้งหมดจำนวน n ตัวใดๆในอาเรย์ A นั้นว่าถูกต้องหรือไม่ เมื่อโจทย์กำหนดอัลกอริทึมให้ดังนี้

```
sum = 0
for  i=1 to length[A]
        sum = sum + A[i]
```

วิธีทำ

ขั้นตอนที่ 1 กำหนด loop invariant ให้เป็น คำสั่งที่ว่า

(S1) **ก่อนจะรันลูป i ใดๆ ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i-1**

A loop invariant is
before running loop i , $sum = \sum_{m=1}^{i-1} A[m]$

ขั้นตอนที่ 2 initialization ตรวจสอบว่าก่อนจะรันลูปแรก ประโยค (S1) loop invariant นั้นเป็นจริง

ซึ่งประโยค (S1) บอกว่า

ก่อนจะรันลูป i ใดๆ ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i-1

ก็แทนค่าประโยค (S1) ได้ว่า

(S2) **ก่อนจะรันลูป 1 ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง 1-1**

ซึ่ง อาเรย์ตัวที่ 1 ถึง 0 ไม่มี ดังนั้นก็ถือว่า ไม่ต้องหาผลรวมในอาเรย์ A

ทีนี้เราก็มาพิสูจน์ประโยค (S2) กันว่าเป็นจริงหรือไม่

เราก็ไปดูโค้ด บรรทัดก่อนเข้าลูป (ซึ่งมีแค่บรรทัดแรก) พบว่า sum=0 ก็ทำให้ประโยค (S2) เป็นจริง

ขั้นตอนที่ 3 maintenance ตรวจสอบว่า **ถ้า**ประโยค (S1) นั้นเป็นจริง ก่อนที่จะรันลูปรอบที่ i **แล้ว** ประโยค (S1) นั้น จะต้องเป็นจริง ก่อนที่จะรันลูปรอบที่ i+1

ซึ่งประโยค (S1) บอกว่า

ก่อนจะรันลูป i ใดๆ ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i-1

ก็แทนค่าประโยค (S1) เพื่อทำการพิสูจน์ได้ว่า

(S3) **ถ้า ก่อนจะรันลูป i ค่า sum เท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i -1**
แล้ว ก่อนจะรันลูป i+1 ค่า sum ต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i

ทีนี้เราก็มาทำการพิสูจน์ประโยค (S3) กันว่าเป็นจริงหรือไม่

เราก็ไปดูโค้ดในลูป (บรรทัดที่ 2-3)

จะเห็นว่า หลังรันลูป ค่าผลรวมตัวเลข จะถูกเพิ่มค่า ด้วยตัวเลขตัวที่ i ในอาเรย์ A

ดังนั้น **ถ้า**ก่อนรันลูป i ค่า sum เท่ากับผลรวมเลขในอาเรย์ A ตัวที่ 1 ถึง i - 1

(sum = A[1]+A[2]+...+A[i-1])

แล้วหลังรันลูป i ค่า sum จะต้องเท่ากับ ผลรวมเลขในอาเรย์ A ตัวที่ 1 ถึง i - 1 บวกกับ ตัวที่ i

(sum = A[1]+A[2]+...+A[i-1]+A[i])

จากโค้ดจะเห็นว่า หลังรันลูป i ก็คือ ก่อนรันลูป i+1

ดังนั้นจะเห็นว่าเราพิสูจน์แล้วว่า ประโยค (S3) เป็นจริง

ขั้นตอนที่ 4 termination ตรวจสอบว่า เมื่อหยุดการทำซ้ำแล้ว ประโยค (S1) ช่วยให้เห็นว่าอัลกอริทึมนั้นถูกต้อง

ซึ่งประโยค (S1) บอกว่า

ก่อนจะรันลูป i ใดๆ ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i-1

ก็แทนค่าประโยค (S1) เพื่อทำการพิสูจน์ได้ว่า

(S4) **ก่อนจะรันลูป n+1 ค่า sum จะต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง n เมื่อ n คือขนาดของอาเรย์**

ซึ่ง ประโยคนี้ สามารถใช้การพิสูจน์ของขั้นตอนที่ 3 ได้เลย เพราะ ในประโยค (S3) เราแสดงให้เห็นว่า

ถ้า ก่อนจะรันลูป i ค่า sum เท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i -1

แล้ว ก่อนจะรันลูป i+1 ค่า sum ต้องเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง i

ดังนั้น เพียงแค่แทน i ด้วย n เมื่อ n คือขนาดของอาเรย์ จะได้ว่า

ก่อนจะรันลูปรอบที่ n+1 ค่า sum จะเท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง n

ก็จะเห็นว่าเราพิสูจน์แล้วว่า (S4) เป็นจริง

ซึ่ง ตัวเลขมี n ตัวก็แปลว่ามีลูป n รอบ ก่อนจะรันลูปรอบที่ n+1 ก็คือหลังการรันลูปรอบที่ n นั่นเอง

แปลว่าเรารันจนครบทุกลูปแล้ว ทำให้เห็นว่า เมื่อเรารันครบทุกลูปแล้ว ค่า sum เท่ากับผลรวมของตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง n ซึ่งก็**คือ เอ๊าพุตของโจทย์ที่เราต้องการ**

ลำดับการพิสูจน์ความถูกต้องของอัลกอริทึมด้วยเทคนิค loop invariants ของตัวอย่างที่ 2 แสดงดัง**ภาพที่ 4**

A loop invariant is

before running at loop i , $sum = \sum_{m=1}^{i-1} A[m]$

Initialization: at loop 1, sum = 0 (True!!)

Maintenance:

If at before running loop i , sum = A[1]+A[2]+…+A[i-1]

then after running loop i , sum = A[1]+A[2]+…+A[i-1]+A[i]

Hence, before running loop i+1 , sum = A[1]+A[2]+…+A[i-1]+**A[i]** (True!!)

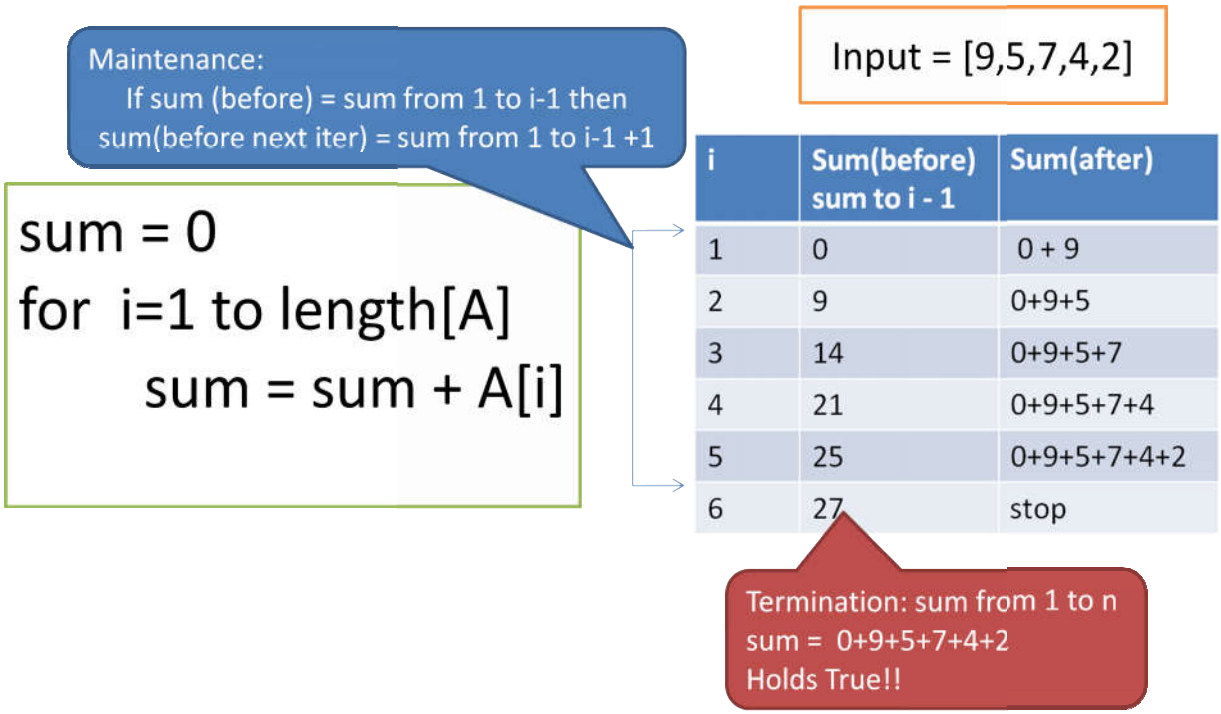
Termination:

Goal(output of program) => $sum = \sum_{i=1}^{n} A[i]$

At start of running at loop n+1, sum = A[1]+A[2]+…+A[n-1]+**A[n]** (True!!)

ภาพที่ 4

เพื่อความเข้าใจเพิ่มเติม การแสดงตัวอย่างการแปลี่ยนแปลงของตัวแปร ในอัลกอริทึมหาค่าผลรวม นั้นแสดงดังต่อไปนี้



แบบฝึกหัด

1. ทำการวิเคราะห์ว่าอัลกอริทึมที่ให้ต่อไปนี้ถูกต้องหรือไม่ หากต้องการแก้ปัญหาโจทย์เพื่อทำการคำนวณการแปลงค่าอินพุตจากหน่วยองศาเป็นหน่วยเรเดียน

a)

```
import math
a = float(input("Enter an angle in degrees: "))
r = a*(22/7)/180
print("%f degrees = %.2f radians and sin(%f) = %.2f and cos(%f) = %.2f" %
      (a,math.pi(a),math.sin(r),math.cos(r)))
```

b)

```
number = math.sin(input('Enter an angle in degrees:'))
Ra = (number*(math.pi))/180
math.sin = number
```

c)

```
degree = int(input("Enter an angle in degrees: "))
import math
radian = (degree*math.pi)/180
sin = math.sin(radian)
cosine = math.cos(radian)
print("%d degrees = %.2f radians and sin(%d) = %.2f and cos(%d) = %.2f"%
      (degree,radian,degree,sin,degree,cosine))
```

d)

```
import math
pi = 3.14
angle = int(input("Enter an angle in degrees: "))
radian = angle*(pi)/180
print("%d degrees = %.2f radian" % angle, radian)
```

2. ทำการวิเคราะห์ความถูกต้องของอัลกอริทึมที่ทำการหาค่าตัวเลขที่มากที่สุดในอาเรย์ หากกำหนดอัลกอริทึมให้ดังต่อไปนี้

```
max = A[1]
for  i=2 to length[A]
        if max < A[i]
                max = A[i]
```

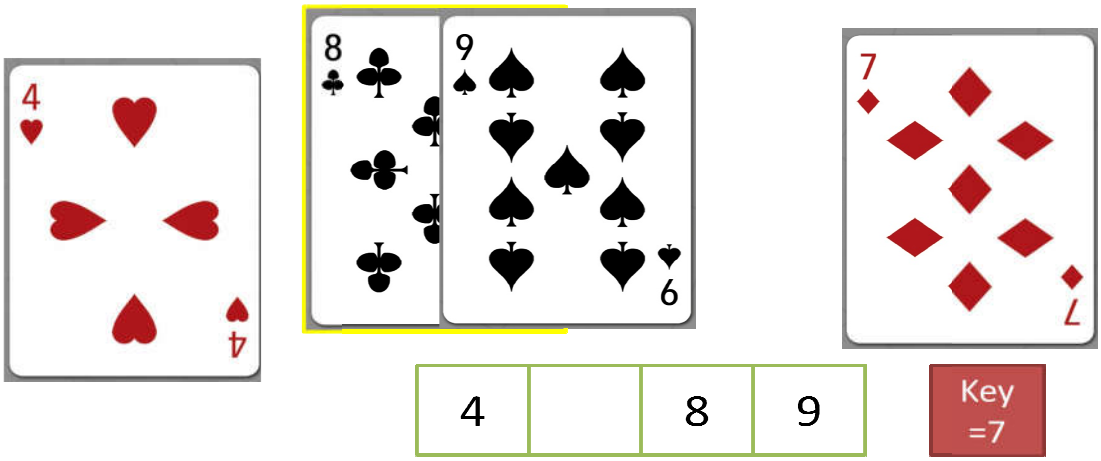
บทที่ ๒ การเรียงลำดับข้อมูลแบบแทรก (insertion sort)

อัลกอริทึมการเรียงลำดับข้อมูลแบบแทรกนั้น มีแนวคิดคล้ายๆกับการเรียงไพ่ แสดงดังรูป**ภาพที่5**

ภาพที่ 5

จะเห็นว่า ไพ่ถูกแบ่งเป็น ไพ่ที่เรียงแล้วในมือ กับไพ่ใบใหม่ ที่เรามีหน้าที่จะต้องหาตำแหน่งแทรกให้ไพ่ใบใหม่

ในการเรียงลำดับข้อมูลแบบแทรกก็ใช้หลักการเดียวกัน คือแบ่งข้อมูล เป็นข้อมูลที่เรียงลำดับแล้ว กับข้อมูลใหม่ แล้วนำข้อมูลใหม่ เปรียบเทียบกับข้อมูลที่เรียงลำดับแล้วทีละตัว หากข้อมูล(ที่เรียงลำดับแล้ว) ตัวที่เปรียบเทียบอยู่มีค่ามากกว่า ข้อมูลใหม่ เราก็จะเลือกข้อมูลตัวถัดไปมาเปรียบเทียบแทน ก็จะคล้ายกับการขยับไพ่ในมือเพื่อหาช่องว่างให้ไพ่ใบใหม่ดังรูป**ภาพที่ 6**



ภาพที่ 6

ซึ่งอัลกอริทึมนี้สามารถเขียนเป็น pseudo code ได้ดังนี้

```
for j=2 to length[A]
        do key = A[ j ]
        insert A[ j ] into the sorted sequence A[1 … j-1]
        i = j - 1
        while i > 0 and A[ i ] > key
                do A[i+1] = A[ i ]
                i = i - 1
        A[i+1]=key
```

<u>ตัวอย่างที่ 3</u> จงเรียงข้อมูลด้วย insertion sort เมื่อกำหนดให้อาเรย์มีข้อมูลเริ่มต้นดังนี้ 5, 2, 4, 6, 1, 3

วิธีทำ

1. เริ่มต้นครั้งแรกกำหนดให้ key คือค่าในอาเรย์ตำแหน่งที่ 2 เสมอ

2. ให้ i = 1 นำค่าในอาเรย์ตำแหน่งที่ i มาเปรียบเทียบกับ key

3. ถ้าหาก ค่าในอาเรย์ตำแหน่ง i มีค่ามากกว่า key ให้ทำการคัดลอกค่าที่ตำแหน่ง i ไปไว้ที่ตำแหน่งถัดไป (i+1)

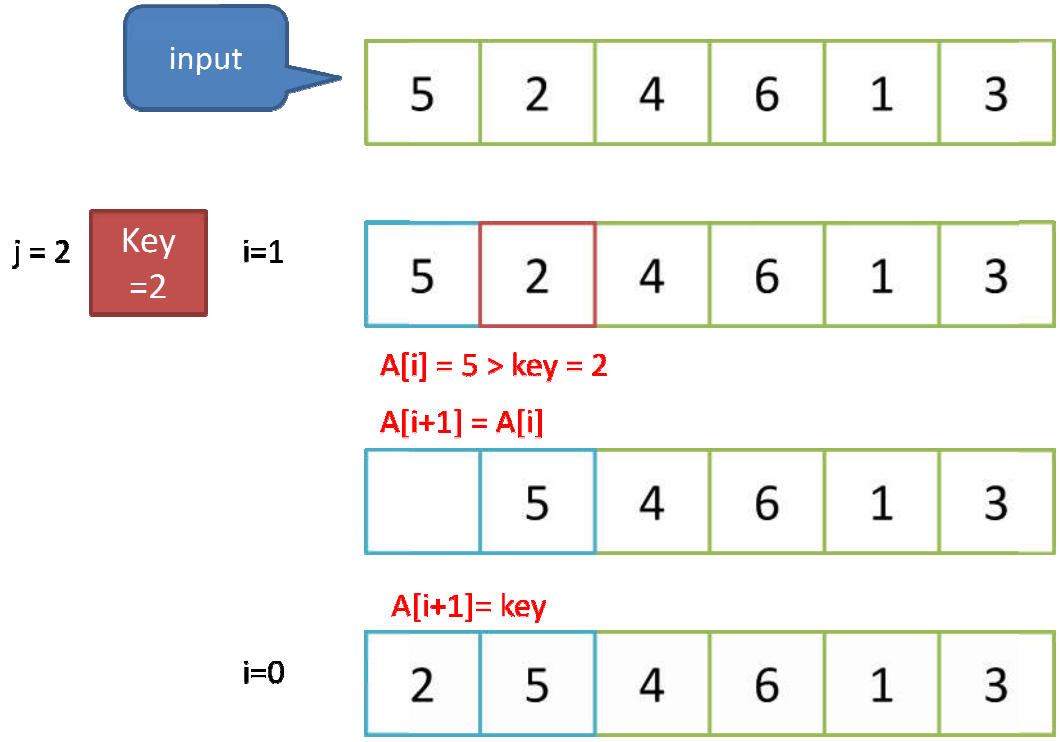
4. แล้วทำการเลื่อนตำแหน่ง i ไปทางซ้าย (ลดค่า i ลง 1 ค่า)

5. ทำซ้ำ ข้อ 3 และข้อ 4 จนกระทั่ง i=0 (เปรียบเทียบครบทุกตัวแล้ว) หรือ

พบว่า ค่าในอาเรย์ตำแหน่ง i มีค่าน้อยกว่า key

ให้อาเรย์ตำแหน่งที่ i+1 มีค่าเท่ากับ key (เป็นการแทรกข้อมูล)

ตัวอย่างลำดับขั้นตอนแสดงดัง**ภาพที่ 7**



j = 3 Key =4 i=2

A[i] = 5 > key = 4 ?

