บทที่ 3 อัลกอริทึม
(Algorithms)

3.1 ความรู้เบื้องต้น (Introduction)

3.2 สัญกรณ์สำหรับอัลกอริทึม (Notation for Algorithms)

3.3 อัลกอริทึมของยุคลิด (The Euclidean Algorithm)

3.4 อัลกอริทึมเรียกซ้ำ (Recursive Algorithms)

3.5 ความซับซ้อนของอัลกอริทึม (Complexity of Algorithms)

แบบฝึกหัด

อัลกอริทึม หมายถึง วิธีการแก้ปัญหา ทีละขั้นตอน

(An **algorithm** is a step-by-step method of solving some problem.)

อัลกอริทึม ขึ้นอยู่กับ หลักของวิชาคณิตศาสตร์ มีบทบาทสำคัญทั้งในวิชาคณิตศาสตร์ และวิชาคอมพิวเตอร์ ในการหาผลเฉลยของปัญหา ซึ่งจะถูก กระทำการ โดย คอมพิวเตอร์ ผลเฉลยนั้นต้องอธิบาย เป็นลำดับ ของขั้นตอนต่างๆ อย่างถูกต้อง

3.1 ความรู้เบื้องต้น (Introduction)

อัลกอริทึม หมายถึง เซต ของ คำสั่งต่างๆ โดยมีคุณสมบัติ หกข้อ ดังนี้

- ความถูกต้อง (precision) ขั้นตอนต่างๆ กำหนดไว้ถูกต้อง
- ความเป็นหนึ่งอย่าง (uniqueness) ผลลัพธ์ ของแต่ละขั้นตอน ของการกระทำการ นิยามเป็นหนึ่งอย่าง และขึ้นอยู่กับอินพุท และผลลัพธ์ของขั้นตอน ก่อนหน้านั้นเท่านั้น
- การจำกัด (finiteness) อัลกอริทึม จบได้ หลังจาก คำสั่ง จำนวนมากเท่าที่ จำกัด ได้รับการกระทำการ
- อินพุท (input) อัลกอริทึม รับอินพุท
- เอ้าพุท (output) อัลกอริทึม ให้เอ้าพุท
- ใช้ได้ทั่วไป (generality) อัลกอริทึม ประยุกต์ใช้กับ เซตของอินพุท

ตัวอย่าง จงพิจารณา อัลกอริทึม ข้างล่างนี้

1. x := a
2. if b > x, then x := b
3. if c > x, then x := c

ซึ่งหา ค่ามากที่สุด จาก เลขสามจำนวน a, b และ c ความคิดของอัลกอริทึมนี้คือ ตรวจดู เลข ทีละ จำนวน และทำสำเนา ค่าที่มากที่สุด ไว้ที่ ตัวแปร x เมื่อจบอัลกอริทึมนี้ x จะเป็นเลขตัวที่มีค่ามากที่สุด ในเลขสามตัวนี้

สัญกรณ์ y := z หมายถึง “ทำสำเนา ค่าของ z ไปที่ y” (“copy the value of z into y) หรือ มีความหมายเหมือนกับ “แทนที่ค่าปัจจุบัน ของ y ด้วยค่าของ z” (“replace the current value of y by the value of z”) เมื่อคำสั่ง y := z ถูกกระทำการ ค่าของ z ไม่เปลี่ยนแปลง เราเรียก เครื่องหมาย := ว่า **ตัวกำหนดค่า** (assignment operator)

ต่อไปจะแสดงให้เห็นว่าอัลกอริทึม ข้างต้นนี้ กระทำการ ค่าของ a, b และ c อย่างไร การจำลองแบบ เช่นนี้ เรียกว่า **การตามรอย** (trace) อันดับแรก สมมติว่า

a = 1 , b = 5 , c = 3

บรรทัดที่ 1	ให้ x มีค่าเท่ากับ a	[x = 1]
บรรทัดที่ 2	b > x	[5 > 1] เป็นจริง
	ดังนั้น ให้ x มีค่าเท่ากับ b	[x = 5]
บรรทัดที่ 3	c > x	[3 > 5] เป็นเท็จ
	ไม่ต้องทำอะไร ดังนั้น x เท่ากับ 5	[x = 5]
	ซึ่งเป็นเลขที่มีค่ามากที่สุด ของ a, b และ c	

