เอกสารประกอบการสอนรายวิชา 305234 **การวิเคราะห์และการออกแบบวิธี**

Algorithm Analysis and Design

จัดทำโดย

อาจารย์จิราพร พุกสุข

ภาควิชาวิศวกรรมไฟฟ้าและคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยนเรศวร

ปรับปรุงล่าสุด ธันวาคม 2563

สำหรับใช้ประกอบการสอน รายวิชา 305234 การวิเคราะห์และออกแบบขั้นตอนวิธี ภาคเรียนปลาย ปีการศึกษา 2563

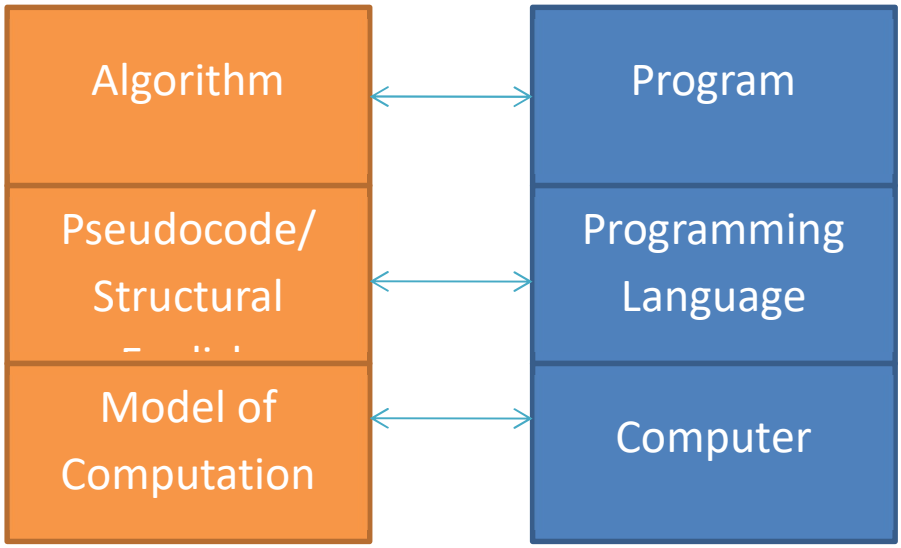
สารบัญ

แนะนำรายวิชา

อัลกอริทึมคือ ลำดับขั้นตอนการคำนวณที่ได้กำหนดไว้เพื่อในการแก้ปัญหาอย่างใดอย่างหนึ่ง ซึ่งจะมีการรับค่าอินพุตและแสดงค่าเอ๊าพุตสำหรับการแก้ปัญหานั้นๆ

อัลกอริทึมใดๆ จะถือว่าถูกต้อง ถ้าหากว่าไม่ว่าจะใส่อินพุตใดๆให้กับอัลกอริทึมนั้น มันจะต้องหยุดทำงานพร้อมกับให้เอ๊าพุตที่ถูกต้อง

แผนภาพที่ 1 แสดงความสัมพันธ์ระหว่างแนวคิดทางทฤษฎี กับการทำแนวคิดนั้นให้สำเร็จจริง



ภาพที่ 1

ใน**การวัดประสิทธิภาพของอัลกอริทึม**นั้น มีการวัดผลหลักๆอยู่สองด้านคือ

1. ความถูกต้องของอัลกอริทึม

    ซึ่งจะต้องตรวจสอบว่าสามารถให้คำตอบได้ถูกต้องไม่ว่าจะเป็นอินพุตใดๆ

2. เวลาที่ใช้ในการประมวลผลอัลกอริทึม

    ซึ่งจะต้องตรวจสอบว่าใช้เวลาน้อยที่สุดในการที่จะทำการประมวลผลให้เสร็จ

ตัวอย่างที่ 1 การวัดประสิทธิภาพของอัลกอริทึม

กำหนดให้มีอัลกอริทึมอยู่ 2 อัลกอริทึมดังนี้

1. insertion sort ใช้เวลาในการคำนวณประมาณ $x.n^2$

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

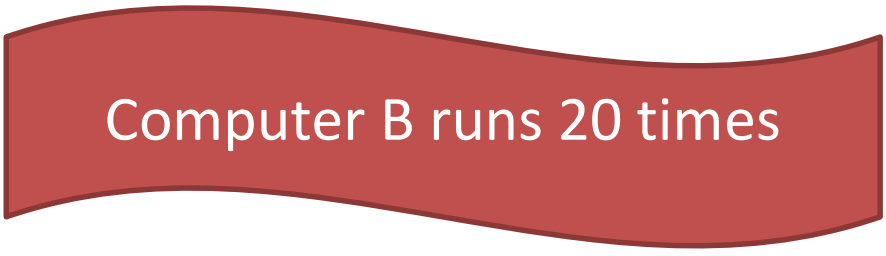
2. คอมพิวเตอร์ B สามารถรันได้ 10^7 คำสั่งต่อวินาที

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 ตามประสิทธิภาพของอัลกอริทึม คือ

1. ความถูกต้องของอัลกอริทึม

    ซึ่งสามารถวิเคราะห์ได้ โดยการใช้เทคนิคการไม่แปรเปลี่ยนของการวนซ้ำ (loop invariants) ในกรณีที่อัลกอริทึมมีขั้นตอนการทำซ้ำ

2. เวลาที่ใช้ในการประมวลผลอัลกอริทึม

    ซึ่งสามารถวิเคราะห์ได้ จากการหาค่าขอบเขตบน ของเวลาที่ใช้ในการรันอัลกอริทึมนั้นๆ ซึ่งสามารถคำนวณคร่าวๆได้จากการดูอัตราการเติบโตของฟังก์ชัน (growth of function)

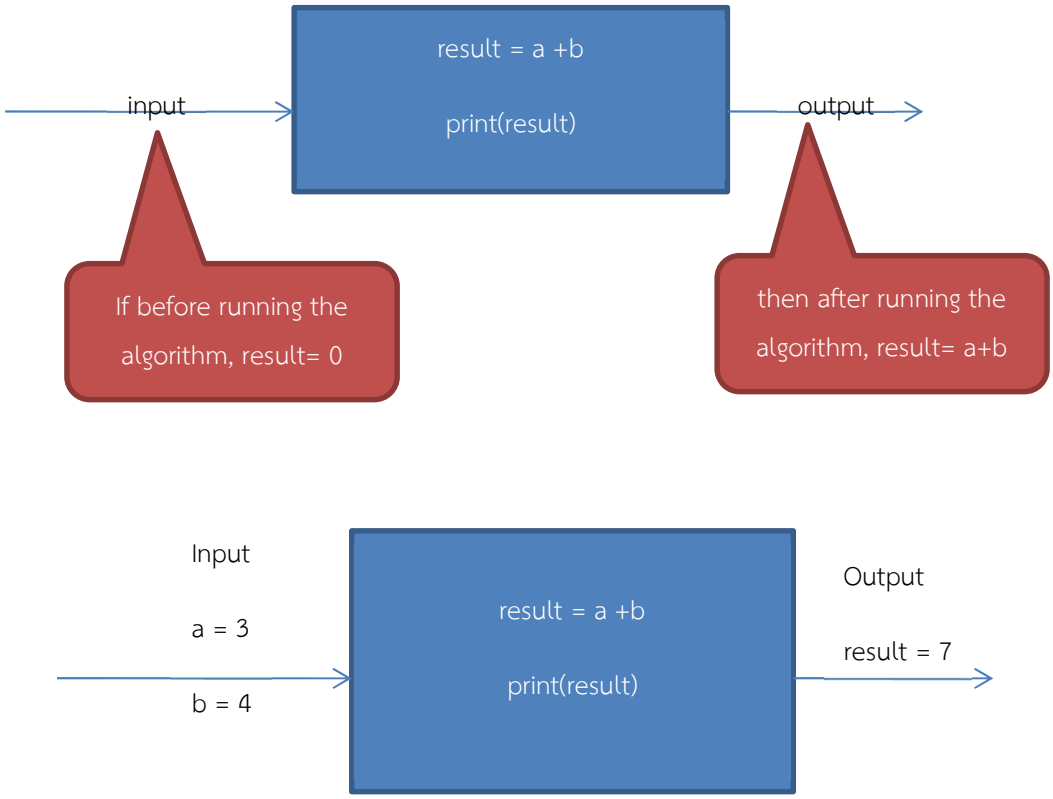
บทที่ ๑ การวิเคราะห์ความถูกต้องของอัลกอริทึม
กับ การไม่แปรเปลี่ยนของการวนซ้ำ (loop invariants)

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

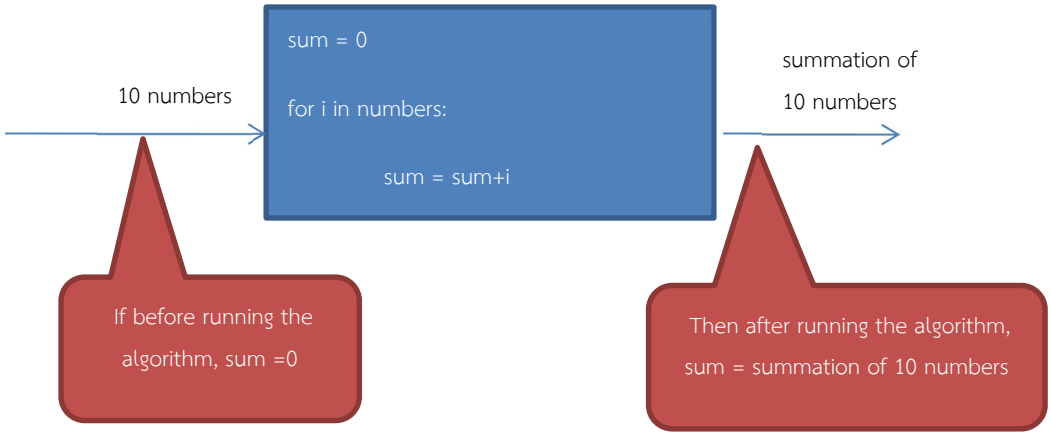
แต่กรณีที่อัลกอริทึมมีขั้นตอนส่วนของการทำซ้ำ จะไม่สามารถใช้เทคนิคการตรวจสอบแบบข้างต้นได้ เนื่องจากจะต้องทำการตรวจสอบว่า ในทุกๆรอบของการทำซ้ำนั้น ค่าของข้อมูลที่ต้องการใช้เพื่อเป็นคำตอบของการแก้โจทย์ปัญหานั้นไม่มีการเปลี่ยนแปลงค่า

อย่างเช่น อัลกอริทึมการหาค่าผลรวมของตัวเลขจำนวน 10 ตัว ที่จะต้องทำการตรวจสอบทุกรอบ(ทั้ง 10 รอบ) ว่า

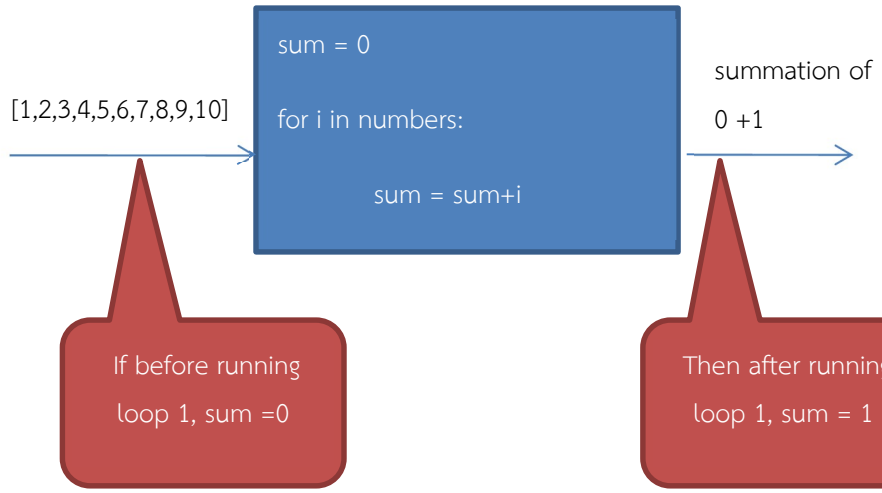
1. ค่าผลรวมนั้นถูกต้องก่อนที่จะรันลูป ในแต่ละรอบ และ

2. ค่าผลรวมนั้นถูกต้องหลังจากที่รันลูปในแต่ละรอบ

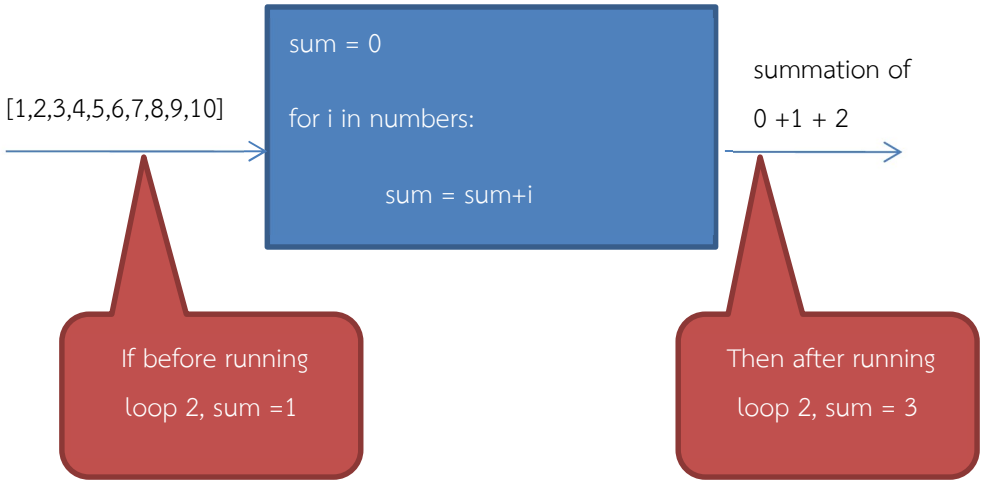
ดังแสดงตัวอย่างใน**ภาพที่ 3**



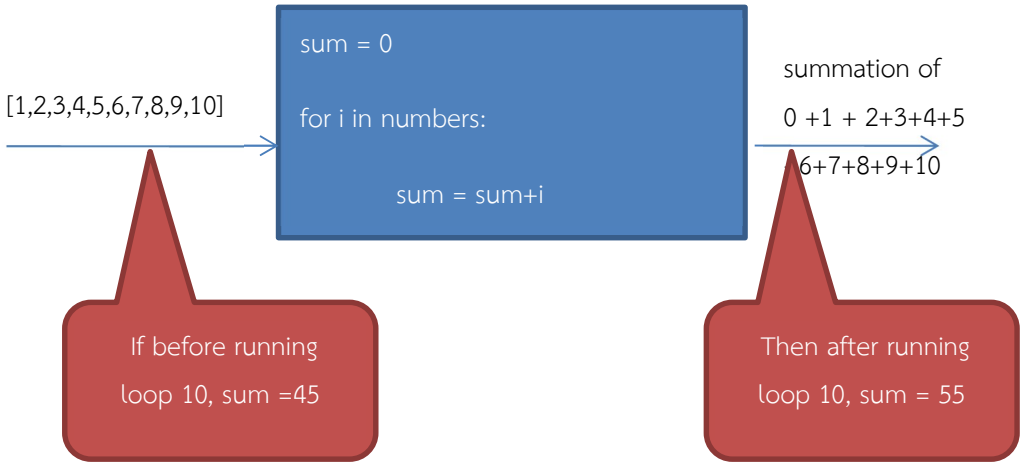
รอบที่ 1



รอบที่ 2



รอบต่อๆไป จนถึง รอบที่ 10



ภาพที่ 3

จากรูปตัวอย่างเราจะพบข้อสังเกตว่า

ก่อนที่จะรันรอบที่ n ใดๆ ค่าผลรวมของตัวเลขจะต้องเป็นผลรวมของตัวเลขตั้งแต่ตัวแรกจนถึงตัวที่ n

Before running loop N, we have sum value of loop N-1

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 จะต้องเห็นว่าเอ๊าพุตของอัลกอริทึมนั้นถูกต้อง

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**

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:

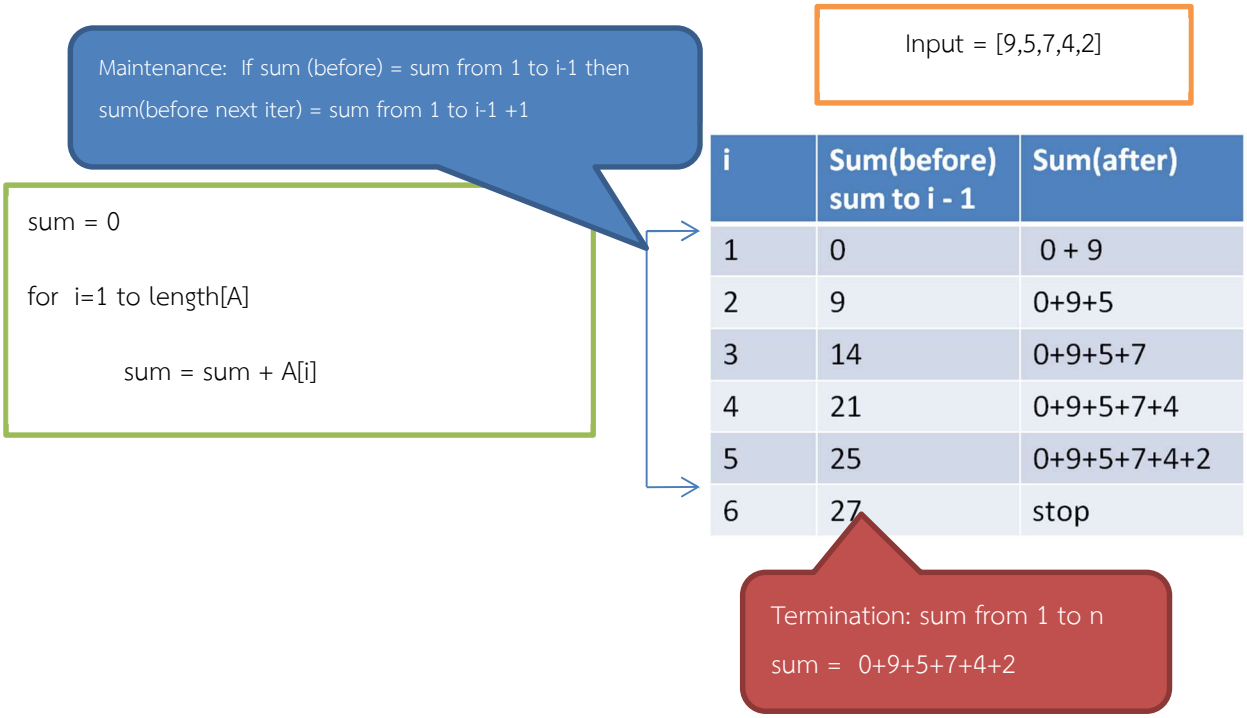
Goal(output of program) => $$sum = \sum_{i=1}^{n} A[i]$$

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

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

ภาพที่ 4

เพื่อความเข้าใจเพิ่มเติม การแสดงตัวอย่างการแปลี่ยนแปลงของตัวแปร ในอัลกอริทึมหาค่าผลรวม นั้นแสดงดังต่อไปนี้



i	Sum(before) sum to i - 1	Sum(after)
1	0	0 + 9
2	9	0+9+5
3	14	0+9+5+7
4	21	0+9+5+7+4
5	25	0+9+5+7+4+2
6	27	stop

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)

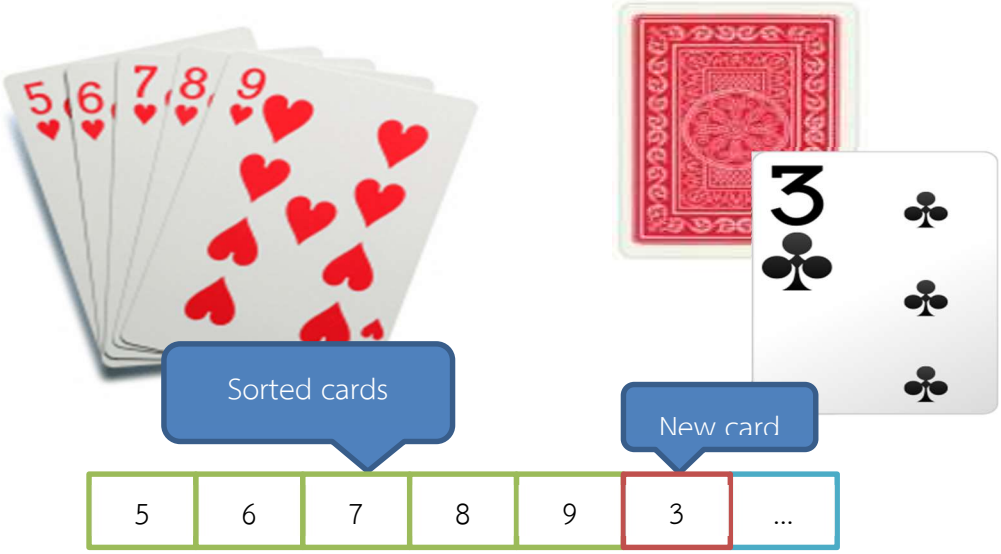
```
import math
pi = 3.14
angle = int(input("Enter an angle in degrees: "))
radian = angle*(pi)/180
print("%d degrees = %.2f radian" % angle, radian)
```

2. ทำการวิเคราะห์ความถูกต้องของอัลกอริทึมที่ทำการหาค่าตัวเลขที่มากที่สุดในอาเรย์ หากกำหนดอัลกอริทึมให้ดังต่อไปนี้

```
max = A[1]

for  i=2 to length[A]

        if max < A[i]

                max = A[i]
```

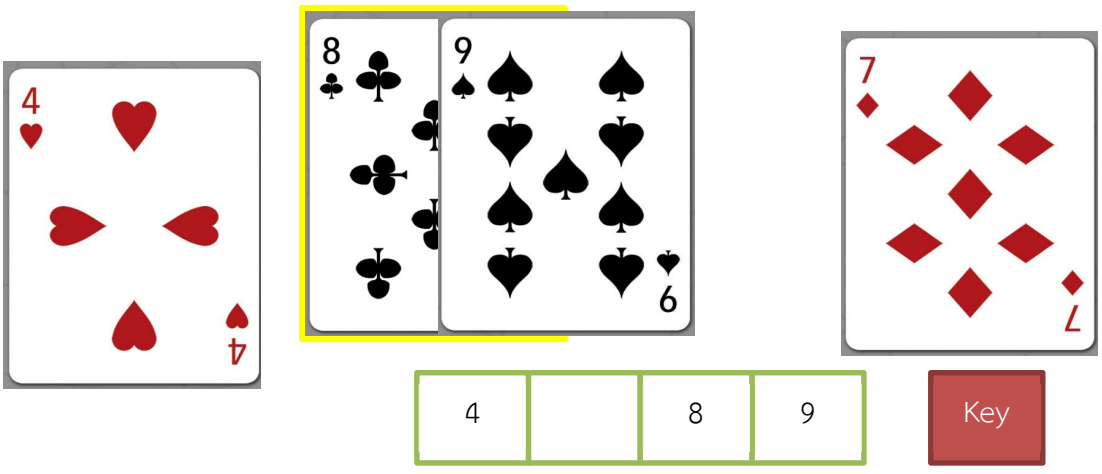
บทที่ ๒ การเรียงลำดับข้อมูลแบบแทรก (insertion sort)

อัลกอริทึมการเรียงลำดับข้อมูลแบบแทรกนั้น มีแนวคิดคล้ายๆกับการเรียงไพ่ แสดงดังรูป**ภาพที่5**