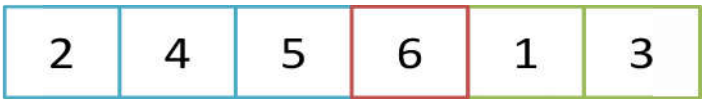
A[i+1] = A[i]

i=1

A[i] = 2 > key = 4 ? (False)

A[i+1]= key

j = 4 Key =6 i=3



A[i] = 5 > key = 6 ? (False)

A[i+1] = A[i]

j = 5 Key =1 i=4

A[i] = 6 > key = 1

A[i+1] = A[i]

i=3

A[i] = 5 > key = 1

A[i+1] = A[i]

i=2

A[i] = 4 > key = 1

A[i+1] = A[i]

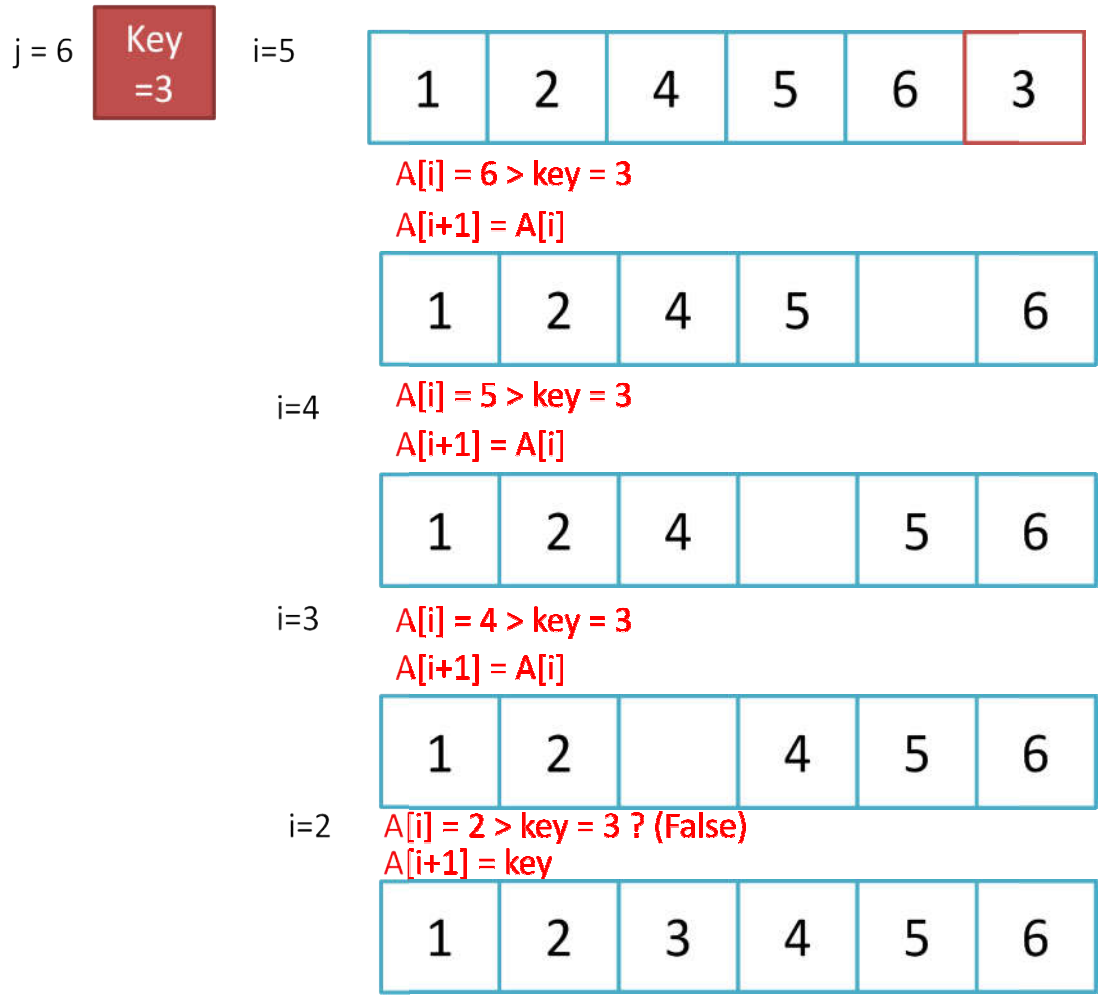
j = 5 Key =1 i=1

A[i] = 2 > key = 1

A[i+1] = A[i]

i=0

A[i+1] = key



ภาพที่ 7

<u>แบบฝึกหัด</u>

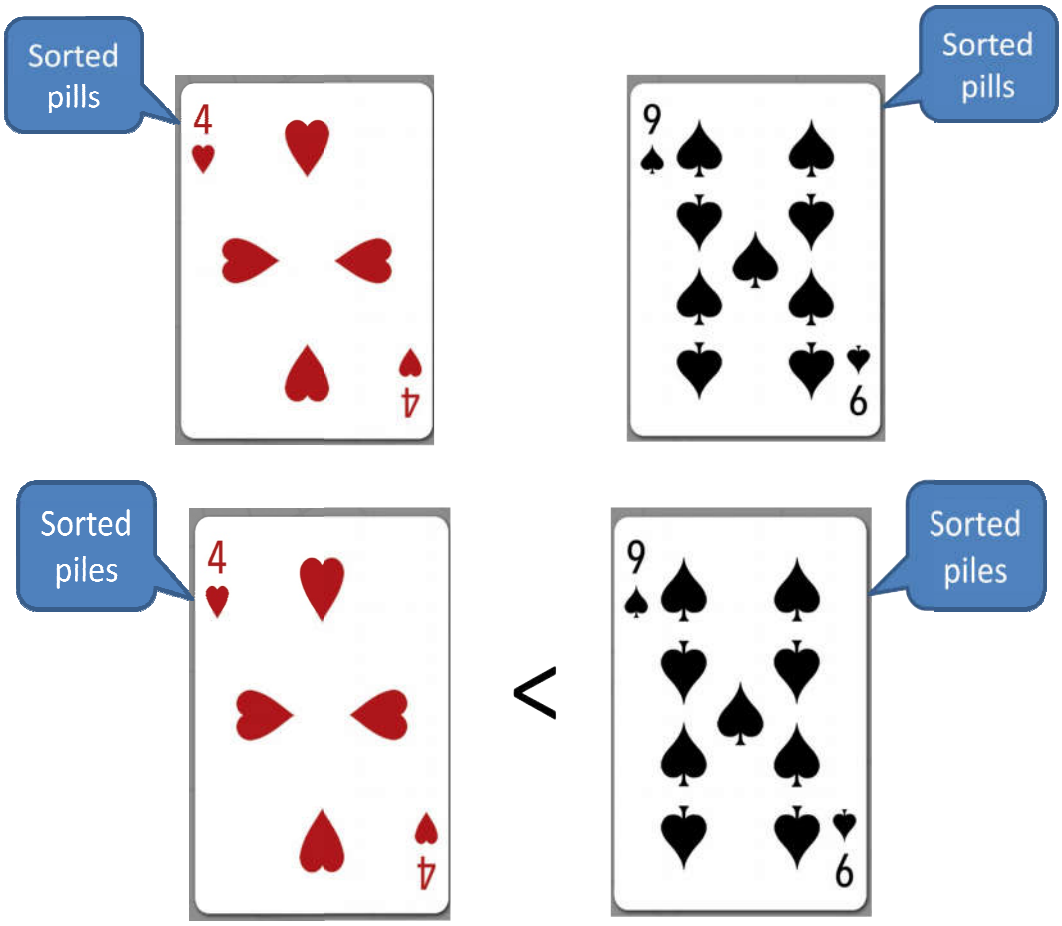
1. จงแสดงวิธีการเรียงลำดับข้อมูลด้วยวิธี insertion sort เมื่อกำหนดข้อมูลให้คือ 9, 5, 7, 4, 2 เรียงลำดับจากน้อยไปมาก

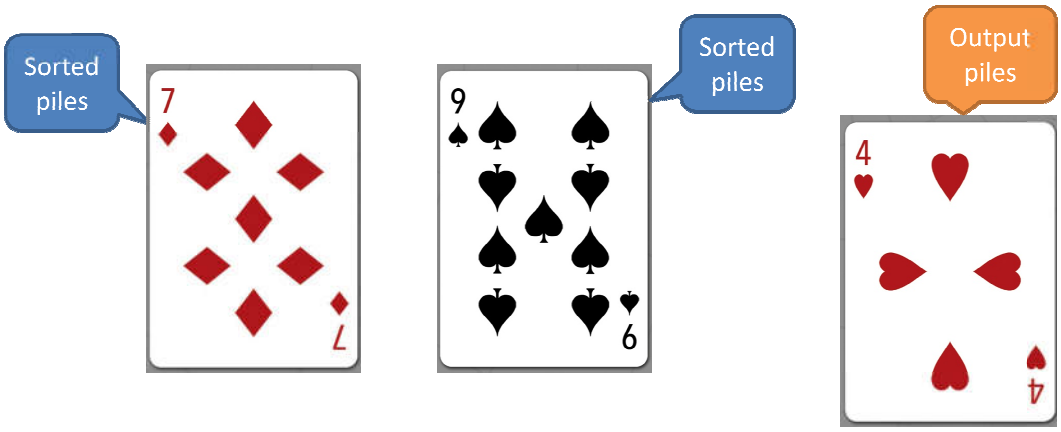
2. จงแสดงวิธีการเรียงลำดับข้อมูลด้วยวิธี insertion sort เมื่อกำหนดข้อมูลให้คือ 2, 1, 0, 1, 2, 5, 6, 2 เรียงลำดับจากน้อยไปมาก

บทที่ ๓ การเรียงลำดับข้อมูลแบบผสาน (merge sort)

หลักการของการเรียงลำดับข้อมูลแบบ merge sort นี้ใช้หลักการเรื่องการแบ่งแยกและเอาชนะ (divide and conquer) เข้ามาช่วยในการเรียงลำดับข้อมูล โดยที่หลักการคือ จะแบ่งกลุ่มข้อมูลออกเป็นข้อมูลที่เรียงลำดับเรียบร้อยแล้ว จำนวน 2 กลุ่ม แล้วนำข้อมูลจากทั้งสองกลุ่มมาเปรียบเทียบกันทีละคู่ ถ้าพบว่าข้อมูลตัวไหนน้อยกว่า ก็จะนำเอาไปเก็บไว้ในชุดข้อมูลที่เป็นเอ้าพุตของอัลกอริทึม

ตัวอย่างของแนวคิดของการเรียงลำดับข้อมูลแบบผสาน โดยเปรียบเทียบตัวอย่างกับไพ่ แสดงให้เห็นดัง **ภาพที่ 8**





ภาพที่ 8

อัลกอริทึมการเรียงลำดับข้อมูลแบบผสาน มีขั้นตอนหลักๆ 3 ขั้นตอน สรุปได้ดังนี้

1. ทำการแบ่งอาเรย์ที่มีขนาด n ตัวออกเป็นอาเรย์ 2 อันที่มีขนาด n/2

2. ทำการเรียงอาเรย์ย่อย(ที่แบ่งออกมา) ทั้ง 2 อาเรย์ โดยใช้วิธีการ merge sort

3. ทำการผสานอาเรย์ทั้ง 2 อาเรย์ที่เรียงเรียบร้อยแล้ว เพื่อสร้างเป็นอาเรย์คำตอบ

ซึ่งอัลกอริทึมการเรียงข้อมูลแบบผสานนั้นสามารถเขียนเป็น pseudo code ได้ดังนี้

Psudocode: merge-sort (A,p,r)

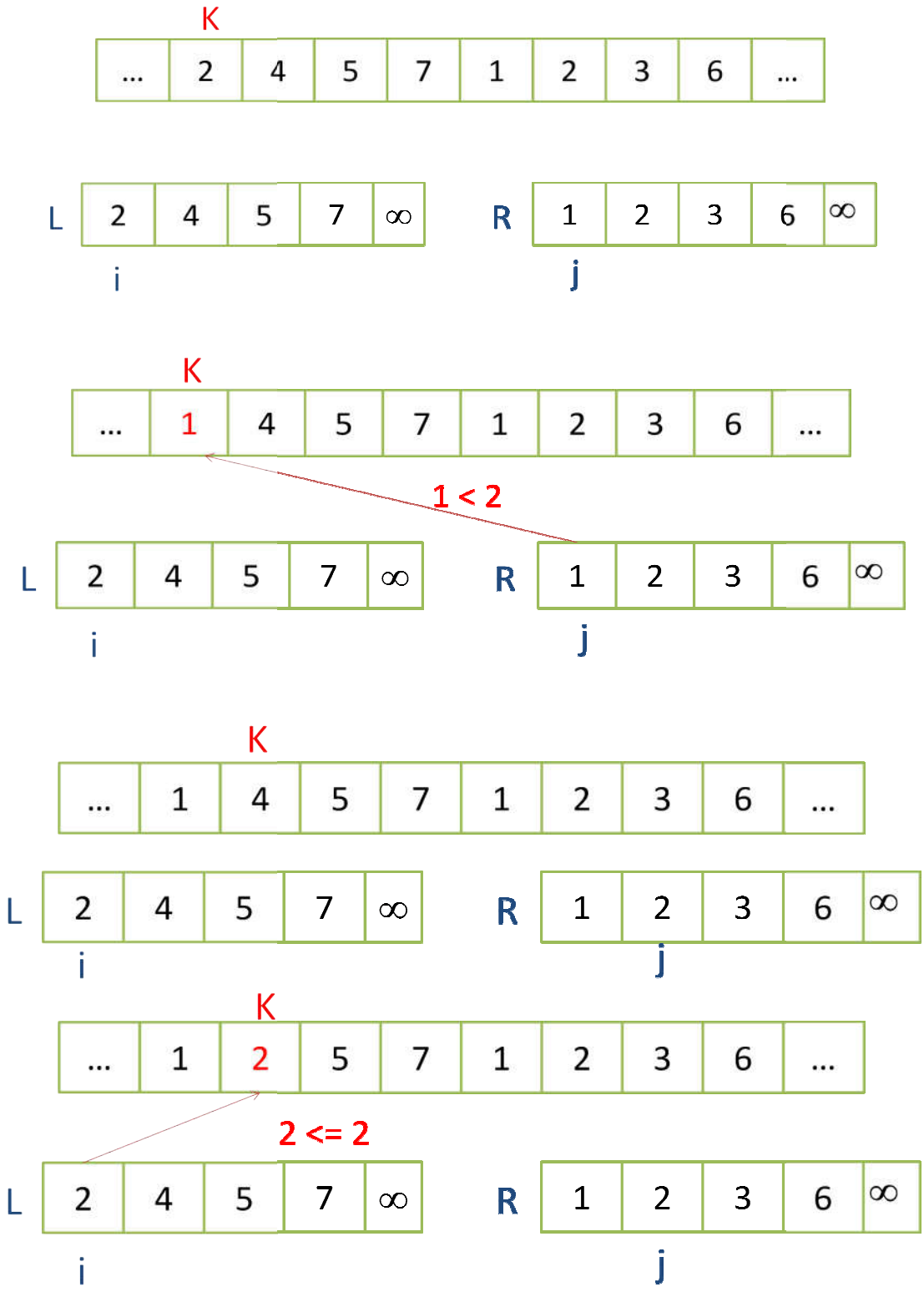
```
If p < r
        then q = L(p+r)/2
                merge-sort(A, p, q)
                merge-sort(A, q+1, r)
                merge(A, p, q, r)
```