ต่อไป ลองสมมติว่า a = 6, b = 1 และ c = 9 แล้ว trace อัลกอริทึม

โปรดสังเกตว่า อัลกอริทึม ตัวอย่างนี้ มีคุณสมบัติครบทั้งหกข้อที่กล่าวไว้ข้างต้น

ขั้นตอนต่างๆ ของอัลกอริทึม ต้องกำหนดไว้ถูกต้อง ขั้นตอนต่างๆ ของตัวอย่างนี้ กล่าวไว้ ถูกต้องอย่างเพียงพอเพื่อให้อัลกอริทึม สามารถเขียนด้วย ภาษาโปรแกรม และกระทำการได้ ด้วยเครื่องคอมพิวเตอร์

กำหนดค่าต่างๆ ของอินพุท แต่ละขั้นตอน ของอัลกอริทึม ให้ผลลัพธ์เป็นหนึ่งอย่าง ตัวอย่างเช่น กำหนดค่า

a = 1 , b = 5 , c = 3

บรรทัดที่ 2 ของอัลกอริทึมตัวอย่าง x จะมีค่าเท่ากับ 5 ไม่ว่า จะให้คน หรือ เครื่องคอมพิวเตอร์ กระทำการ อัลกอริทึม ก็ตาม

อัลกอริทึม จบลง หลังจากขั้นตอน จำนวนมากอย่างจำกัด ให้คำตอบกับคำถาม ตัวอย่างเช่น อัลกอริทึมนี้ จบ หลังจาก ทำสามขั้นตอน และ ให้ ค่ามากที่สุด ของ เลขสามจำนวน ที่กำหนดให้

อัลกอริทึม รับ อินพุท และ ให้ เอ้าพุท อัลกอริทึมตัวอย่างนี้ รับ อินพุท เป็นค่า ของ a, b และ c และ ให้ เอ้าพุท เป็น ค่าของ x

อัลกอริทึม ต้องใช้ได้ทั่วไป อัลกอริทึม ตัวอย่าง สามารถหาค่ามากที่สุด ของ เลขสามจำนวนใดๆ (any three numbers)

แบบฝึกหัด

1. จงเขียน อัลกอริทึม หา สมาชิกตัวที่มีค่าน้อยที่สุด ระหว่าง a, b, และ c

   (Write an algorithm that finds the smallest element among a, b, and c.)

2. จงเขียน อัลกอริทึม หา สมาชิกตัวที่มีค่าน้อยที่สุด เป็น อันดับสอง จาก a, b และ c โดยสมมติว่า ค่าของ a, b และ c ไม่ซ้ำกัน

(Write an algorithm that finds the second smallest element among a, b, and c. Assume that the values of a, b, and c are distinct.)

3.2 สัญกรณ์ สำหรับ อัลกอริทึม (Notation for Algorithms)

ถึงแม้ว่า บางครั้ง ภาษาธรรมชาติ (ordinary language) จะพอเพียง สำหรับ เขียนอัลกอริทึม แต่ นักคณิตศาสตร์ และนักคอมพิวเตอร์ จำนวนมาก ชอบ **รหัสเทียม (pseudocode)** เนื่องจาก ความถูกต้อง, โครงสร้าง และความเป็นสากลของมัน (because of its precision, structure, and universality) รหัสเทียม มีลักษณะเหมือนกับชื่อของมัน คือ มันคล้ายกับ รหัสจริง (โปรแกรม) ของภาษา เช่น Pascal และ C รหัสเทียม มี หลาย เวอร์ชัน (versions)

ตัวอย่าง

Algorithm 3.2.1 Finding the Maximum of three numbers.

อัลกอริทึมนี้ ค้นหาเลขที่มีค่ามากที่สุด ของเลข a, b, และ c (This algorithm finds the largest of the numbers a, b, and c.)