ภาพที่ 5

จะเห็นว่า ไพ่ถูกแบ่งเป็น ไพ่ที่เรียงแล้วในมือ กับไพ่ใบใหม่ ที่เรามีหน้าที่จะต้องหาตำแหน่งแทรกให้ไพ่ใบใหม่ในการเรียงลำดับข้อมูลแบบแทรกก็ใช้หลักการเดียวกัน คือแบ่งข้อมูล เป็นข้อมูลที่เรียงลำดับแล้ว กับข้อมูลใหม่แล้วนำข้อมูลใหม่ 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
```

ตัวอย่างที่ 3 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)

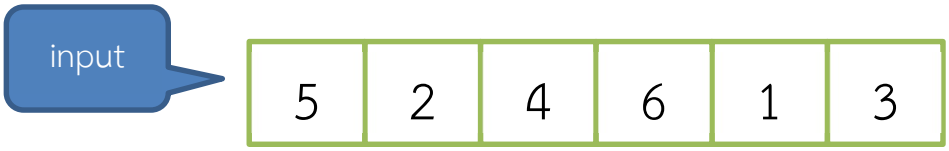
4. แล้วทำการเลื่อนตำแหน่ง i ไปทางซ้าย (ลดค่า i ลง 1 ค่า)

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

พบว่า ค่าในอาเรย์ตำแหน่ง i มีค่าน้อยกว่า key

ให้อาเรย์ตำแหน่งที่ i+1 มีค่าเท่ากับ key (เป็นการแทรกข้อมูล)

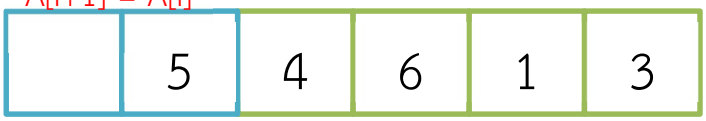
ตัวอย่างลำดับขั้นตอนแสดงดัง**ภาพที่ 7**



j = 2 Key i=1

A[i] = 5 > key = 2

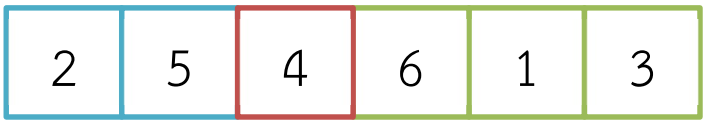
A[i+1] = A[i]



A[i+1]= key

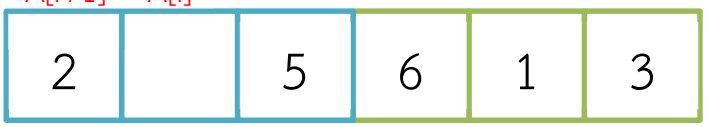
i=0

j = 3 Key i=2



A[i] = 5 > key = 4 ?

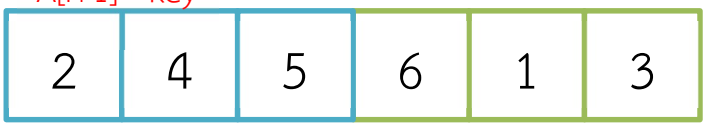
A[i+1] = A[i]



i=1

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

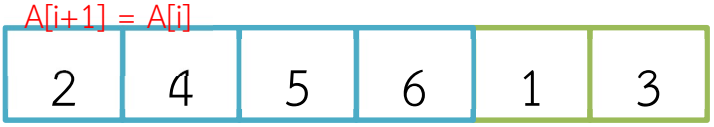
A[i+1]= key



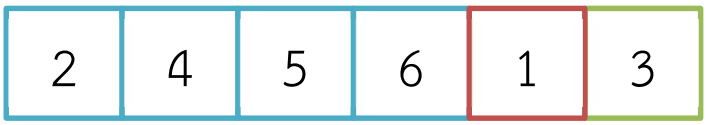
j = 4 Key i=3

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

A[i+1] = A[i]



j = 5 Key i=4



A[i] = 6 > key = 1

A[i+1] = A[i]

i=3

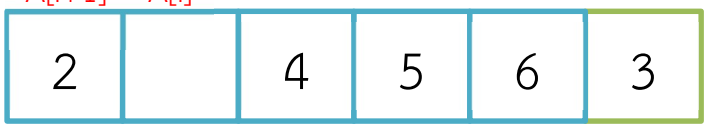
A[i] = 5 > key = 1

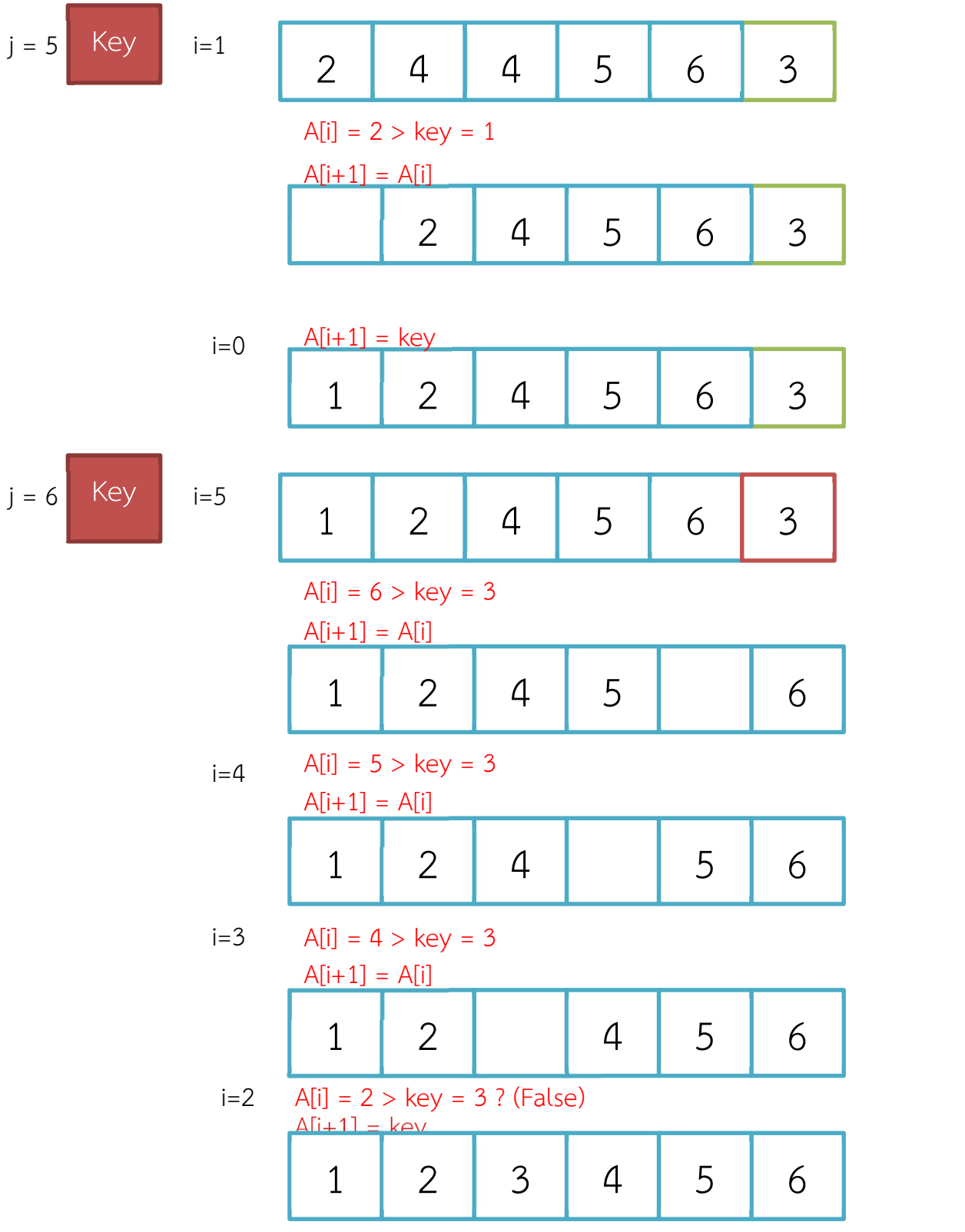
A[i+1] = A[i]

i=2

A[i] = 4 > key = 1

A[i+1] = A[i]




j = 5
Key
i=1
2 4 4 5 6 3
A[i] = 2 > key = 1
A[i+1] = A[i]
2 4 5 6 3
i=0
A[i+1] = key
1 2 4 5 6 3
j = 6
Key
i=5
1 2 4 5 6 3
A[i] = 6 > key = 3
A[i+1] = A[i]
1 2 4 5 6
i=4
A[i] = 5 > key = 3
A[i+1] = A[i]
1 2 4 5 6
i=3
A[i] = 4 > key = 3
A[i+1] = A[i]
1 2 4 5 6
i=2
A[i] = 2 > key = 3 ? (False)
1 2 3 4 5 6

ภาพที่ 7

<u>แบบฝึกหัด</u>

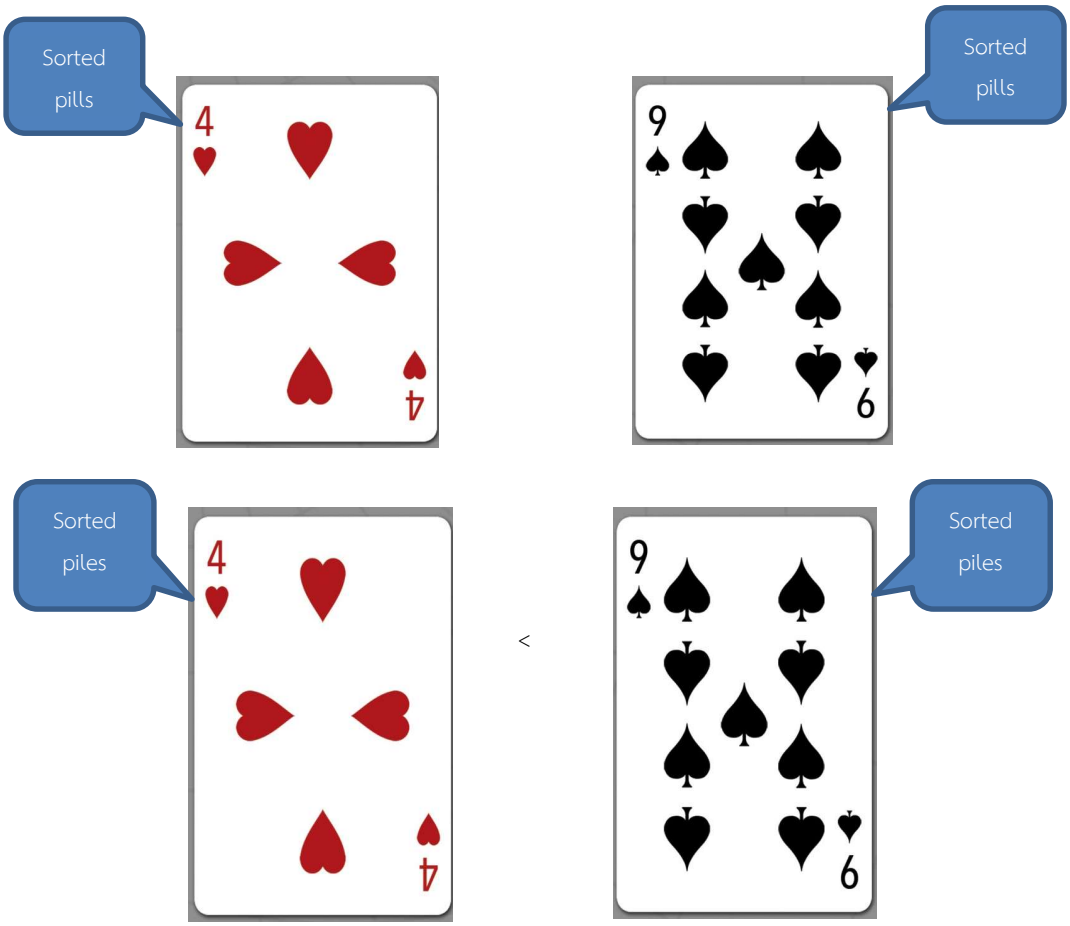
1. จงแสดงวิธีการเรียงลำดับข้อมูลด้วยวิธี insertion sort เมื่อกำหนดข้อมูลให้คือ 9, 5, 7, 4, 2 เรียงลำดับจากน้อยไปมาก

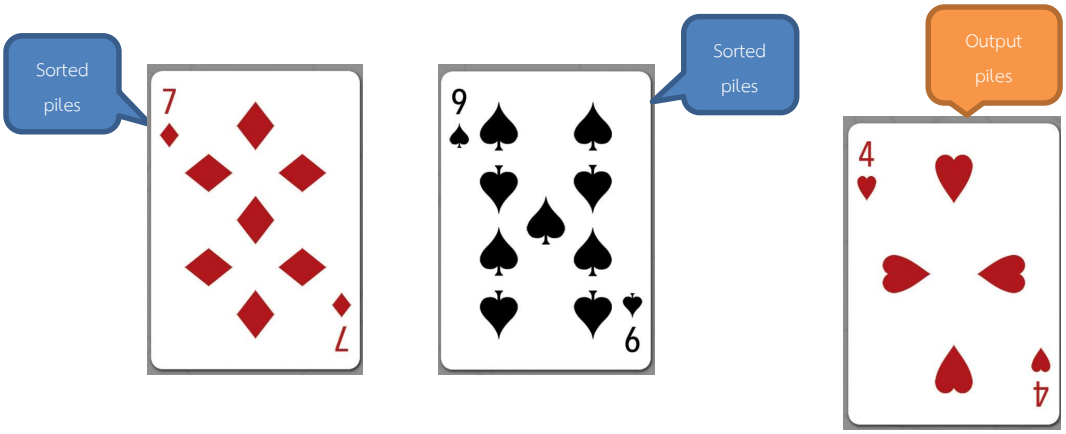
2. จงแสดงวิธีการเรียงลำดับข้อมูลด้วยวิธี insertion sort เมื่อกำหนดข้อมูลให้คือ 2, 1, 0, 1, 2, 5, 6, 2 เรียงลำดับจากน้อยไปมาก

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)

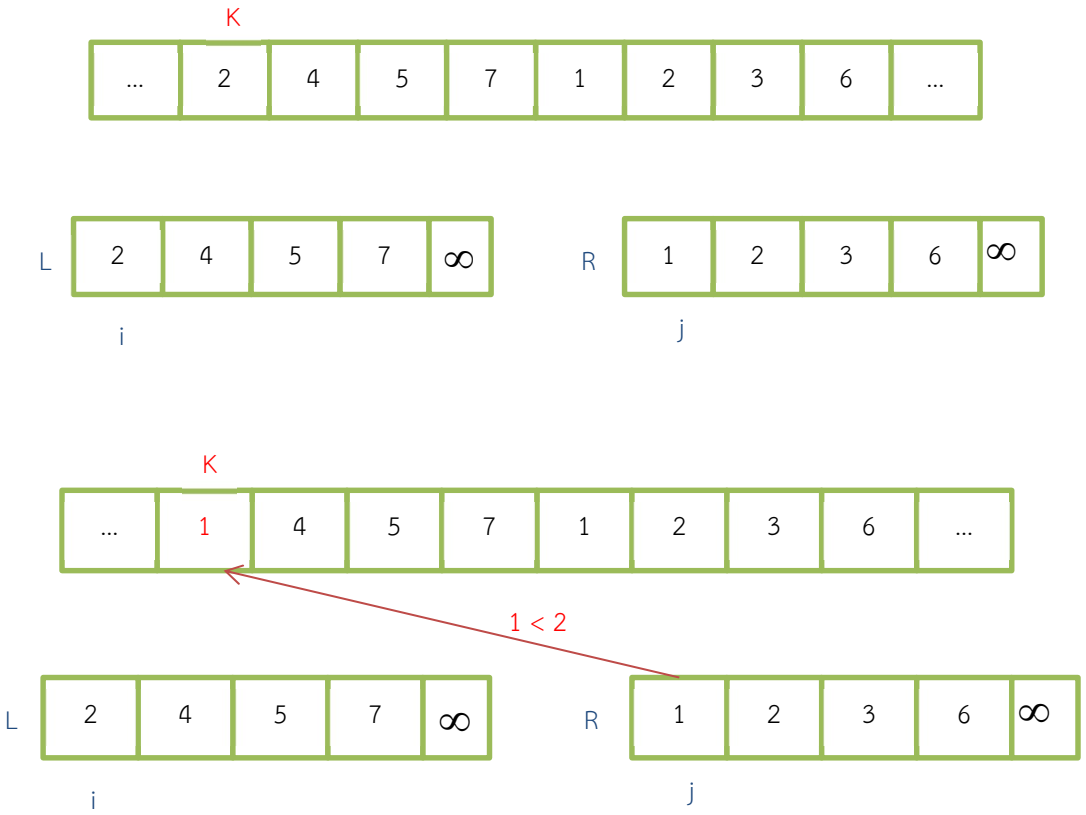
```
If p < r
        then q = L(p+r)/2
                merge-sort(A, p, q)
                merge-sort(A, q+1, r)
                merge(A, p, q, r)
```

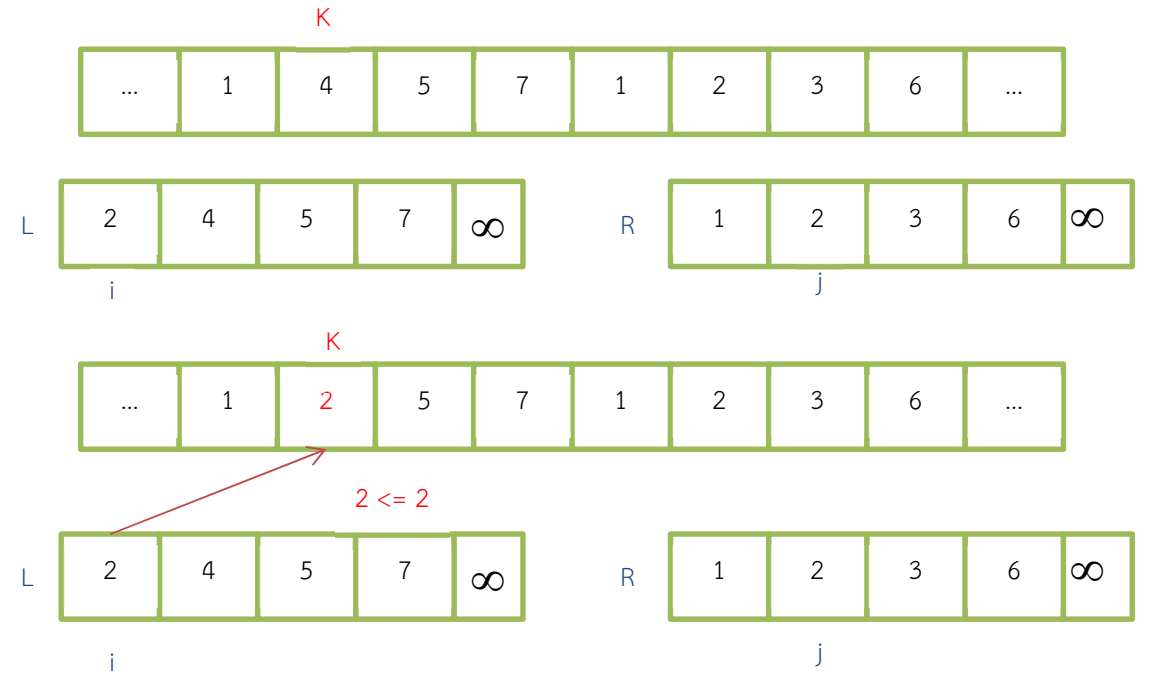
ซึ่งอัลกอริทึมฟังก์ชัน merge สามารถเขียนเป็น pseudo code ได้ดังนี้

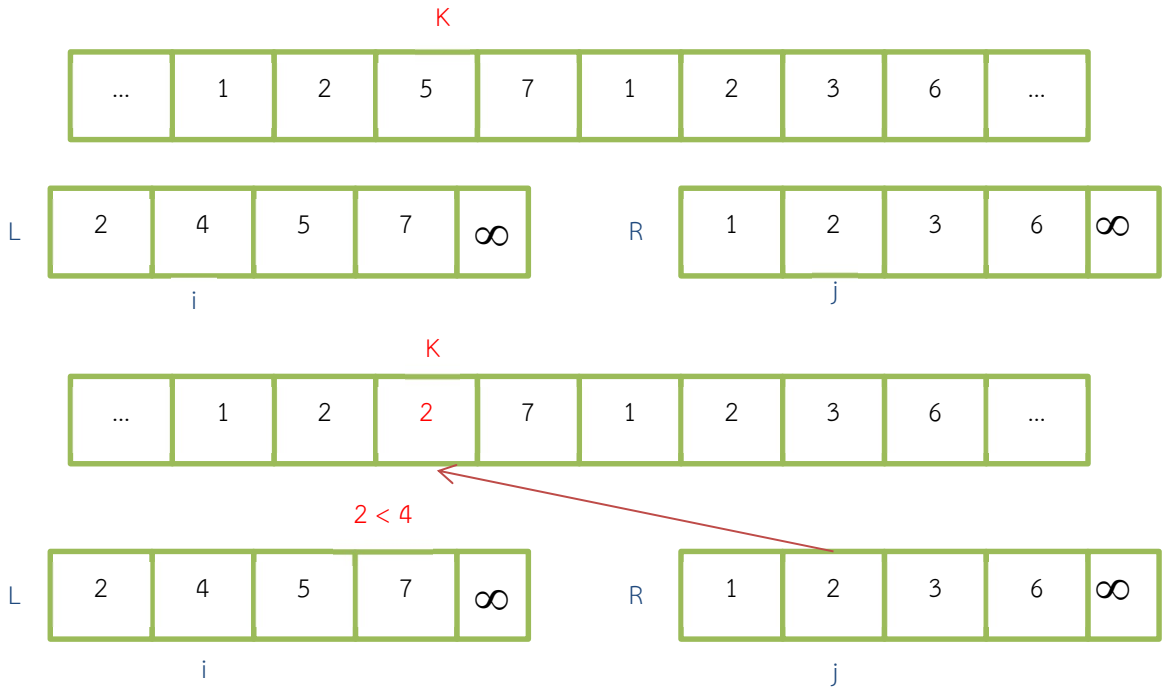
Psudocode: merge(A,p,q,r)

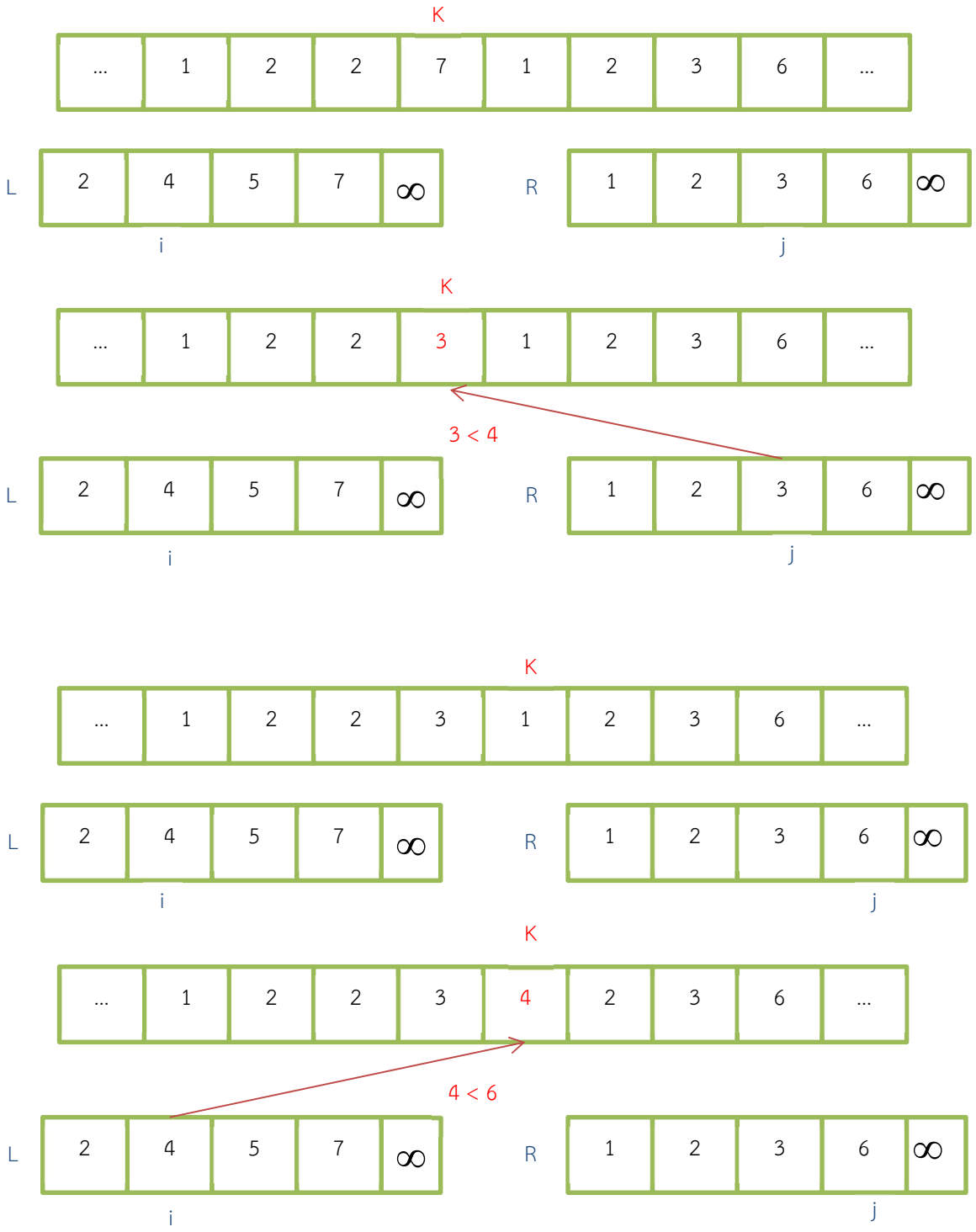
```
N1 = q – p + 1