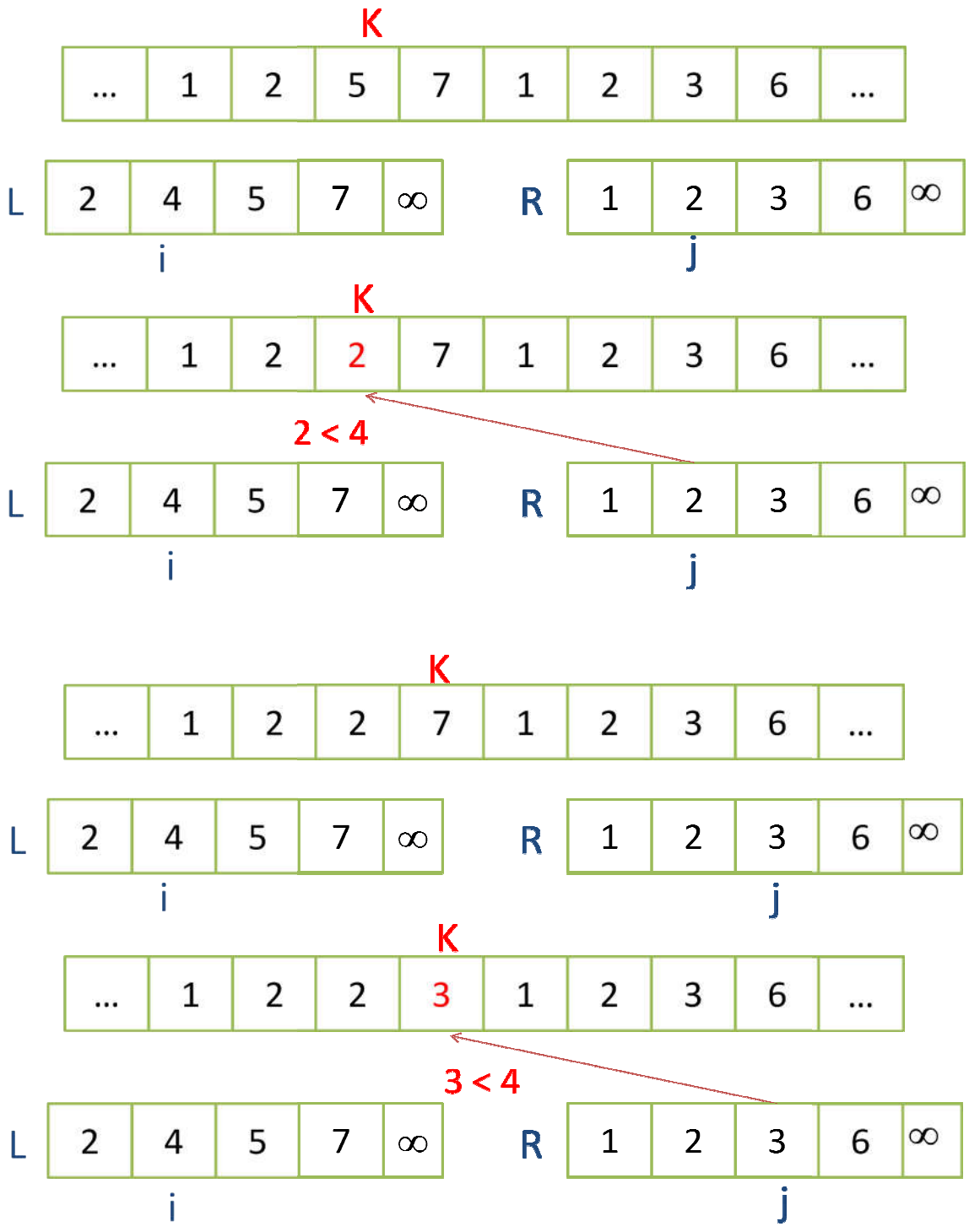
ซึ่งอัลกอริทึมฟังก์ชัน merge สามารถเขียนเป็น pseudo code ได้ดังนี้

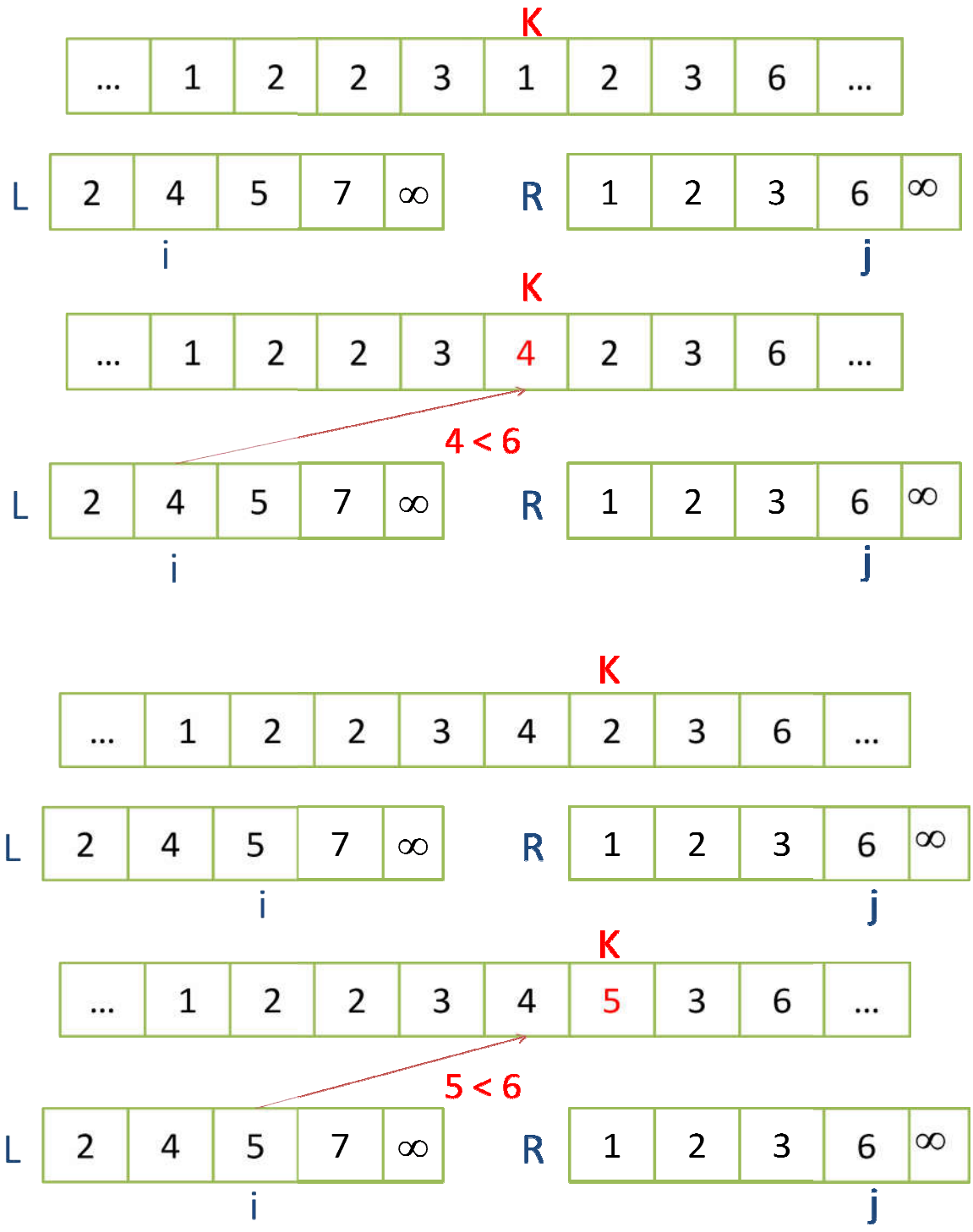
Psudocode: merge(A,p,q,r)

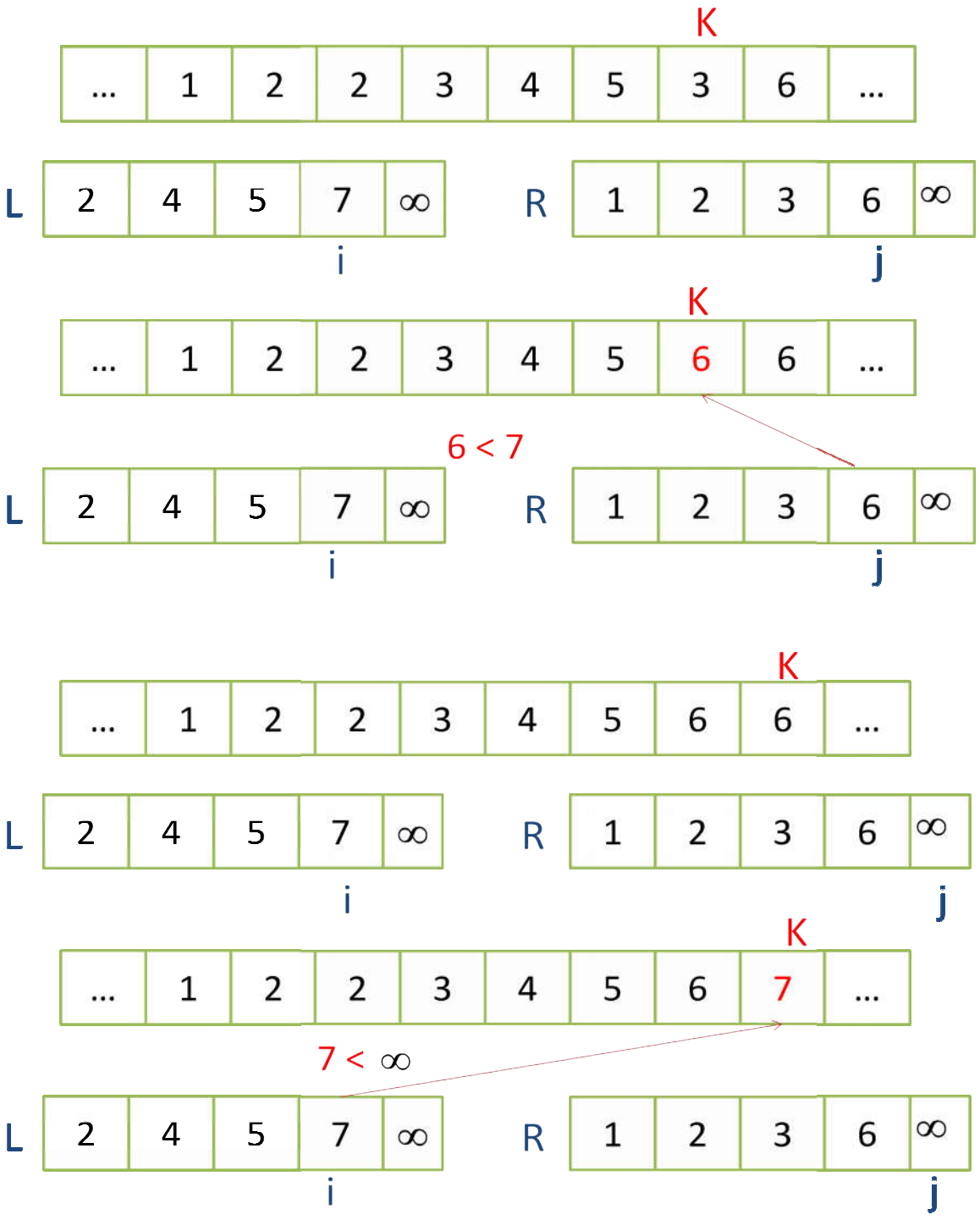
```
N1 = q – p + 1
N2 = r – q
Create arrays L[1..N1+1] and R[1...N2+1]
For i = 1 to N1
        do L[i]  = A[ p + i -1]
For j = 1 to N2
        do R[j] = A[q + j]
L[N1+1] =
R[N2+1] =
i = 1
j = 1
For k=p to r
        do if L[ i ] <= R[ j ]
                then A[k] = L[ i ]
                     i = i+1
                else A[k] = R[ j ]
                     j = j+1
```

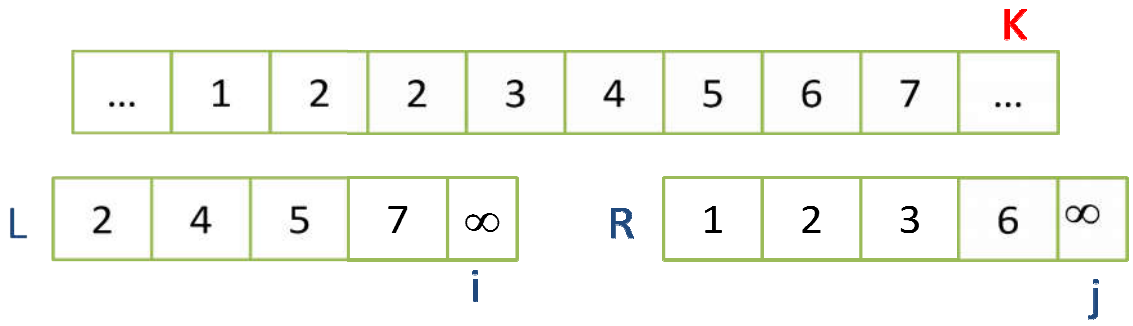
<u>ภาพที่ 9</u> แสดงตัวอย่างการทำงานของอัลกอริทึมฟังก์ชัน merge โดยสมมติให้มีอาเรย์ที่ยังไม่ได้เรียงข้อมูล เป็นอาเรย์ขนาด 8 ตัว ดังนั้นจะเริ่มจากการแบ่งอาเรย์ออกเป็น 2 อาเรย์ที่มีขนาดอันละ 4 ตัว แล้วนำอาเรย์ย่อย (ที่มีขนาด 4 ตัว) นำไปเรียงลำดับข้อมูล โดยใช้อัลกอริทึมฟังก์ชัน merge


K
... 2 4 5 7 1 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 4 5 7 1 2 3 6 ...
1 < 2
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 4 5 7 1 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 5 7 1 2 3 6 ...
2 <= 2
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j


K
... 1 2 5 7 1 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 7 1 2 3 6 ...
2 < 4
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 7 1 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 1 2 3 6 ...
3 < 4
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j


K
... 1 2 2 3 1 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 2 3 6 ...
4 < 6
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 2 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 5 3 6 ...
5 < 6
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j


K
... 1 2 2 3 4 5 3 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 5 6 6 ...
6 < 7
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 5 6 6 ...
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j
K
... 1 2 2 3 4 5 6 7 ...
7 < ∞
L 2 4 5 7 ∞
R 1 2 3 6 ∞
i
j



ภาพที่ 9

ตัวอย่างที่ 4 กำหนดอาเรย์ขนาด 8 มีข้อมูล คือ 5, 2, 4, 7, 1, 3, 2, 6 จงแสดงการทำงานของอัลกอริทึม merge sort เพื่อใช้ในการเรียงลำดับข้อมูล

วิธีทำ

1. เริ่มต้นแบ่งอาเรย์จากขนาด 8 ออกเป็นอาเรย์ขนาด 4 ตัวจำนวน 2 กลุ่ม

2. นำอาเรย์ย่อยที่มีขนาด 4 มาแบ่งขนาดอีกครั้ง ได้เป็นอาเรย์ขนาด 2 ตัวจำนวน 2 กลุ่ม

เนื่องจากมีอาเรย์ย่อยขนาด 4 ตัวสองกลุ่ม ดังนั้นทั้งหมด เราจะได้อาเรย์ขนาด 2 ตัวจำนวน 4 กลุ่ม

3. นำอาเรย์ย่อยที่มีขนาด 2 ตัว มาแบ่งขนาดอีกครั้ง ได้เป็นอาเรย์ขนาด 1 ตัวจำนวน 2 กลุ่ม