Input : Three numbers a, b, and c

Output : x, the largest of a, b, and c

```
1. procedure max (a, b, c)
2. x := a
3. if b > x then // if b is larger than x, update x
4.      x := b
5. if c > x then // if c is larger than x, update x
6.      x := c
7. return (x)
8. end max
```

อัลกอริทึมข้างต้นนี้ มี ชื่อเรื่อง (title) มีคำอธิบายอัลกอริทึมอย่างย่อ อินพุทและเอ้าพุทของอัลกอริทึม และโปรซีเดอร์ ซึ่งประกอบด้วย คำสั่งต่างๆ (instructions) ของอัลกอริทึม

อัลกอริทึม 3.2.1 ประกอบด้วย หนึ่งโปรซีเดอร์ เพื่อให้ง่ายในการอ้างถึงแต่ละบรรทัด ภายในโปรซีเดอร์ บางครั้ง เราจะใส่เลขที่บรรทัด (line number) ไว้

โดยทั่วไป โครงสร้าง if-then มีรูปแบบ ดังนี้

```
if p then
    action
```

ถ้า เงื่อนไข p เป็นจริง, action จะถูกกระทำการ และการควบคุม จะส่งไปยัง ข้อความสั่ง ตามหลัง action

ถ้า เงื่อนไข p เป็นเท็จ การควบคุม จะส่งไปยัง ข้อความสั่ง ตามหลัง action ทันที

อีกทางเลือกหนึ่ง คือ โครงสร้าง **if-then-else** มีรูปแบบดังนี้

```
if p then
    action 1
else
    action 2
```

ถ้า เงื่อนไข p เป็นจริง, action 1 (ไม่ใช่ action 2) จะถูกกระทำการ และการควบคุม ส่งไปยัง ข้อความสั่ง ตามหลัง action 2

ถ้า เงื่อนไข p เป็นเท็จ, action 2 (ไม่ใช่ action 1) จะถูกกระทำการ และการควบคุม ส่งไปยัง ข้อความสั่ง ตามหลัง action 2

จะเห็นว่า เราใช้ ย่อหน้า (identation) เพื่อแสดงถึง ข้อความสั่งต่างๆ ซึ่งประกอบกันขึ้นเป็น action นอกจากนี้แล้ว ถ้า action ประกอบด้วย ข้อความสั่ง หลาย บรรทัด (multiple statements) เราคั่น (delimit) ข้อความสั่งเหล่านั้น ด้วยคำว่า **begin** และ **end**

ตัวอย่าง multiple statement action ในข้อความสั่ง **if**

```
if x ≥ 0 then
    begin
    x := x - 1
    a := b + c
    end
```

เครื่องหมาย slash สองตัว (//) หมายถึง การเริ่มต้น ของ **คอมเมนต์** (comment) ซึ่งจะขยาย ไปจน จบบรรทัด จากในตัวอย่าง อัลกอริทึม 3.2.1 ได้แก่

// if b is larger than x, update x

คอมเมนต์ จะช่วย ให้ผู้อ่าน เข้าใจ อัลกอริทึม ได้ง่ายขึ้น แต่ ข้อความ ส่วนนี้ จะไม่ได้รับการกระทำการ

ข้อความสั่ง **return** (x) ซึ่ง จบ (terminate) โปรซีเดอร์ และส่งกลับ ค่าของ x ไปยัง **ผู้เรียก** (invoker) ของโปรซีเดอร์ ถ้าไม่มีค่าส่งกลับ ข้อความสั่ง **return** (ไม่มี (x)) แสดงการจบ โปรซีเดอร์ เท่านั้น และจะอยู่ก่อน บรรทัด **end**

โปรซีเดอร์ ซึ่งมี ข้อความสั่ง **return** (x) หมายถึง **ฟังก์ชัน** (function) ซึ่ง โดเมน (domain) ประกอบด้วย ค่า ถูกต้องทั้งหมด สำหรับพารามิเตอร์ (parameters) และ พิสัย (range) ของฟังก์ชัน คือ เซตของค่าทั้งหมด ซึ่ง จะถูกส่งกลับ โดย โปรซีเดอร์

เมื่อใช้รหัสเทียม เราจะใช้ตัวดำเนินการคำนวณ (arithmetic operators) ดังนี้ +, -, * (คูณ) และ / (หาร) เช่นเดียวกับ ตัวดำเนินการสัมพันธ์ (relational operators) =, $\neq$, <, >, $\leq$ และ $\geq$