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
```

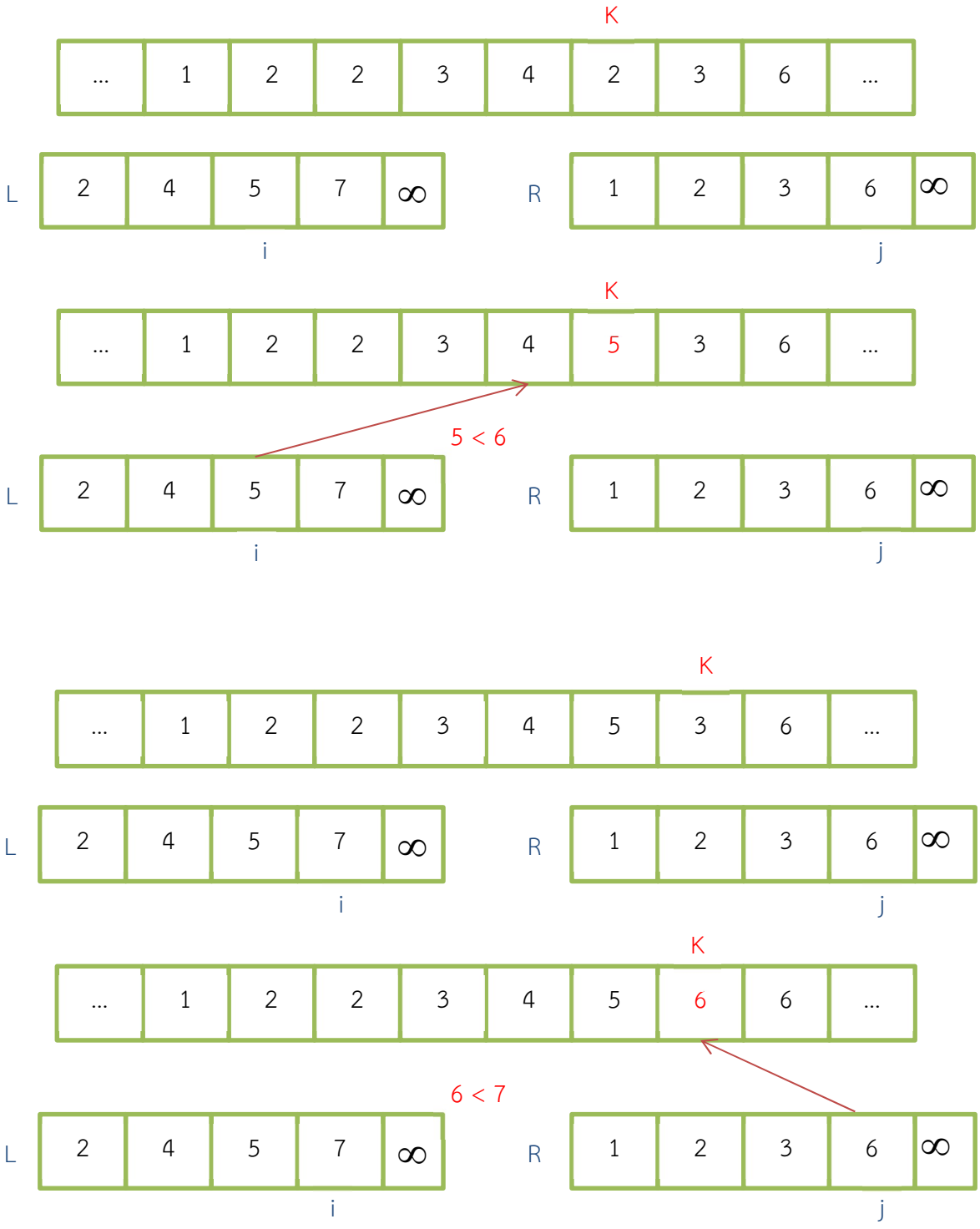

<u>ภาพที่ 9</u> แสดงตัวอย่างการทำงานของอัลกอริทึมฟังก์ชัน merge โดยสมมติให้มีอาเรย์ที่ยังไม่ได้เรียงข้อมูล เป็นอาเรย์ขนาด 8 ตัว ดังนั้นจะเริ่มจากการแบ่งอาเรย์ออกเป็น 2 อาเรย์ที่มีขนาดอันละ 4 ตัว แล้วนำอาเรย์ย่อย (ที่มีขนาด 4 ตัว) นำไปเรียงลำดับข้อมูล โดยใช้อัลกอริทึมฟังก์ชัน merge

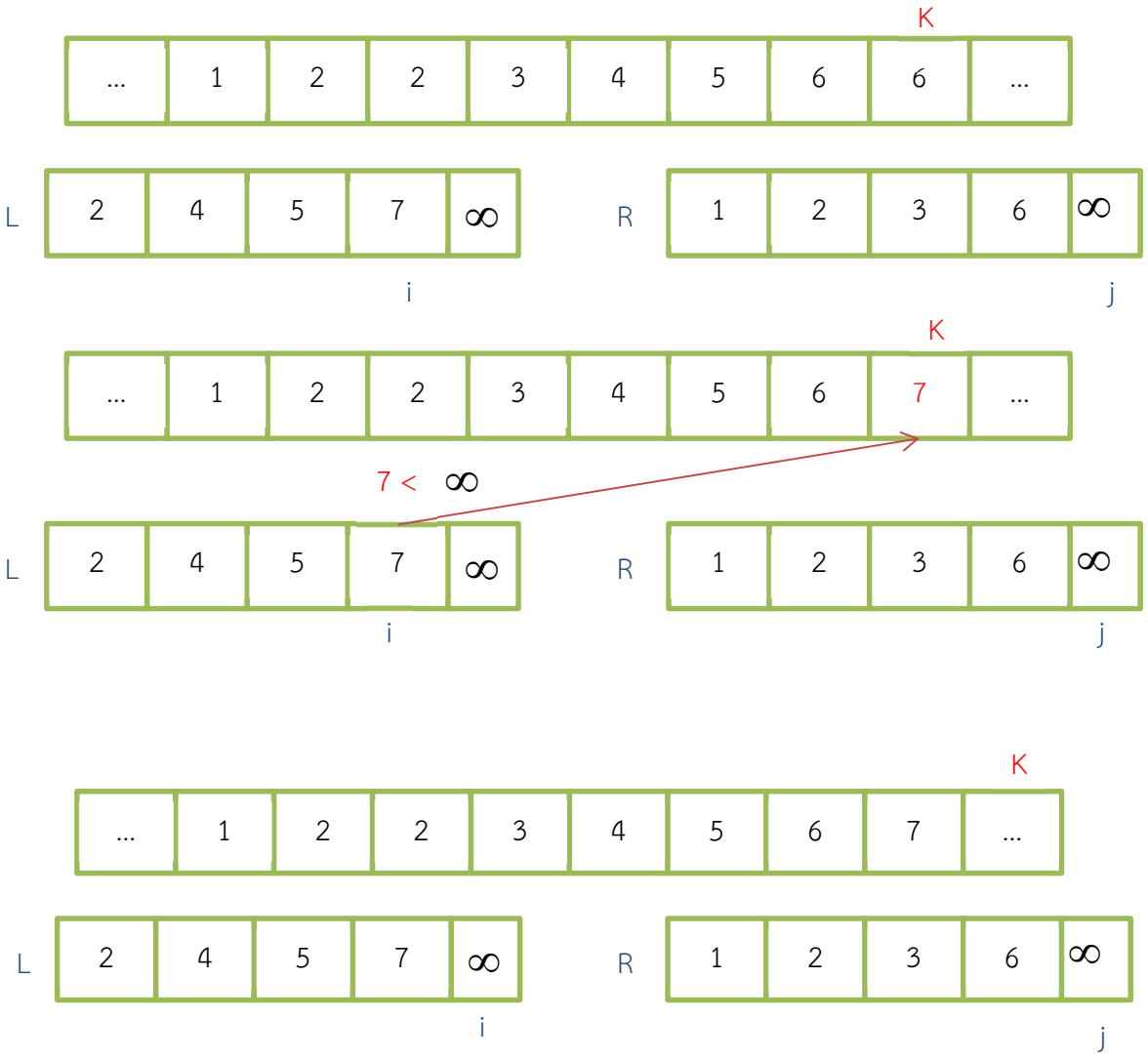



K
... 1 4 5 7 1 2 3 6 ...
L
2 4 5 7 ∞
R
1 2 3 6 ∞
i
j
K
... 1 2 5 7 1 2 3 6 ...
2 <= 2
L
2 4 5 7 ∞
R
1 2 3 6 ∞
i
j


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



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

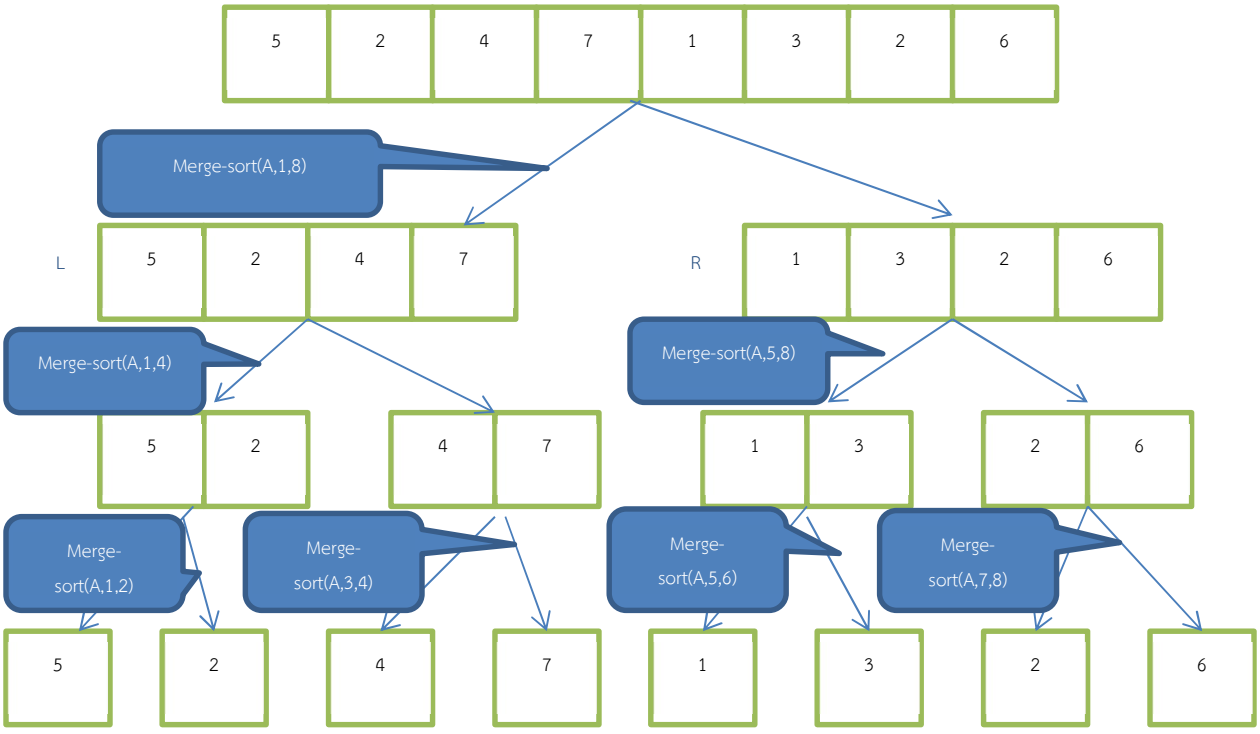
เนื่องจากมีอาเรย์ย่อยขนาด 2 ตัวสี่กลุ่ม ดังนั้นทั้งหมด เราจะได้อาเรย์ขนาด 1 ตัวจำนวน 8 กลุ่ม

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

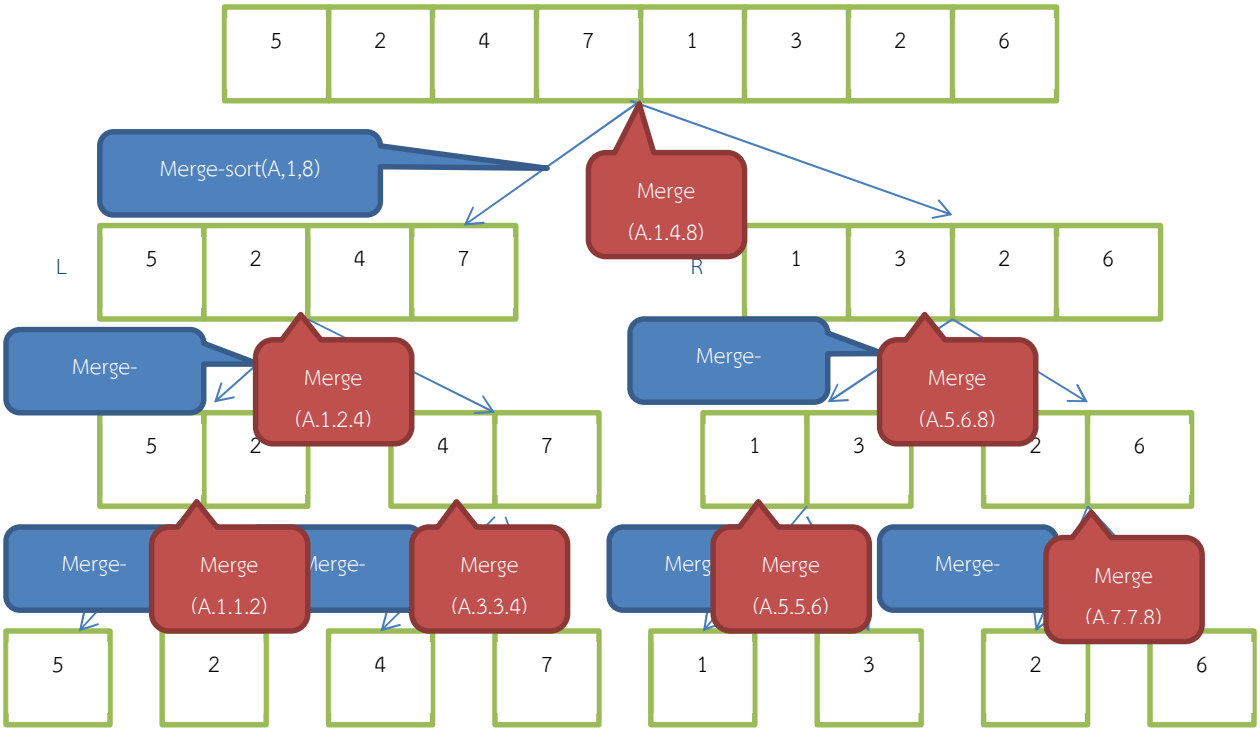
5. ทำการเรียงข้อมูลและผสานอาเรย์ย่อย โดยใช้ฟังก์ชัน merge ผสาน อาเรย์ขนาด 2 ตัว ทีละสองกลุ่ม ดังนั้น ผลลัพธ์ทั้งหมด จะได้อาเรย์ที่เรียงลำดับข้อมูลแล้ว เป็นอาเรย์ขนาด 4 ตัว จำนวน 2 กลุ่ม

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

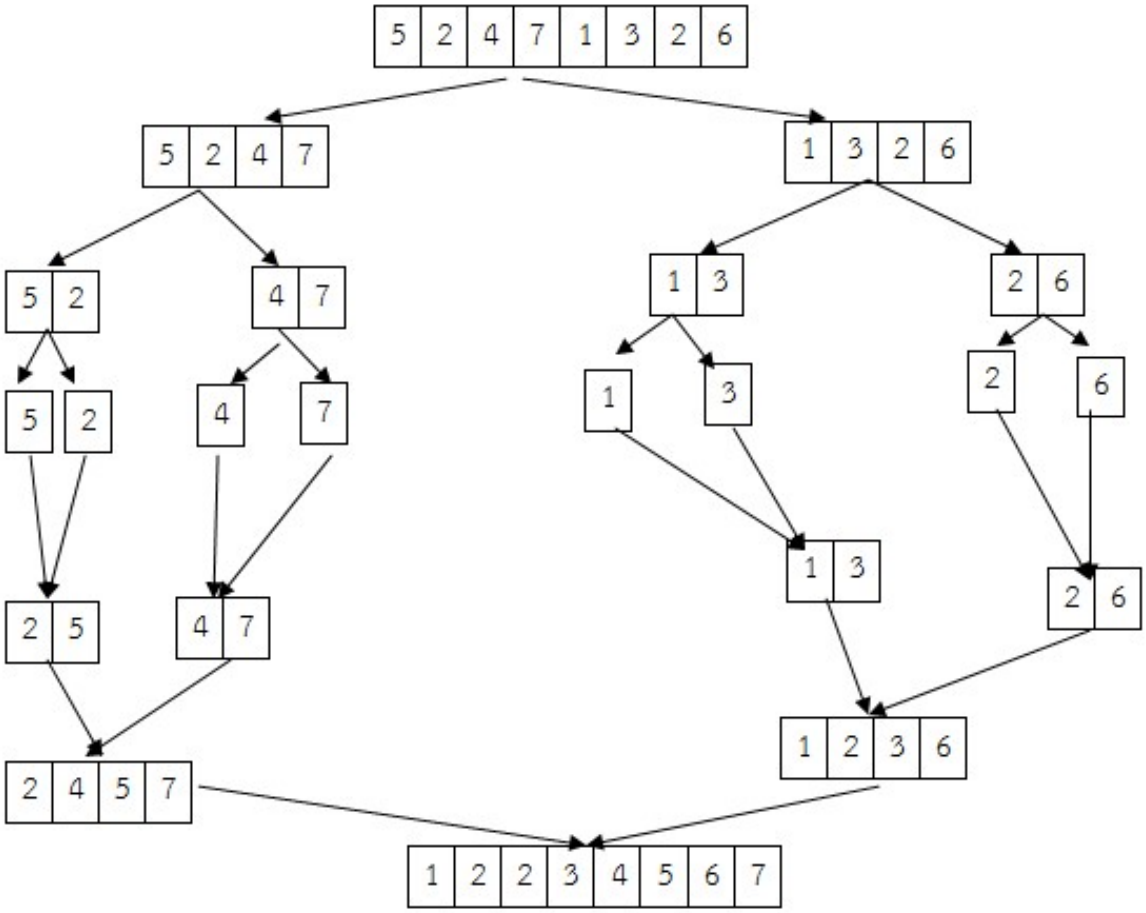
ภาพต่อไปนี้แสดงลำดับการเรียกใช้อัลกอริทึม merge-sort เพื่อแบ่งอาเรย์และเรียงลำดับข้อมูล



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

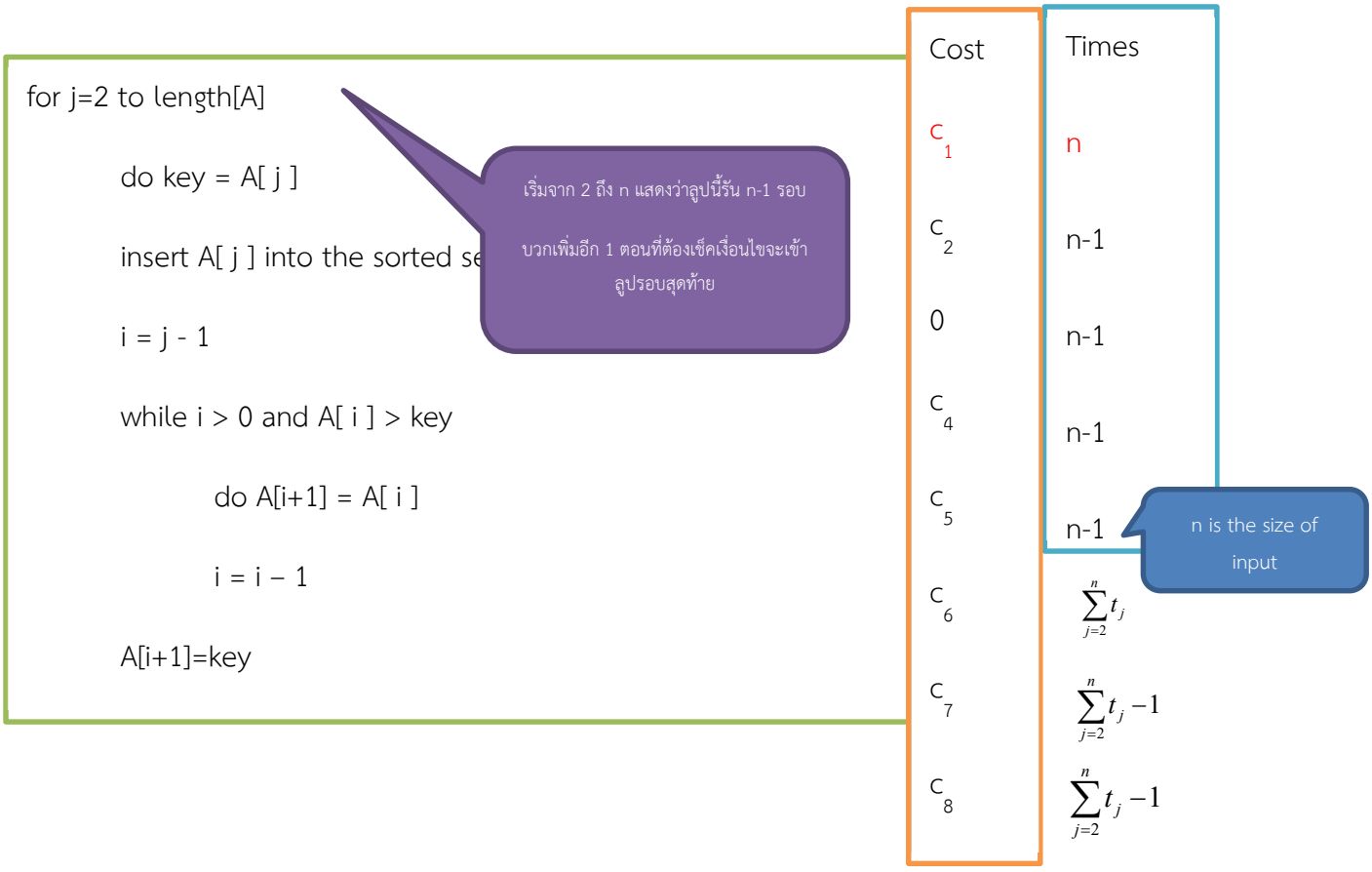
1. ทำการกำหนดชื่อให้แก่ pseudo code แต่ละบรรทัดของอัลกอริทึม (เอาไว้ใช้กำหนดเป็นชื่อตัวแปรของค่าคงที่ในการคำนวณ)

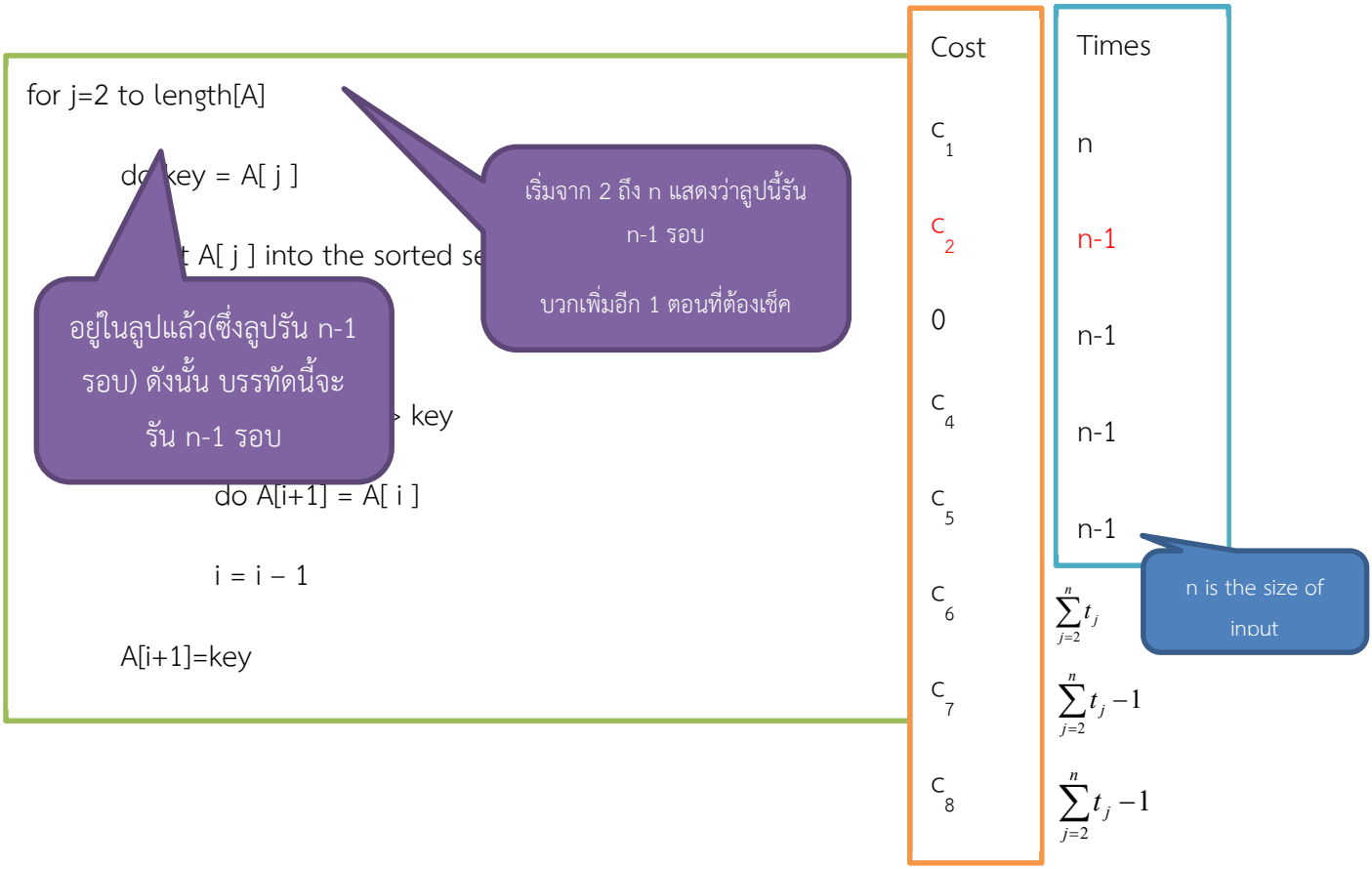
2. ทำการคำนวณเวลาที่ใช้ในการประมวลผลของโค้ดแต่ละบรรทัด โดยเทคนิคการเติบโตของฟังก์ชัน (growth of function) - หาก เป็นเพียงโค้ดที่ไม่ได้อยู่ในลูป ก็ใช้ growth of function คือ ค่าคงที่

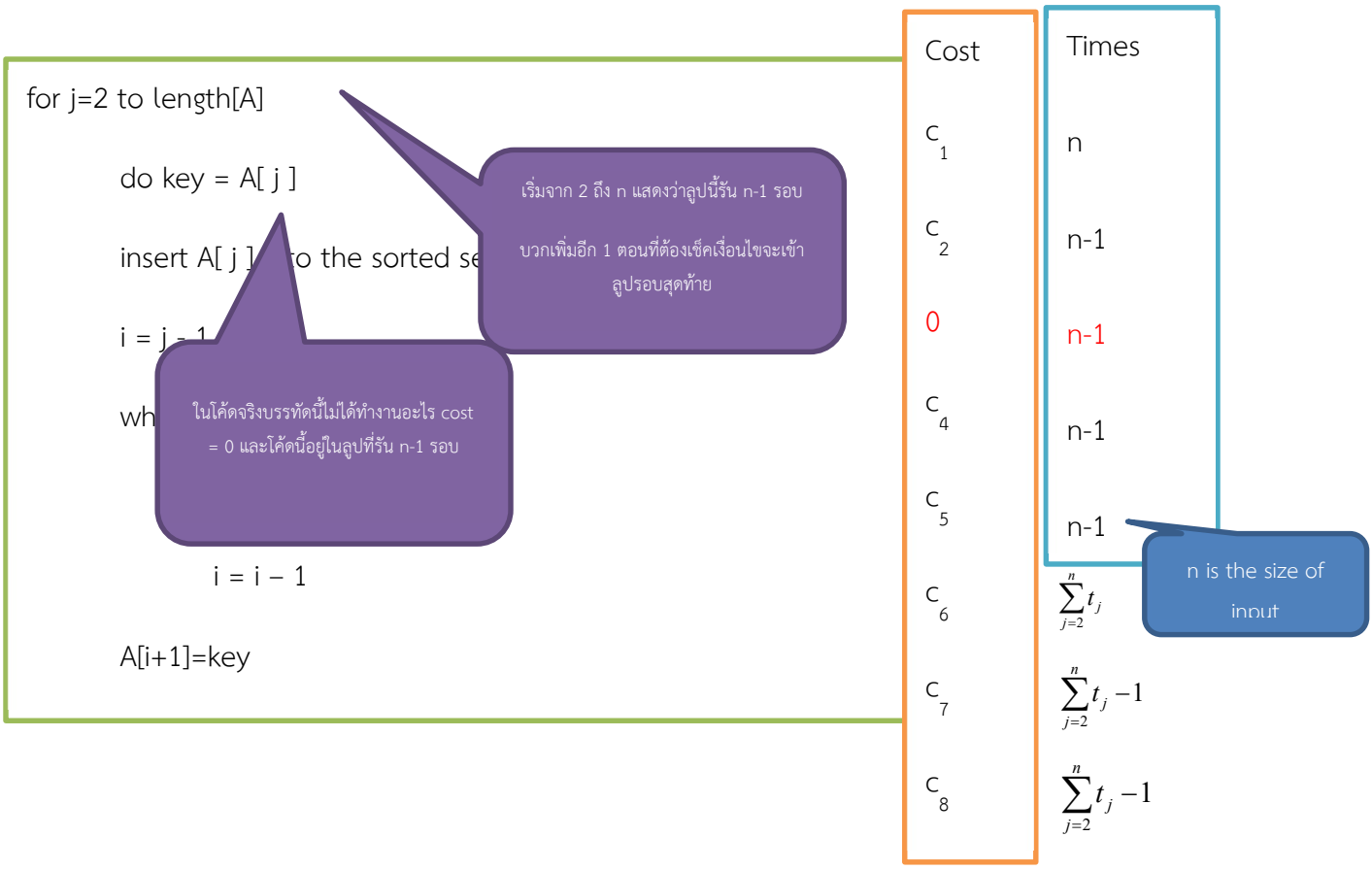
- หากเป็นโค้ดที่อยู่ในลูป ก็ใช้ growth of function ที่เท่ากับขนาดของอินพุต (นับว่าอย่างมากสุดต้องวนลูปกี่รอบ)

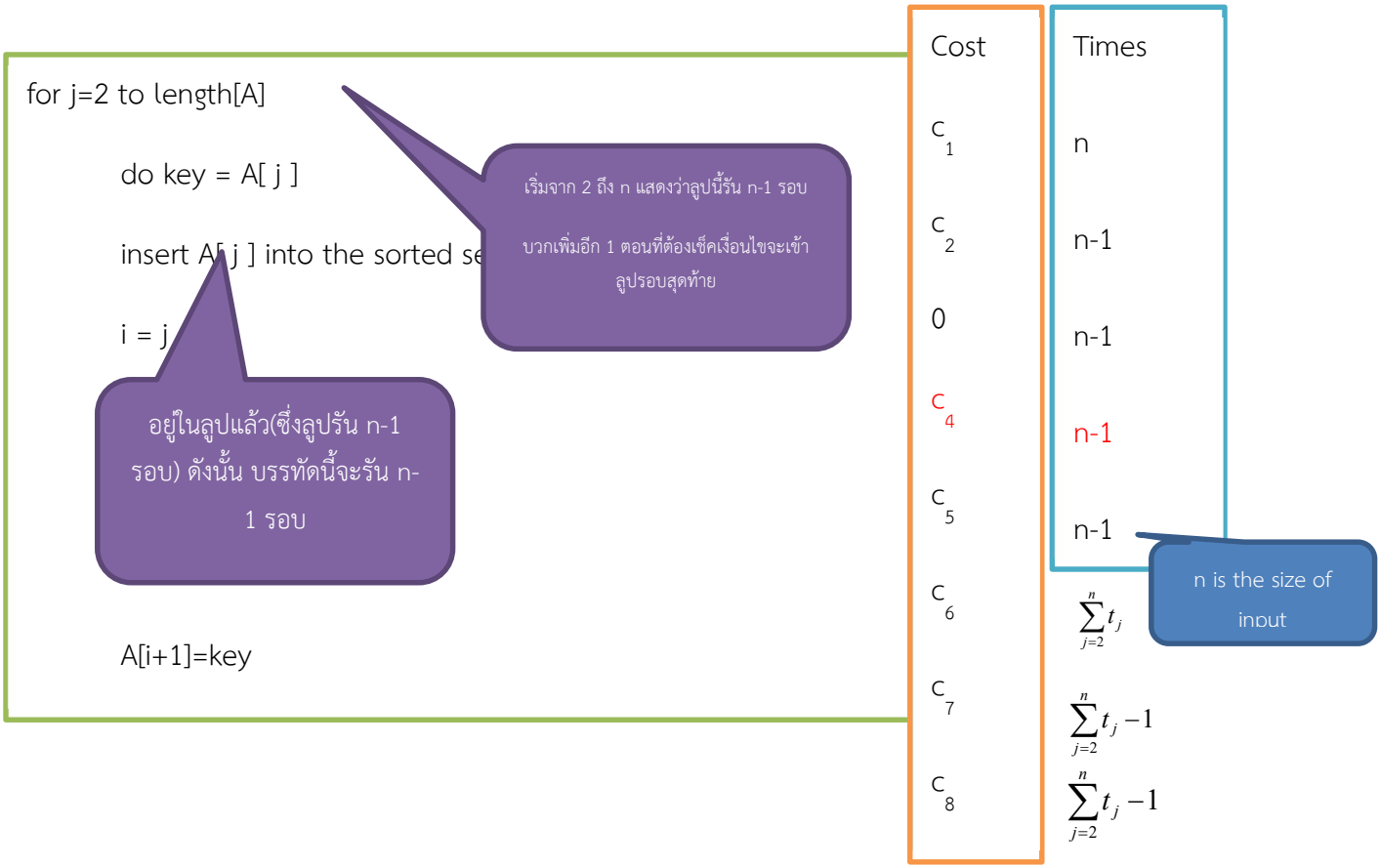
3. วิเคราะห์หาขอบเขตของเวลาที่ต้องการใช้ประมวลผลมากที่สุด โดยนำค่าคงที่ในข้อ 1 คูณกับเวลาในข้อ 2 หาผลรวมของทั้งหมด จะได้เวลาที่คาดว่าต้องใช้ในการคำนวณอัลกอริทึมนี้

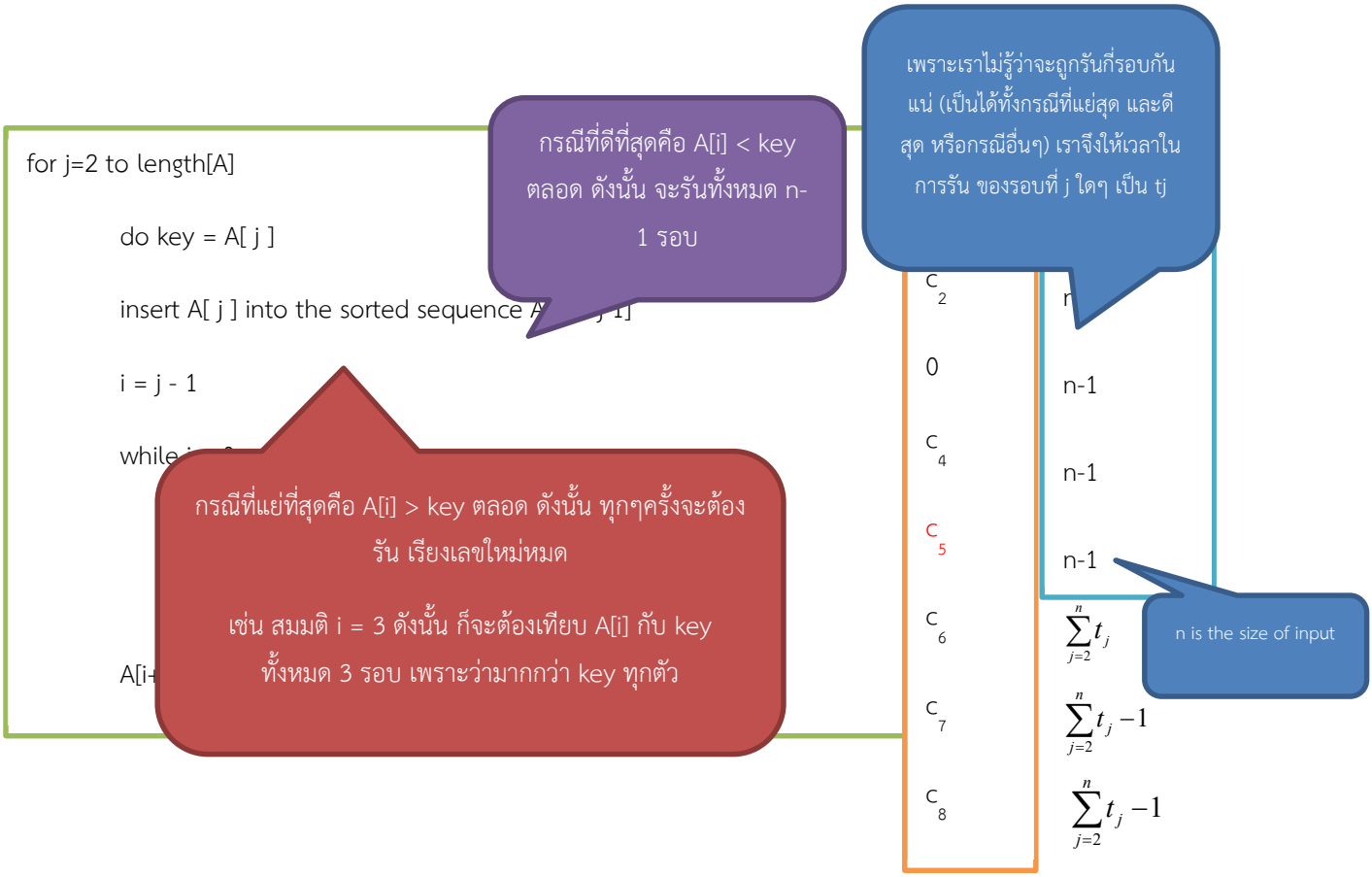
ดังนั้นขั้นตอน การวิเคราะห์ running time ของอัลกอริทึม insertion sort จะเป็นดังต่อไปนี้

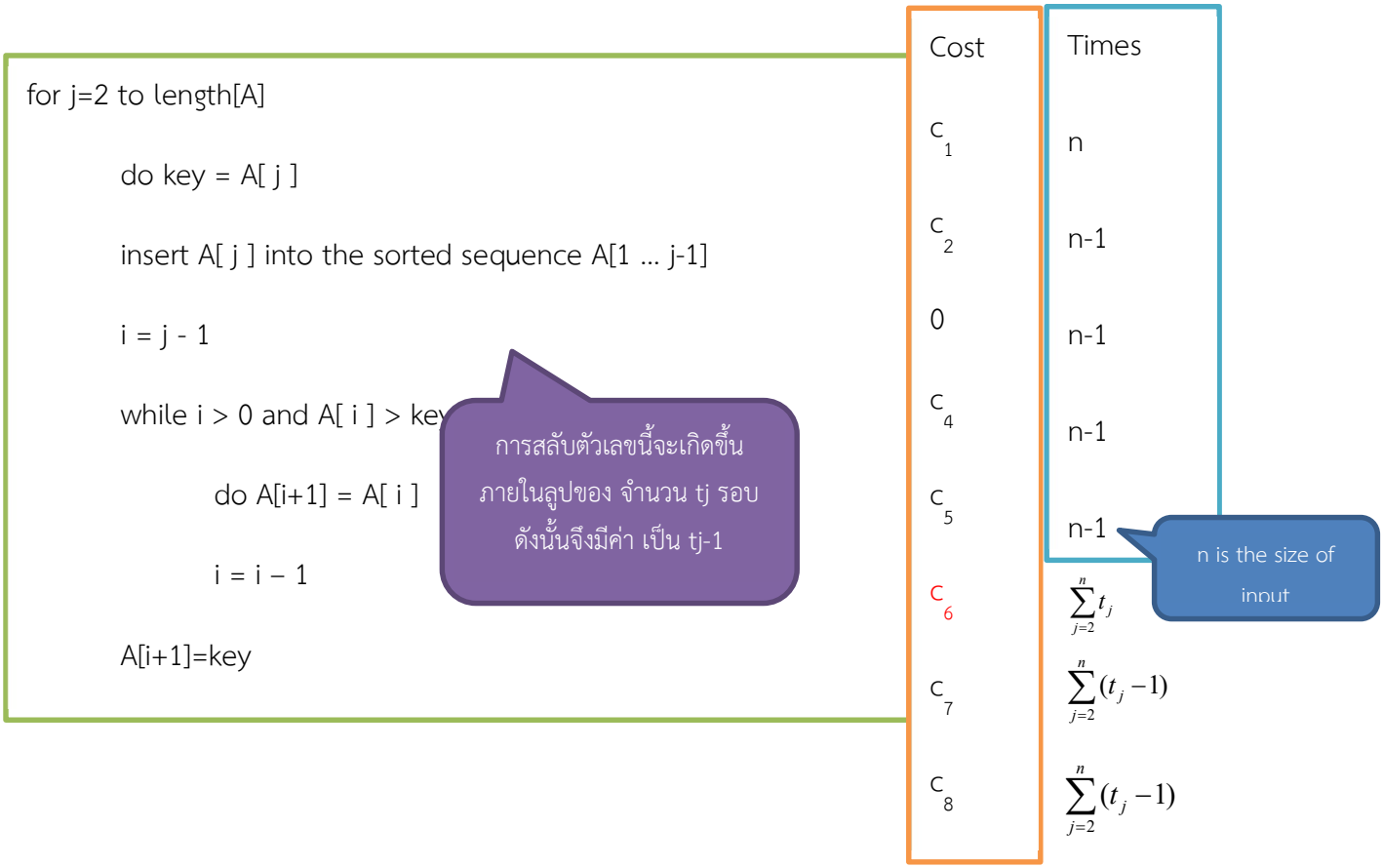


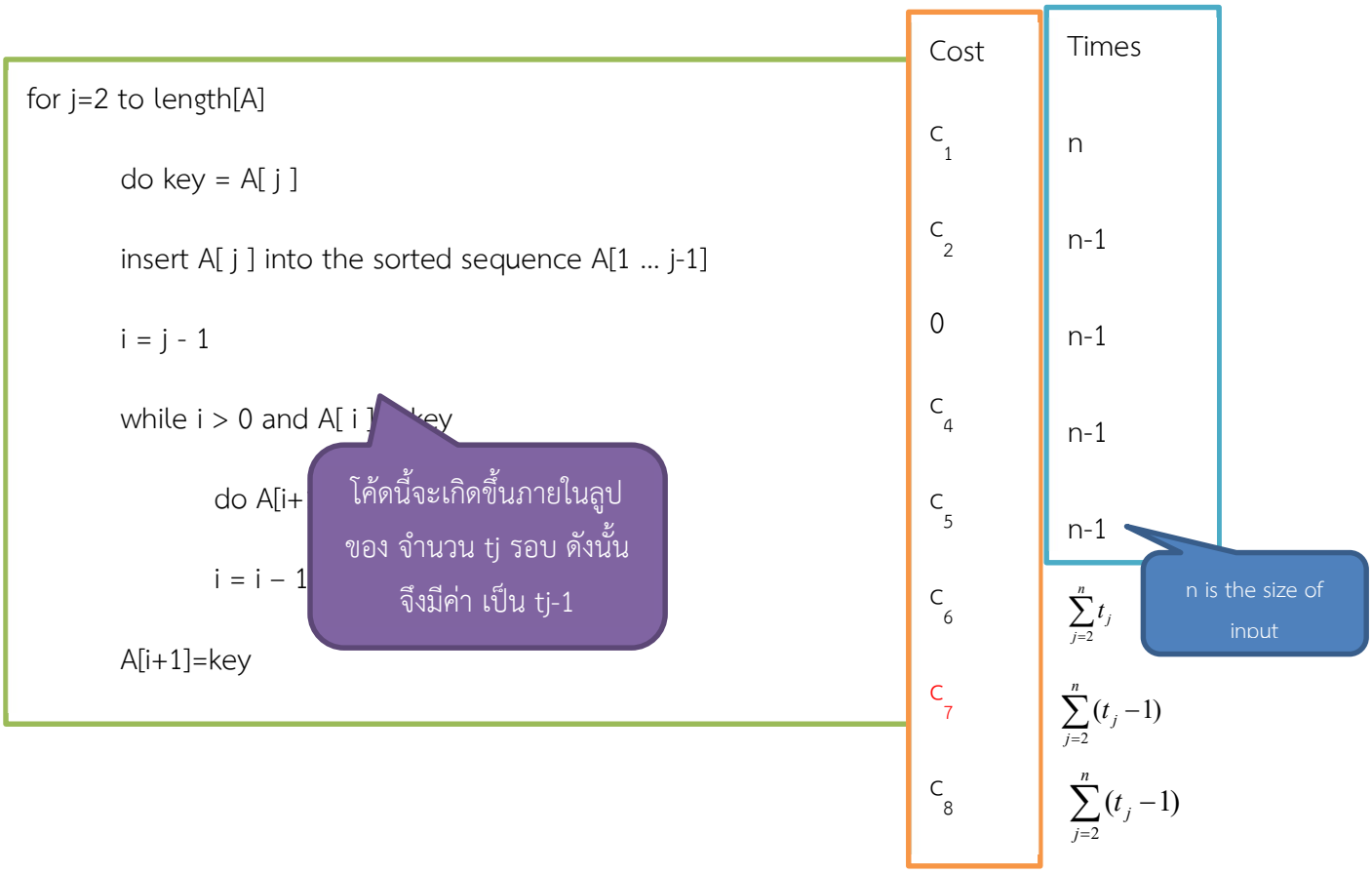

for j=2 to length[A]
do key = A[j]
t A[j] into the sorted se
key
do A[i+1] = A[i]
i = i – 1
A[i+1]=key
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ
บวกเพิ่มอีก 1 ตอนที่ต้องเช็ค
อยู่ในลูปแล้ว(ซึ่งลูปรัน n-1 รอบ) ดังนั้น บรรทัดนี้จะรัน n-1 รอบ
Cost
c1
c2
0
c4
c5
c6
c7
c8
Times
n
n-1
n-1
n-1
n-1
$\sum_{j=2}^{n} t_j$
$\sum_{j=2}^{n} t_j - 1$
$\sum_{j=2}^{n} t_j - 1$
n is the size of input

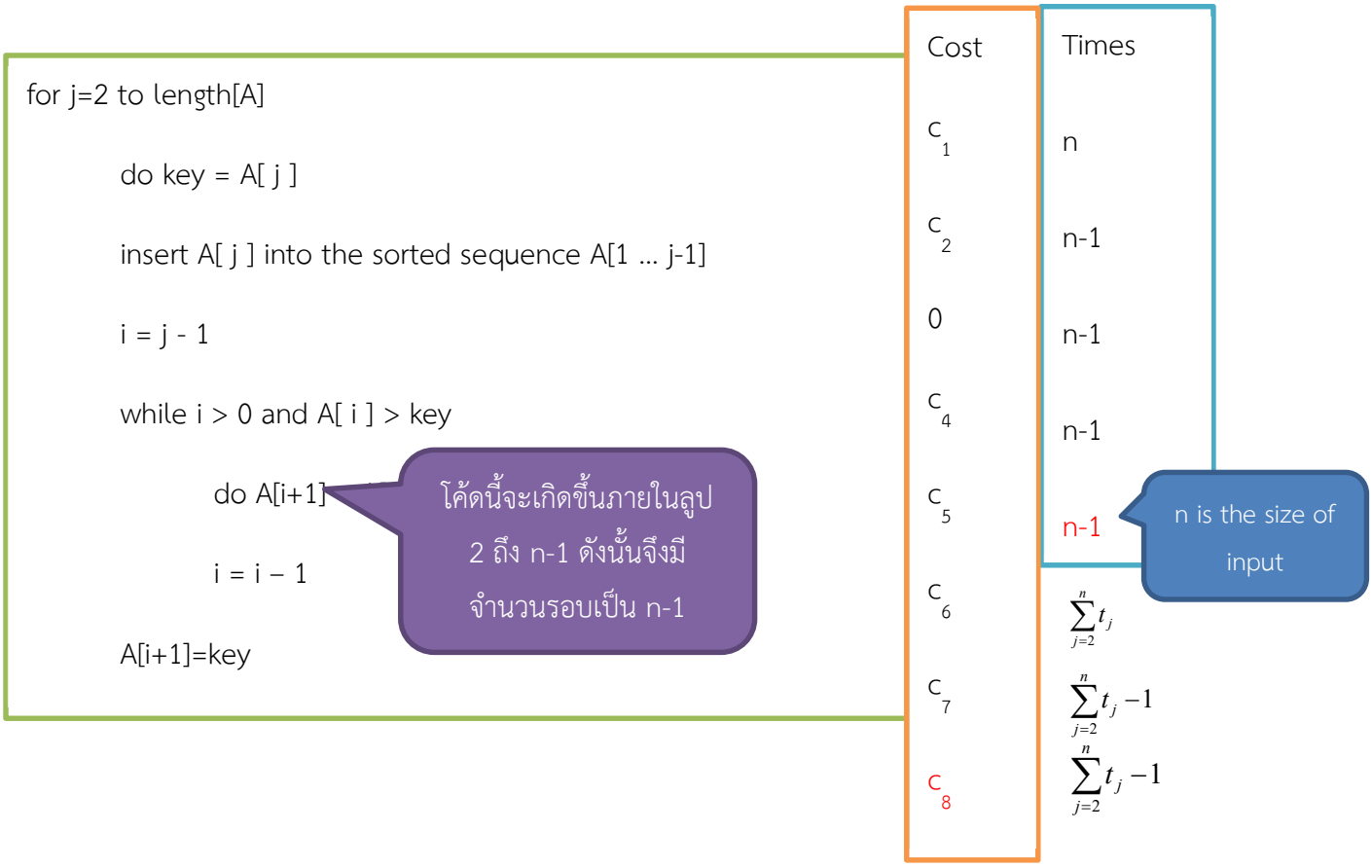

for j=2 to length[A]
do key = A[j]
insert A[j] to the sorted se
i = j - 1
wh
i = i – 1
A[i+1]=key
เริ่มจาก 2 ถึง n แสดงว่าลูปนี้รัน n-1 รอบ
บวกเพิ่มอีก 1 ตอนที่ต้องเช็คเงื่อนไขจะเข้าลูปรอบสุดท้าย
ในโค้ดจริงบรรทัดนี้ไม่ได้ทำงานอะไร cost = 0 และโค้ดนี้อยู่ในลูปที่รัน n-1 รอบ
Cost
C1
C2
0
C4
C5
C6
C7
C8
Times
n
n-1
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 is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the sorted se
i = j
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
C1
C2
0
C4
C5
C6
C7
C8
Times
n
n-1
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 is the size of input


for j=2 to length[A]
do key = A[j]
insert A[j] into the sorted sequence A
i = j - 1
while
A[i+
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 ทุกๆครั้งจะต้องรัน เรียงเลขใหม่หมด
เช่น สมมติ i = 3 ดังนั้น ก็จะต้องเทียบ A[i] กับ key ทั้งหมด 3 รอบ เพราะว่ามากกว่า key ทุกตัว
C2
0
C4
C5
C6
C7
C8
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 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
การสลับตัวเลขนี้จะเกิดขึ้นภายในลูปของ จำนวน tj รอบ ดังนั้นจึงมีค่า เป็น tj-1
Cost
c1
c2
0
c4
c5
C6
c7
c8
Times
n
n-1
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 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]
do A[i+
i = i – 1
A[i+1]=key
โค้ดนี้จะเกิดขึ้นภายในลูป ของ จำนวน tj รอบ ดังนั้น จึงมีค่า เป็น tj-1
Cost
C1
C2
0
C4
C5
C6
C7
C8
Times
n
n-1
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 is the size of input



ดังนั้น **running time ของอัลกอริทึม insertion sort** จะเป็นดังสมการต่อไปนี้

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

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

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

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

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 และเป็นเป็นตัวเลขที่เรียงลำดับกันเรียบร้อยแล้ว 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 และเป็นเป็นตัวเลขที่เรียงลำดับกัน

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

อ้างอิงจากขั้นตอน maintenance ก็จะพบว่า เราได้พิสูจน์ไว้แล้วว่า ข้อความข้างต้นเป็นจริง

จบการพิสูจน์ ดังนั้น อัลกอริทึมนี้ถูกต้อง

<u>หมายเหตุ</u> 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)

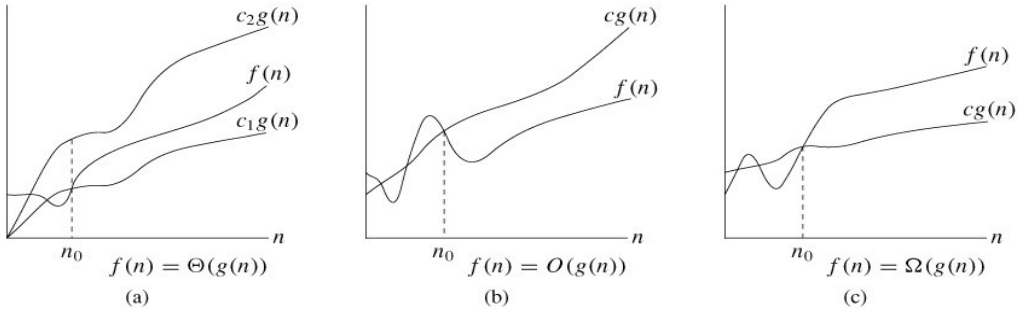
2.Big-O (O)

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 $f(n) \in o(g(n))$

ตัวอย่างที่ 7 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 $g(n) = n$

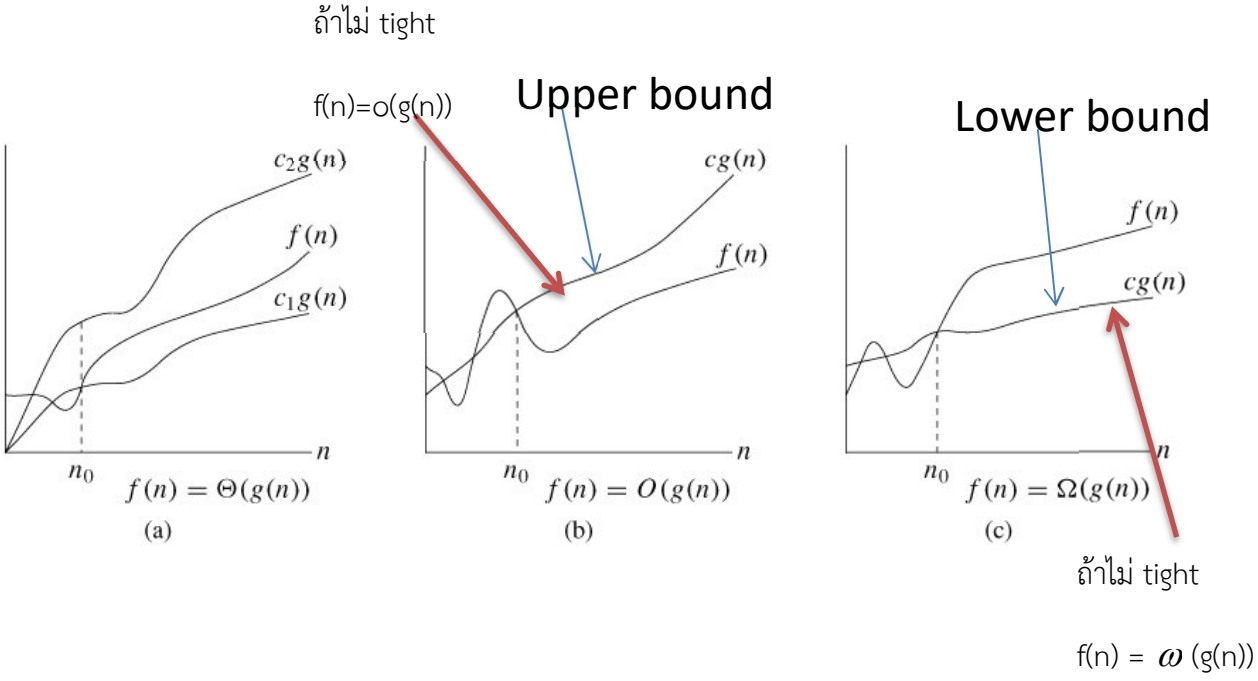
วิธีทำ เราต้องทำการพิสูจน์ว่า เราสามารถหาค่าคงที่ c และ n_0 ที่ทำให้สมการ $0 \le c\, g(n) < f(n)$ เป็นจริงสำหรับ n ทุกตัวที่มีค่ามากกว่า n_0

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**<u>ภาพที่ 11</u>** อธิบายความสัมพันธ์ของ asymptotic notation ทั้งห้าแบบ



ภาพที่ 11

<u>แบบฝึกหัด</u>

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 $g(n) = n^2$

บทที่ ๗ การเวียนบังเกิด (recurrence)

การเวียนบังเกิด (recurrence) คือสมการหรือความไม่เท่ากันที่ซึ่งถูกใช้ในการอธิบายฟังก์ชันให้อยู่ในรูปของค่าของอินพุตที่มีขนาดไม่มาก เช่น จากสมการที่อธิบายฟังก์ชัน T(N) ทั้งสองเงื่อนไขนี้

$$T(N) = \Theta(1) \quad if\ n = 1$$
$$T(N) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad if\ n > 1$$

เราสามารถเขียนได้ว่า $T(N) = \Theta(n \lg n)$

วิธีการที่ใช้สำหรับแก้ปัญหาของ recurrence มีอยู่ 3 วิธีดังนี้

1. Substitution method
2. Recursion tree method
3. Master method

วิธีที่ 3 Master method

มีไว้สำหรับแก้ปัญหา recurrence ที่มีลักษณะฟังก์ชันอยู่ในรูปแบบ

$$T(N) = a\,T\left(\frac{n}{b}\right) + f(n)$$

โดยที่ a,b เป็นค่าคงที่ a ≥ 1 และ b > 1 และ f(n) เป็น asymptotically positive function

ซึ่ง recurrence ที่มีรูปแบบนี้ อธิบายถึงเวลาที่ใช้ในการรันของอัลกอริทึม ที่มีการแบ่งปัญหาขนาด n ใดๆ ออกเป็นปัญหาย่อย ที่แต่ละอันมีขนาด n/b แทน

สำหรับวิธีการแก้ปัญหา recurrence วิธีนี้มีกฎเกณฑ์ที่เรียกว่า **Master theorem** เพื่อใช้วิเคราะห์อยู่ 3 กรณี นิยามไว้ดังนี้

กำหนดให้ a ≥ 1 และ b > 1 เป็นค่าคงที่ใดๆ และให้ f(n) เป็นฟังก์ชัน และให้ T(n) เป็น recurrence ที่มีลักษณะดังนี้ $T(N) = a\,T\left(\frac{n}{b}\right) + f(n)$

ซึ่งค่า n/b สามารถตีความได้ว่าเป็นค่าขอบเขตบนของ n/b $\left(\left\lceil\frac{n}{b}\right\rceil\right)$ หรือ ขอบเขตล่างของ n/b $\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$ แล้วเราจะหาค่า asymptotic ของสมการ T(n) ได้ดังนี้

1. ถ้าหาก $f(n) = O\left(n^{\log_b a-\varepsilon} \right)$ โดยที่ ε คือค่าคงที่ใดๆที่มากกว่า 0 แล้ว
$$T(n) = \Theta(n^{\log_b a-\varepsilon})$$
2. ถ้าหาก $f(n) = \Theta(n^{\log_b a})$ แล้ว
$$T(n) = \Theta(n^{\log_b a} \lg n)$$
3. ถ้าหาก $f(n) = \Omega(n^{\log_b a+\varepsilon})$ โดยที่ ε คือค่าคงที่ใดๆที่มากกว่า 0 และ ถ้า a.f(n/b) ≤ c.f(n) โดยที่ c คือค่าคงที่ c<1 และ n คือค่าคงที่ที่มากพอ แล้ว
$$T(n) = \Theta(f(n))$$

<u>ตัวอย่างที่ 1</u> กำหนดให้ $\mathrm{T(n)} = 9T\left(\frac{n}{3}\right) + n$ จงหาค่า asymptotic function ให้กับ T(n)

วิธีทำ ก่อนอื่นต้องทำการวิเคราะห์ก่อนว่า จะสามารถใช้ master theorem ในการแก้ปัญหาได้หรือไม่ ถ้าได้ จะเข้ากับกรณีใด

จากโจทย์ ค่า a= 9 , b=3 และ f(n) = n

ลองวิเคราะห์ $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

ข้อสังเกต จะเห็นว่า $f(n) = n < n^2$ ดังนั้นน่าจะเข้ากรณีที่ 1

ทดลองตรวจสอบเงื่อนไข กรณีที่ 1 ต้องตรวจสอบว่า $f(n) = O\left(n^{\log_b a-\varepsilon} \right)$ หรือไม่

ซึ่งเราแทนค่า $O\left(n^{\log_3 9-\varepsilon} \right)$ ถ้ากำหนดค่าให้ $\varepsilon = 1$ (กำหนดค่าเท่าไหร่ก็ได้ที่น้อยๆ)

จะได้ $O\left(n^{\log_3 8} \right) = O(n^{<2})$

ซึ่งจะเห็นว่า $f(n) = n \in O(n^{<2})$ ดังนั้นเราสามารถใช้ master theorem กรณีที่ 1 ได้เพราะเงื่อนไขเป็นจริง

ก็จะได้คำตอบว่า
$$T(n) = \Theta\left(n^{\log_b a-\varepsilon}\right) = \Theta\left(n^{\log_3 9-1}\right)$$

ตัวอย่างที่ 2 กำหนดให้ $\mathrm{T(n)} = T\left(\frac{2n}{3}\right) + 1$ จงหาค่า asymptotic function ให้กับ T(n)

วิธีทำ ก่อนอื่นต้องทำการวิเคราะห์ก่อนว่า จะสามารถใช้ master theorem ในการแก้ปัญหาได้หรือไม่ ถ้าได้ จะเข้ากับกรณีใด

จากโจทย์ ค่า a= 1 , b=3/2 และ f(n) = 1

ลองวิเคราะห์ $n^{\log_b a} = n^{\log_{3/2} 1} = \Theta(n^0) = \Theta(1)$

ข้อสังเกต จะเห็นว่า $f(n) = 1 = n^0$ ดังนั้นน่าจะเข้ากรณีที่ 2

ทดลองตรวจสอบเงื่อนไข กรณีที่ 2 ต้องตรวจสอบว่า $f(n) = \Theta(n^{\log_b a})$ หรือไม่

ซึ่งเราแทนค่า $\Theta\left(n^{\log_{3/2} 1} \right)$ จะได้ $\Theta\left(n^{\log_{3/2} 1} \right) = \Theta(n^0)$

ซึ่งจะเห็นว่า $f(n) = 1 \in \Theta(n^0)$ ดังนั้นเราสามารถใช้ master theorem กรณีที่ 2 ได้เพราะเงื่อนไขเป็นจริง

ก็จะได้คำตอบว่า

$$T(n) = \Theta\left(n^{\log_b a} \lg n \right) = \Theta(n^0 \lg n) = \Theta(\lg n)$$

ตัวอย่างที่ 3 กำหนดให้ $\mathrm{T(n)} = 3T\left(\frac{n}{4}\right) + n \lg n$ จงหาค่า asymptotic function ให้กับ T(n)

วิธีทำ ก่อนอื่นต้องทำการวิเคราะห์ก่อนว่า จะสามารถใช้ master theorem ในการแก้ปัญหาได้หรือไม่ ถ้าได้ จะเข้ากับกรณีใด

จากโจทย์ ค่า a= 3 , b=4 และ f(n) = n lg n

ลองวิเคราะห์ $n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793})$

ข้อสังเกต จะเห็นว่า $f(n) = n \lg n > n^{0.793}$ ดังนั้นน่าจะเข้ากรณีที่ 3

ทดลองตรวจสอบเงื่อนไข กรณีที่ 3 ต้องตรวจสอบว่า $f(n) = \Omega(n^{\log_b a + \varepsilon})$ หรือไม่

ซึ่งเราแทนค่า $\Omega\left(n^{\log_4 3 + 0.2} \right)$ จะได้ $\Omega\left(n^{\log_4 3.2} \right) = \Omega(n^{<1})$

ซึ่งจะเห็นว่า $f(n) = n \lg n \ \in \ \Omega(n^{<1})$

สำหรับกรณีที่ 3 ต้องทำการตรวจสอบต่อว่า a.f(n/b) ≤ c.f(n) โดยที่ c คือค่าคงที่ c<1 หรือไม่

ทดลองแทนค่า โดยให้ c= ¾ จะได้ว่า

$$3.\left(\frac{n}{4}\right)\lg\frac{n}{4} \leq \left(\frac{3}{4}\right) n \lg n$$

ซึ่งเป็นจริง ดังนั้นเราสามารถใช้ master theorem กรณีที่ 3 ได้เพราะเงื่อนไขเป็นจริง

ก็จะได้คำตอบว่า

$$T(n) = \ \Theta(\ f(n)) = \Theta(\ n \lg n)$$

<u>แบบฝึกหัด</u>

1. จงหาค่า asymptotic notation เมื่อกำหนดให้ $T(n) = 4T\left(\frac{n}{3}\right) + 5n$
2. จงหาค่า asymptotic notation เมื่อกำหนดให้ $T(n) = 3T\left(\frac{n}{3}\right) + 5n$
3. จงหาค่า asymptotic notation เมื่อกำหนดให้ $T(n) = 2T\left(\frac{n}{3}\right) + 5n$

บทที่ ๘ การวิเคราะห์ความน่าจะเป็นและอัลกอริทึมแบบสุ่ม (probabilistic analysis and randomized algorithm)

จากบทที่ผ่านมาจะเห็นว่า เราจะสนใจการวิเคราะห์หาว่า worse case running time คือการวิเคราะห์หาเวลาที่ใช้ในการรันอัลกอริทึมที่นานที่สุด สำหรับอินพุตขนาด n ใดๆ นั้นเป็นเท่าไหร่

เช่น ถ้าเราใช้ insertion sort ทำการเรียงข้อมูลจากน้อยไปมาก แต่ปรากฏว่าอินพุตที่ได้รับเข้ามานั้นเป็นข้อมูลที่เรียงจากมากไปน้อยอยู่ ก็จะทำให้ใช้เวลาในการรันอัลกอริทึมนานมาก ถือเป็น worse case ของ insertion sort เลย

อย่างไรก็ตามในความเป็นจริงนั้น เราอาจจะสนใจค่าเฉลี่ยในการใช้เวลารันอัลกอริทึม (average running time) ที่เกิดขึ้นจากการรันอินพุตทั่วๆไป (typical input) ซึ่ง typical input นั้นคืออินพุตที่เราสมมุติว่า ลำดับของข้อมูลทุกๆแบบนั้นมีโอกาสเกิดขึ้นเท่าๆกัน

คำถามคือเราไม่รู้ว่าอินพุตที่รับเข้ามานั้นจะมาในรูปแบบ best case หรือ worse case ถ้าหากว่าได้รับอินพุตเข้ามาเป็น worse case อย่างตัวอย่างเรื่อง insertion sort ข้างต้น อัลกอริทึมก็จะรันนานมาก ดังนั้นเราจะสามารถแก้ไขอินพุตเพื่อให้ลดการเกิด worse case ได้ไหมด้วยการใช้การสุ่ม

Probabilistic Analysis (การวิเคราะห์ความน่าจะเป็น)

ในการที่เราจะทำให้อินพุตเกิด average case นั้นเราจะต้องทำการวิเคราะห์ความน่าจะเป็นในการเกิดรูปแบบของอินพุตก่อน เพื่อใช้ในการสร้างอัลกอริทึมแบบสุ่ม (randomized algorithm) เพื่อมาช่วยทำให้อินพุตกลายเป็น average case ซึ่งเราจะใช้หลักการคณิตศาสตร์ทฤษฎีความน่าจะเป็นเข้ามาช่วยในการวิเคราะห์

เนื้อหาต่อจากนี้เป็นทฤษฎีพื้นฐานความน่าจะเป็นที่เกี่ยวข้อง

การสลับลำดับ (Permutation)

การสลับข้อมูลของเซตจำกัดใดๆ (permuation of a set) คือ ลำดับข้อมูลของสมาชิกทั้งหมดในเซตนั้น โดยที่สมาชิกแต่ละตัวจะปรากฏแค่ครั้งเดียวเท่านั้น

เช่น ถ้า S={a,b,c}, แล้ว จะมี permutation ของ S ทั้งหมด 6 แบบคือ abc, acb, bac, bca, cab, cba

ถ้าเราหา k-permuation of S คือ ลำดับข้อมูลสมาชิก จำนวน k ตัวของเซต S โดยที่ไม่มีสมาชิกตัวไหนปรากฏมากกว่าหนึ่งครั้ง เช่น ถ้า S = {a,b,c,d} แล้ว หา k-permutation โดยกำหนดให้ k=2 (เรียกอีกชื่อว่า 2-permutations of S) ก็จะได้ออกมา 12 แบบคือ ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc

ซึ่งจำนวนรูปแบบของ k-permutation สำหรับเซตที่มีขนาด n ใดๆ มีสมการดังนี้

$$n(n-1)(n-2)...(n-k+1) = \frac{n!}{(n-k)!}$$

คิดได้จาก เรามีวิธีเลือกสมาชิกตัวแรกทั้งหมด n แบบ ส่วนสมาชิกตัวที่สองเลือกได้ n-1 แบบ สมาชิกตัวถัดๆไปก็เลือกได้ n-2, n-3, ..., n-k+1 แบบ จนกระทั่งเลือกครบ k ตัว

การจัดกลุ่ม (k-combination)

การจัดกลุ่มข้อมูลของเซตจำกัดใดๆ คือ เซตย่อยทั้งหมดของ S ที่มีสมาชิก ขนาด k ตัว เช่น ถ้า S={a,b,c,d}, แล้วจะมี 2-combinations of S จำนวนทั้งหมด 6 เซตย่อย คือ {a,b}, {a,c}, {a,d}, {b,c}, {b,d}, {c,d} หรือเขียนอีกแบบได้เป็น ab, ac, ad, bc, bd, cd

จำนวนของกลุ่มข้อมูล k-combination ของเซตใดๆสามารถหาค่าอ้างอิงได้ในรูปแบบจำนวนของ k-permutation ของเซตนั้นๆ หาได้จากสมการ

$$\frac{n!}{k!(n-k)!}$$

สัมประสิทธิ์ทวินาม (binomial coefficient)

เราใช้คำว่า “n เลือก k” เพื่อบอกจำนวนของ k-combination ของเซตขนาด n ใดๆ เขียนตามรูปแบบสมการ

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

ซึ่งสมการนี้ symmetric ทั้งสำหรับ k และ n-k ดังนั้น

$$\binom{n}{k} = \binom{n}{n-k}$$

ซึ่งตัวเลขนี้เราเรียกว่า **binomial coefficients** เพราะว่าเลขเหล่านี้ปรากฏอยู่บนการกระจายสมการทวินาม

$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ จะเห็นว่า n, k เป็นเลขสัมประสิทธิ์ที่ปรากฏในแต่ละพจน์ของการกระจาย

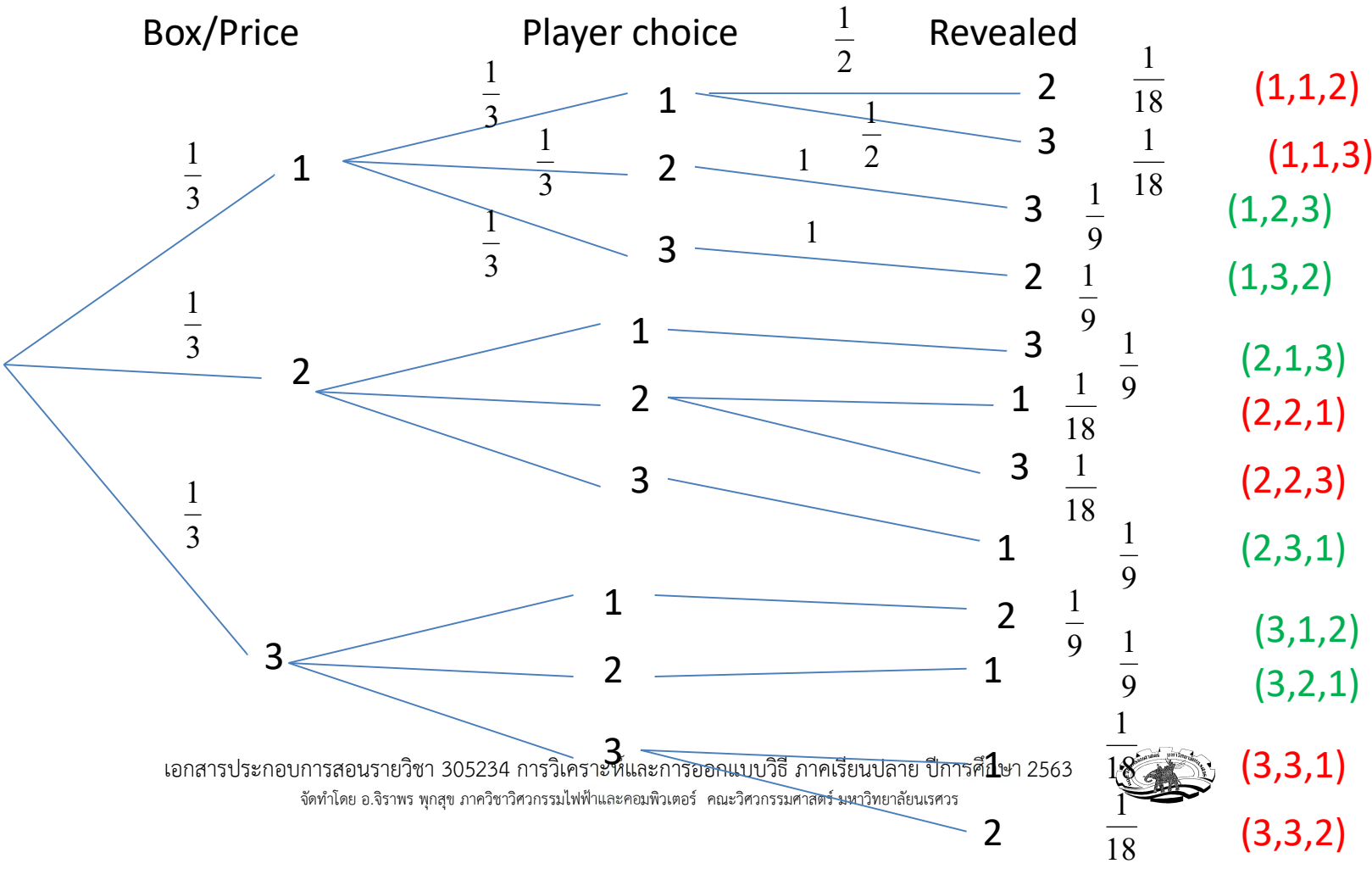
ความน่าจะเป็น

Sample space คือเซตที่แสดงเอ้าพุตทั้งหมดที่จะเกิดขึ้นได้ เราจะเรียกสมาชิกแต่ละตัวว่า elementary event ซึ่ง **elementary event** แต่ละตัวสามารถพิจารณาได้เป็น เอ้าพุตที่เป็นไปได้จากการทดลอง ส่วน **Event** คือเซตย่อยของ sample space ตัวอย่างเช่น ถ้าเราโยนลูกเต๋า (fair dice) 2 ลูก จะได้ว่ามี sample space S = {1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 2-1 ,..., 6-5, 6-6} และ event ของการเกิดเลขตัวเดียวกันบนลูกเต๋าทั้งสองลูก คือ {1-1,2-2,3-3,4-4,5-5,6-6}

ตัวอย่างที่ 1 Monty Hall Problem มีประตูอยู่ 3 บานและด้านหลังของประตูบานหนึ่งจะมีรางวัลใหญ่ กติกาคือให้ผู้เล่นเลือกเพียง 1 บานจากประตูทั้ง 3 บาน เมื่อผู้เล่นเลือกแล้ว พิธีกรจะทำการเปิดประตู 1 ใน 2 บานที่เหลือแล้วพิธีกรจะถามผู้เล่นว่าต้องการจะเปลี่ยนใจจากตัวเลือกที่เลือกไปแล้วหรือไม่ คำถามคือ ถ้าหากผู้เล่นเลือกที่จะเปลี่ยนใจ โอกาสที่จะได้รางวัลจะเป็น ½ หรือ 2/3

วิธีทำ จากแผนภาพจะเห็นว่า ถ้าหากประตูที่มีรางวัลคือบานที่ 1 ถ้าผู้เล่นเลือกบานที่ 1 และพิธีกรเปิดประตูบานที่ 2 เราก็จะคำนวณ ค่าความน่าจะเป็นได้ 1/3 . 1/3 . 1/2 = 1/18 เมื่อคำนวณทุกกิ่งของแผนภาพจะได้ว่าโอกาสที่ผู้เล่นจะชนะถ้าเปลี่ยนใจ จะเท่ากับ 2/3

If switch, pr(win)= $\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}=\frac{2}{3}$



ถ้าเรามี event A,B ใดๆ เราเรียกสอง event นี้ว่า **mutually exclusive** ถ้าหาก $A \cap B = \emptyset$

การกระจายของความน่าจะเป็น **probability distribution Pr{}** บน sample space S คือการแมปจาก event ใน S ไปยังเลขจำนวนจริงใดๆ ที่ **probability axioms** เหล่านี้เป็นจริง

1. Pr{A} ≥ 0 เมื่อ Aคือ eventใดๆ
2. Pr{S} = 1
3. $\Pr(A \cup B) = \Pr\{A\} + \Pr\{B\}$ เมื่อ A,B คือ mutually exclusive event

<u>ตัวอย่างที่ 2</u> ทอยลูกเต๋า (fair dice) จำนวนสองลูก sample space S = {1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 2-1 ,..., 6-5, 6-6} ดังนั้นความน่าจะเป็นของการเกิด elementary event แต่ละอันจะเท่ากับ 1/36

ถ้าเราหาความน่าจะเป็นของการเกิดตัวเลขเดียวกันจากการโยนลูกเต๋าทั้งสองลูก จะคิดได้ดังนี้

Pr{1-1,2-2,3-3,4-4,5-5,6-6} = Pr{1-1} + Pr{2-2}+Pr{3-3}+Pr{4-4}+Pr{5-5}+Pr{6-6} = 1/36*6 = 1/6

เราจะถือว่า **probability distribution** มีลักษณะแบบ **discrete** ถ้าหากว่า sample space นั้นเป็นเซตที่มีลักษณะเป็น finite หรือ countably infinite

กำหนดให้ S เป็น sample space และถ้าเรามี event A ใดๆ ดังนั้น

เนื่องจาก elementary event มีลักษณะเป็น mutually exclusive ดังนั้น

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$$

ถ้า S เป็น finite set และ ทุกๆสมาชิก a **∈** S มีค่าความน่าจะเป็น ค่าจะเท่ากับ $\Pr\{a\} = \frac{1}{|S|}$

ดังนั้นเราจะถือได้ว่า เรามี uniform probability distribution on S คือเลือกสมาชิกของ S มาแบบสุ่ม

<u>ตัวอย่างที่ 3</u> ทอยลูกเต๋า (fair dice) จำนวน 1 ลูก ค่าความน่าจะเป็นที่จะออกตัวเลขแต่ละตัวคือ 1/6

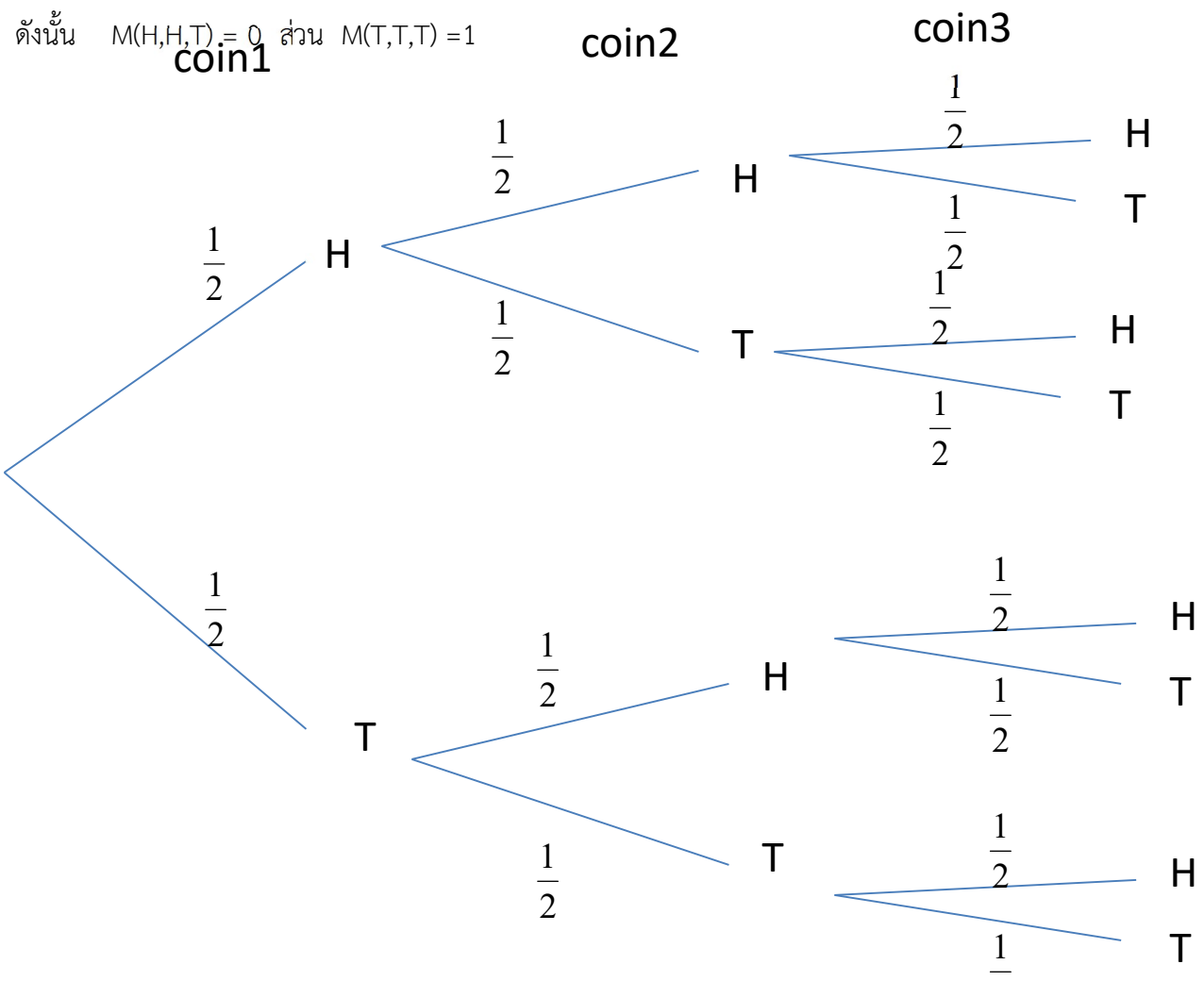
ถ้าหากว่าเราโยนลูกเต๋า n ครั้ง เราก็จะได้ uniform probability distribution บน sample space S ={1,2,3,4,5,6}^{n} เป็นเซตที่มีขนาด 6^n ซึ่ง elementary event แต่ละตัวจะมีค่าความน่าจะเป็นเท่ากับ 1/ 6^n

ดังนั้น ถ้ากำหนดให้ event A = {มีตัวเลข i ใดๆเกิดขึ้นทั้งหมด n ครั้ง สำหรับ i = 1...6 }
A ก็จะเป็นเซตย่อยของ S ที่มีขนาด ของ A เท่ากับ 6 ซึ่งก็คือ |A|= 6 ดังนั้น ความน่าจะเป็นของ event A ก็คือ

$$Pr\{A\}=6/\ 6^{n}=\ 1/\ 6^{n-1}$$

Random variable (ที่เป็น discrete) X ใดๆ คือฟังก์ชันที่แมปจาก sample space (ที่เป็น finite หรือ countably infinite) ไปยังเลขจำนวนจริง ตัวอย่างเช่น กำหนดให้ X_1 เป็น random variable แสดงค่าผลลัพธ์ของการทอยลูกเต๋าลูกแรก X_2 เป็น random variable แสดงค่าผลลัพธ์ของการทอยลูกเต๋าลูกที่สอง แล้วกำหนดให้ X เป็น random variable ที่แสดงค่าผลรวมของลูกเต๋าทั้งสองลูก $X = X_1 + X_2$.

<u>ตัวอย่างที่ 4</u> ทอยเหรียญ (fair coin) จำนวน 3 เหรียญ ค่าความน่าจะเป็นแสดงดังในแผนภาพ สมมติให้ ตัวแปรแบบสุ่ม (**Random variable**) R เท่ากับ จำนวนของการออกหัว ดังนั้น R(H,T,H) = 2
และให้ ฟังก์ชันบ่งชี้ตัวแปรแบบสุ่ม (**Indicator random var** or Bernoulli random var) M
มีค่าเท่ากับ 1 ถ้าหากว่าเหรียญทุกเหรียญออกเหมือนกันหมด
มีค่าเท่ากับ 0 ถ้าหากว่าเหรียญออกต่างกัน
ดังนั้น M(H,H,T) = 0 ส่วน M(T,T,T) =1



กำหนดให้ มี random variable X และเลขจำนวนจริง x และเรากำหนดค่า event X = x ให้เป็นเซตคือ {s **∈** S: X(s) = x} ดังนั้น

$$\Pr\{X = x\} = \sum_{s \in S: X(s) = x} \Pr\{s\}$$

ฟังก์ชัน f(x) = Pr{X=x} คือ **probability density function** of the random variable X

ดังนั้นจาก probability axioms จะได้ว่า

$$\Pr\{X = x\} \geq 0 \quad and \quad \sum_{x} \Pr\{X = x\} = 1$$

ตัวอย่างเช่น ทอยลูกเต๋า(fair dice) สองลูก ทำให้มี elementary event ที่เป็นไปได้ถึง 36 แบบใน sample space เราสมมติว่า probability distribution นั้น uniform ดังนั้น elementary event แต่ละ s **∈** S มีค่าความน่าจะเป็นเท่าๆกันคือ Pr{s} = 1/36

กำหนดให้ X เป็น random variable แสดงค่าที่มากที่สุดระหว่างตัวเลข 2 ตัวที่โชว์บนลูกเต๋าที่ทอย ดังนั้นเราจะมี Pr{X=3} = 5/36 เนื่องจากว่า elementary event ที่เป็นไปได้คือ {1-3, 2-3, 3-3, 3-2, 3-1} (อีเว้นที่มีเลข 3 เป็นค่ามากที่สุดระหว่าง 2 ลูกเต๋าที่ทอยได้)

ถ้าให้ X,Y เป็น random variable แล้ว ฟังก์ชัน F(x,y) = Pr{X=x and Y=y} จะเป็นฟังก์ชัน **joint probability density** ซึ่ง

- ถ้าเรากำหนดค่าคงให้Y คือ y

$$\Pr\{Y = y\} = \sum_{x} \Pr\{X = x \; and \; Y = y\}$$

- ถ้าเรากำหนดค่าคงที่ให้X คือ x

$$\Pr\{X = x\} = \sum_{y} \Pr\{X = x \; and \; Y = y\}$$

ตัวแปร random variable สองค่า X,Y ใดๆ จะ independent กัน ถ้าหากว่า event X=x และ Y=y เป็น independent กัน เมื่อกำหนดให้ x,y คือค่าคงที่ ดังสมการ

$$\Pr\{X = x \; and \; Y = y\} = \Pr\{X = x\} \, . \Pr\{Y = y\}$$

ค่า expected value (หรือค่า expectation หรือค่า mean) ของ discrete random variable X ใดๆ คือ

$$E[X] = \sum_{x} x \, . Pr\{X = x\}$$

ตัวอยางเช่น ในการโยนเหรียญ (fair coin) จำนวน 2 เหรียญ ซึ่งมีกติกาว่า ถ้าออกหัว คุณจะได้ 3 ดอลล่าร์ ต่อเหรียญ และถ้าออกก้อย คุณจะได้ 2 ดอลล่าร์ต่อเหรียญ ค่า expected value ของ X จะคำนวณได้ดังนี้

E[X] = 6.Pr{HH} + 1.Pr{1H,1T} – 4. Pr{TT}

= 6(1/4) + 1(1/2) – 4(1/4) = 1

ความเป็นเชิงเส้นของ expectation (linearity of the expectation) ก็คือ การที่ค่า expectation ของผลรวมของ random variable สองค่า เท่ากับผลรวมของ expectation ของแต่ละ random variable ดังสมการต่อไปนี้

$$E[X + Y] = E[X] + E[Y]$$

ถ้าหากว่า X เป็น random variable แล้ว สำหรับ g(x) ที่เป็นฟังก์ชันใดๆ เราจะกำหนดค่า random variable ใหม่ ชื่อ g(X) ได้ ถ้าหากว่าหาค่า expectation ของ g(X) ได้ แล้วดังนั้น

$$E[g(X)] = \sum_{x} g(x) \Pr\{X = x\}$$

กำหนดให้ g(x) = ax เมื่อ a คือค่าคงที่ใดๆ แล้ว E[a.X] = a.E[X]

ถ้าหากมี random variable X,Y สองค่าใดๆ ที่ independent กัน และแต่ละตัวแปรมีการกำหนดค่า expectation ของตัวเองอยู่ แล้ว E[XY] = E[X] . E[Y]

ค่า **variance** เป็นค่าที่แสดงว่า random variable X อยู่ห่างจากค่า mean เท่าไหร่ หากมีค่า mean คือ E[X]

Var[X] = $E[X^2] – E^2[X]$

Var[X+Y] = Var[X]+Var[Y] ถ้าหากว่า X,Y เป็น independent

ค่า **standard deviation** ของ random variable X คือ $\sqrt{Var[X]}$

ตัวอย่างที่ 5 ถ้าเรามี random variable X,Y โดยที่กำหนดค่า Pr{X=1/4}=Pr{X=3/4}= ½ และ Pr{Y=0}=Pr{Y=1} = ½

ดังนั้น E[X] = ¼.1/2 + ¾.1/2 = ½

E[Y] = 0.1/2 + 1.1/2 = ½.

แต่ว่า ค่าที่แท้จริงของ Y ห่างจากค่า mean มากกว่าค่าที่แท้จริงของ X เทียบกับ mean ซึ่งคำนวณได้จากการหาค่า variance ดังนี้

$E[X^2] = (1/4)^2.1/2 + (3/4)^2.1/2 = 5/16$

$Var[X] = 5/16 - (1/2)^2 = 1/16$

$E[Y^2] = 0.1/4 + 1^2.1/2 = 1/2$

$Var[Y] = 1/2 - (1/2)^2 = 1/4$

ค่า **Bernoulli trial** คือการทดลองที่มีเอ๊าพุตที่เป็นไปได้เพียงแค่ 2 แบบ คือ **success** โดยที่จะมีค่าความน่าจะเป็นคือ p และ **failure** มีค่าความน่าจะเป็นคือ q=1-p

สมมติว่าเรามีลำดับของการทดลอง Bernoulli คำถามคือต้องมีการทดลองกี่ครั้งจนกว่าเราจะได้ **success**

กำหนดให้ random variable X คือจำนวนการทดลองที่ต้องทำเพื่อให้ได้ success ดังนั้น

$Pr\{X=k\}=q^{k-1}\ p$

ดังนั้นเราจะมี failure ทั้งหมด k-1 ครั้งก่อนจะได้ success ซึ่ง probability distribution แบบนี้เรียกว่า **geometric distribution**

สมมติให้ q < 1 ค่า expectation ของ geometric distribution คือ

$$E[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q}\sum_{k=1}^{\infty} kq^{k} = \frac{p}{q}\cdot\frac{q}{(1-q)^2} = \frac{1}{p}$$

ดังนั้นโดยเฉลี่ย จะต้องทำการทดลอง 1/p ครั้งก่อนที่จะได้ success

ค่า varience เท่ากับ $Var[X] = q/p^2$

ตัวอย่างเช่น สมมติว่าเราทอยลูกเต๋าพร้อมกัน 2 ลูกจนกว่าจะได้ค่าเลข 7 หรือ 11 ถึงจะหยุดทอยลูกเต๋า ดังนั้นก็จะมีเอ๊าพุตที่เป็นไปได้ทั้งหมด 6 แบบที่จะทำให้ได้เลข 7 และ มี 2 แบบที่ทำให้ได้เลข 11 ดังนั้น ความน่าจะเป็นที่จะ success คือ p = 8/36 = 2/9 ดังนั้นเราต้องทอยลูกเต๋า 1/p = 9/2 = 4.5 ครั้งโดยเฉลี่ยที่จะทำให้ได้ตัวเลขค่า 7 หรือ 11

สมมติว่าเรามีลำดับของ Bernoulli trial คำถามคือถ้าอยากรู้ว่า จะเกิดกรณี success กี่ครั้ง ถ้าเราทำการทดลองทั้งหมด n ครั้ง โดยที่ success มีความน่าจะเป็นคือ p และ failure มีความน่าจะเป็น q = 1-p

กำหนดให้ random variable X เป็นจำนวนครั้งที่ success สำหรับการทดลอง n ครั้ง ดังนั้น

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}$$

เนื่องจากว่า มี $\binom{n}{k}$ วิธีที่จะเลือก k จากการทดลอง n ครั้ง โดยที่ k คือ success และความน่าจะเป็นของการเกิดแต่ละครั้งคือ $p^k q^{n-k}$ ซึ่ง probability distribution แบบนี้เรียกว่า **binomial distribution**

เพื่อความสะดวกในการเรียกชื่อ กลุ่มของ binomial distributions จะใช้สัญลักษณ์

$$b(k; n; p) = \binom{n}{k} p^k (1-p)^{n-k}$$

ค่า expectation ของ random variable ที่มี binomial distribution คือ E[X] = np

ดังนั้นค่า variance เท่ากับ Var[X] = npq

<u>แบบฝึกหัด</u>

1. มีคน 20 คนอยู่ในงานปาร์ตี้ แล้วโยนเสื้อของตัวเองปนกัน ต่อมาให้กลับไปหยิบเสื้อ คำถามคือจงหาว่าโดยเฉลี่ยแล้ว(expected value) จะมีกี่คนที่จะเจอเสื้อของตัวเอง
2. มีบอล 5 ลูก มีถัง 5 ใบ โยนบอลทั้ง 5 ลูกนั้น แล้วจงหาว่า expected value (ค่าเฉลี่ย) ที่บอลจะลงในถังใบที่ 1 จะมีกี่ลูก
3. มีถัง 5 ใบ ให้หาว่าโดยเฉลี่ยแล้ว จะต้องโยนกี่ครั้งถึงจะมีบอลลงในถังใบที่ 2 (บอลลงครั้งแรก)

บทที่ ๙ การวิเคราะห์อัลกอริทึมแบบสุ่ม (analysing randomized algorithms)

ค่า **indicator random variable** I{A} ที่เกี่ยวข้องกับ event A ถูกกำหนดให้เป็นดังนี้

I{A} = 1 ถ้าเกิดอีเว้น A
0 ถ้าไม่เกิดอีเว้น A

ตัวอย่างเช่น ในการโยนเหรียญ (fair coin) เรากำหนดค่า expected number ของการออกหัว

โดยที่มี sample space S = {H,T} มีความน่าจะเป็น Pr{H}=Pr{T}=1/2

เรากำหนด indicator random variable X_H เกี่ยวข้องกับการออกหัวของการโยนเหรียญ ซึ่งให้เป็นอีเว้น H ดังนั้น

X_H = I{H} = 1 ถ้าเกิด H
0 ถ้าเกิด T

ค่า expected number ของการเกิดหัวในการโยนเหรียญหนึ่งเหรียญ ก็คือค่า expected value ของ X_H

$E[X_H]$ = E [I{H}] = 1.Pr{H} + 0.Pr{T} = 1.(1/2)+ 0.(1/2) = ½

Lemma: กำหนดให้ sample space S และ event A ใน S และให้ X_A = I{A} ดังนั้น $E[X_A]$ = Pr{A}

ตัวอย่างที่ 1 ถ้าเราโยนเหรียญ (fair coin) ทั้งหมด n ครั้งแล้วต้องการคาดการณ์ว่าจะออกหัวกี่ครั้ง

วิธีทำ กำหนดให้ X_i เป็น indicator random variable โดยที่

X_i = I{H} = 1 ถ้าออกหัวในการโยนเหรียญครั้งที่ i

= 0 ถ้าออกก้อยในการโยนเหรียญครั้งที่ i

แล้วกำหนดให้ X เป็น indicator random variable ที่แสดงค่าจำนวนครั้งทั้งหมดที่โยนเหรียญแล้วออกหัว จากการโยนเหรียญทั้งหมด n ครั้ง ดังนั้น $X = \sum_{i=1}^{n} X_i$

ดังนั้นค่า expectation จะเท่ากับ

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$$

หมายเหตุ: $E[X_i]$= E[I{H}] = 1.Pr{H} + 0.Pr{T} = 1.(1/2) + 0.(1/2) = ½

ปัญหาเรื่อง The Hiring Problem คือปัญหาที่สมมติว่าเราต้องการจ้างพนักงานใหม่ แล้วแต่ละวันจะสัมภาษณ์ผู้สมัครเพียงหนึ่งคนเท่านั้น หลังจากสัมภาษณ์ก็จะตัดสินใจว่าจะจ้างคนนั้นหรือไม่ ถ้าหากว่าคนที่มาสัมภาษณ์ดีกว่าพนักงานคนปัจจุบัน คุณก็จะตัดสินใจจ้างไว้

อัลกอริทึมในการจ้างพนักงานแสดงดังรูป