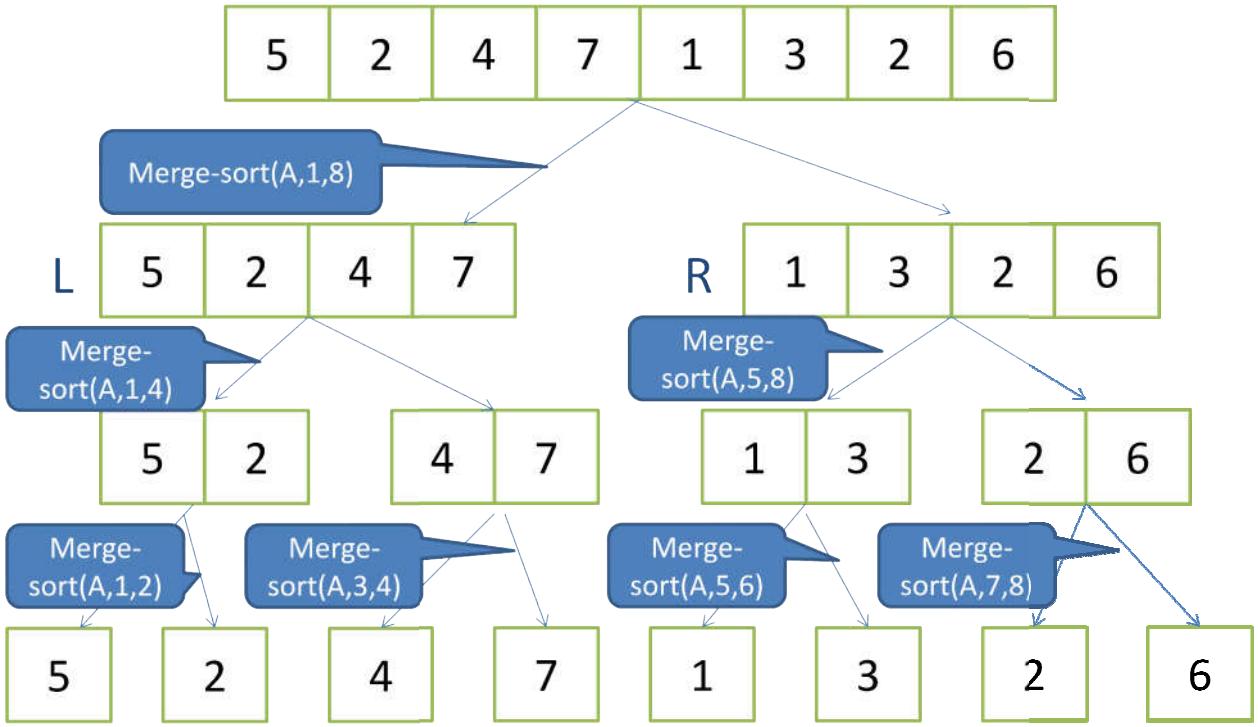
เนื่องจากมีอาเรย์ย่อยขนาด 2 ตัวสี่กลุ่ม ดังนั้นทั้งหมด เราจะได้อาเรย์ขนาด 1 ตัวจำนวน 8 กลุ่ม

4. ทำการเรียงข้อมูลและผสานอาเรย์ย่อย โดยใช้ฟังก์ชัน merge เริ่มจาก ผสาน อาเรย์ขนาด 1 ตัว ทีละสองกลุ่ม ดังนั้นผลลัพธ์ทั้งหมด จะได้อาเรย์ที่เรียงลำดับข้อมูลแล้ว เป็นอาเรย์ขนาด 2 ตัว จำนวน 4 กลุ่ม

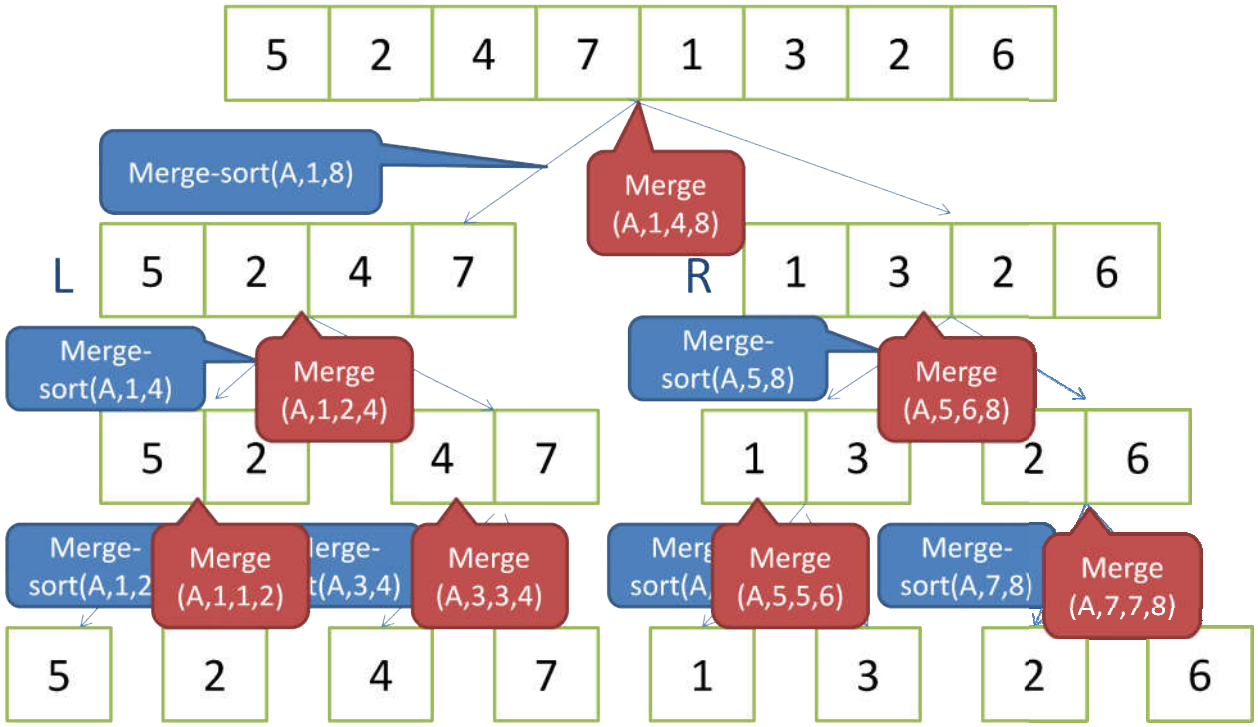
5. ทำการเรียงข้อมูลและผสานอาเรย์ย่อย โดยใช้ฟังก์ชัน merge ผสาน อาเรย์ขนาด 2 ตัว ทีละสองกลุ่ม ดังนั้น ผลลัพธ์ทั้งหมด จะได้อาเรย์ที่เรียงลำดับข้อมูลแล้ว เป็นอาเรย์ขนาด 4 ตัว จำนวน 2 กลุ่ม

6. ทำการเรียงข้อมูลและผสานอาเรย์ย่อย โดยใช้ฟังก์ชัน merge ผสาน อาเรย์ขนาด 4 ตัว สองกลุ่ม ดังนั้นผลลัพธ์ทั้งหมด จะได้อาเรย์ที่เรียงลำดับข้อมูลแล้ว เป็นอาเรย์ขนาด 8 ตัว เป็นคำตอบ

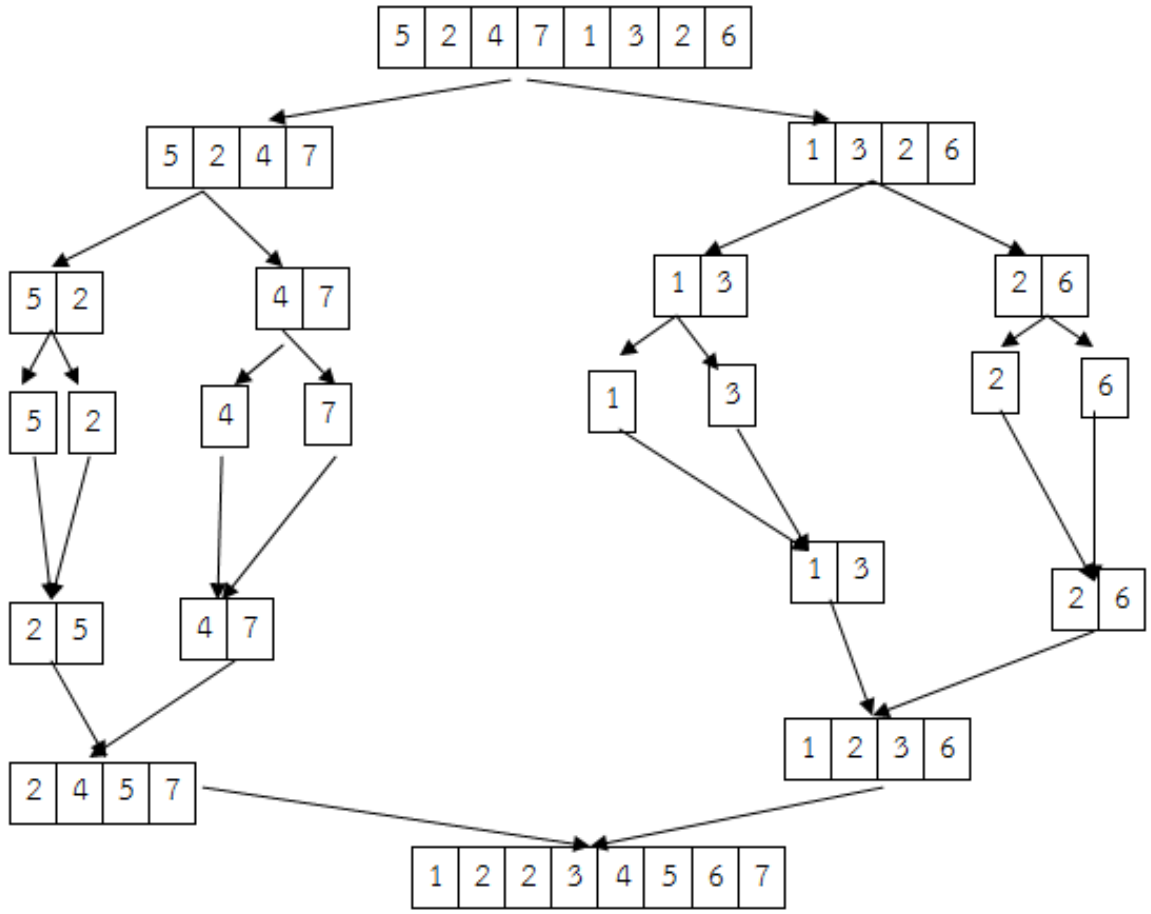
ภาพต่อไปนี้แสดงลำดับการเรียกใช้อัลกอริทึม merge-sort เพื่อแบ่งอาเรย์และเรียงลำดับข้อมูล



ภาพต่อไปนี้แสดงลำดับการเรียกใช้อัลกอริทึมฟังก์ชัน merge เพื่อเรียงลำดับข้อมูลและผสานอาเรย์



ดังนั้นโดยสรุปจะได้แผนภาพแสดงขั้นตอนการเรียงลำดับข้อมูลแบบผสาน ดังนี้



<u>แบบฝึกหัด</u>

1. จงแสดงวิธีการเรียงลำดับข้อมูลในอาเรย์ขนาด 6 ตัว ที่มีข้อมูลคือ 9, 5, 7, 4, 2, 8 เรียงลำดับจากน้อยไปมาก

2. จงแสดงวิธีการเรียงลำดับข้อมูลในอาเรย์ขนาด 8 ตัว ที่มีข้อมูลคือรหัสนิสิต เรียงลำดับจากน้อยไปมาก

บทที่ ๔ วิเคราะห์ประสิทธิภาพของอัลกอริทึมการเรียงลำดับข้อมูลแบบแทรก

ดังที่อธิบายไว้ในบทที่ 1 เกี่ยวกับการวิเคราะห์ประสิทธิภาพของอัลกอริทึม ซึ่งจะมีการวิเคราะห์สองด้านด้วยกันคือ ความถูกต้องของอัลกอริทึม และเวลาที่ใช้ในการประมวลผลของอัลกอริทึม

ขั้นแรกเราจะทำการวิเคราะห์ความถูกต้องของอัลกอริทึมการเรียงลำดับข้อมูลแบบแทรก (insertion sort) โดยใช้เทคนิค loop invariants

จากอัลกอริทึมของ insertion sort คือ

```
for j=2 to length[A]
        do key = A[ j ]
        insert A[ j ] into the sorted sequence A[1 ... j-1]
        i = j - 1
        while i > 0 and A[ i ] > key
                do A[i+1] = A[ i ]
                i = i - 1
        A[i+1]=key
```

จะสามารถวิเคราะห์ความถูกต้องของอัลกอริทึมด้วยเทคนิค loop invariants ได้ตามขั้นตอน ดังนี้

1. ทำการกำหนดประโยค loop invariant ได้คือ

ก่อนที่จะรันลูปรอบที่ j ใดๆ อาเรย์ A ตัวที่ 1 ถึง j-1 เป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว

2. ขั้นตอน initialization

พิสูจน์ว่า **ก่อนที่จะรันลูปรอบที่ j=2 ตัวเลขในอาเรย์ A ตั้งแต่ตัวที่ 1 ถึง 2-1 เป็นเลขที่เรียงลำดับเรียบร้อยแล้ว**

เนื่องจาก A[1..1] คือ A[1] มีเลขตัวเดียวถือว่าเรียงลำดับเรียบร้อย ข้อพิสูจน์เป็นจริง

3. ขั้นตอน maintenance

พิสูจน์ว่า **ถ้าก่อนที่จะรันลูปรอบที่ j ใดๆ อาเรย์ A ตัวที่ 1 ถึง j-1 เป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว ก่อนที่จะรันลูปรอบที่ j+1 อาเรย์ A ตัวที่ 1 ถึง j เป็นตัวเลขที่เรียงลำดับกันเรียบร้อย**

เมื่อตรวจสอบที่อัลกอริทึม หลังรันรอบที่ j จะพบว่า

ถ้าหาก A[i] > A[j] เมื่อ i < j แล้วจะทำการแทรก A[j] ไว้ที่ตำแหน่งก่อน i ก็จะทำให้อาเรย์เรียงลำดับดังนี้ A[1...i-1] < A[j] < A[i] ซึ่งเราตั้งสมมติฐานแล้วว่า A[1...i-1] เรียงลำดับกันเรียบร้อย ดังนั้นจะเห็นว่า A[1...i] เรียงลำดับกันเรียบร้อย

สำหรับการแทรกอาเรย์ A[j] เมื่อไหร่ก็ตามที่เราพบว่า A[i] > A[j] ซึ่ง i < j ด้วยอัลกอริทึมบรรทัดที่ A[i+1] = A[j] ก็จะทำให้แทรกได้ แต่ถ้าหาก A[i] <= A[j] ซึ่ง i < j ด้วยอัลกอริทึมบรรทัดที่ A[i+1] = A[j] ก็จะทำให้อาเรย์เรียงลำดับดังนี้ A[1...i-1] < A[i] < A[j]

ดังนั้นจะเห็นว่าพิสูจน์ได้ว่า ก่อนจะรันลูปรอบที่ j+1 อาเรย์ตัวที่ 1 ถึง j เป็นตัวเลขที่เรียงลำดับกันเรียบร้อย

4. ขั้นตอน termination

พิสูจน์ว่า หลังจากจบลูป อาเรย์ทั้งหมดเรียงลำดับกันเรียบร้อย

ดังนั้นเราสามารถใช้บทพิสูจน์ได้ว่า

ก่อนที่จะรันลูปรอบที่ n+1 ใดๆ อาเรย์ A ตัวที่ 1 ถึง n เป็นตัวเลขที่เรียงลำดับกันเรียบร้อย

อ้างอิงจากขั้นตอน maintenance ก็จะพบว่า เราได้พิสูจน์ไว้แล้วว่า ข้อความข้างต้นเป็นจริง

จบการพิสูจน์ ดังนั้น อัลกอริทึมนี้ถูกต้อง

สรุปขั้นตอนทั้งหมดได้ดังภาพต่อไปนี้

loop invariant = before running loop j, all elements in A[1 … j – 1] are in sorted order.

Initialization:
Before running loop 2 , all elements in = A[1…1] are sorted. (True!!)