```
Pseudo code: Hire-Assistant(n)
best = 0 // dummy candidate
for i = 1 to n
        do interview candidate i
        if candidate i is better than candidate best
                then best = i
                        hire candidate i
```

สำหรับค่าใช้จ่าย คุณจะต้องจ่ายมากถ้าจ้างไว้ แต่จะจ่ายเพียงเล็กน้อยถ้าไม่จ้าง

กำหนดให้ c_i เป็นค่าใช้จ่ายในการสัมภาษณ์ และ c_h คือค่าใช้จ่ายหากจ้างไว้ และให้ m เป็นจำนวนคนที่จ้างทั้งหมด ดังนั้น

ค่าใช้จ่ายทั้งหมดคือ $O(n\ c_i + m\ c_h)$ สำหรับ worst-case ถ้าเราจ้างทุกคนเลยหลังการสัมภาษณ์แต่ละคนแต่ละวัน ค่าใช้จ่ายทั้งหมดจะเท่ากับ $O(n\ c_h)$

<u>ตัวอย่างที่ 2</u> วิเคราะห์ปัญหาเรื่อง The Hiring Problem

วิธีทำ กำหนดให้ X_i เป็น indicator random variable ที่แสดงอีเว้นคือ ผู้สมัครคนที่ i จะถูกจ้าง โดยที่

X_i = I{ผู้สมัครคนที่ i ถูกจ้าง} = 1 ถ้าผู้สมัครคนที่ i ถูกจ้าง
= 0 ถ้าผู้สมัครคนที่ i ไม่ถูกจ้าง

และให้ $X = X_1 + X_2 + \ldots + X_n$

จาก lemma $E[X_i]$ = Pr{ผู้สมัครคนที่ i ถูกจ้าง}

พิจารณาดูพบว่า ผู้สมัครคนที่ i ถูกจ้างเมื่อคนที่ i ดีกว่าผู้สมัครคนที่ 1 ถึงคนที่ i-1 ดังนั้น ผู้สมัครคนที่ i จะมีความน่าจะเป็นคือ 1/i ที่จะมีคุณสมบัติดีกว่าคนที่ 1 ถึงคนที่ i-1 และดังนั้นความน่าจะเป็นที่ผู้สมัครคนที่ i จะถูกจ้างคือ 1/i

จาก lemma $E[X_i]$ = Pr{ผู้สมัครคนที่ i ถูกจ้าง} เราจึงสรุปได้ว่า $E[X_i] = 1/i$

ทีนี้เรามาคำนวณหา E[X] ได้ดังนี้

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n}[X_i] = \sum_{i=1}^{n}\frac{1}{i} = \ln n + O(1)$$

Lemma: สมมติให้ผู้สมัครเข้ามาสัมภาษณ์ด้วยลำดับที่สุ่ม อัลกอริทึม Hire-Assistant จะมีค่า hiring cost ทั้งหมดเท่ากับ O (c_h ln n)

Pseudo code: Randomized-Hire-Assistant(n)

```
randomly permute the list of candidates
best = 0 // dummy candidate
for i = 1 to n
        do interview candidate i
        if candidate i is better than candidate best
                then best = i
                        hire candidate i
```

วิธีการทำให้ข้อมูลกลายเป็นข้อมูลแบบสุ่ม (Randomization method)

ในการสุ่มข้อมูลอินพุตด้วยการสลับตำแหน่งของอินพุตอาเรย์ เราจะทำการกำหนดค่า priority P[i] ให้กับสมาชิกแต่ละตัวในอาเรย์ A[i] แล้วจึงทำการเรียงลำดับสมาชิกของอาเรย์ A ตามลำดับค่า priority ซึ่งมีวิธีการทำ 2 วิธีดังนี้

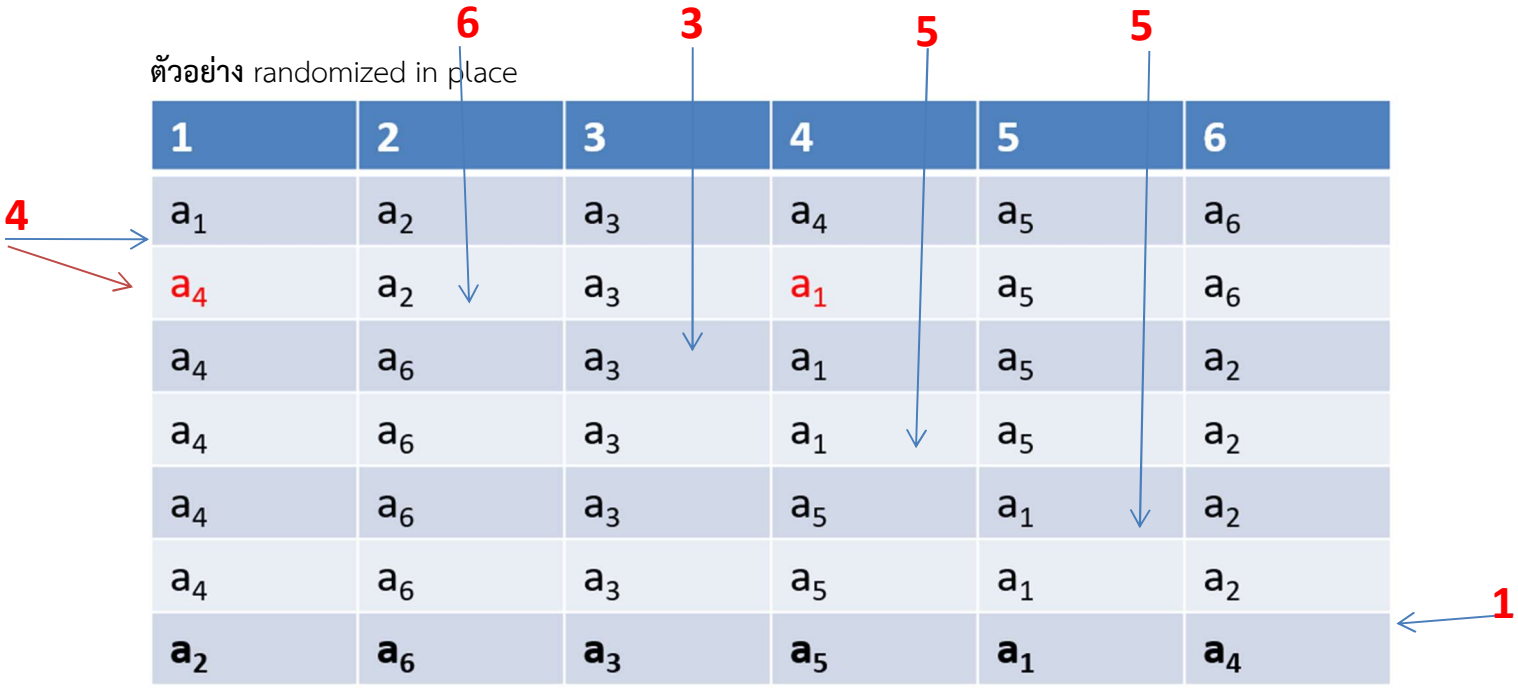
1. **Permute by sorting** เราทำการกำหนดช่วงค่า priority อยู่ในช่วงของ 1 ถึง n^3 เพื่อให้แน่ใจว่าค่า priority จะไม่ซ้ำกัน ดังโค้ดด้านล่าง ซึ่ง เวลาที่ใช้ในโค้ดบรรทัดที่ 4 จะขึ้นอยู่กับ sorting algorithm ที่เราเลือกใช้ เช่นหากเลือกใช้ merge sort จะใช้เวลาประมาณ $\Theta(n \lg n)$

```
Pseudo code: Permute-By-Sorting(A)
N = length[A]
for i = 1 to n
        do P[i] = Random(1,n^3)
sort A, using P as sort keys
return A
```

2. Randomized in place เป็นการเรียงอาเรย์ในตำแหน่งที่มี ซึ่งกระบวนการนี้จะใช้เวลาประมาณ O(n) ดังโค้ดด้านล่าง

```
Pseudo code: Randomize-In-Place(A)
N = length[A]
for i = 1 to n
        do swap (A[i], A[Random(1,n)]
```

ตัวอย่าง randomized in place



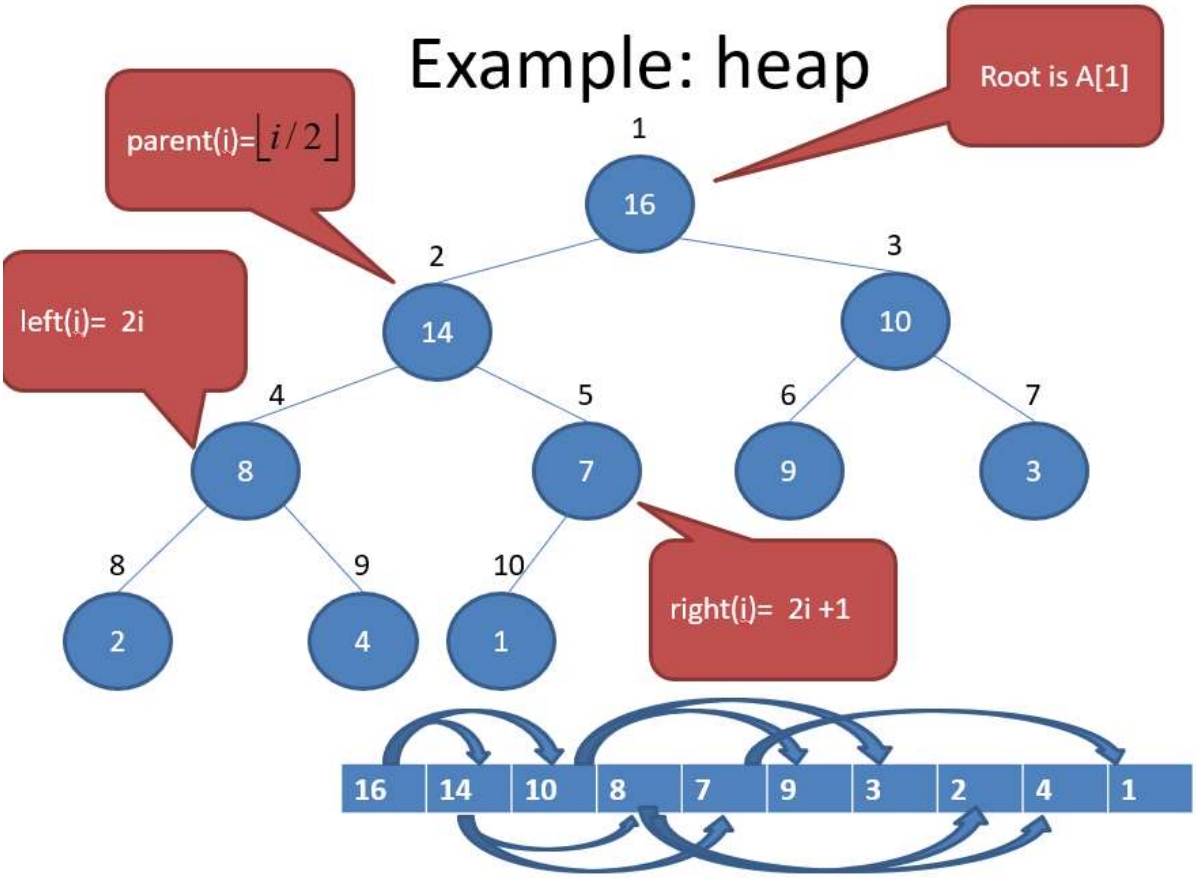
บทที่ ๑๐ การเรียงลำดับข้อมูลแบบฮีป (heap and heap sort)

โครงสร้างข้อมูลแบบ (binary) heap คืออาเรย์ของวัตถุที่มีลักษณะเกือบคล้ายกับ binary tree แต่ละโหนดของทรีก็คือสมาชิกของอาเรย์

อาเรย์ A ใดๆที่ใช้แสดงเป็นโครงสร้างแบบ heap นั้นจะเป็นวัตถุที่ประกอบด้วย 2 ค่าคือ

- length[A] คือจำนวนสมาชิกที่อยู่ในอาเรย์
- heap-size[A] จำนวนของสมาชิกที่อยู่ใน heap ในอาเรย์ A

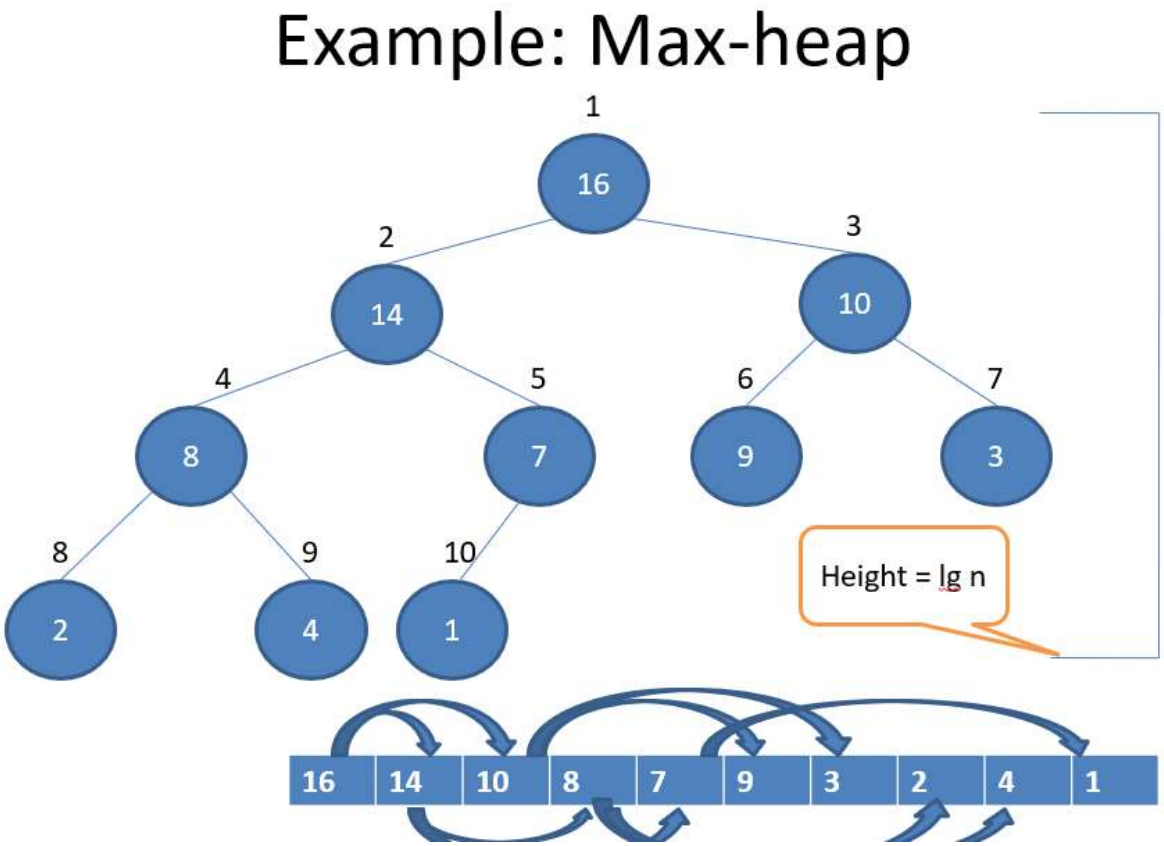
ตัวอย่าง heap แสดงดังรูป



สำหรับ **binary heap** มีอยู่ด้วยกันสองชนิดคือ

Max-heaps ซึ่ง สำหรับทุกโหนด i ที่ไม่ใช่ root node ค่าของ A[parent(i)] ≥ A[i]
Min-heaps ซึ่ง สำหรับทุกโหนด i ที่ไม่ใช่ root node ค่าของ A[parent(i)] ≤ A[i]

ตัวอย่าง max heaps แสดงดังรูป



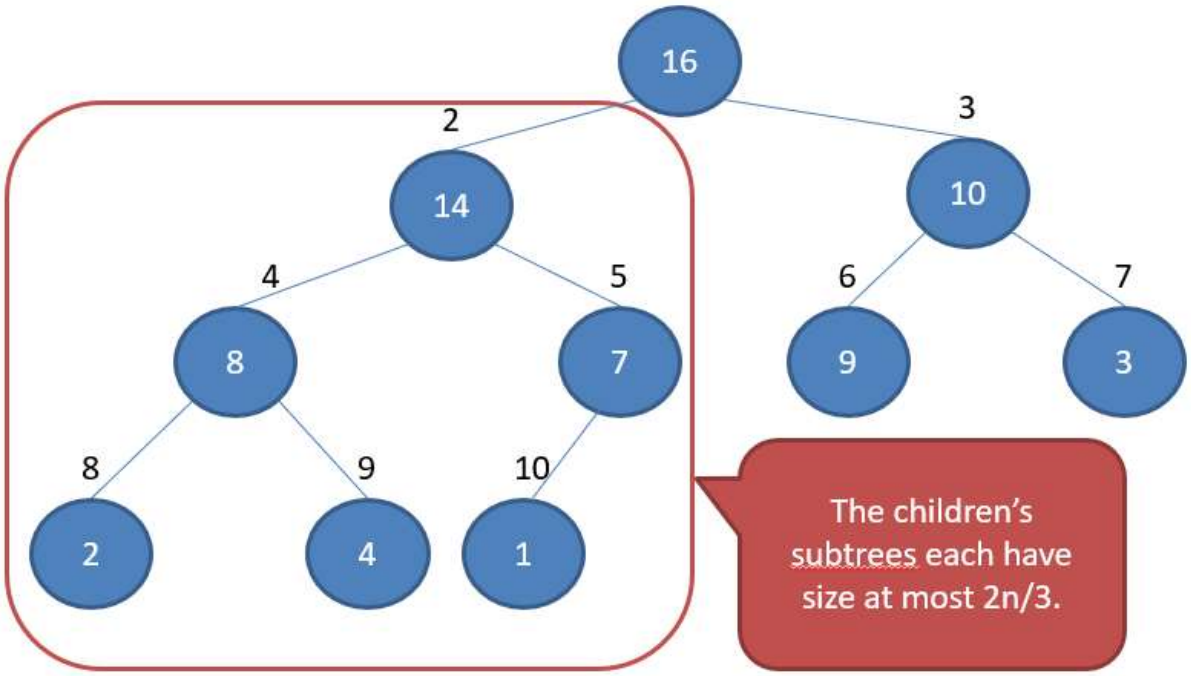
สำหรับอัลกอริทึมที่ใช้ในการจัดเรียงโหนดของ max heaps มีดังนี้

Psudo code: Max-Heapify(A,i)

```
l = left(i)
r = right(i)
if l ≤ heap-size[A] and A[l] > A[i]
        then largest = l
else    largest = i
if  r ≤ heap-size[A]  and A[r] > A[largest]
        then largest = r
If largest != i
        then exchanged A[i] and A[largest]
            Max-Heapify(A,largest)
```

วิเคราะห์เวลาที่ใช้ในการรันอัลกอริทึม Max-Heapify(A,i) จะพบว่าจากรูปด้านล่างค่า recurrence คือ

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$



ใช้ master theorem วิเคราะห์ได้ว่า เรามี a=1, b=3/2, f(n)= 1 ดังนั้น

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

เนื่องจาก $f(n) = \Theta(n^0) = \Theta(1)$ เราก็สามารถใช้กรณีที่ 2 ของ master theorem ได้ก็จะสามารถสรุปได้ว่าเวลาที่ใช้ในการรันอัลกอริทึม Max-heapify คือ $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

สำหรับ**การสร้าง max heaps** ขึ้นมาเราใช้อัลกอริทึมที่ชื่อว่า build max heap ซึ่งมีรายละเอียดดังโค้ดต่อไปนี้

Psudo code: Build-Max-Heap(A)