Maintenance:
If before running loop j , all elements in = A[1… j-1] are sorted.
then after running loop j, if A[i] > A[j] for i < j then A[j] will be inserted before position i and A[1…i-1] are sorted where A[1…i-1] < A[j] < A[i] .
when we found A[i] > A[j] for i< j then A[i+1] = A[j] where A[1…i] are sorted. Hence A[1…j] are sorted.
Hence before running loop j+1, A[1…j] are sorted. (True!!)

Termination: at starting of loop n+1, A[1…n] are sorted (True!!)

ต่อไปเราจะทำ**การวิเคราะห์เวลาที่ใช้ในการประมวลผล (running time) ของอัลกอริทึม insertion sort** ซึ่งมีขั้นตอนดังนี้

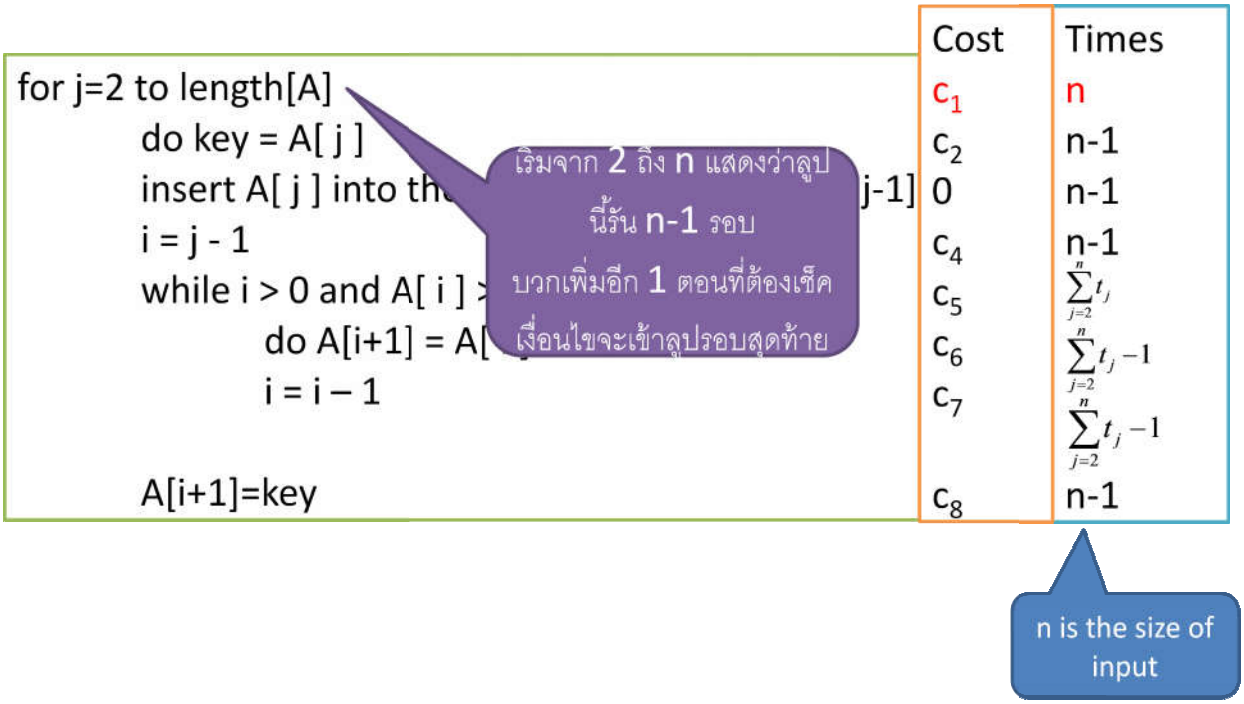
1. ทำการกำหนดชื่อให้แก่ pseudo code แต่ละบรรทัดของอัลกอริทึม (เอาไว้ใช้กำหนดเป็นชื่อตัวแปรของค่าคงที่ในการคำนวณ)

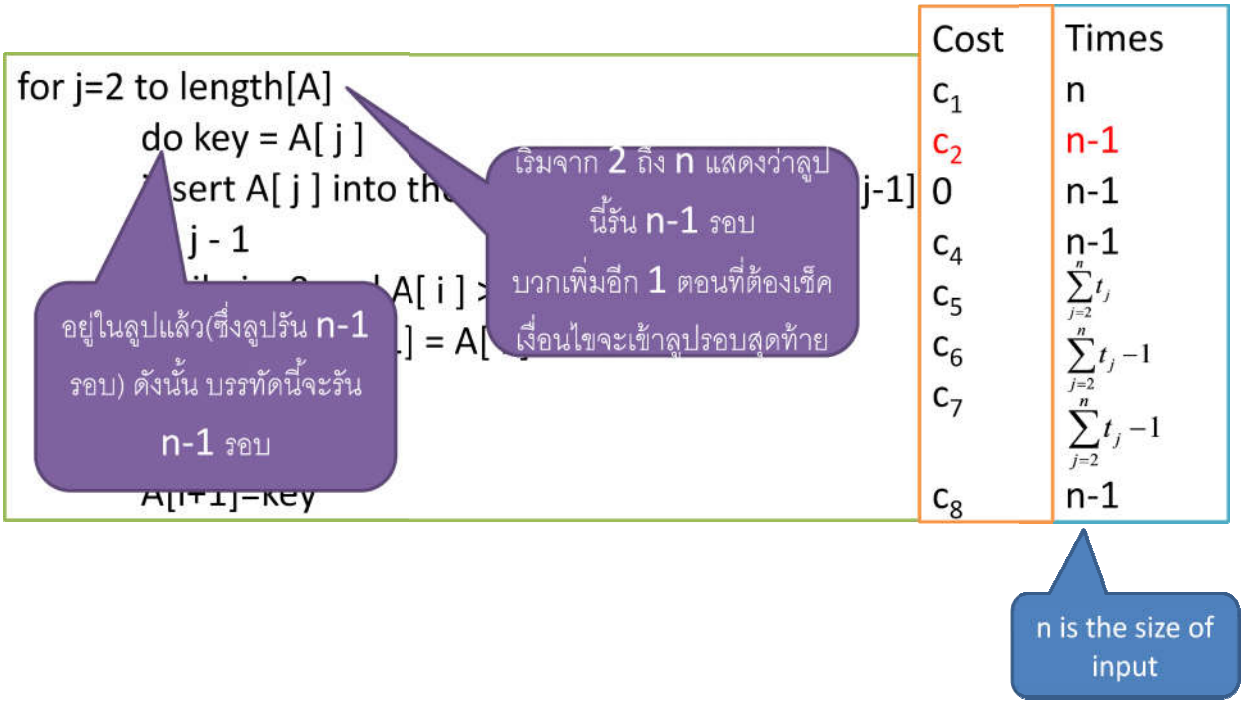
2. ทำการคำนวณเวลาที่ใช้ในการประมวลผลของโค้ดแต่ละบรรทัด โดยเทคนิคการเติบโตของฟังก์ชัน (growth of function) - หาก เป็นเพียงโค้ดที่ไม่ได้อยู่ในลูป ก็ใช้ growth of function คือ ค่าคงที่

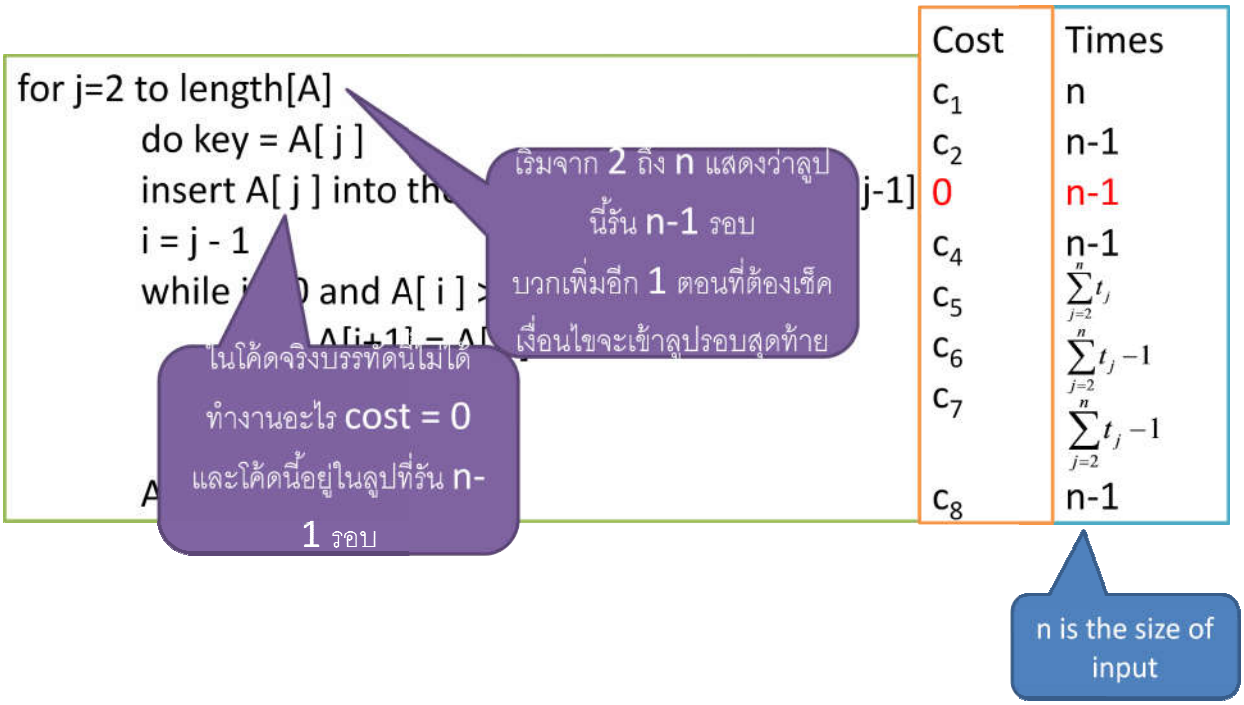
- หากเป็นโค้ดที่อยู่ในลูป ก็ใช้ growth of function ที่เท่ากับขนาดของอินพุต (นับว่าอย่างมากสุดต้องวนลูปกี่รอบ)

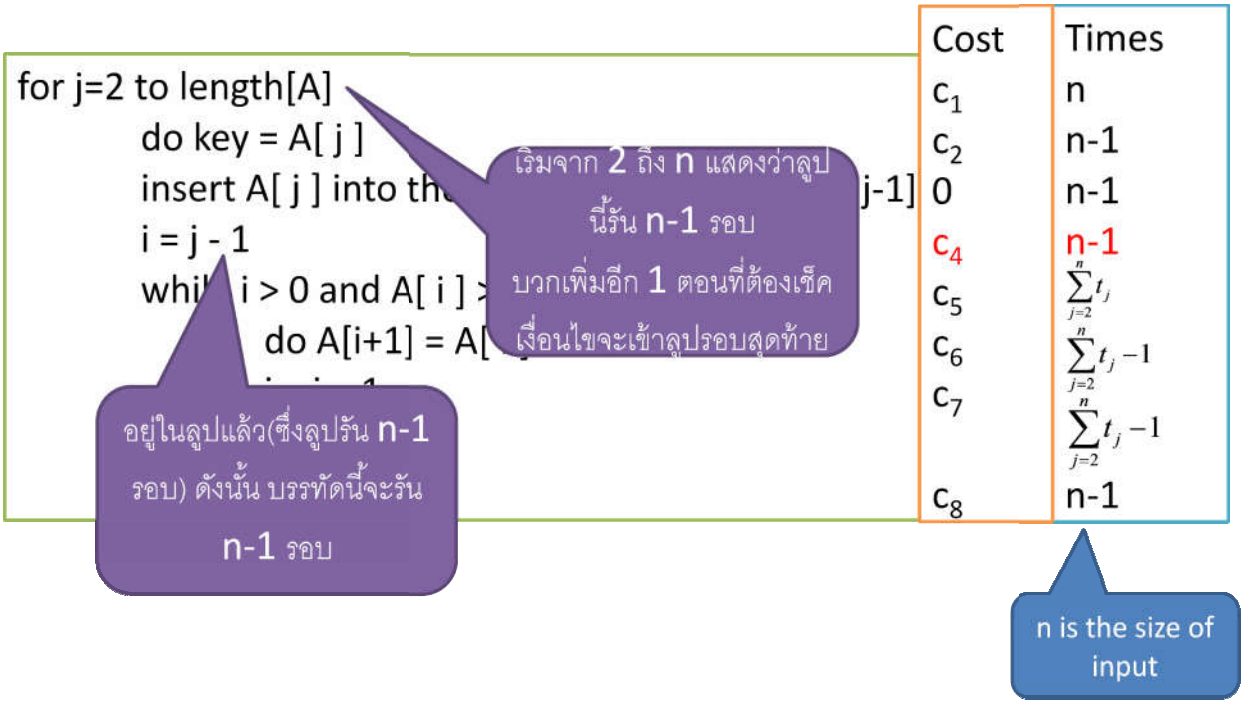
3. วิเคราะห์หาขอบเขตของเวลาที่ต้องการใช้ประมวลผลมากที่สุด โดยนำค่าคงที่ในข้อ 1 คูณกับเวลาในข้อ 2 หาผลรวมของทั้งหมด จะได้เวลาที่คาดว่าต้องใช้ในการคำนวณอัลกอริทึมนี้

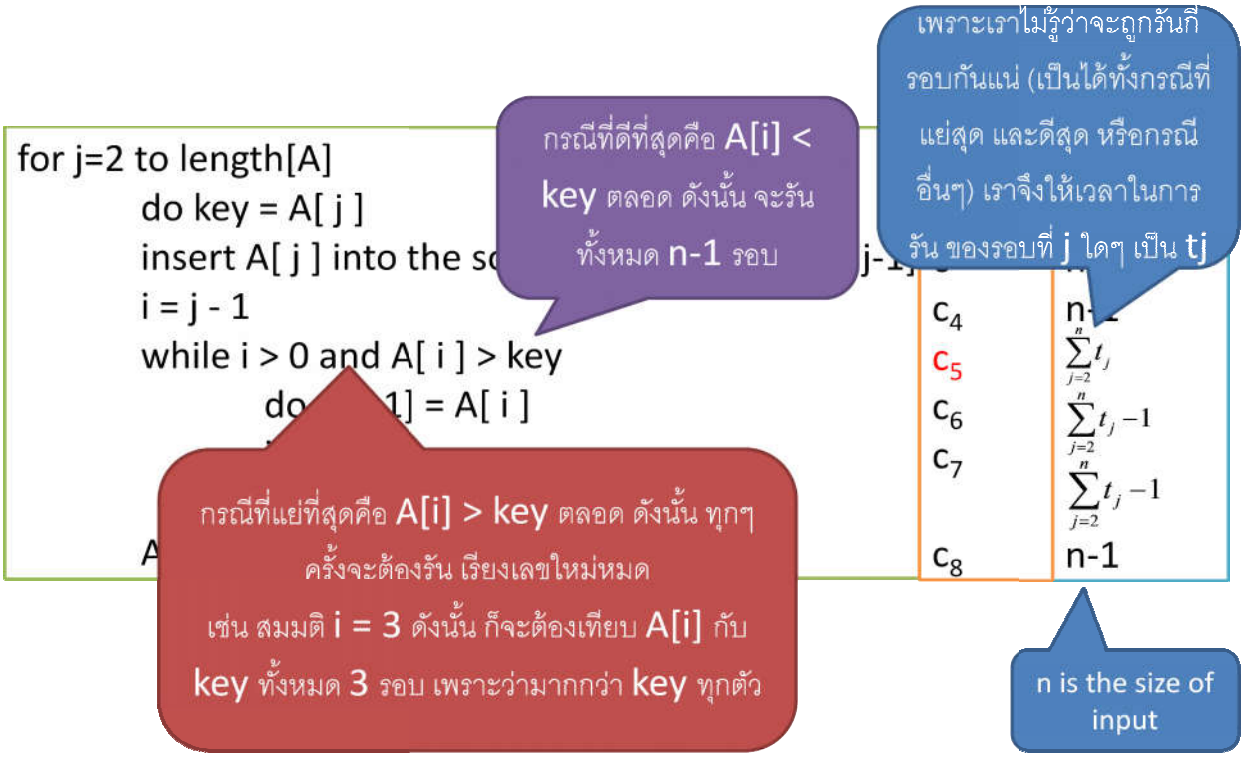
ดังนั้นขั้นตอน การวิเคราะห์ running time ของอัลกอริทึม insertion sort จะเป็นดังต่อไปนี้