```
heap-size[A] = length[A]
for i = ⌊length[A]/2⌋ downto 1
        do Max-Heapify(A,i)
```

วิเคราะห์ความถูกต้องของอัลกอริทึม Build-Max-Heap(A) โดยการใช้เทคนิค loop invariants ได้ดังนี้

กำหนดให้ loop invariant คือ ก่อนที่จะเริ่มแต่ละรอบของ for loop ในบรรทัดที่ 2-3 นั้น แต่ละโหนด i+1, i+2,...,n จะต้องเป็น root ของ max-heap

พิสูจน์แต่ละขั้นตอนดังนี้

Initialization:

ก่อนที่จะรันรอบที่ 1 ค่า $i = \lfloor n/2 \rfloor$ แต่ละโหนด $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ ก็จะเป็น leaf node และดังนั้น root ก็ต้องเป็น max-heap (True!!)

Maintenance:

ถ้า children ของโหนด i มีค่ามากกว่า i แล้ว ทั้งสองโหนดนั้นจะเป็น root ของ max-heap

เงื่อนไขกำหนดให้เรียกฟังก์ชัน Max-Heapify(A,i) เพื่อทำให้ โหนด i เป็น root เสร็จแล้วทำการลดเลขจำนวนรอบลงเพื่ออัพเดทค่า loop invariant ของรอบถัดไป (True!!)

Termination:

ตอนจบ i =0 แต่ละโหนด 1,2,....,n เป็น root ของ max-heap ซึ่งโหนด 1 ก็เป็น root (True!!)

วิเคราะห์เวลาที่ใช้ในการรันอัลกอริทึม Build-Max-Heap(A) ได้ดังนี้

	Times
heap-size[A] = length[A]	1
for i = $\lfloor length[A]/2 \rfloor$ downto 1	n/2+1
do Max-Heapify(A,i)	n/2.O(lg n)

ดังนั้น $T(n) = O(n \lg n)$

สุดท้ายเมื่อเราทำการสร้าง max heaps ขึ้นมาแล้ว เราต้องทำการ**เรียงโหนดทั้ง**หมดด้วยอัลกอริทึมที่ชื่อว่า heap sort ซึ่งมีรายละเอียดดังโค้ดต่อไปนี้

Psudo code: Heapsort(A)

```
Build-Max-Heap(A)
for i = length[A]  downto 2
        do exchange A[1] and A[i]
        heap-size[A] = heap-size[A] -1
        Max-Heapify(A,1)
```

วิเคราะห์เวลาที่ใช้ในการรันอัลกอริทึม Heapsort(A) ได้ดังนี้

	Times
Build-Max-Heap(A)	O(n)
for i = length[A] downto 2	n
do exchange A[1] and A[i]	n-1
heap-size[A] = heap-size[A] -1	n-1
Max-Heapify(A,1)	n-1 . O(lg n)

ดังนั้น $T(n) = O(n \lg n)$

<u>แบบฝึกหัด</u>

1. ให้ทำการเรียงข้อมูลต่อไปนี้จากน้อยไปมากด้วยวิธีการ heap sort 8, 17, 12 , 15 ,92 , 16, 11, 52, 41

บทที่ ๑๑ การเรียงลำดับข้อมูลแบบรวดเร็ว (quick sort)

การเรียงลำดับข้อมูลแบบรวดเร็ว มีแนวคิดอยู่บนพื้นฐานของ divide and conquer ซึ่ง worst-case running time คือ $\Theta(n^2)$ เมื่อ n คือขนาดของอินพุตอาเรย์ และ Expected running time คือ $\Theta(n \lg n)$

อัลกอริทึมของ Quick sort มีดังนี้

Pseudo code: Quicksort(A, p,r)

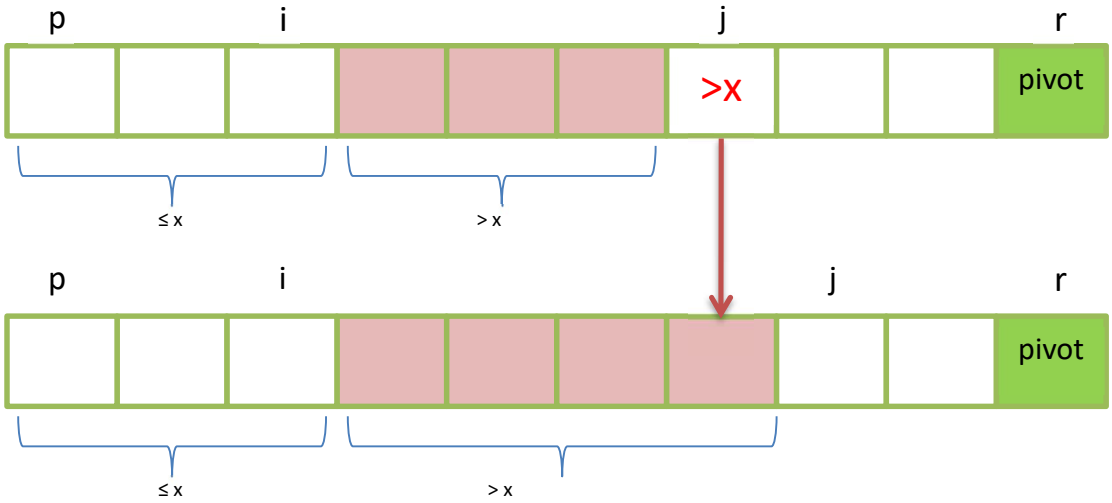
```
if p < r
        then q = Partition(A,p,r)
                Quicksort(A,p, q-1)
                Quicksort(A, q+1 , r)
```

ซึ่งฟังก์ชัน partition คือการแบ่งข้อมูลเป็น 2 กลุ่ม ซึ่งกลุ่มนึงจะมีค่ามากกว่าค่าที่ใช้อ้างอิง ส่วนอีกกลุ่มจะเก็บค่าที่มีค่าน้อยกว่าค่าที่ใช้อ้างอิง ส่วนค่าอ้างอิง เรียกว่า ค่า pivot

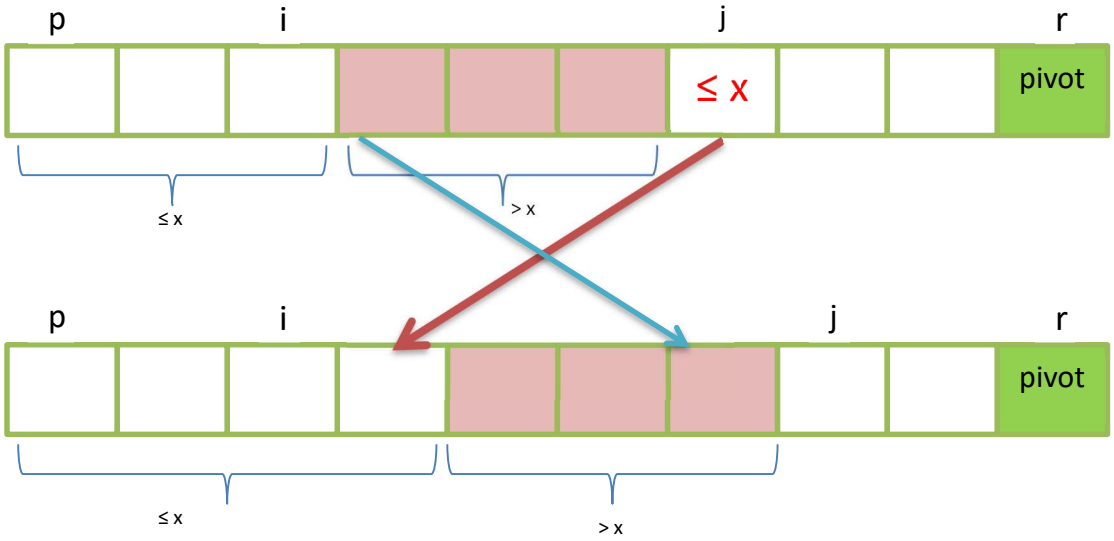
Psuedo code: Partition(A, p,r)

```
x = A[r]
i = p-1
for j = p to r-1
        do if A[j] <= x
                then i = i +1
                exchange A[ i ] and A[ j ]
exchange A[i+1] and A[r]
return i+1
```

ซึ่งหลักการ partition แสดงดังแผนภาพ จะเห็นว่าอาเรย์ถูกแบ่งเป็นกลุ่มที่มีค่ามากกว่า pivot กับกลุ่มที่มีค่าน้อยกว่าหรือเท่ากับ pivot เมื่อเลขตัวที่กำลังเช็คนั้นมีค่ามากกว่า pivot เลขนั้นก็จะถูกเพิ่มในฝั่งของอาเรย์ที่เป็นกลุ่มค่ามากกว่า x



ถ้าหากเลขที่กำลังเช็คนั้นมีค่าน้อยกว่าหรือเท่ากับ pivot เลขนั้นก็จะไปเพิ่มฝั่งกลุ่มที่มีค่าน้อยกว่าหรือเท่ากับ x ดังรูป



วิเคราะห์ความถูกต้องของอัลกอริทึม quick sort

กำหนดให้ **loop invariant** ประกอบไปด้วยเงื่อนไข 3 ข้อดังนี้

ก่อนที่จะรันลูปแต่ละรอบสำหรับ array ที่มีค่าตำแหน่ง k ใดๆ เงื่อนไขเหล่านี้ต้องเป็นจริง

1. $p \le k \le i$, then $A[k] \le x$

2. If $i+1 \le k \le j-1$, then $A[k] > x$

3. If $k=r$ then $A[k] = x$

Initialization: ก่อนที่จะรันลูปรอบที่ 0 ค่า i= p-1 และ j=p

เงื่อนไขที่ 1 : ระหว่างตำแหน่ง p และ i ไม่มีค่าข้อมูล

เงื่อนไขที่ 2: ระหว่างตำแหน่งที่ i+1 และ j-1 ไม่มีค่าข้อมูล

เงื่อนไข 3: บรรทัดที่ 1 x = A[k]
ดังนั้นเงื่อนไขทั้ง 3 ข้อเป็นจริง (True!!)

Maintenance:

เมื่อ A[j]<=x, ค่า i จะเพิ่มและเราจะสลับค่ากันระหว่าง A[i] กับ A[j] ดังนั้นค่า j ก็จะเพิ่ม เงื่อนไขที่ 1 ก็เป็นจริง

เมื่อ A[j] >= x ค่า j จะเพิ่ม ดังนั้นเงื่อนไขที่ 2 เป็นจริง

จากบรรทัดที่ 1 ทำให้เงื่อนไขที่ 3 เป็นจริง

ดังนั้นเงื่อนไขทั้ง 3 ข้อเป็นจริง (True!!)

Termination:

เมื่อจบการรัน j = r อาเรย์จะถูกแบ่งออกเป็น 3 ส่วนตามเงื่อนไขด้านบน (True!!)

วิเคราะห์เวลาที่ใช้ในการรันอัลกอริทึม quick sort

เราแบ่งการวิเคราะห์ออกเป็น 2 กรณีดังนี้

1) Worst-case partitioning ถ้าแบ่งพาทิชั่นออกเป็นสมาชิก n-1 ตัวกับ 0 ตัว ค่า running time จะเป็น $T(n) = T(n-1) + \Theta(n)$ ดังนั้น running time คือ $\Theta(n^2)$
2) Best-case partitioning ถ้าแบ่งพาทิชั่นออกเป็นสมาชิกกลุ่มละ n/2 ตัว ค่า running time จะเป็น $T(n) \leq 2T(n/2) + \Theta(n)$ ดังนั้น running time คือ $\Theta(n \lg n)$

ซึ่งพบว่าค่า average running time เองก็มีค่าใกล้เคียงกับ best-case running time เช่นกัน

สำหรับ quick sort อัลกอริทึม เราสามารถออกแบบเวอร์ชันที่เป็น randomized quick sort ได้ดังโค้ดต่อไปนี้

```
Randomized-Partition(A,p,r)
i = Random(p,r)
exchange A[r] and A[i]
return Partition(A,p,r)
```

```
Randomized-Quicksort(A,p,r)
if p < r
        then q = Randomized-Partition(A,p,r)
                Randomized-Quicksort(A,p, q-1)
                Randomized-Quicksort(A, q+1 , r)
```

<u>แบบฝึกหัด</u>

1. ให้ทำการเรียงข้อมูลต่อไปนี้จากน้อยไปมากด้วยวิธีการ quick sort 16, 14 , 51 , 2 ,15 , 19, 17 , 13

บทที่ ๑๒ การเรียงลำดับข้อมูลในเวลาเชิงเส้น (sorting in linear time)

อัลกอริทึมที่ใช้เรียงข้อมูลที่กล่าวถึงก่อนหน้านี้ทั้งหมด เป็นอัลกอริทึมที่เปรียบเทียบค่ากันระหว่างตัวเลขในอินพุตเท่านั้น ซึ่งกระบวนการเปรียบเทียบระหว่างตัวเลขนี้ จะมี wost case running time ประมาณ $\Omega(n \lg n)$ ในบทนี้จะพิจารณาอัลกอริทึมอื่นๆที่เรียงข้อมูลโดยใช้เวลาน้อยกว่า อัลกอริทึมก่อนหน้า มีดังนี้

1. **Counting sort** เป็นอัลกอริทึมที่ใช้เรียงข้อมูล ที่มีหลักการคือการนับค่าของอินพุตและทำตำแหน่งสำหรับเรียงข้อมูล ซึ่งมีรายละเอียดโค้ดดังนี้

```
for  i=0 to k
        do C[ i ] = 0
for  j=1 to length[A]
        do C[A[ j ]] = C[A[ j ]]+1
for i = 1 to k
        do C[i] = C[i] + C[i-1]
for j=length[A] downto 1
        do B[C[A[ j ]]] = A[ j ]
        C[A[ j ]] = C[A[ j ]] -1
```

การวิเคราะห์ running time ของ counting sort พบว่า ถ้าเรามีอินพุตขนาด n ใดๆ ซึ่งค่าข้อมูลจะอยู่ในช่วง 0 ถึง k เท่านั้น ถ้ากำหนดให้ k คือเลขจำนวนเต็มใดๆ ดังนั้น

- โค้ดบรรทัดที่ 1-2 ใช้เวลา $\Theta(k)$
- โค้ดบรรทัดที่ 3-4 ใช้เวลา $\Theta(n)$
- โค้ดบรรทัดที่5-6 ใช้เวลา $\Theta(k)$
- โค้ดบรรทัดที่7-9 ใช้เวลา $\Theta(n)$

ดังนั้นโดยรวม การเรียงข้อมูลจะใช้เวลา $\Theta(k+n)$ ถ้าเราให้ running time ของ k ประมาณ k = O(n) running time โดยรวมของ quick sort คือ $\Theta(n)$ ซึ่ง Counting sort นั้นมีลักษณะ stable (คำว่า stable คือ ตัวเลขที่มีค่าเท่ากัน ที่อยู่ในอินพุตลำดับก่อนหลัง เมื่อเรียงแล้วก็ต้องอยู่ลำดับก่อนหลังตามนั้นด้วย)

2. **Radix sort** เป็นอัลกอริทึมที่ใช้เรียงข้อมูล ซึ่งมีลักษณะเป็น stable ด้วย ซึ่งมักจะใช้อัลกอริทึมนี้ในการเรียงข้อมูลที่มีการบันทึกเป็นหลายๆส่วน เช่น เรียงวันที่ (เนื่องจากวันที่มี ปี เดือน และวัน แยกกัน) รายละเอียดโค้ดมีดังนี้

```
for  i = 1 to d      // d is the highest-order digit
        do use a stable sort to sort array A on digit i
```

การวิเคราะห์ running time ของ radix sort พบว่า เมื่อตัวเลขแต่ละหลักอยู่ในช่วงค่า 0 ถึง k-1 และ k เป็นค่าที่ไม่มากนัก ดังนั้นการรันตัวเลข n ตัวใดๆ ที่แต่ละหลัก จะใช้เวลา $\Theta(n+k)$ ซึ่งข้อมูลอินพุตมีทั้งหมด d หลัก ดังนั้นเวลาที่ใช้ในการรันทั้งหมดคือ $\Theta(d(n+k))$ เมื่อ d เป็นค่าคงที่ และ k = O(n) อัลกอริทึม radix sort จะใช้เวลารันแค่ linear time
ถ้ากำหนดให้ตัวเลข n จำนวนใดๆ มี b bit ต่อตัวเลขและเป็นเลขจำนวนเต็มบวก ค่า r ≤ b ดังนั้น อัลกอริทึม radix sort จะใช้เวลา $\Theta((b/r)(n+2^r))$

3. **Bucket sort** เป็นอัลกอริทึมที่ใช้เรียงข้อมูล โดยที่กำหนดให้อินพุตถูกสร้างแบบสุ่มโดยที่สมาชิกจะมีค่าอยู่ในช่วง [0,1) วิธีการคือทำการแบ่งช่วง [0,1) เป็นค่าเท่าๆกัน n ช่วง (ตรงนี้เราเรียกว่า bucket) ให้ทำการกระจายอินพุต n ตัวลงไปใส่ใน bucket แล้วก็ทำการเรียงตัวเลขในแต่ละ bucket แล้วจึงจะดึงค่าออกจาก bucket ตามลำดับ ก็จะทำให้ได้ข้อมูลที่เรียงกันเรียบร้อย อัลกอริทึม มีโค้ดดังนี้

```
n = length[A]
for i = 1 to n
        do insert A[i] into list B[          ]
for i = 0 to n-1
        do sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ... , B[n-1] together in order.
```

การวิเคราะห์ running time ของ bucket sort พบว่าขึ้นอยู่กับโค้ดบรรทัดที่ 5 เราก็ทำการวิเคราะห์ cost ของการเรียกใช้อัลกอริทึม insertion sort ในบรรทัดที่ 5 และกำหนดให้จำนวนครั้งของการเรียกใช้ insertion sort เป็น 2 -1/n ดังนั้น running time ของ bucket sort คือ

$$T(n) = \Theta(n) + n.O\left(2 - \frac{1}{n}\right) = \Theta(n)$$

<u>แบบฝึกหัด</u> ให้ทำการเรียงข้อมูลต่อไปนี้จากน้อยไปมากด้วยวิธีการ counting sort 2, 5, 0, 1, 1, 3, 4, 1, 4, 2

บทที่ ๑๓ ต้นไม้ทวิภาค (binary search tree)

Binary search tree จะมีการจัดการในรูปแบบของ binary tree โดยที่โหนดแต่ละอันจะมีโครงสร้างข้อมูลสามส่วนคือ left กับ right และ p ซึ่งจะทำหน้าที่ชี้ไปยังโหนดที่เป็นโหนด left child โหนด right child และ parent ตามลำดับ

คุณสมบัติของ binary search tree มีดังนี้

กำหนดให้ x เป็นโหนดที่อยู่ใน binary search tree ถ้าหากว่า y เป็นโหนดที่อยู่ใน subtree ด้านซ้ายของ x แล้ว key[y] ≤ key[x] ถ้าหากว่า y เป็นโหนดที่อยู่ใน subtree ด้านขวาของ x แล้ว key[x] ≤ key[y]

ซึ่งเวลาที่ใช้ในการรันจะอยู่ที่ประมาณ ความสูงของ tree โดยที่ค่าความสูงของ tree ที่คาดไว้ (expected height of a binary search tree) จะประมาณ O(lg n) ดังนั้นโดยทั่วไปแล้ว จะใช้เวลาเฉลี่ยในการทำ operation ต่างๆอยู่ที่ $\Theta(\lg n)$

สำหรับ operation ต่างๆที่ทำกับ binary search tree มีดังนี้

1. Inorder-tree-walk คืออัลกอริทึมในการแสดงค่าในโหนดของ binary search tree แต่ละโหนด ซึ่งมีโค้ดดังนี้

```
Pseudo code: Inorder-Tree-Walk(x)
if x != NIL
        then Inorder-Tree-Walk(left[x])
        print key[x]
        Inorder-Tree-Walk(right[x])
```

2. Tree-search คืออัลกอริทึมที่ใช้ในการเสิร์ชหาค่าข้อมูลใน binary search tree มีโค้ดดังนี้

```
Pseudo code: Tree-Search(x,k)
if x = NIL or k = key[x]
        then return x
if k < key[x]
        then return Tree-Search(left[x],k)
        else return Tree-Search(right[x],k)
```

```
Pseudo code: Iterative-Tree-Search(x,k)
while x != NIL and k != key[x]
        do if k < key[x]
                then x = left[x]
                else x = right[x]
return x
```

3. Tree-minimum และ Tree-maximum คืออัลกอริทึมที่ใช้หาค่าน้อยที่สุดและค่ามากที่สุดใน binary search tree ซึ่งมีโค้ดดังนี้

```
Pseudo code: Tree-Minimum(x)
while left[x] != NIL
        do x = left[x]
return x
```

```
Pseudo code: Tree-Maximum(x)
while right[x] != NIL
        do x = right[x]
return x
```

4. Tree-successor คือ อัลกอริทึมที่ทำหน้าที่หาค่าข้อมูลที่มากถัดต่อไปจากตัวปัจจุบันใน binary search tree ซึ่งมีโค้ดดังนี้

```
Pseudo code: Tree-successor(x)
if right[x] != NIL
        then return Tree-Minimum(right[x])
Y=p[x]
while y!=NIL and x = right[y]
        do x=y
        y=p[y]
return y
```

5. Tree-insert คืออัลกอริทึมที่ใช้ทำการเพิ่มเลขตัวใหม่ลงไปใน binary search tree ซึ่งมีโค้ดดังนี้

Pseudo code: Tree-Insert(T,z)

```
y = NIL
x = root[ T ]
while  x != NIL
        do y = x
        if key[ z ] < key[x]
        then x = left[x]
        else x = right[x]
p[z]  = y
if y = NIL
        then root[ T ] = z
        else if key[z] < key[y]
                then left[y] = z
                else right[y] = z
```

6. Tree-delete คืออัลกอริทึมที่ใช้ทำการลบโหนดออกจาก binary search tree ซึ่งมีโค้ดดังนี้

Pseudo code: Tree-Delete (T,z)

```
if  left[z] = NIL or right[z] = NIL
        then y = z
        else y = Tree-Successor(z)
if  left[y] != NIL
        then x=left[y]
        else x = right[y]
If x != NIL
        then p[x] = p[y]
If p[y] = NIL
        then root[T] = x
        else if y = left[ p[y]]
                then left[ p[y]] = x
                else right[ p[y]] = x
if y != z
        then key[z] = key[y]
        copy y's satellite data into z
return y
```

บทที่ ๑๔ ต้นไม้เอวีแอล (AVL tree)

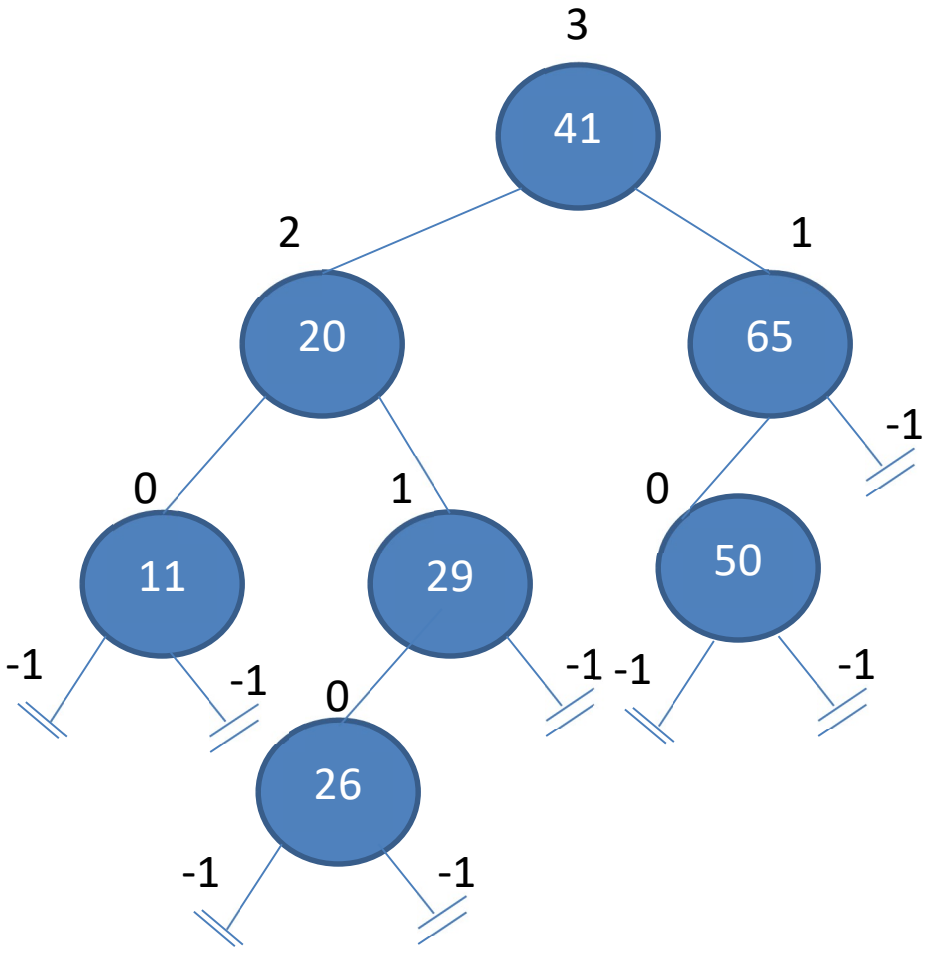
คือการทำให้ binary search tree นั้นมีความสูงของ tree แบบสมดุล (balance)

ความสูงของทรี (height of a tree) คือความยาวของ path ที่ยาวที่สุดตั้งแต่ root จนถึง leaf

ความสูงของโหนด (height of a node) คือความยาวของ path ที่ยาวที่สุดตั้งแต่ node นั้นๆจนถึง leaf

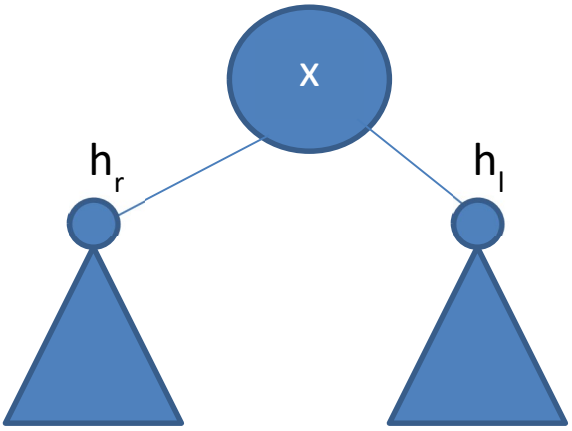
ตัวอย่างเช่นภาพด้านล่าง ความสูงของ tree คือ 3 ส่วนความสูงของโหนดเลข 50 คือ 0

ดังนั้น ความสูงของโหนด = ค่าที่มากที่สุดระหว่าง โหนดลูกด้านซ้าย กับโหนดลูกด้านขวา



ALV tree คือ binary search tree ที่มีความสูง balance ซึ่งมีเงื่อนไขว่า สำหรับแต่ละโหนด x ใดๆ ความสูงของ subtree ด้านซ้ายและด้านขวาของ x จะต้องมีความแตกต่างกันไม่เกิน 1 ดังนั้น AVL tree ที่มีจำนวนโหนด n โหนด จะมีความสูงคือ O(lg n)

$$|h_r - h_r| \leq 1$$



ในการสร้าง AVL tree ให้มีความสูง balance นั้น ในกรณี worse case คือเมื่อ sub tree ด้านขวา มีความสูงมากกว่า sub tree ด้านซ้าย อย่างน้อย 1 เสมอๆ

ดังนั้นเมื่อวิเคราะห์ความสูงของ AVL tree จะพบว่า ถ้าหาก เรากำหนดให้ N_h คือเลขจำนวนโหนดน้อยที่สุดใน ALV tree ที่มีความสูง h ดังนั้น

$N_h = 1 + N_{h-1} + N_{h-2}$

$N_h > F_h$ เมื่อให้ F_h เป็นฟังก์ชัน Fibonacci

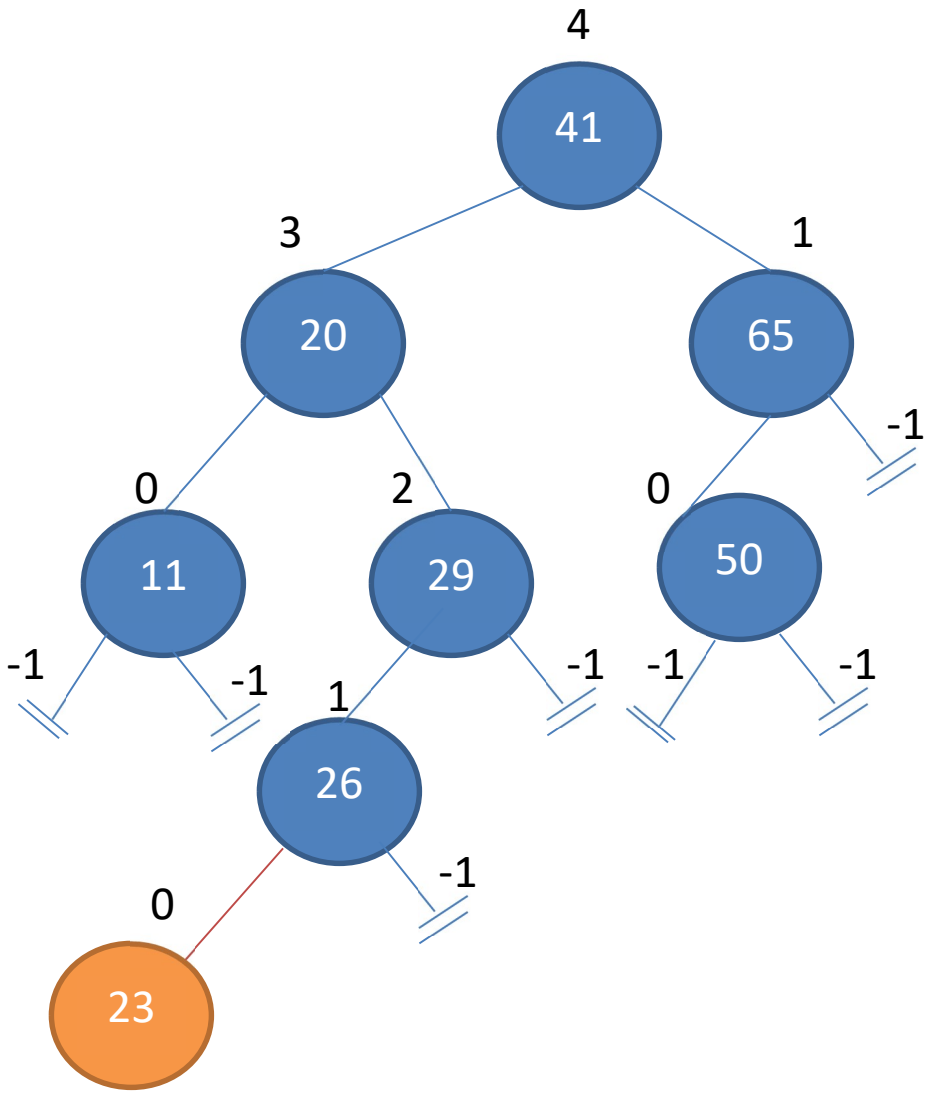
$N_h > 1.618^h / 5^{1/2}$

กำหนดให้ $N_h = n$

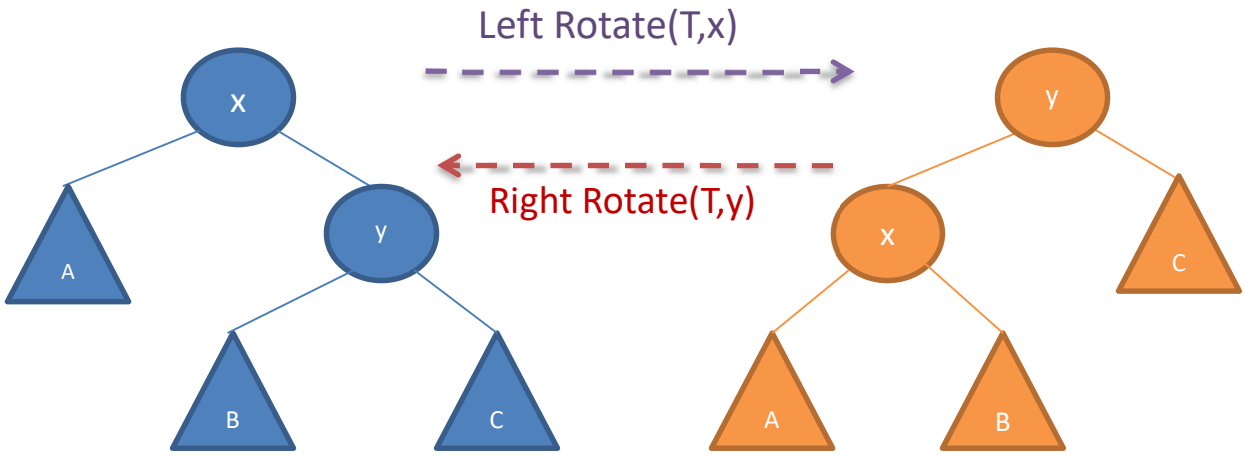
$1.618^h / 5^{1/2} < n$

จะได้ว่า $h < 1.440 \lg n$ ดังนั้น ความสูงของ ALV tree จะประมาณ lg n

การเพิ่มข้อมูลใน AVL tree ทำได้ โดยการใช้วิธีการ binary search tree insert แล้วจึงทำการ rotate เพื่อให้ความสูงของทรี balance ดังตัวอย่างด้านล่างนี้ เมื่อทำการ insert โหนดที่มีเลข 23 เข้าไป แล้วพบว่า tree ไม่ balance ดังรูป



ก็จะต้องทำการ rotate ทรีเพื่อให้ความสูงนั้น balance ซึ่งหลักการ rotate มี 2 แบบคือ left rotate และ right rotate แสดงดังรูป



สำหรับอัลกอริทึมในการทำ left rotate มีดังนี้

Pseudo code: Left-Rotate(T,x)

```
y = right[x]
right[x] = left[y]
if left[y] != nil[T]
        then p[left[y]] = x
p[y] = p[x]
if p[x]= nil[T]
        then root[T] = y
else if x = left[p[x]]
        then left[p[x]] = y
else right[p[x]] = y
left[y] =x
p[x] = y
```

เมื่อเราสร้าง AVL tree ซึ่งมีลักษณะเป็น balanced binary search tree แบบนี้แล้วทำให้เวลาที่ใช้ในการเรียงข้อมูล ใช้น้อยลงดังนี้

1. การ insert ข้อมูล n จำนวน จะใช้เวลาเพียง O(n lg n)
2. การแสดงข้อมูลในทรี (inorder tree traversal) ใช้เพียง O(n)

<u>แบบฝึกหัด</u>

1. ทำการสร้าง AVL tree โดยใช้ฟังก์ชัน AVL insert เมื่อกำหนดให้อินพุตคือ 4, 7 , 5, 10, 23 , 65, 73

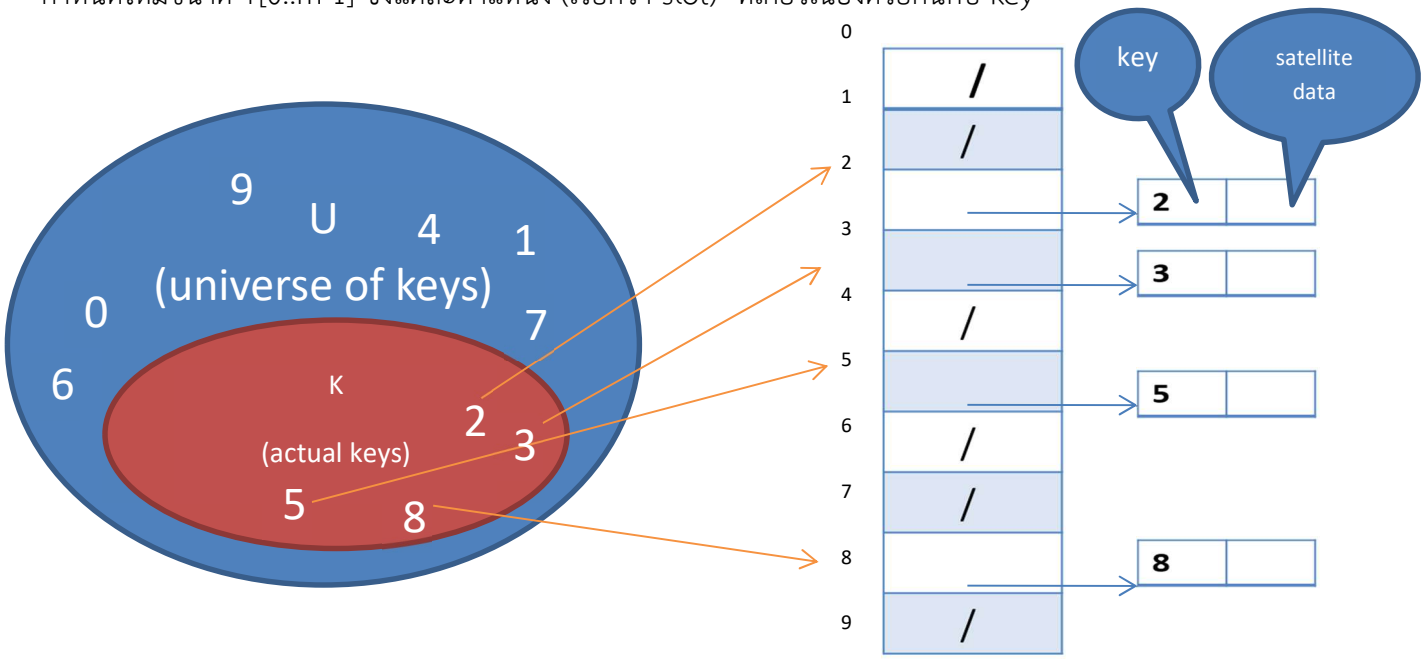
บทที่ ๑๕ ตารางแฮช (Hash tables)

หลักการของ dictionary (Abstract data type) คือการจัดการชุดข้อมูลแต่ละอันด้วย ค่าคีย์ข้อมูล ซึ่งสามารถทำได้หลาย operation ดังนี้

1. INSERT(item)
2. DELETE(item)
3. SEARCH(key)

hash table เป็นโครงสร้างข้อมูลที่ใช้สำหรับการสร้าง dictionary ซึ่ง worse case running time สำหรับการ search คือ O(n) แต่ว่า expected time คือ O(1)

ก่อนอื่นเรามาดูโครงสร้างข้อมูลอีกแบบที่เรียกว่า **direct address table** ก่อน ซึ่งมีเซตของ universe U = {0,1,...,m-1} โดยที่ m มีค่าไม่มากนัก และให้ key เป็นค่าที่อยู่ใน U ดังนั้น direct-address table คืออาเรย์ที่กำหนดให้มีขนาด T[0..m-1] ซึ่งแต่ละตำแหน่ง (เรียกว่า slot) ที่เกี่ยวเนื่องด้วยกันกับ key



ซึ่งเวลาที่ใช้ในการทำ operation ต่างๆ จะใช้เวลาเพียงแค่ O(1) เท่านั้น ซึ่งโค้ดของแต่ละ operation มีดังนี้

```
DIRECT-ADDRESS-SEARCH(T, k)
        return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
        T[key[x]] = x
```

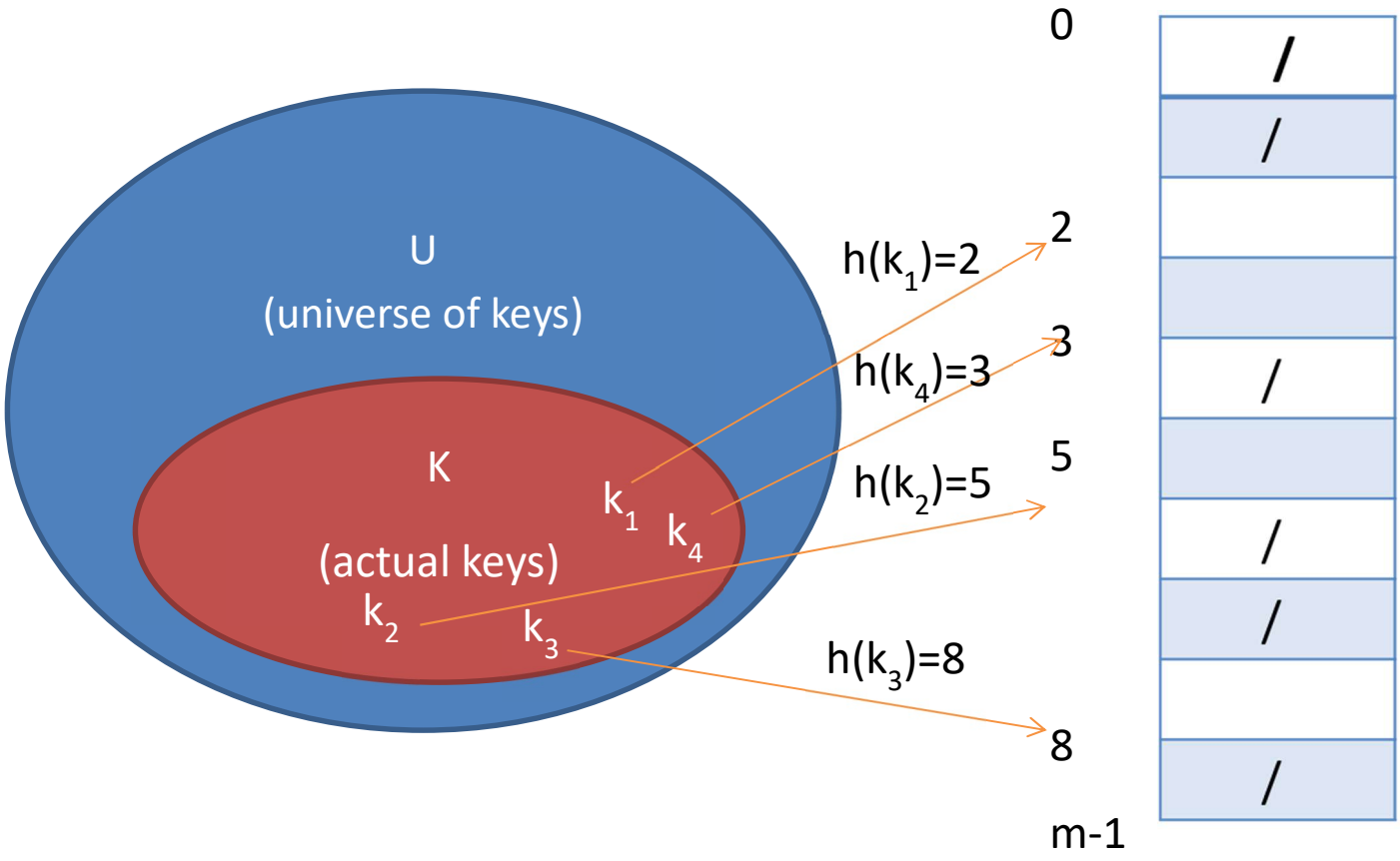
```
DIRECT-ADDRESS-DELETE (T, k)
        T[key[x]] = NIL
```

ข้อเสียของการทำ direct addressing table คือ

1. ค่าคีย์ อาจจะไม่เป็นเลข ที่ไม่ใช่เลขติดลบ ซึ่งวิธีการแก้ไขก็สามารถทำได้โดย ใช้ prehash เพื่อแมปจากคีย์ไปยังเลขที่ไม่เป็นเลขติดลบ
2. Direct address table ต้องการใช้หน่วยความจำมาก วิธีการแก้ไขก็สามารถทำได้โดยใช้หลักการ hashing

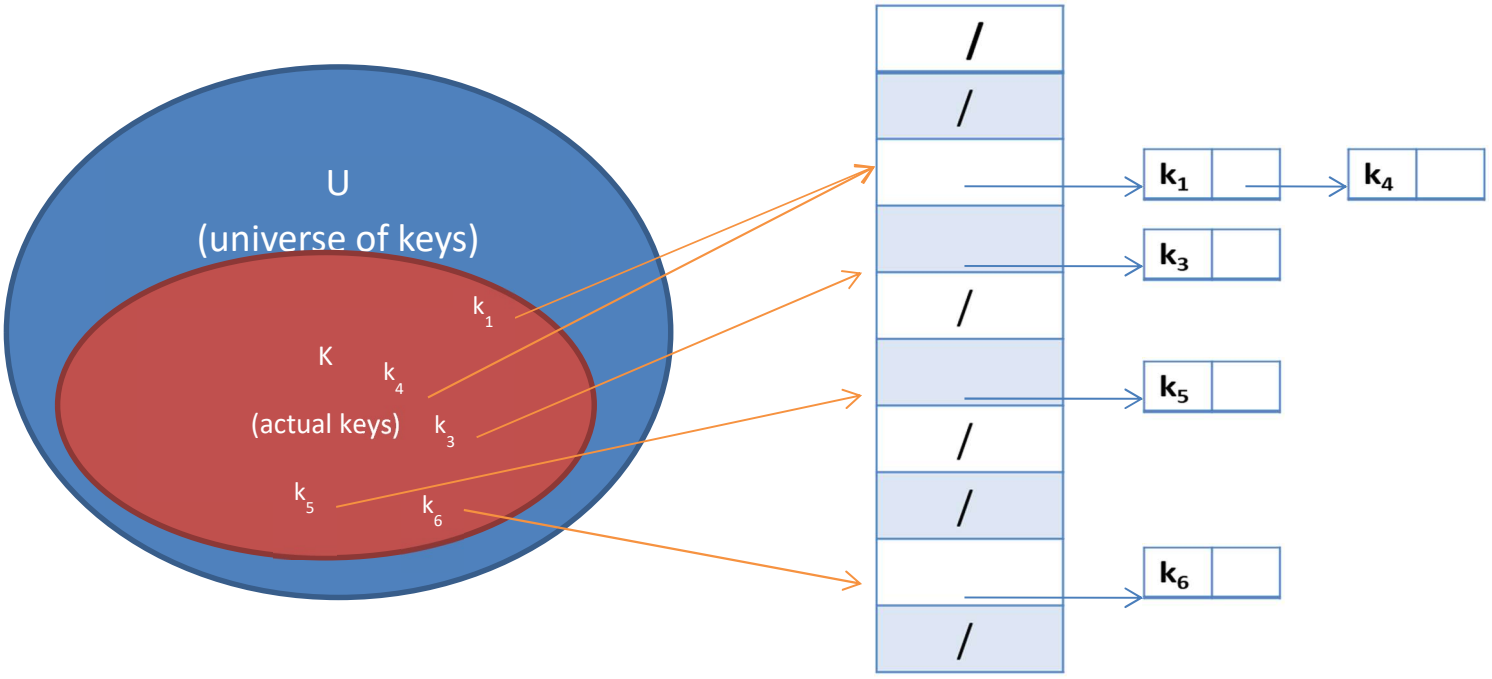
Hash table ถูกสร้างขึ้นมาเพื่อใช้ในการจัดการข้อมูลเหมือน direct address table แต่ว่าวิธีนี้เราจะใช้ฟังก์ชัน hash ในการคำนวณหา slot ที่จะเก็บ key ดังนั้น ฟังก์ชัน hash จะทำการแมป ค่าคีย์ใน universe ไปยัง slot ของ hash table ที่มีขนาด T[0..m-1] ได้เป็นฟังก์ชัน h: U -> {0,1,...,m-1}

เราถือว่าสมาชิกที่มีค่าคีย์ k ใดๆ จะถูก hash ให้ตรงกับ slot h(k) หรือเรียกอีกอย่างว่า h(k) คือค่า hash ของ k



ในการทำ hashing จะมีปัญหาหนึ่งเรียกว่า collision คือการที่ ใส่ค่าลงไปใน hash ฟังก์ชันแล้วได้ค่า slot เดียวกัน วิธีการแก้ไขมี 2 วิธีดังนี้

1. Hash with chaining คือการสร้างลิสไว้เก็บคีย์ในกรณีที่ slot นึงมีคีย์มากกว่า 1 ค่า ดังรูป



ซึ่งเวลาที่ใช้ในการทำ operation ต่างๆ จะใช้เวลาเพียงแค่ O(1) เท่านั้น ซึ่งโค้ดของแต่ละ operation มีดังนี้

CHAINED-HASH-SEARCH(T, k) search for an element with key k in list T[h(k)]
CHAINED-HASH-INSERT(T, x) insert x at the head of list T[h(key[x])]
CHAINED-HASH-DELETE (T, k) delete x from the list T[h(key[x])]

วิเคราะห์ running time ของอัลกอริทึม hash with chaining เริ่มจากเราตั้งสมมติฐานว่า ไม่ว่าจะมีอินพุตอะไรเข้ามา โอกาสที่อินพุตเลขนั้นจะถูก hash เข้าไปใส่ใน slot ทั้งหมด m สล็อตนั้นมีค่าเท่าๆกัน ซึ่งเรียกสมมติฐานนี้ว่า simple uniform hashing ดังนั้น

ให้ j = 0, 1 , ..., m-1 แล้วกำหนดให้ความยาวของลิส T[j] เท่ากับ n_j ดังนั้น

$$n = n_0 + n_1 + ...+ n_{m-1}$$

ค่าเฉลี่ยของ n_j คือ $E[n_j] = \alpha = n/m$

เราสมมติต่อว่าค่า hash h(k) ถูกคำนวณด้วยเวลา O(1) ดังนั้นเวลาที่ใช้ในการเสิร์ชหา key k ใดๆนั้นก็จะใช้เวลาขึ้นอยู่กับความยาว $n_{h(k)}$ ของลิส T[h(k)] พิจารณา 2 กรณีคือ

1. กรณีที่เสิร์ชไม่สำเร็จ ซึ่ง ค่า expected time ที่จะเสิร์ชไม่สำเร็จคือ expected time ที่จะต้องเสิร์ชไปจนถึงสมาชิกตัวสุดท้ายของลิส T[h(k)] ซึ่ง ลิส T[h(k)] มีค่าความยาว expected $=E[n_{h(k)}] = \alpha$ ดังนั้น ค่า expected จำนวนของสมาชิกในลิสที่จะต้องเอามาคิดคือ α และค่าเวลาทั้งหมดที่ต้องใช้ (รวมถึงเวลาที่ต้องคำนวณ h(k) ด้วย) $= O(1+ \alpha)$

2. กรณีที่เสิร์ชสำเร็จ และหาค่า k เจอ ซึ่งค่า expected time ที่จะเสิร์ชหาสมาชิก x ใดๆ สำเร็จคือต้องเสิร์ชมากกว่าจำนวนของสมาชิกที่ปรากฏก่อน x ไป 1 ตัว กำหนดให้ x_i คือสมาชิกตัวที่ i ที่ insert ใส่ลงไปใน table เมื่อ i= 1,2,...,n และให้ $k_i = key[x_i]$
สำหรับค่าคีย์ k_i และ k_j เรากำหนดค่า random variable $X_{ij} = I\{h(k_i)=h(k_j)\}$ ซึ่งจากสมมติฐานที่ว่าเรามี simple uniform hashing ดังนั้น $Pr\{h(k_i)=h(k_j)\} = 1/m$ และ $E[X_{ij}]= 1/m$
ดังนั้น expected จำนวนของสมาชิกในลิสที่จะต้องถูกค้นหาจนกว่าจะสำเร็จคือ

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E\left[X_{ij}\right])$$

$$=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m})$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{nm}(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i)$$

$$= 1 + \frac{1}{nm}(n^2 - \frac{n(n+1)}{2})$$

$$= 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

ถ้าหากว่าจำนวน slot ของ hash table มีค่าอย่างน้อยเป็นสัดส่วนของจำนวนสมาชิกใน table เราจะมี n = O(m) และดังนั้น α=n/m = O(m)/m = O(1) และการค้นหาจะใช้เวลาโดยเฉลี่ยแค่ O(1) เท่านั้น

ก่อนที่จะไปดูวิธีแก้ปัญหาของ collision แบบที่สอง เรามาดูรูปแบบของ hash function กันก่อน

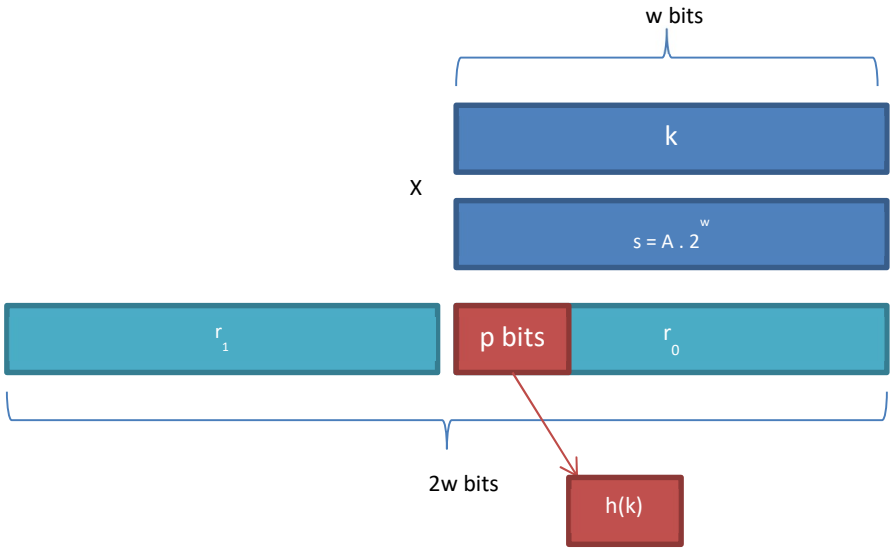
ฟังก์ชัน Hash ที่ดีต้องตรงกับเงื่อนไขสมมติฐานที่ว่าเป็น simple uniform hashing ก็คือ คีย์แต่ละค่านั้นมีโอกาสที่จะ hash ให้ไปลงใน slot ทั้งหมด m slot ได้เท่าๆกัน ซึ่งเงื่อนไขนี้ยากมากที่จะตรวจสอบได้

ลักษณะของสมการ hash function มีหลายวิธีดังตัวอย่างต่อไปนี้

The division method มีสมการคือ h(k) = k mod m ตัวอย่างเช่น ถ้าให้ hash table มีขนาด m=12 และ k = 100 ดังนั้น h(k)=4

เรามักจะเลี่ยงบางค่าของ m เช่น m ไม่ควรเป็นเลขยกกำลัง 2 แต่ว่านิยมที่จะให้ m เป็นจำนวนเฉพาะมากกว่า

The multiplication method มีสมการคือ h(k) = [m(k A mod 1)] หรือคือ
h(k) = [(k.A) mod 2^w]>>(w-p) โดยที่ ค่า A อยู่ในช่วง 0 < A < 1 (แนะนำว่า ค่า A = ($5^{1/2}$ -1)/2 = 0.6180339887) และค่า m = 2^p โดยกำหนดให้ k มีจำนวน w bit



ตัวอย่างเช่น ให้ k = 123456 , p = 14, m = 2^{14} = 16384, w = 32 ถ้าให้ A = s/ 2^{32} ซึ่งก็ประมาณ ($5^{1/2}$ -1)/2 จะได้ค่า A = 2654435769 และ k.s = 327706022297664 ดังนั้น k.s = (76300. 2^{32}) + 17612864 จะพบว่า r_0 = (76300. 2^{32}) และ r_1 = 17612864 ตัวเลข 14 บิตซ้ายของ r_0 จะทำให้ได้ค่า h(k) = 67

Universal hashing มีสมการคือ h(k) = [(ak+b) mod p] mod m โดยที่ a, b เป็นค่าแบบสุ่มอยู่ {0,1,..,p-1} และ p คือเลขจำนวนเฉพาะที่มีค่ามากกว่าขนาดของ universe ดังนั้น

กรณี worst case ค่า k_i != k_j แล้ว Pr{h(k_i)= h(k_j)} = 1/m

2. Open addressing จะมีโครงสร้างที่กำหนดให้ แต่ละตำแหน่งของ table เก็บค่าสมาชิกที่เปลี่ยนแปลงได้ หรือเก็บค่า NIL ซึ่งจะไม่มี chaining และแต่ละ slot จะเก็บค่าเดียวเท่านั้น

เมื่อทำการเสิร์ชหาค่า จะต้องหาจนกว่าจะเจอค่าหรือจนกว่าแน่ใจว่าไม่มีค่านั้น ซึ่ง open addressing นั้นทำให้ hash table เก็บค่าจนเต็มทำให้ insert ค่าเพิ่มอีกไม่ได้แล้ว ดังนั้น load factor **α** จะมีค่าไม่เกิน 1

ในการทำ insertion โดยใช้ open addressing เราต้องประเมิน (เรียกอีกอย่างว่า probe) has tableไปเรื่อยๆ จนกว่าจะเจอช่องว่าง แทนที่จะกำหนดค่าตายตัวว่าเป็น 0,1,...m-1 ลำดับของตำแหน่งของข้อมูลจะขึ้นอยู่กับ key

ฟังก์ชัน hash จะอยู่ในรูปแบบ h: U x {0,1,...,m-1} -> {0,1,...,m-1} สำหรับทุกๆค่าคีย์ k, ลำดับของ probe คือ <h(k,0) , h(k,1), ... ,h(k, m-1)> ซึ่งจะเป็น permutation ของ <0,1,...,m-1>

ซึ่งอัลกอริทึมของ hash table แสดงดังต่อไปนี้

Pseudo code: HASH-INSERT (T, k)

```
i = 0
repeat j = h(k,i)
        if T[ j ] = NIL
                then T [j ] = k
                        return j
        else i = i+1
until i = m
error “hash table overflow”
```

Pseudo code: HASH-SEARCH (T, k)

```
i = 0
repeat j = h(k,i)
        if T[ j ] = k
                then return j
        i = i+1
until T[ j ]=NIL or i =m
return NIL
```

สำหรับสมการ hash function ที่ใช้ในการ probe มีหลายสมการดังนี้

Linear probing คือ ถ้ากำหนดให้ ฟังก์ชัน hash เดิมคือ h’: U -> {0,1,...,m-1} และ วิธีการของ linear probing ใช้ฟังก์ชัน hash คือ h(k,i) = (h’(k) + i) mod m เมื่อ i = 0,1,...,m-1 ซึ่งข้อเสียของวิธีนี้คือใช้เวลานานมากในการสร้าง slot และค่าเวลาเฉลี่ยการเสิร์ชข้อมูลก็เพิ่มขึ้น

Quadratic probing คือ ถ้ากำหนดให้ฟังก์ชัน hash เดิมคือ h': U -> {0,1,...,m-1} และวิธีการของ quadratic probing ใช้ฟังก์ชัน hash คือ $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ เมื่อ i = 0,1,...,m-1 และ c_1 กับ c_2 ไม่เท่ากับ 0

Double probing คือ ถ้ากำหนดให้ฟังก์ชัน hash เดิมคือ h': U -> {0,1,...,m-1} และวิธีการของ double probing ใช้ฟังก์ชัน hash คือ $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$ เมื่อ i = 0,1,...,m-1 และ $h_1(k)$ และ $h_2(k)$ เป็น hash function ใดๆ ค่าของ $h_2(k)$ ต้องมีค่าเป็นจำนวนเฉพาะเทียบกับขนาดของ hash table (m)

วิเคราะห์ running time ของ open addressing จะพบว่าเราจะมีข้อมูลได้มากที่สุด 1 ข้อมูลต่อ 1 slot ดังนั้น $n \le m$ ซึ่งหมายถึงว่า $\alpha \le 1$ เราสมมติฐานว่า มี uniform hashing ดังนั้น ลำดับของการ probe <h(k,0) , h(k,1), ... ,h(k, m-1)> จะใช้เพื่อ insert หรือ search หาค่าคีย์แต่ละคีย์ ซึ่งประมาณเท่าๆกับ permutation ลำดับใดๆ ของ (0,1,...,m-1)

ค่า expected ของจำนวนครั้งที่ทำ probe ในกรณีที่ search ไม่สำเร็จอยู่ที่สูงสุดคือ $1/(1-\alpha)$ ดังนั้น การ insert ข้อมูลลงไปใน opening address hash table ด้วย load factor α จะต้องการสูงสุด $1/(1-\alpha)$ probe โดยเฉลี่ย เมื่อสมมติฐานว่ามี uniform hashing

ค่า expected ของจำนวนครั้งที่ทำ probe ในกรณีที่ search สำเร็จก็จะอยู่ที่สูงสุดคือ $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

บทที่ ๑๖ วิถีสั้นสุดจากแหล่งเดียว (single source shortest path)

ปัญหาบางอย่างต้องการหาระยะที่สั้นที่สุด เช่นคนขับรถต้องการหาทางที่สั้นที่สุดจาก เมือง Chicago ไปเมือง Boston โดยที่มีแผนที่ของอเมริกาให้ คำถามคือ เราจะหาทางที่สั้นที่สุดได้อย่างไร

รูปแบบของปัญหา shortest path

กำหนดให้ weighted directed graph G=(V,E) โดยที่ฟังก์ชัน weight คือ w : E -> R
เราจะทำการแมป edge เข้ากับค่า weight ทำได้ดังนี้
กำหนดให้ path p = (v_0 , v_1 ,..., v_k) และ (v_i , v_{i+1}) ∈ E เมื่อ 0 ≤ i < k
ค่า weight ของ path p = (v_0 , v_1 ,..., v_k) คือผลรวมของ weight ของ edge ที่มีทั้งหมด ดังสมการ

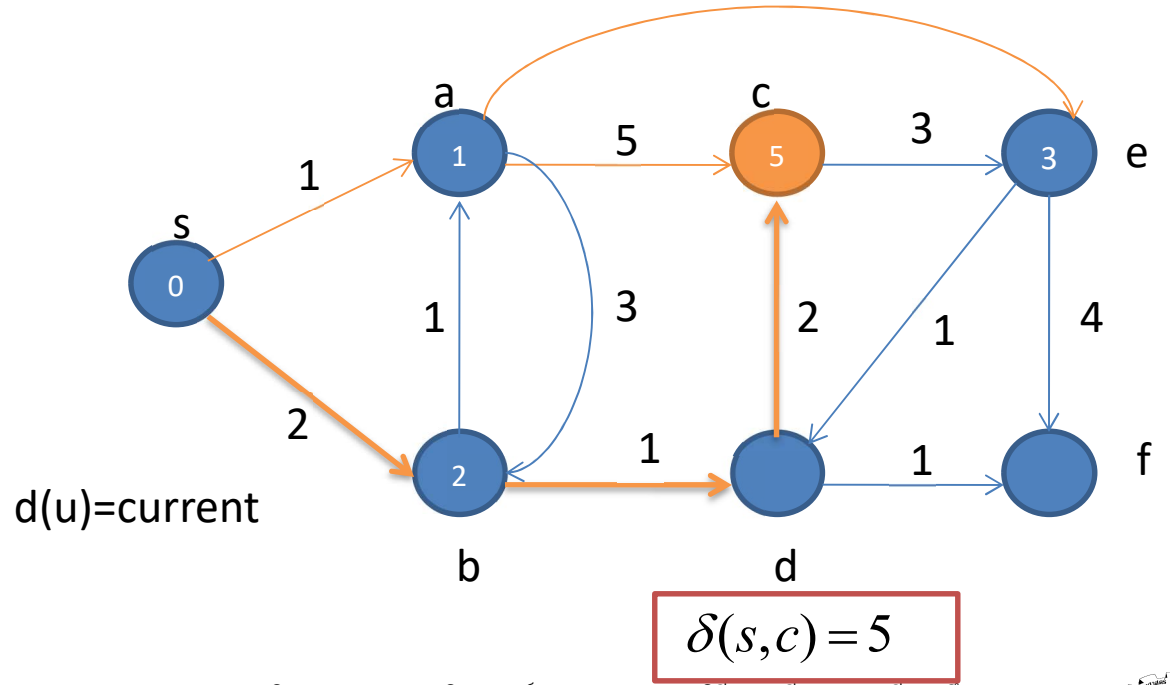
$$w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$$

เรากำหนดค่า weight ของระยะที่สั้นที่สุดจาก u ไปยัง v โดยที่

$$\delta(u,v)=\min\{w(p):u\overset{p}{\rightarrow}v\}$$

ถ้าหากว่าไม่มี path จาก u ไปยัง v แล้ว $\delta(u,v)=\infty$

ตัวอย่างดังรูป จะเห็นว่าระยะที่สั้นที่สุดจาก s ไปยัง c คิดมาจากหาระยะที่สั้นที่สุดของ path =(s,a,c) และ (s,b,d,c) และ (s,b,a,c) ซึ่งพบว่าระยะที่สั้นที่สุดคือ path ของ (s,b,d,c)



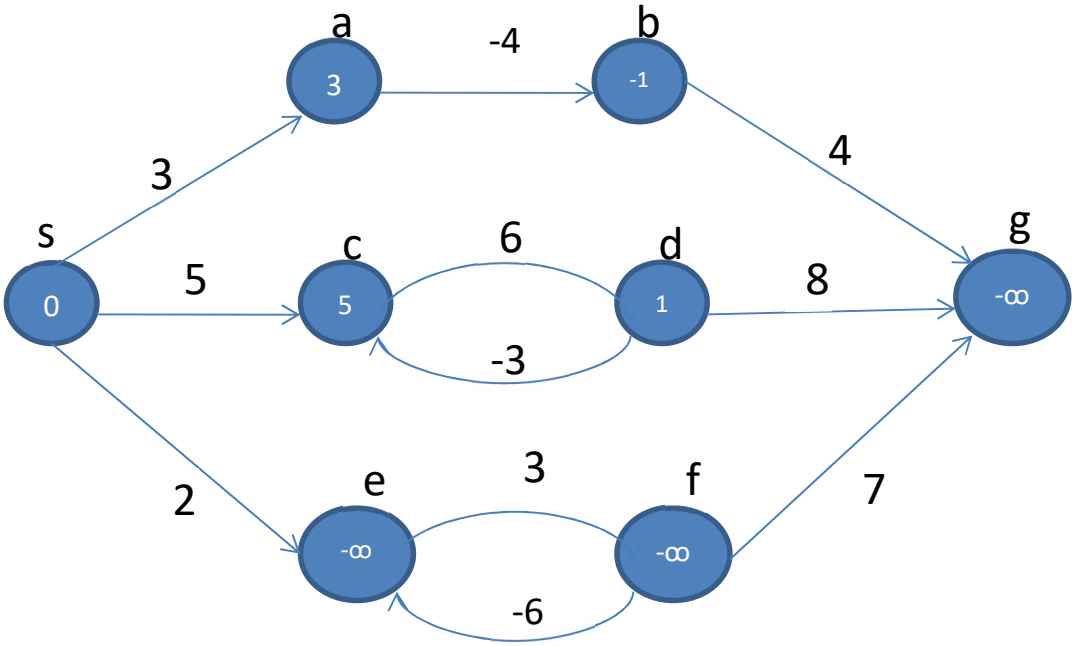
การนำเสนอปัญหา shortest path ทำได้ดังนี้

กำหนดให้กราฟ G=(V,E) สำหรับแต่ละโหนด v ∈ V เราให้ **predecessor** ¶[v] คือโหนดก่อนหน้าโหนด v มีค่าคือชื่อโหนดอื่นที่อยู่ก่อนหน้า หรือคือค่า NIL

เรากำหนดให้ d(v) เป็นค่าที่อยู่ข้างในโหนด เพื่อแสดงค่าของ weight ปัจจุบัน

เรากำหนดให้ ¶[v] เมื่อ v คือโหนดใดๆ ซึ่งแทนค่าโหนดก่อนหน้า ใน path ที่ดีที่สุดที่จะไปยังปลายทาง v

Negative weighted edges คือ edge ที่มีค่า weight เป็นลบ และหากมี negative weight cycle จากโหนดเริ่มต้น จะทำให้เราหาค่า shortest path ไม่ได้ ดังนั้นถ้ามี **negative-weight cycle** บน path จากโหนดเริ่มต้น s ไปยังปลายทาง v เราจะให้ $\delta(s,v) = -\infty$ ตัวอย่างของ negative weight cycle ดังรูป



โครงสร้างทั่วไปของ shortest path มีสองขั้นตอนหลักดังนี้

1. **Initialize single source** ทำได้โดย
   สำหรับโหนด u ∈ V เรากำหนดให้ d[v] = ∞ และ ¶[u] = NIL และ d[s] = 0
2. **Relaxation** ทำได้โดย
   เลือก edge(u,v) และทำการ relax(u,v) โดยเช็คเงื่อนไขต่อไปนี้:
   ถ้า d[v] > d[u] + w(u,v) แล้ว d[v] = d[u] + w(u,v) และให้ ¶[v] = u

โค้ดของอัลกอริทึม initialize single source และ relaxation มีดังนี้

Pseudo code: Initialize-Single-Source(G,s)