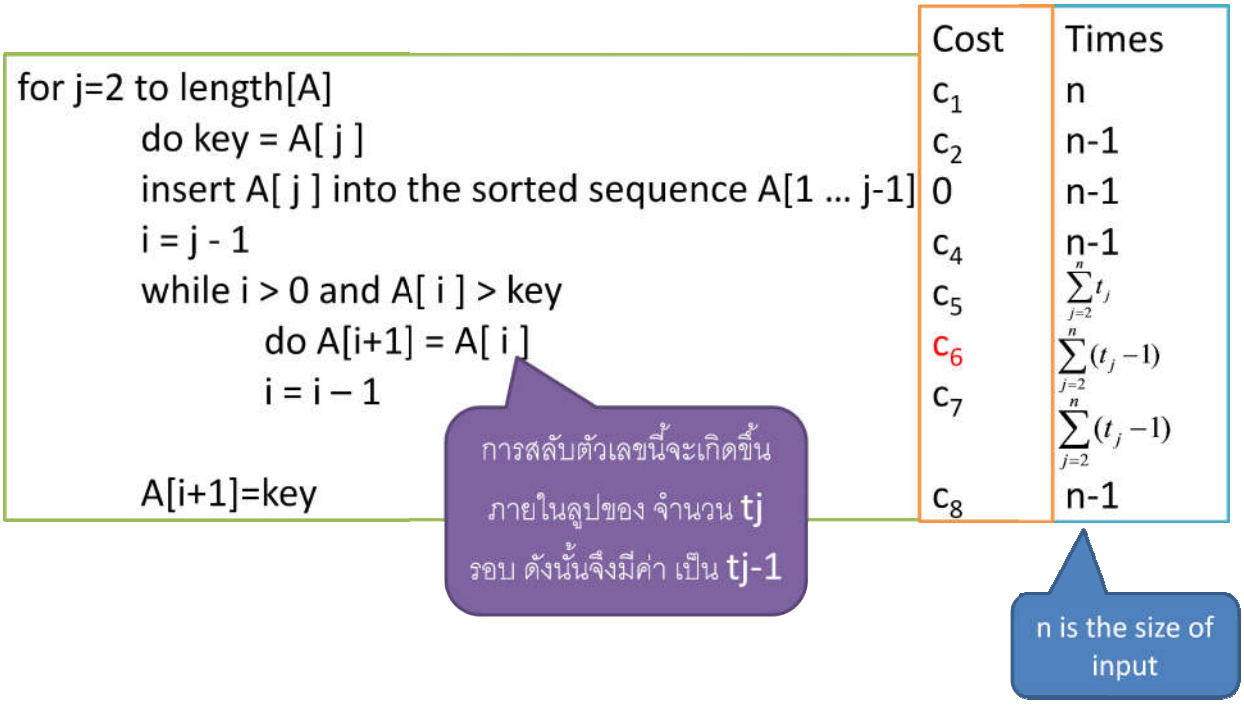

for j=2 to length[A]
do key = A[j]
insert A[j] into th
j-1]
i = j - 1
while i > 0 and A[i] >
do A[i+1] = A[
i = i – 1
A[i+1]=key
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ บวกเพิ่มอีก 1 ตอนที่ต้องเช็คเงื่อนไขจะเข้าลูปรอบสุดท้าย
Cost
Times
c_1 n
c_2 n-1
c_4 0 n-1
c_5 n-1
c_6 $\sum_{j=2}^{n} t_j$
c_7 $\sum_{j=2}^{n} t_j - 1$
c_8 $\sum_{j=2}^{n} t_j - 1$
n-1
n is the size of input

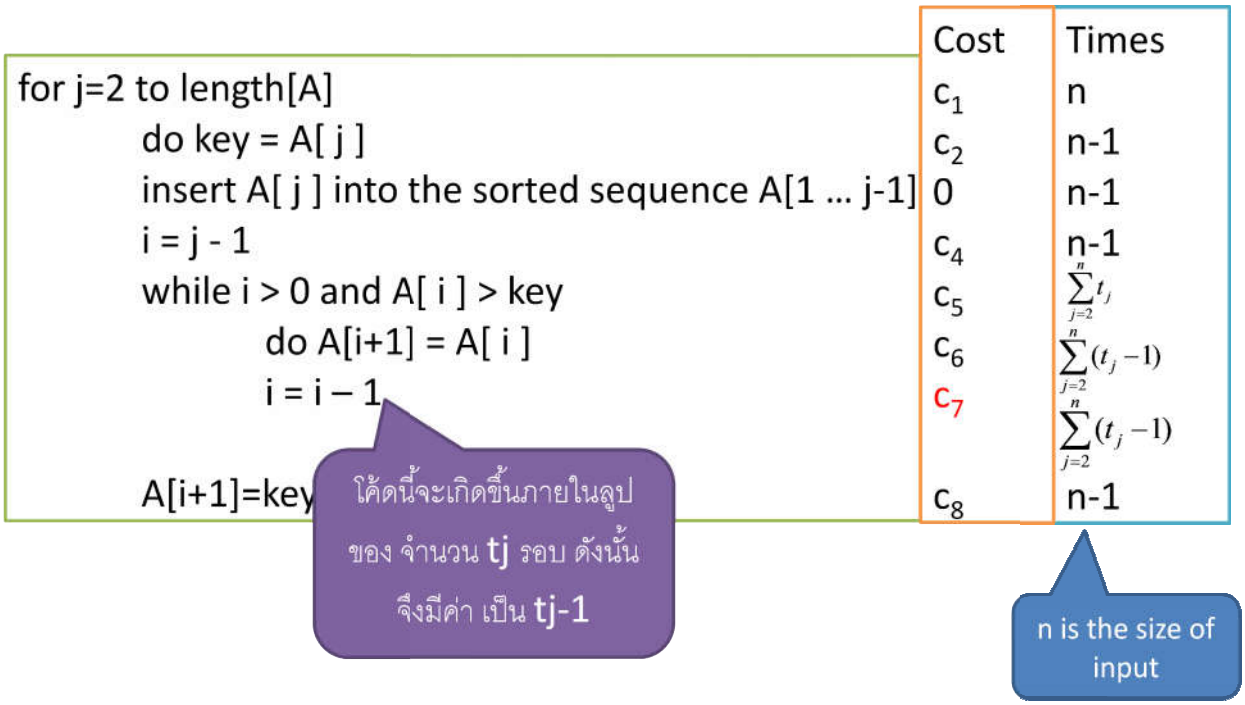

for j=2 to length[A]
do key = A[j]
sert A[j] into th
j-1]
j - 1
A[i] >
] = A[
A[i+1]=key
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ บวกเพิ่มอีก 1 ตอนที่ต้องเช็คเงื่อนไขจะเข้าลูปรอบสุดท้าย
อยู่ในลูปแล้ว(ซึ่งลูปรัน n-1 รอบ) ดังนั้น บรรทัดนี้จะรัน n-1 รอบ
Cost
Times
c_1 n
c_2 n-1
c_4 0 n-1
c_5 n-1
c_6 $\sum_{j=2}^{n} t_j$
c_7 $\sum_{j=2}^{n} t_j - 1$
c_8 $\sum_{j=2}^{n} t_j - 1$
n-1
n is the size of input

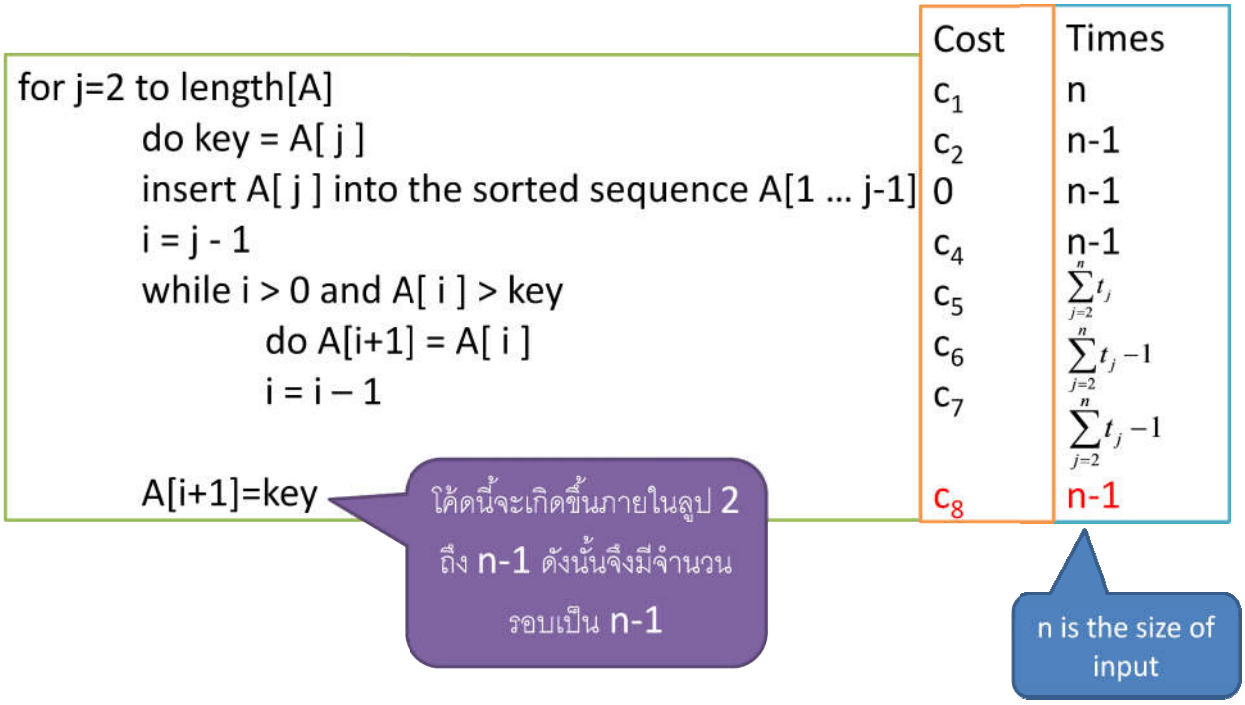

for j=2 to length[A]
do key = A[j]
insert A[j] into th
j-1]
i = j - 1
while i
0 and A[i] >
A[i+1] = A[
A
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ บวกเพิ่มอีก 1 ตอนที่ต้องเช็คเงื่อนไขจะเข้าลูปรอบสุดท้าย
ในโค้ดจริงบรรทัดนี้ไม่ได้ทำงานอะไร cost = 0 และโค้ดนี้อยู่ในลูปที่รัน n-1 รอบ
Cost
Times
c_1 n
c_2 n-1
0 n-1
c_4 n-1
c_5 $\sum_{j=2}^{n} t_j$
c_6 $\sum_{j=2}^{n} t_j - 1$
c_7 $\sum_{j=2}^{n} t_j - 1$
c_8 n-1
n is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into th
j-1]
i = j - 1
whi
i > 0 and A[i] >
do A[i+1] = A[
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ บวกเพิ่มอีก 1 ตอนที่ต้องเช็คเงื่อนไขจะเข้าลูปรอบสุดท้าย
อยู่ในลูปแล้ว(ซึ่งลูปรัน n-1 รอบ) ดังนั้น บรรทัดนี้จะรัน n-1 รอบ
Cost
Times
c_1 n
c_2 n-1
0 n-1
c_4 n-1
c_5 $\sum_{j=2}^{n} t_j$
c_6 $\sum_{j=2}^{n} t_j - 1$
c_7 $\sum_{j=2}^{n} t_j - 1$
c_8 n-1
n is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the so
i = j - 1
while i > 0 and A[i] > key
do
1] = A[i]
A
Cost
Times
c_4
c_5
c_6
c_7
c_8
$\sum_{j=2}^{n} t_j$
$\sum_{j=2}^{n} t_j - 1$
$\sum_{j=2}^{n} t_j - 1$
n-1
กรณีที่ดีที่สุดคือ A[i] < key ตลอด ดังนั้น จะรันทั้งหมด n-1 รอบ
เพราะเราไม่รู้ว่าจะถูกรันกี่รอบกันแน่ (เป็นได้ทั้งกรณีที่แย่สุด และดีสุด หรือกรณีอื่นๆ) เราจึงให้เวลาในการรัน ของรอบที่ j ใดๆ เป็น tj
กรณีที่แย่ที่สุดคือ A[i] > key ตลอด ดังนั้น ทุกๆ ครั้งจะต้องรัน เรียงเลขใหม่หมด
เช่น สมมติ i = 3 ดังนั้น ก็จะต้องเทียบ A[i] กับ key ทั้งหมด 3 รอบ เพราะว่ามากกว่า key ทุกตัว
n is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the sorted sequence A[1 … j-1]
i = j - 1
while i > 0 and A[i] > key
do A[i+1] = A[i]
i = i – 1
A[i+1]=key
Cost
Times
c_1
c_2
0
c_4
c_5
c_6
c_7
c_8
n
n-1
n-1
n-1
$\sum_{j=2}^{n} t_j$
$\sum_{j=2}^{n} (t_j - 1)$
$\sum_{j=2}^{n} (t_j - 1)$
n-1
การสลับตัวเลขนี้จะเกิดขึ้นภายในลูปของ จำนวน tj รอบ ดังนั้นจึงมีค่า เป็น tj-1
n is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the sorted sequence A[1 … j-1]
i = j - 1
while i > 0 and A[i] > key
do A[i+1] = A[i]
i = i – 1
A[i+1]=key
Cost
Times
c_1 n
c_2 n-1
0 n-1
c_4 n-1
c_5 $\sum_{j=2}^{n} t_j$
c_6 $\sum_{j=2}^{n} (t_j - 1)$
c_7 $\sum_{j=2}^{n} (t_j - 1)$
c_8 n-1
โค้ดนี้จะเกิดขึ้นภายในลูป ของ จำนวน tj รอบ ดังนั้น จึงมีค่า เป็น tj-1
n is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the sorted sequence A[1 … j-1]
i = j - 1
while i > 0 and A[i] > key
do A[i+1] = A[i]
i = i – 1
A[i+1]=key
Cost
Times
c_1 n
c_2 n-1
0 n-1
c_4 n-1
c_5 $\sum_{j=2}^{n} t_j$
c_6 $\sum_{j=2}^{n} t_j - 1$
c_7 $\sum_{j=2}^{n} t_j - 1$
c_8 n-1
โค้ดนี้จะเกิดขึ้นภายในลูป 2 ถึง n-1 ดังนั้นจึงมีจำนวน รอบเป็น n-1
n is the size of input

ดังนั้น **running time ของอัลกอริทึม insertion sort** จะเป็นดังสมการต่อไปนี้

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

ใน**กรณีที่ดีที่สุด best case** (คือข้อมูลตัวเลขในอินพุตเรียงกันจากน้อยไปมากอยู่แล้ว) จะเป็น

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_1 + c_2 + c_4 + c_5 + c_8)$$

กรณีที่แย่ที่สุด worst case (คือข้อมูลตัวเลขในอินพุตเรียงกันจากมากไปหาน้อย) จะเป็น

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

หมายเหตุ $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

จะเห็นว่าในกรณี best case ของ insertion sort จะใช้เวลาเป็น linear function of n

ในขณะที่ worst case จะใช้เวลาถึง quadratic function of n

บทที่ ๕ วิเคราะห์ประสิทธิภาพของอัลกอริทึมการเรียงลำดับข้อมูลแบบผสาน

สำหรับการวิเคราะห์ประสิทธภาพของอัลกอริทึมการเรียงลำดับข้อมูลแบบผสานนี้ เราจะเน้นไปที่การวิเคราะห์อัลกอริทึมของฟังก์ชัน merge ที่เดียว เนื่องจากเป็นฟังก์ชันที่มีการทำซ้ำ และเป็นฟังก์ชันสำคัญ

psuedo code ของอัลกอริทึมฟังก์ชัน merge(A,p,q,r) เป็นดังนี้

```
N1 = q – p + 1
N2 = r – q
Create arrays L[1..N1+1] and R[1…N2+1]
For i = 1 to N1
        do L[i]  = A[ p + i -1]
For j = 1 to N2
        do R[j] = A[q + j]
L[N1+1] =  ∞
R[N2+1] =  ∞
i = 1
j = 1
For k=p to r
        do if L[ i ] <= R[ j ]
                then A[k] = L[ i ]
                        i = i+1
                else A[k] = R[ j ]
                        j = j+1
```

ขั้นแรกเราจะทำการ**วิเคราะห์ความถูกต้องของอัลกอริทึมการเรียงลำดับข้อมูลแบบผสาน** (merge sort) โดยใช้เทคนิค loop invariants ตามขั้นตอนดังนี้

before running loop k, all elements in A[p ... k – 1] contains k-p smallest number of L[1..N1+1] , R[1...N2+1]are in sorted order and,

L[i] , R[j] are the smallest number of their arrays that have never been copied into A.