```
for each vertex v in V[G]
        do d[v] = ∞
           ¶[v] = NIL
d[s] = 0
```

Pseudo code: Relaxation(u,v,w)

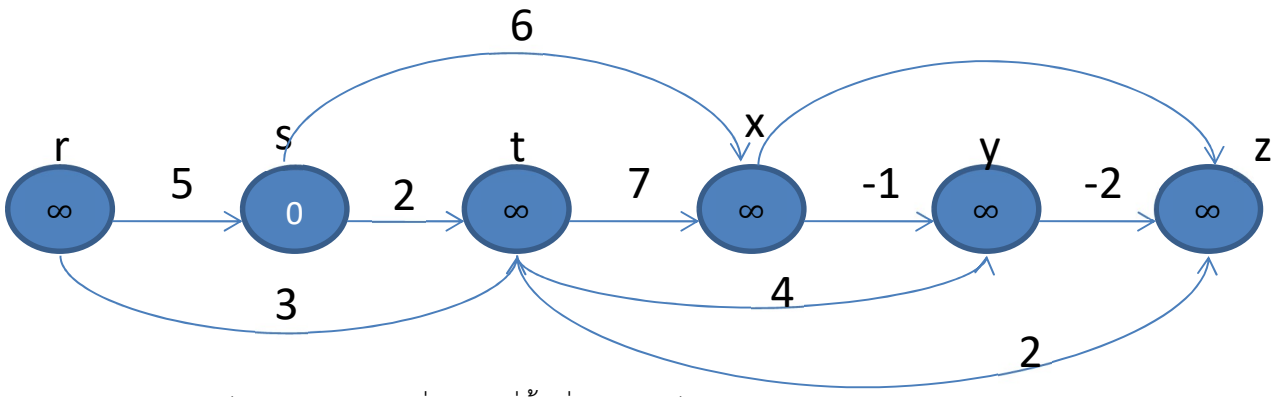
```
if  d[v] > d[u] + w(u,v)
        then d[v] = d[u] + w(u,v)
             ¶[v] = u
```

สำหรับอัลกอริทึมที่ใช้ในการแก้ปัญหา shortest path มีด้วยกันหลายวิธี ซึ่งที่เราจะกล่าวถึงมี 3 วิธีดังต่อไปนี้

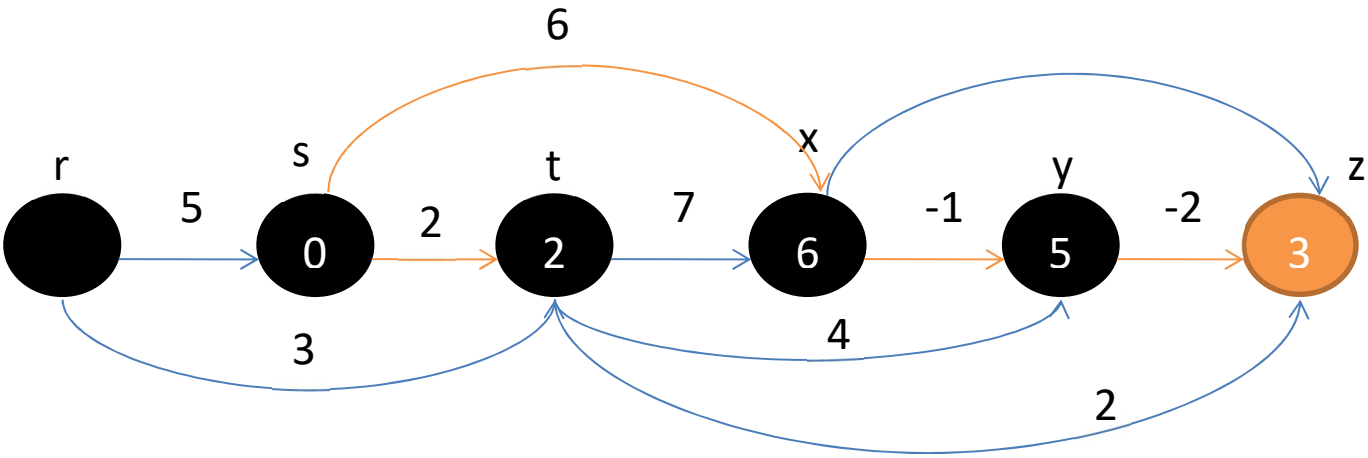
1. **Shortest path in directed acyclic graphs (DAG)** เป็นวิธีการที่เราคำนวณหาระยะที่สั้นที่สุดของ single source โดยใช้ running time เพียงแค่ O(V+E) โดยใช้วิธี relaxation กับ edge ของ weighted **directed acyclic graph**(dag) ซึ่งอัลกอริทึมมีดังนี้

```
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G,s)
for  each vertex u, taken in topologically sorted order
        do for each vertex v ϵ Adj[u]
              do RELAX(u, v, w)
```

ตัวอย่างที่ 1 จงหา shortest path จาก s ไปยัง z ของรูปต่อไปนี้ด้วยวิธี DAG



วิธีทำ จะได้คำตอบดังรูป ซึ่งระยะที่สั้นที่สุดจาก s ไปยัง z คือ path =(s,x,y,z) ด้วยระยะ weight=3



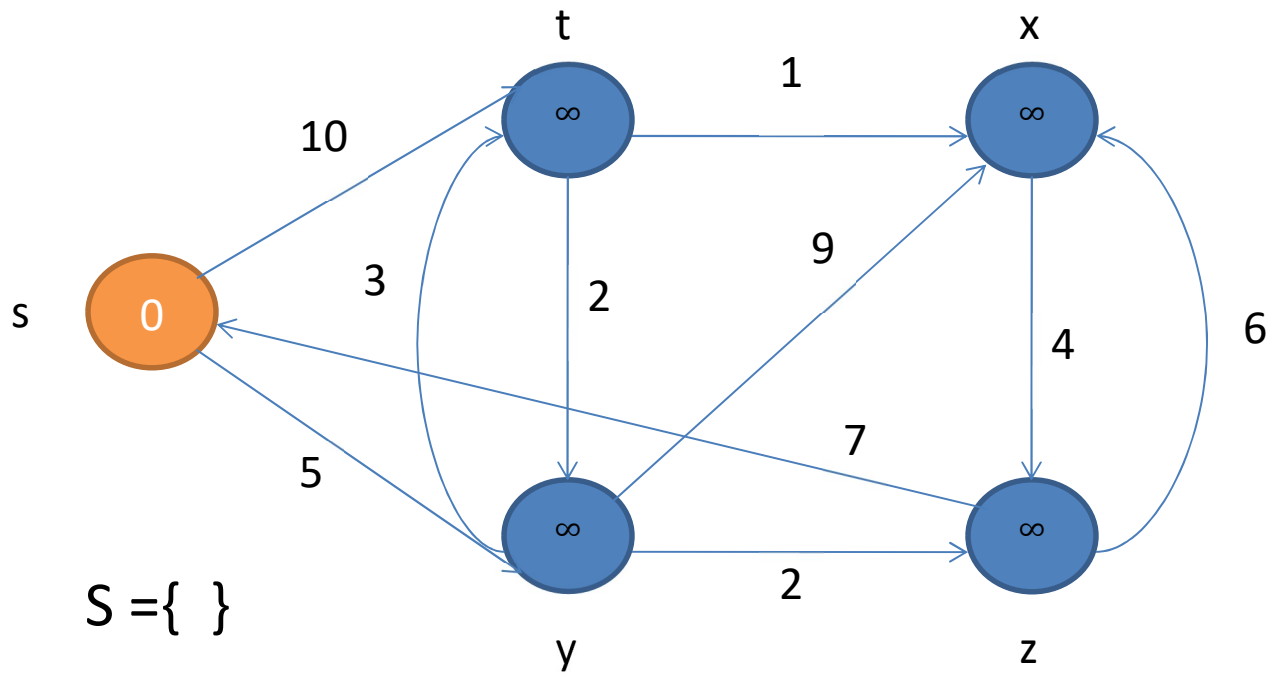
2. **Dijkstra algorithm** เป็นการแก้ปัญหาในกรณีที่ ในกราฟ weighted directed graph G = (V,E) นั้น ทุกๆ edge ไม่มี negative edge เลย เราจะสมมติว่า w(u,v) ≥ 0 สำหรับแต่ละ edge (u,v) **∈** E ซึ่งรายละเอียดโค้ดเป็นดังต่อไปนี้

Pseudo code: Dijkstra(G,w,s)

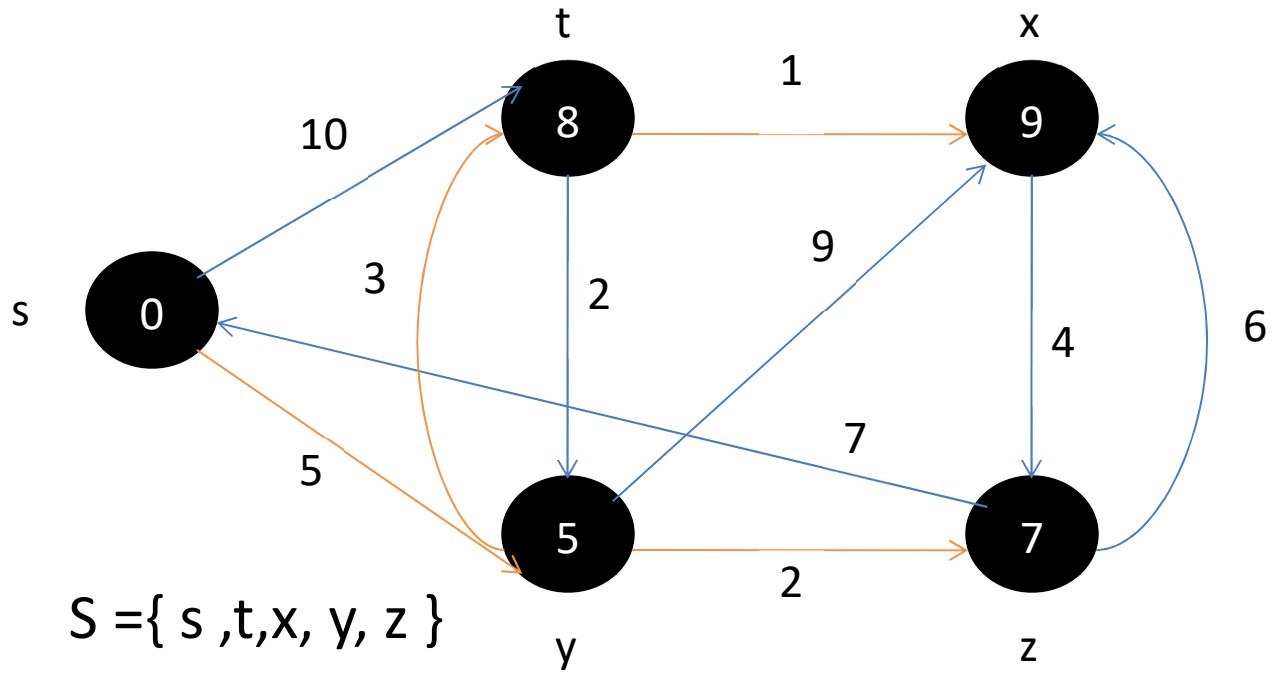
```
INITIALIZE-SINGLE-SOURCE(G,s)
S = Ø
Q = V[G]
while Q != Ø
        do u = EXTRACT-MIN(Q)
           S = S U {u}
           for each vertex v ϵ Adj[u]
                do RELAX(u,v,w)
```

วิเคราะห์ Dijkstra algorithm พบว่า running time ของมันจะขึ้นอยู่กับเราใช้วิธีไหนในการทำ min-priority queue ถ้าหากว่าเราใช้วิธี binary min-heap ซึ่งมี running time O(lg V) ถ้าหากว่าทุกๆ โหนดนั้นสามารถเดินทางไปได้จากโหนดเริ่มต้น ดังนั้นเวลารวมทั้งหมดคือ O((V+E)lg V) = O(E lg V)

ตัวอย่างที่ 2 จงหา shortest path จาก s ไปยัง z ของรูปต่อไปนี้ด้วยวิธี Dijkstra



วิธีทำ จะได้คำตอบดังรูป ซึ่งระยะทางที่สั้นที่สุดคือ path = (s,y,z) ด้วยระยะ weight = 7



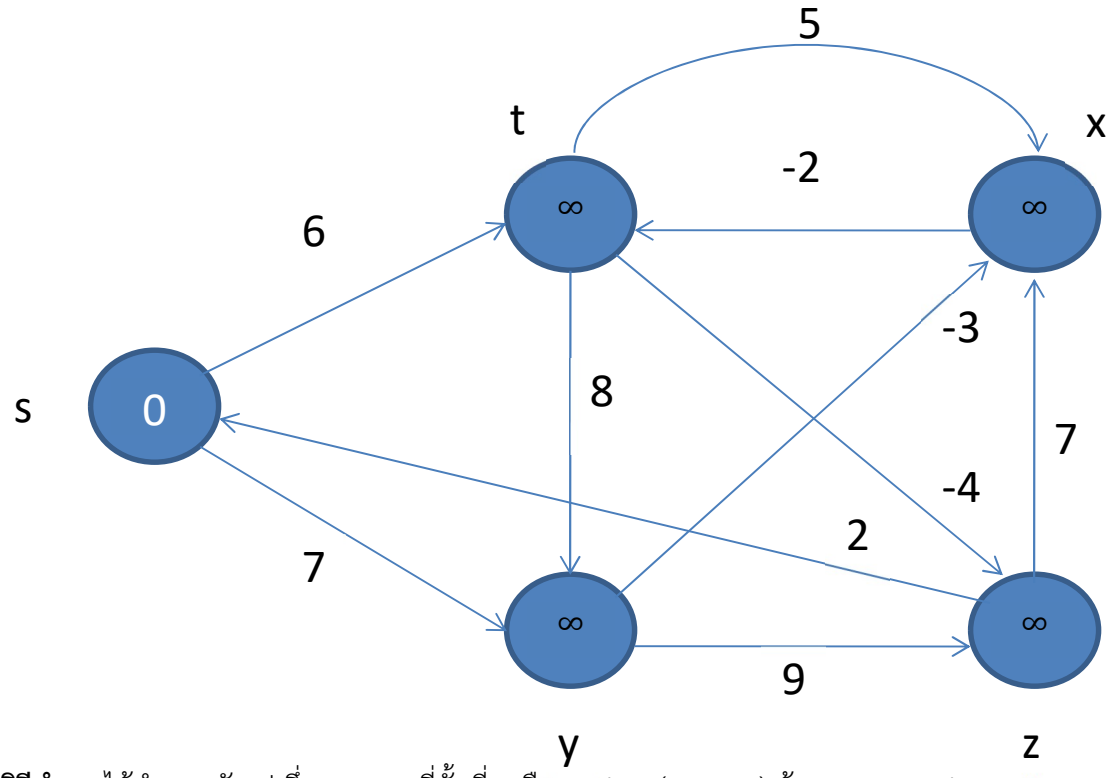
3. **Bellman-Ford algorithm** เป็นการแก้ปัญหา shortest-paths ในกรณีทั่วๆไปที่ค่า weight ของ edge อาจจะเป็นเลขติดลบ ซึ่งอัลกอริทึมนี้จะให้ค่า True/False เพื่อบอกว่า มี negative-weight cycle ที่สามารถเข้าถึงได้จากโหนดต้นทางหรือไม่ ถ้าหากว่ามี negative-weight cycle อัลกอริทึมก็จะบอกว่าไม่สามารถหาคำตอบให้กับปัญหา shortest path นี้ได้ ถ้าหากว่าไม่มี cycle ที่มีเลขติดลบ อัลกอริทึมก็จะสร้างคำตอบของ shortest path มาให้

Pseudo code: Bellman-Ford(G,w,s)

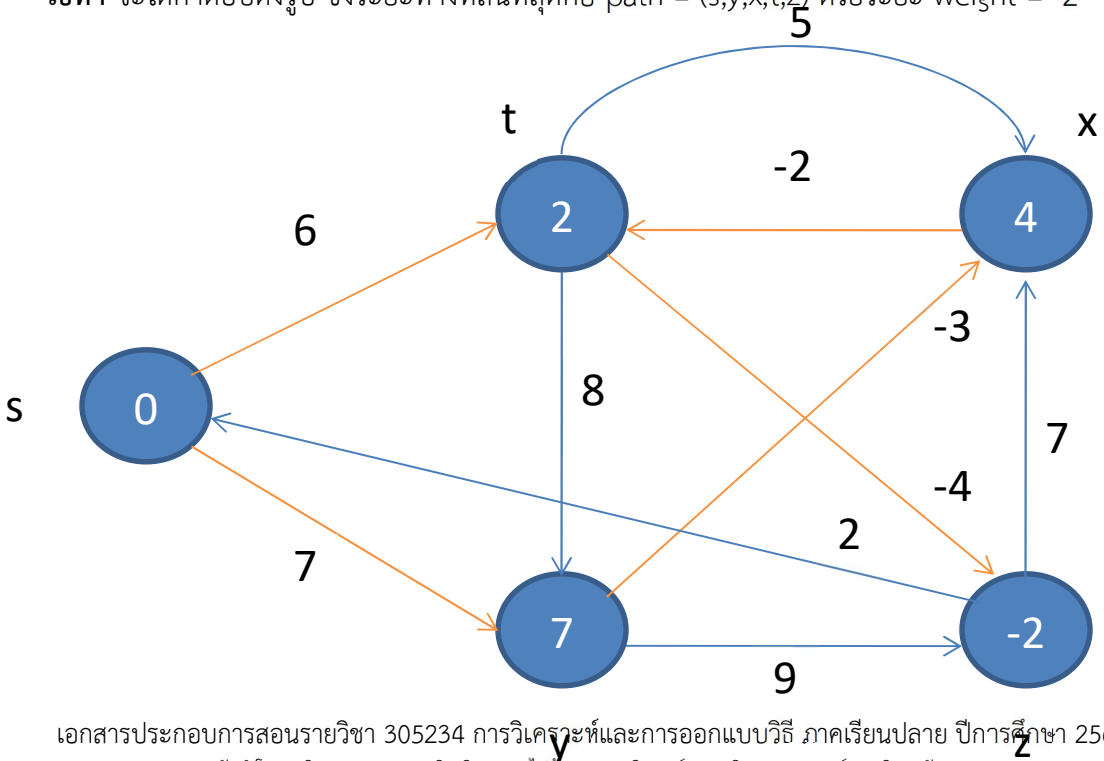
```
INITIALIZE-SINGLE-SOURCE(G,s)
for i =1 to |V[G]| -1
        do  for each edge (u,v) ϵ E[G]
                do RELAX(u, v , w)
for each edge (u,v) ϵ E[G]
        do if d[v] > d[u] + w(u,v)
                then return false
return true
```

วิเคราะห์ Bellman-Ford พบว่า ค่า running time ประมาณ O(VE) เนื่องจากขั้นตอน initialization ใช้เวลาประมาณ O(V) และโหนดอีก |V|-1 โหนดจะต้องเอา edgeมาหาค่าในโค้ดบรรทัดที่ 2-4 ทำให้ใช้เวลาไป O(E) และสำหรับ loop ในโค้ดบรรทัดที่ 5-7 ใช้เวลา O(E)

<u>ตัวอย่างที่ 3</u> จงหา shortest path จาก s ไปยัง z ของรูปต่อไปนี้ด้วยวิธี Bellman-Ford



วิธีทำ จะได้คำตอบดังรูป ซึ่งระยะทางที่สั้นที่สุดคือ path = (s,y,x,t,z) ด้วยระยะ weight = -2



บทที่ ๑๗ กำหนดการพลวัต (dynamic programming)

เป็นหลักการ ซึ่งจะทำการแก้ปัญหาด้วยการแบ่งเป็น ปัญหาย่อยๆแล้วเอาคำตอบของปัญหาย่อยมารวมกัน ซึ่งจะคล้ายกับวิธี divide-and–conquer แต่ว่า dynamic programming จะใช้ได้เฉพาะกรณีที่ปัญหาย่อยๆนั้นไม่ได้เป็นอิสระต่อกัน หมายถึง ปัญหาย่อยๆนั้นแชร์บางส่วนกันอยู่

dynamic-programming อัลกอริทึมจะแก้ปัญหาย่อยๆแค่ครั้งเดียวและบันทึกคำตอบไว้ในตารางเพื่อลดเวลาที่จะต้องใช้ในการคำนวณคำตอบ ซึ่งอัลกอริทึมนี้มักจะถูกใช้ในงานด้าน optimization ซึ่งสามารถมีคำตอบได้หลากหลายแบบ ดังนั้นเป้าหมายคือจะต้องหาคำตอบที่มี optimal (น้อยสุดหรือมากสุด)

ตัวอย่างเช่น ปัญหา Fibonacci มีสมการคือ ซึ่งเราต้องการที่จะคำนวณหา F_n

- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$

หากเราใช้วิธีการต่างๆกันในการแก้ปัญหา Fibonacci ดังนี้

1. ใช้วิธี **Naïve Recursive Algorithm** ดังโค้ดด้านล่าง จะพบว่าเวลาที่ใช้ในการรันเป็น exponential เพราะ T(n) = T(n-1) + T(n-2) + $\Theta(1)$ ดังนั้น T(n) >= 2 T(n-2) จะได้ว่า $T(n) = \Theta(2^{n/2})$

```
Pseudo code: fib(n)
        if n <= 2
                then f = 1
        else    f = fib(n-1) + fib(n-2)
return f
```

2. ใช้วิธี **Memoized Dynamic Programming Algorithm** ดังโค้ดด้านล่าง จะพบว่า fib(k) จะทำการคำนวณแค่ครั้งเดียว เมื่อตอนถูกเรียกใช้ครั้งแรกเท่านั้น ซึ่ง ค่า running time ในการเรียกส่วนที่มีการจำค่าไว้ สำหรับทุกๆ k ใดๆ คือ $\Theta(1)$ และจำนวนครั้งในการเรียกส่วนที่ไม่ได้จำค่าไว้ คือ n
   fib(1), fib(2), … , fib(n)
   และส่วนที่ไม่มี recursive ก็มี running time เท่ากับ $\Theta(1)$
   ดังนั้น running time ของอัลกอริทึมนี้ เท่ากับ $\Theta(n)$

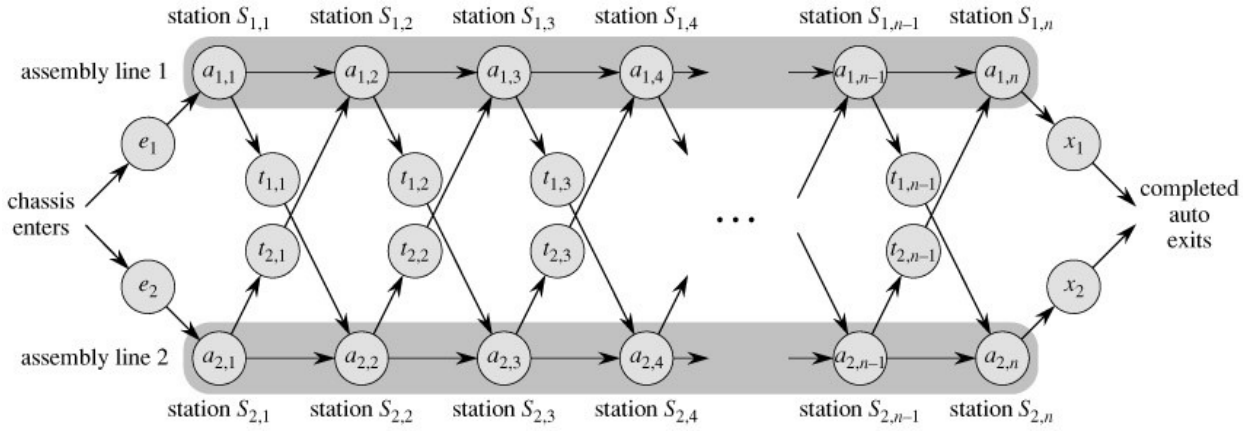
```
Pseudo code: fib(n)
memo= { }
        if n is in memo
                then return memo[n]
        if n <=2
                then f =1
        else    f = fib(n-1) + fib(n-2)

        memo[n] = f
return f
```

ทีนี้เรามาดูวิธี Dynamic programming กันบ้างว่ามีลักษณะอย่างไร ซึ่งหลักการโดยทั่วไปคือมีการจำและใช้คำตอบซ้ำๆกับปัญหาย่อยๆ ดังนั้น dynamic programming คือการ recursion และ memorization ค่า running time ก็จะเท่ากับ จำนวนของปัญหาย่อย คูณกับ เวลาที่ใช้ต่อปัญหาย่อย

ดังตัวอย่าง **Bottom-up Dynamic programming algorithm** ต่อไปนี้ ที่มี running time = $\Theta(n)$

```
fib= { }
for  k from 1 to n :
        if k <=2
                then f =1
        else    f = fib[k-1] + fib[k-2]

        fib[k] = f
return fib[n]
```

ปัญหานึงที่นิยมเอา dynamic programming algorithm เข้ามาใช้ คือ assembly line scheduling ดังตัวอย่างด้านล่าง



ที่มาภาพ Thomas H. Corment, et al. Introduction to algorithms. 3rd ed

ปัญหา **assembly line scheduling** คือการหาวิธีที่ทำให้การผลิตในโรงงานเร็วที่สุด ซึ่งมีอยู่ 2ทางเลือกคือ มาจาก station $S_{1,j-1}$ และไปต่อที่ station $S_{1,j}$

หรือ มาจาก station $S_{2,j-1}$ และย้ายไปยัง station $S_{1,j}$

เมื่อ $a_{i,j}$ คือค่าเวลาที่ใช้ในการผลิต ณ station i,j ใดๆ และ $t_{i,j}$ คือเวลาที่ใช้ในการย้ายจาก station i,j ไปยัง station อื่น และ e_i คือเวลาที่ใช้ในการเข้าสู่ระบบ ณ ทาง assembly line i ใดๆ กับ x_i คือเวลาที่ใช้ในการออกจากระบบ ณ assembly line i ใดๆ

ซึ่งวิธีแก้ปัญหานี้เป็น recursive โดยที่ เรากำหนดให้ f*แทนค่าเวลาที่ใช้น้อยที่สุดในการผลิต ดังนั้น

$f^* = \min(f_1[n] + x_1 , f_2[n] + x_2)$ และกำหนดให้

$f_1[1] = e_1 + a_{1,1}$

$f_2[1] = e_2 + a_{2,1}$

ดังนั้น การแก้ปัญหา recursive นี้จะมีสมการดังนี้

$f_1[j] = f_1[j-1] + a_{1,j}$, and

$f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$

$f_1[j] = \min (f_1[j-1] + a_{1,j} \ , f_2[j-1] + t_{2,j-1} + a_{1,j})$

$f_2[j] = f_2[j-1] + a_{2,j}$, and

$f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$

$f_2[j] = \min (f_2[j-1] + a_{2,j} \ , f_1[j-1] + t_{1,j-1} + a_{2,j})$

ซึ่งก็คือ

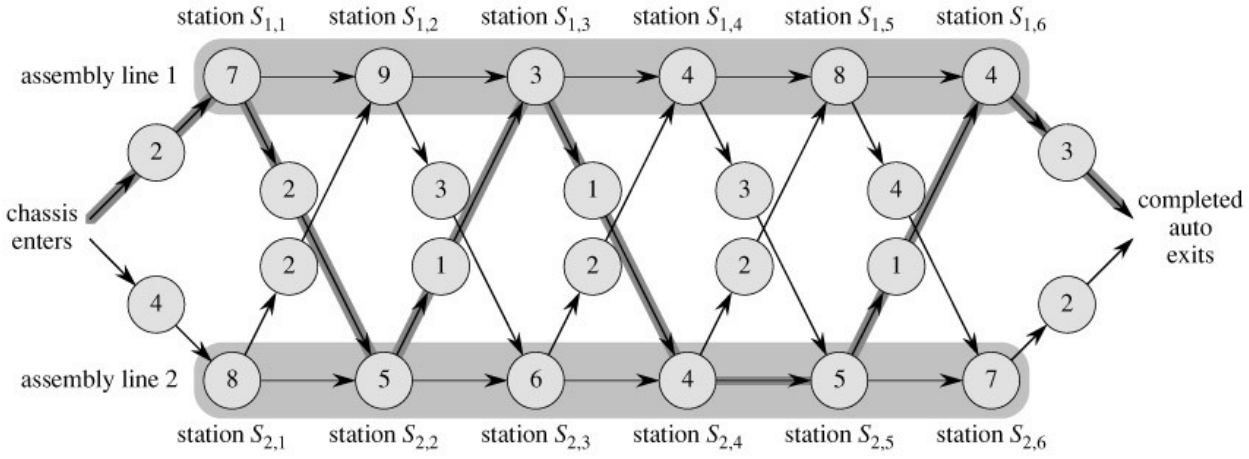
$f_1[j] = e_1 + a_{1,1}$ ถ้าหาก $j = 1$

$f_1[j] = \min (f_1[j-1] + a_{1,j} \ , \ f_2[j-1] + t_{2,j-1} + a_{1,j})$ ถ้าหาก $j \geq 2$

$f_2[1] = e_2 + a_{2,1}$ ถ้าหาก $j=1$

$f_2[j] = \min (f_2[j-1] + a_{2,j} \ , f_1[j-1] + t_{1,j-1} + a_{2,j})$ ถ้าหาก $j \geq 2$

ดังนั้นจะคำนวณคำตอบของปัญหาในตัวอย่างนี้ได้ดังรูป



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

ที่มาภาพ Thomas H. Corment, et al. Introduction to algorithms. 3rd ed

เมื่อทำการวิเคราะห์ค่า running time ของอัลกอริทึมต่างๆ ในการแก้ปัญหา พบว่า ถ้าหากเราใช้ recursion ค่า running time จะเท่ากับ $\Theta(2^{\frac{n}{2}})$ ถ้าหากเราใช้วิธี bottom-up dynamic programming ค่า running time จะเหลือแค่ $\Theta(n)$

สำหรับโค้ดที่ใช้ในการแก้ปัญหา assembly line scheduling มีดังนี้

Pseudo code: Fastest-Way(a, t, e, x, n)

$f_1[1] = e_1 + a_{1,1}$

$f_2[1] = e_2 + a_{2,1}$

for j = 2 to n

do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$

then $f_1[j] = f_1[j-1] + a_{1,j}$

$l_1[j] = 1$

else $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$

$l_1[j] = 2$

if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$

then $f_2[j] = f_2[j-1] + a_{2,j}$

$l_1[j] = 2$

else $f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$

$l_1[j] = 1$

if $f_1[n] + x_1 \leq f_2[n] + x_2$

then $f^* = f_1[n] + x_1$

$l^* = 1$

else $f^* = f_2[n] + x_2$

$l^* = 2$

เอกสารอ้างอิง

[1] Thomas H. Corment, et al. Introduction to algorithms. 3rd ed, The MIT Press, 2009.