1. ทำการกำหนดประโยค loop invariant ได้คือ

ก่อนที่จะรันลูปรอบที่ k ใดๆ เงื่อนไข 1) อาเรย์ A ตัวที่ p ถึง k-1 จะมีตัวเลขที่น้อยที่สุดจำนวน k-p ตัว ได้มาจากอาเรย์ L ตัวที่ 1 ถึง N1+1 และ อาเรย์ R ตัวที่ 1 ถึง N2+1 และเป็นเป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว และ เงื่อนไข 2) อาเรย์ L ตำแหน่งที่ i กับ อาเรย์ R ตำแหน่งที่ j เป็นตัวเลขที่น้อยที่สุดของอาเรย์ของตัวเอง ซึ่งยังไม่เคยถูกคัดลอกไปเป็นคำตอบในอาเรย์ A เลย

2. ขั้นตอน initialization

พิสูจน์ว่า **ก่อนที่จะรันลูปรอบที่ k=1 เงื่อนไข 1) อาเรย์ A ตัวที่ p ถึง 0 จะมีตัวเลขที่น้อยที่สุดจำนวน 1-p ตัว ได้มาจากอาเรย์ L ตัวที่ 1 ถึง N1+1 และ อาเรย์ R ตัวที่ 1 ถึง N2+1 และเป็นเป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว และเงื่อนไข 2) อาเรย์ L ตำแหน่งที่ i กับ อาเรย์ R ตำแหน่งที่ j เป็นตัวเลขที่น้อยที่สุดของอาเรย์ของตัวเอง ซึ่งยังไม่เคยถูกคัดลอกไปเป็นคำตอบในอาเรย์ A เลย**

จากอัลกอริทึม พิสูจน์เงื่อนไขที่ 1) ก่อนจะรันลูปรอบแรก k=p ดังนั้น อาเรย์ A ตัวที่ p ถึง 0 มันไม่มี จึงถือว่า A[p...k-1] เรียงลำดับเรียบร้อยแล้ว

พิสูจน์เงื่อนไขที่ 2) i = j = 1 ดังนั้น ขนาดของอาเรย์ L, R มีตัวเลขในอาเรย์แค่ 1 จำนวน จึงเป็นเลขที่น้อยที่สุด

3. ขั้นตอน maintenance

พิสูจน์ว่า **ถ้าก่อนที่จะรันลูปรอบที่ k ใดๆ เงื่อนไข 1) อาเรย์ A ตัวที่ p ถึง k-1 จะมีตัวเลขที่น้อยที่สุดจำนวน k-p ตัว ได้มาจากอาเรย์ L ตัวที่ 1 ถึง N1+1 และ อาเรย์ R ตัวที่ 1 ถึง N2+1 และเป็นเป็นตัวเลขที่**

เรียงลำดับกันเรียบร้อยแล้ว และ เงื่อนไข 2) อาเรย์ L ตำแหน่งที่ i กับ อาเรย์ R ตำแหน่งที่ j เป็นตัวเลขที่น้อยที่สุดของอาเรย์ของตัวเอง ซึ่งยังไม่เคยถูกคัดลอกไปเป็นคำตอบในอาเรย์ A เลย

แล้ว ก่อนที่จะรันลูปรอบที่ k+1 เงื่อนไข 1) อาเรย์ A ตัวที่ p ถึง k จะมีตัวเลขที่น้อยที่สุดจำนวน k-p+1 ตัว ได้มาจากอาเรย์ L ตัวที่ 1 ถึง N1+1 และ อาเรย์ R ตัวที่ 1 ถึง N2+1 และเป็นเป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว และ เงื่อนไข 2) อาเรย์ L ตำแหน่งที่ i กับ อาเรย์ R ตำแหน่งที่ j เป็นตัวเลขที่น้อยที่สุดของอาเรย์ของตัวเอง ซึ่งยังไม่เคยถูกคัดลอกไปเป็นคำตอบในอาเรย์ A เลย

จากอัลกอริทึม พิสูจน์เงื่อนไขที่ 1) หลังรันรอบที่ k จะพบว่า

i) ถ้าหาก L[i] < R[j] แล้ว L[i] เป็นค่าที่น้อยสุด และถูกคัดลอกไปไว้ในอาเรย์ A ตำแหน่งที่ k ซึ่งเรามีสมมติฐานอยู่ว่าอาเรย์ A ตำแหน่ง p ถึง k-1 ถูกเรียงเรียบร้อยอยู่แล้ว และมีเลขที่น้อยสุดอยู่ k-p ตัว เมื่อคัดลอกเพิ่มไป ดังนั้น เราจะได้อาเรย์ A ที่มีตัวเลขที่น้อยสุดจำนวน k-p+1 ตัว

ii) ถ้าหาก L[i] > R[j] แล้ว R[i] เป็นค่าที่น้อยสุด และถูกคัดลอกไปไว้ในอาเรย์ A ตำแหน่งที่ k ซึ่งเรามีสมมติฐานอยู่ว่าอาเรย์ A ตำแหน่ง p ถึง k-1 ถูกเรียงเรียบร้อยอยู่แล้ว และมีเลขที่น้อยสุดอยู่ k-p ตัว เมื่อคัดลอกเพิ่มไป ดังนั้น เราจะได้อาเรย์ A ที่มีตัวเลขที่น้อยสุดจำนวน k-p+1 ตัว

ดังนั้นเงื่อนไขที่ 1) เป็นจริง ส่วนเงื่อนไขที่ 2) เป็นจริงเนื่องจาก อาเรย์ L,R เป็อาเรย์ที่เรียงอยู่แล้ว ดังนั้น L[i], R[j] เป็นตัวเลขที่น้อยที่สุดของอาเรย์ตัวเอง และยังไม่เคยถูกคัดลอกคำตอบ เพราะเมื่อถูกคัดลอกไปที่อาเรย์ A ค่าตำแหน่ง i, j จะต้องเลื่อนไปตัวถัดไป

4. ขั้นตอน termination

พิสูจน์ว่า หลังจากจบลูป อาเรย์ทั้งหมดเรียงลำดับกันเรียบร้อย

ดังนั้นเราสามารถใช้บทพิสูจน์ได้ว่า

ก่อนที่จะรันลูปรอบที่ n+1 ใดๆ เงื่อนไข 1) อาเรย์ A ตัวที่ p ถึง n จะมีตัวเลขที่น้อยที่สุดจำนวน n-p+1 ตัว ได้มาจากอาเรย์ L ตัวที่ 1 ถึง N1+1 และ อาเรย์ R ตัวที่ 1 ถึง N2+1 และเป็นเป็นตัวเลขที่เรียงลำดับกัน

เรียบร้อยแล้ว และ เงื่อนไข 2) อาเรย์ L ตำแหน่งที่ i กับ อาเรย์ R ตำแหน่งที่ j เป็นตัวเลขที่น้อยที่สุดของอาเรย์ของตัวเอง ซึ่งยังไม่เคยถูกคัดลอกไปเป็นคำตอบในอาเรย์ A เลย

อ้างอิงจากขั้นตอน maintenance ก็จะพบว่า เราได้พิสูจน์ไว้แล้วว่า ข้อความข้างต้นเป็นจริง

จบการพิสูจน์ ดังนั้น อัลกอริทึมนี้ถูกต้อง

สำหรับการวิเคราะห์ running time ของอัลกอริทึม merge sort เราจะอธิบายอีกครั้งในบทที่ 7 เรื่อง recurrence

บทที่ ๖ สัญกรณ์เชิงเส้นกำกับ (asymptotic notation)

สิ่งที่เราต้องคำนึงถึงในการเลือกใช้หรือการออกแบบอัลกอริทึมคือเวลาในการคำนวณอัลกอริทึมนั้นๆ ในกรณีที่เรามีอินพุตขนาดเพิ่มขึ้นเป็นจำนวนมาก

ดังนั้นเราจำเป็นต้องวิเคราะห์ เวลาที่ใช้ในการประมวลผลอัลกอริทึม โดยการหาขอบเขตบน(หรือคือการประมาณการเวลาที่มากสุดที่เป็นไปได้) ของเวลาที่จะใช้ในการคำนวณอัลกอริทึม ซึ่งวิธีการประมาณการนี้เราจะใช้สัญกรณ์เชิงเส้นกำกับ

สัญกรณ์เชิงเส้นกำกับ (asymptotic notation) คือ สัญลักษณ์ที่ใช้อธิบายการเติบโตของฟังก์ชัน เพื่อที่จะนำมาอธิบายเวลาที่ใช้ในการประมวลผลของอัลกริทึม ซึ่งมีด้วยกันอยู่ 5 สัญลักษณ์คือ

1. Big-theta (Θ)

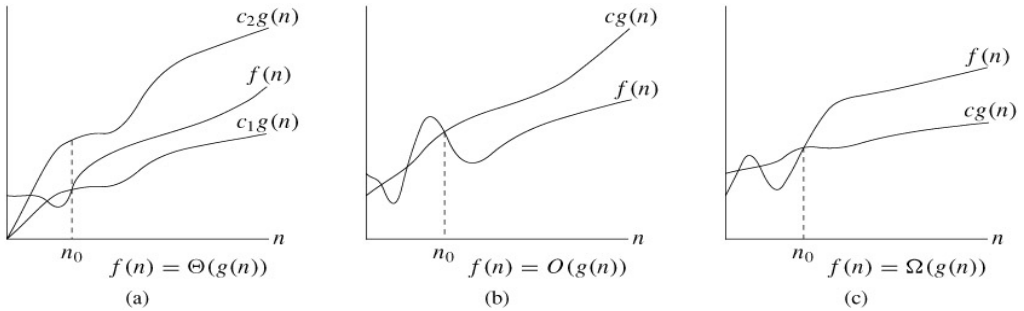
2.Big-O (O)

3.Big-omega (Ω)

4.Little-O (o)

5.Little-omega (ω)

แสดงดังรูปตัวอย่างภาพที่ 10



ภาพที่ 10

1. Big-theta

มี**นิยาม**คือ

$\Theta(g(n)) =$ { f(n): there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0$ }

จากนิยามคือ ถ้าเรามีฟังก์ชัน g(n) ใดๆอยู่ แล้วต้องการหาค่า Big-theta ของมัน เราจะต้องหาค่าคงที่ที่เป็นบวก c_1, c_2 และ n_0 แล้วเรานำค่าคงที่นั้นไปแทนสมการ $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ ถ้าเราสามารถหาฟังก์ชัน f(n) ใดๆที่ทำให้สมการนี้เป็นจริงได้ สำหรับ n ทุกตัวที่มีค่ามากกว่า n_0 แล้ว Big-theta ของ g(n) ก็คือ เซตของฟังก์ชัน f(n) นั้น (ใช้คำว่าเซต หมายความว่า เราอาจจะหา f(n) ได้หลายฟังก์ชันก็ได้)

สังเกตดูจากภาพที่10 ได้แสดงให้เห็นว่าฟังก์ชัน f(n)จะมีค่าอยู่ตรงกลางระหว่างฟังก์ชัน c_1 g(n) และ c_2 g(n) เสมอ

เราจะเขียนสัญลักษณ์ว่า $f(n) = \Theta(g(n))$ เพื่อแทนความหมาย $f(n) \in \Theta(g(n))$

<u>ตัวอย่างที่ 5</u> จงทำการตรวจสอบว่า $f(n) \in \Theta(g(n))$ เมื่อกำหนดให้ ฟังก์ชัน $f(n) = \frac{1}{2}\ n^2 - 3n$ และ$g(n) = n^2$

วิธีทำ เราต้องทำการพิสูจน์ว่า เราสามารถหาค่าคงที่ c_1, c_2 และ n_0 ที่ทำให้สมการ $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

ดังนั้นเริ่มจากแทนค่าฟังก์ชัน f(n) , g(n) ลงไปในสมการ จะได้ $0 \le c_1 n^2 \le \frac{1}{2}\ n^2 - 3n \le c_2 n^2$

จัดรูปสมการเพื่อให้เหลือค่าตัวแปร n ตัวเดียว โดยการหารสมการทั้งสองข้างด้วย n^2 จะได้ $0 \le c_1 \le \frac{1}{2}\ -\frac{3}{n} \le c_2$

ทีนี้เวลาวิเคราะห์หาค่า c_1, c_2 และ n_0 ควรจะทำการตรวจสอบทีละค่า

1) เริ่มจากพิจารณาข้างซ้ายของสมการก่อน $0 \le c_1 \le \frac{1}{2}\ -\frac{3}{n}$

ก็จะเห็นว่าเมื่อค่า n มีจำนวนมากๆ ค่าของ c_1 จะลู่เข้าสู่ค่าคงที่คือ 1/2 และพบว่าที่ n=6 ค่า c_1=0 แต่ว่าจากนิยามบอกว่าค่า c_1 จะต้องเป็นค่าบวก (>0) เพื่อให้ได้ค่า c_1 เป็นค่าบวก เราจึงเลือกค่า n_0 =7 เมื่อแทนค่าลงไปในสมการที่ n=7 จะได้ c_1=1/14 ซึ่งจะสรุปได้ว่า ที่ n>= n_0 (คือ n>=7) จะพบว่าค่า c_1 <= 1/14 ก็จะเห็นว่าสามารถหาค่า c_1 และ n_0 ที่ทำให้ สมการข้างต้นเป็นจริง

2) แล้วก็จึงจะพิจารณาข้างขวาของสมการ $0 \le \frac{1}{2} - \frac{3}{n} \le c_2$

ก็จะเห็นว่าถ้าค่า n อยู่ในช่วง 1-5 จะได้ค่า c_2 เป็นค่าติดลบ แต่เมื่อค่า n มีจำนวนมากๆ ค่าของ c_2 จะลู่เข้าสู่ค่าคงที่คือ 1/2 และพบว่าที่ n=6 ค่า c_2=0 แต่ว่าจากนิยามบอกว่าค่า c_2 จะต้องเป็นค่าบวก (>0) เพื่อให้ได้ค่า c_2 เป็นค่าบวก เราจึงเลือกค่า n_0 =7 เมื่อแทนค่าลงไปในสมการที่ n=7 จะได้ c_2=1/14 ซึ่งจะสรุปได้ว่า ที่ n>= n_0 (คือ n>=7) จะพบว่าค่า c_2 >= 1/2 ก็จะเห็นว่า สามารถหาค่า c_2 และ n_0 ที่ทำให้ สมการข้างต้นเป็นจริง

จากการพิสูจน์ในข้อ 1) และ 2) เราพบว่า เราสามารถหาค่าคงที่ c_1, c_2 และ n_0 ที่ทำให้ สมการ $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ เป็นจริง ดังนั้นสรุปได้ว่า $f(n) \in \Theta(g(n))$

ข้อสังเกต เรามีวิธีการตรวจสอบ ว่าค่าคงที่ c_1, c_2 และ n_0 ที่เราเลือกนั้นทำให้สมการเป็นจริงหรือไม่ โดยการทดลองแทนค่าคงที่และฟังก์ชัน กลับเข้าไปในสมการตั้งต้นที่เราต้องการตรวจสอบ ถ้าหากว่าได้ค่าทั้งสองข้างของสมการไม่เป็นจริงแสดงว่า น่าจะมีข้อผิดพลาดเกิดขึ้นในการเลือกค่าคงที่

ตัวอย่างเช่น หากเราต้องการตรวจสอบการเลือกค่า ค่าคงที่ c_1, c_2 และ n_0 ของตัวอย่างที่ 5 ข้างต้น

1) เริ่มจากการแทนค่าสมการข้างซ้ายก่อน (ดูเฉพาะ c_1 ก่อน)
จากตัวอย่างที่ 5 เราเลือก n_0 =7 และ c_1=1/14 เมื่อแทนค่าลงไปในสมการ $0 \le c_1 n^2 \le \frac{1}{2} n^2 - 3n$
ก็จะได้ $0 \le (\frac{1}{14} \times 7^2) \le (\frac{1}{2} \times 7^2 - 3 \times 7)$ จะเห็นว่าค่าของสมการเป็นจริง

2) แทนค่าสมการข้างขวา (ดูเฉพาะ c_2)
จากตัวอย่างที่ 5 เราเลือก n_0 =7 และ c_2=1/2 เมื่อแทนค่าลงไปในสมการ $0 \le \frac{1}{2} n^2 - 3n \le c_2 n^2$
ก็จะได้ $0 \le (\frac{1}{2} \times 7^2 - 3 \times 7) \le (\frac{1}{2} \times 7^2)$ จะเห็นว่าค่าของสมการเป็นจริง

2. Big-O

มี**นิยาม**คือ

$O(g(n)) =$ { f(n): there exist positive constants c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$ }

จากนิยามคือ ถ้าเรามีฟังก์ชัน g(n) ใดๆอยู่ แล้วต้องการหาค่า Big-O ของมัน เราจะต้องหาค่าคงที่ c ที่เป็นบวก และ n_0 แล้วเรานำค่าคงที่นั้นไปแทนสมการ $0 \le f(n) \le c\, g(n)$ ถ้าเราสามารถหาฟังก์ชัน f(n) ใดๆที่ทำให้สมการนี้เป็นจริงได้ สำหรับ n ทุกตัวที่มีค่ามากกว่า n_0 แล้ว Big-O ของ g(n) ก็คือ เซตของฟังก์ชัน f(n) นั้น (ใช้คำว่าเซตหมายความว่า เราอาจจะหา f(n) ได้หลายฟังก์ชันก็ได้)

สังเกตดูจากภาพที่10 ได้แสดงให้เห็นว่าฟังก์ชัน f(n)จะมีค่าอยู่ต่ำกว่าฟังก์ชัน c g(n) เสมอ

เราจะเขียนสัญลักษณ์ว่า $f(n) = O(g(n))$ เพื่อแทนความหมาย $f(n) \in O(g(n))$

<u>ตัวอย่างที่ 6</u> จงทำการตรวจสอบว่า $f(n) = O(g(n))$ เมื่อกำหนดให้ ฟังก์ชัน $f(n) = 3n^2$ และ $g(n) = n^2$

วิธีทำ เราต้องทำการพิสูจน์ว่า เราสามารถหาค่าคงที่ c และ n_0 ที่ทำให้สมการ $0 \le f(n) \le c\, g(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

ดังนั้นเริ่มจากแทนค่าฟังก์ชัน f(n) , g(n) ลงไปในสมการ จะได้ $0 \le 3n^2 \le c\ n^2$

จัดรูปสมการเพื่อให้เหลือค่าตัวแปร n ตัวเดียว โดยการหารสมการทั้งสองข้างด้วย n^2 จะได้ $0 \le 3 \le c$

เนื่องจาก ไม่มีตัวแปร n ในสมการหมายความว่า n เป็นอะไรก้ได้เราให้ n_0 =1 ทำให้สรุปได้ว่า ที่ n>= n_0 (คือ n_0 >=1) จะพบว่าที่ n_0 =1 ไม่ว่าค่า n เป็นอะไรที่ มากกว่า n_0 เราสามารถหาค่า c >=3 ที่ทำให้ สมการข้างต้นเป็นจริง ดังนั้นสรุปได้ว่า $f(n) \in O(g(n))$

3. Little-O

มี**นิยาม**คือ

$o(g(n)) =$ { f(n): for any positive constant c > 0, there exists a constant $n_0 > 0$ such that $0 \le f(n) < c\,g(n)$ for all $n \ge n_0$ }

จากนิยามคือ ถ้าเรามีฟังก์ชัน g(n) ใดๆอยู่ แล้วต้องการหาค่า Little-O ของมัน ไม่ว่าค่าคงที่ c จะเป็นค่าอะไรก็ตามที่มากกว่าศูนย์ เราจะต้องหาค่า n_0 ที่มากกว่าศูนย์ แล้วเรานำค่าคงที่นั้นไปแทนสมการ $0 \le f(n) < c\,g(n)$ ถ้าเราสามารถหาฟังก์ชัน f(n) ใดๆที่ทำให้สมการนี้เป็นจริงได้ สำหรับ n ทุกตัวที่มีค่ามากกว่า n_0 แล้ว Little-O ของ g(n) ก็คือ เซตของฟังก์ชัน f(n) นั้น (ใช้คำว่าเซต หมายความว่า เราอาจจะหา f(n) ได้หลายฟังก์ชันก็ได้)

เราจะเขียนสัญลักษณ์ว่า $f(n) = o(g(n))$ เพื่อแทนความหมาย $f(n) \in o(g(n))$

<u>ตัวอย่างที่ 7</u> จงทำการตรวจสอบว่า $f(n) \in o(g(n))$ เมื่อกำหนดให้ ฟังก์ชัน $f(n) = 2n$ และ $g(n) = n^2$

วิธีทำ เราต้องทำการพิสูจน์ว่า สำหรับค่าคงที่ c ทุกๆค่า เราสามารถหาค่าคงที่ n_0 ที่ทำให้สมการ $0 \le f(n) < c\,g(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

ดังนั้นเริ่มจากแทนค่าฟังก์ชัน f(n) , g(n) ลงไปในสมการ จะได้ $0 \le 2n < cn^2$

จัดรูปสมการเพื่อให้เหลือค่าตัวแปร n ตัวเดียว โดยการหารสมการทั้งสองข้างด้วย n จะได้ $0 \le 2 < cn$

ลองแทนค่าถ้า c =1 จะได้ n > 2 ดังนั้น n_0 >2 ถ้าเราเลือกค่า c ที่ =2 จะได้ n > 1 ดังนั้น n_0 >1

ถ้า c = k ใดๆ ค่า n = 2/k ดังนั้นไม่ว่า c เป็นค่าอะไร เราจะสามารถหา n_0 > 0 ได้

สมมติเราทดสอบสมการด้วยการเลือก c = 1, n_0=3 จะเห็นว่า สมการเป็นจริง แต่หาก เลือก c=1, n_0=2 สมการจะไม่เป็นจริง เพราะฉะนั้นต้องเลือก c >=1, n_0 >2 (คือ n_0 >=3) ก็จะเห็นว่า ไม่ว่าค่า c จะเป็นค่าอะไรก็ตามที่มากกว่า 0 เราสามารถหาค่าของ n_0 >=3 ที่ทำให้ สมการข้างต้นเป็นจริง ดังนั้นสรุปได้ว่า $f(n) \in o(g(n))$

ข้อสังเกตระหว่าง Big-O กับ Little-O

- จะเห็นว่า Big-O จะครอบคลุมกรณีมากกว่า คือ ถ้าเรา วิเคราะห์สมการเวลาไว้ tight ก็ได้ หรือจะไม่ tight bound ก็ได้
- แต่ว่า Little-O จะครอบคลุมเฉพาะกรณีที่ ไม่ tight bound
- Tight bound คือกรณีที่สมการไม่ฟิตกัน อาจจะเพราะเรา over estimate เช่น กรณี $f(n)=2n$ และ $g(n)=n^2$ แล้วเราจะหาว่า $f(n)\in o(g(n))$ จะหาด้วย little-o ได้เพราะว่า ไม่ tight bound ดังนั้น ถ้าสมการ ไม่ tight bound เราจะพบค่า Little-O ของมัน และ Big-O ของมันด้วย

4. Big-Omega

มี**นิยาม**คือ

$\Omega(g(n)) =$ {f(n): there exist positive constants c and n_0 such that $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$ }

จากนิยามคือ ถ้าเรามีฟังก์ชัน g(n) ใดๆอยู่ แล้วต้องการหาค่า Big-omega ของมัน เราจะต้องหาค่าคงที่ที่เป็นบวก c และ n_0 แล้วเรานำค่าคงที่นั้นไปแทนสมการ $0 \le c\,g(n) \le f(n)$ ถ้าเราสามารถหาฟังก์ชัน f(n) ใดๆที่ทำให้สมการนี้เป็นจริงได้ สำหรับ n ทุกตัวที่มีค่ามากกว่า n_0 แล้ว Big-omega ของ g(n) ก็คือ เซตของฟังก์ชัน f(n) นั้น (ใช้คำว่าเซต หมายความว่า เราอาจจะหา f(n) ได้หลายฟังก์ชันก็ได้)

สังเกตดูจากภาพที่10 ได้แสดงให้เห็นว่าฟังก์ชัน f(n)จะมีค่าอยู่ตรงด้านบนฟังก์ชัน c g(n) เสมอ

เราจะเขียนสัญลักษณ์ว่า $f(n)=\Omega(g(n))$ เพื่อแทนความหมาย $f(n)\in\Omega(g(n))$

ตัวอย่างที่ 8 จงทำการตรวจสอบว่า $f(n)\in\Omega(g(n))$ เมื่อกำหนดให้ ฟังก์ชัน $f(n)=3n^2$ และ $g(n)=n$

วิธีทำ เราต้องทำการพิสูจน์ว่า สำหรับค่าคงที่ c ทุกๆค่า เราสามารถหาค่าคงที่ n_0 ที่ทำให้สมการ $0 \le c\,g(n) \le f(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

ดังนั้นเริ่มจากแทนค่าฟังก์ชัน f(n) , g(n) ลงไปในสมการ จะได้ $0 \le cn \le 3n^2$

จัดรูปสมการเพื่อให้เหลือค่าตัวแปร n ตัวเดียว โดยการหารสมการทั้งสองข้างด้วย n^2 จะได้ $0 \le \frac{c}{n} \le 3$

ถ้า n = 1 แล้วจะได้ c <= 3 ถ้า n=2 แล้ว c <= 6, n=3, c<=9 ต่อไปเรื่อยๆ จะเห็นว่า ไม่ว่า n จะเป็นเลขอะไร เราสามารถหาค่าคงที่ c ที่มากกว่า 0 ที่ทำให้สมการนี้เป็นจริงได้ ดังนั้นก็จะได้ว่า ที่ n_0 =1 เราสามารถหาค่า 0< c <=3 ได้ สรุปได้ว่า $f(n) \in \Omega(g(n))$

5. Little-omega

มี**นิยาม**คือ

$\omega(g(n)) =$ { f(n): for any positive constants c > 0 and n_0 such that $0 \le c\, g(n) < f(n)$ for all $n \ge n_0$ }

จากนิยามคือ ถ้าเรามีฟังก์ชัน g(n) ใดๆอยู่ แล้วต้องการหาค่า Little-omega ของมัน ไม่ว่าค่าคงที่ c จะเป็นค่าอะไรก็ตามที่มากกว่าศูนย์ เราจะต้องหาค่า n_0 ที่มากกว่าศูนย์ แล้วเรานำค่าคงที่นั้นไปแทนสมการ $0 \le c\, g(n) < f(n)$ ถ้าเราสามารถหาฟังก์ชัน f(n) ใดๆที่ทำให้สมการนี้เป็นจริงได้ สำหรับ n ทุกตัวที่มีค่ามากกว่า n_0 แล้ว Little-omega ของ g(n) ก็คือ เซตของฟังก์ชัน f(n) นั้น (ใช้คำว่าเซต หมายความว่า เราอาจจะหา f(n) ได้หลายฟังก์ชันก็ได้)

เราจะเขียนสัญลักษณ์ว่า $f(n) = \omega(g(n))$ เพื่อแทนความหมาย $f(n) \in \omega(g(n))$

<u>ตัวอย่างที่ 9</u> จงทำการตรวจสอบว่า $f(n) \in \omega(g(n))$ เมื่อกำหนดให้ ฟังก์ชัน $f(n) = 3n^2$ และ $g(n) = n$

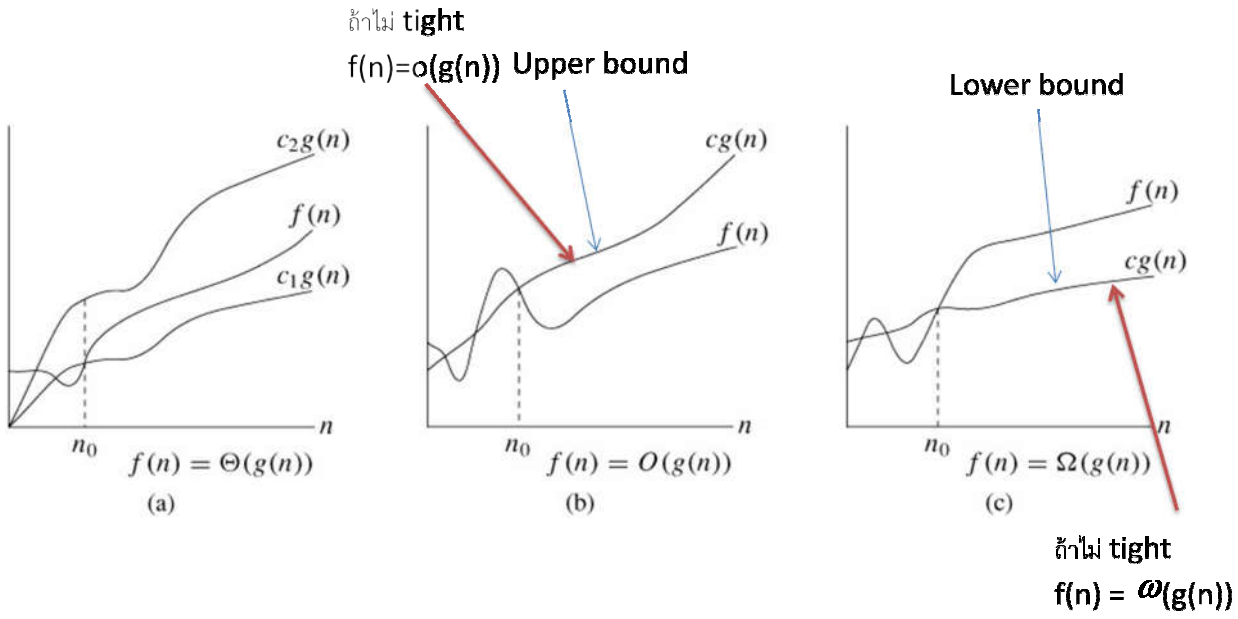
วิธีทำ เราต้องทำการพิสูจน์ว่า เราสามารถหาค่าคงที่ c และ n_0 ที่ทำให้สมการ $0 \le c\, g(n) < f(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

ดังนั้นเริ่มจากแทนค่าฟังก์ชัน f(n) , g(n) ลงไปในสมการ จะได้ $0 \le cn < 3n^2$

จัดรูปสมการเพื่อให้เหลือค่าตัวแปร n ตัวเดียว โดยการหารสมการทั้งสองข้างด้วย n^2 จะได้ $0 \le \frac{c}{n} \le 3$

ถ้าหาก c =3 เราก็จะได้ค่า n = 1 เพื่อให้ พจน์ c/n มันน้อยกว่า 3 ตามสมการ สมมติ c=1 เราก็จะได้ค่า n > 0 เช่นกัน(หารแล้วปัดขึ้น) ดังนั้นไม่ว่าค่า c เป็นอะไร เราจะได้ค่า n > 0 ดังนั้นเราจะสามารถหา n_0 > 0 ได้ สรุปได้ว่า $f(n) \in \omega(g(n))$

รูป**ภาพที่ 11** อธิบายความสัมพันธ์ของ asymptotic notation ทั้งห้าแบบ



ภาพที่ 11

แบบฝึกหัด

1. จงตรวจสอบว่า $f(n) \in \Theta(g(n))$ เมื่อกำหนดให้ $f(n) = 6n^3$ และ $g(n) = n^2$
2. จงตรวจสอบว่า $f(n) \in \Theta(g(n))$ เมื่อกำหนดให้ $f(n) = 2n^4 - 4n$ และ $g(n) = n^2$
3. จงตรวจสอบว่า $f(n) \in O(g(n))$ เมื่อกำหนดให้ $f(n) = 3n + 5$ และ $g(n) = n^2$
4. จงตรวจสอบว่า $f(n) \in o(g(n))$ เมื่อกำหนดให้ $f(n) = 3n^2$ และ $g(n) = n^2$
5. จงตรวจสอบว่า $f(n) \in \Omega(g(n))$ เมื่อกำหนดให้ $f(n) = 3n^2$ และ $g(n) = n^2$
6. จงตรวจสอบว่า $f(n) \in \omega(g(n))$ เมื่อกำหนดให้ $f(n) = 3n^2$ และ $g(n) = n^2$