ตัวดำเนินการตรรกะ (logical operators) ได้แก่ **and, or** และ **not** เราจะใช้ เครื่องหมาย = หมายถึง ตัวดำเนินการเท่ากัน (equality operator) และใช้ เครื่องหมาย := หมายถึง ตัวกำหนดค่า (assignment operator) บางครั้ง เราจะใช้ ข้อความสั่ง เป็นทางการ น้อยกว่านี้ (ตัวอย่างเช่น เลือก สมาชิก x ใน S) เมื่อต้องการทำสิ่งอื่นๆ ซึ่งไม่สนใจ ความหมาย โดยทั่วไป ผลเฉลยของ แบบฝึกหัดนั้น ต้องการ อัลกอริทึม ที่ควรจะเขียน ในรูปแบบซึ่งแสดงให้เห็น เช่น ในอัลกอริทึม 3.2.1

บรรทัดต่างๆ ของ โปรซีเดอร์ ซึ่ง ถูกกระทำการตามลำดับ ปกติได้แก่ ข้อความสั่งกำหนดค่า (assignment statements), ข้อความสั่งมีเงื่อนไข (conditional or if statements), ลูป (loops), ข้อความสั่ง ส่งคืน (return statements) และการรวมกัน ของ ข้อความสั่งเหล่านี้

โครงสร้าง ลูป ซึ่งมีประโยชน์ อีก หนึ่งชนิด ได้แก่ **while loop** มีรูปแบบ ดังนี้

```
while p do
   action
```

ซึ่ง action จะถูกกระทำการซ้ำๆ กัน ตราบใดที่ p เป็นจริง เราเรียก action ว่า **body** ของ ลูป

เช่นเดียวกับ ใน ข้อความสั่ง if กล่าวคือ ถ้า action ประกอบด้วย ข้อความสั่งมากกว่า หนึ่งอย่าง (multiple statements) เรา จำกัด ข้อความสั่งเหล่านี้ ด้วยคำว่า **begin** และ **end** ซึ่งจะแสดงให้เห็น ในอัลกอริทึมตัวอย่าง ข้างล่างนี้

Algorithm 3.2.2 Finding the Largest Element in a Finite Sequence

อัลกอริทึมนี้ ค้นหาเลขที่มีค่ามากที่สุด ในลำดับ $s_1, s_2, \ldots, s_n$ เวอร์ชันนี้ ใช้ while ลูป

```
Input  : The sequence s1, s2, . . . , sn and the length n of the sequence
output : large, the largest element in this sequence
procedure find-large (s, n)
    large := s1
    i := 2
    while i ≤ n do
        begin
        if si > large then // a larger value was found
            large := si
        i := i + 1
        end
    return (large)
end find-large
```

จง trace (ตามรอย) อัลกอริทึม 3.2.2 เมื่อ n = 4 และ S คือ ลำดับ

$s_1 = -2$, $s_2 = 6$, $s_3 = 5$, $s_4 = 6$

ในอัลกอริทึม 3.2.2 เราทำขั้นตอน ตามลำดับ โดยใช้ ตัวแปร i ซึ่งเป็น ค่าจำนวนเต็ม 1 ถึง n, มีลูปชนิดพิเศษ *เรียกว่า for loop ใช้กันมาก และบ่อยครั้ง* ใช้ แทน while loop มีรูปแบบดังนี้

```
for var := init to limit do
    action
```

เหมือนกับ ข้อความสั่ง if และ while ลูป ที่กล่าวมาแล้ว ถ้า action ประกอบด้วย multiple statements เราจำกัดข้อความสั่งต่างๆ ด้วยคำว่า **begin** และ **end** เมื่อ for ลูป ถูกกระทำการ, action จะถูกกระทำการ สำหรับค่าต่างๆ ของ var จาก init ถึง limit พูดชัดเจนคือ init และ limit เป็น นิพจน์ ซึ่งมีค่าเป็น จำนวนเต็ม

ครั้งแรก ตัวแปร var กำหนด ให้เป็นค่า init, ถ้า var $\leq$ limit กระทำการ action จากนั้น บวก 1 ให้กับ var, กระบวนการนี ทำซ้ำๆ กัน, การทำซ้ำ ต่อเนื่องจนกระทั่ง var > limit โปรดสังเกตว่า ถ้า init > limit, action จะไม่ได้รับการกระทำการ แต่อย่างใด

Algorithm 3.2.3 Finding the largest element in a finite sequence

Input : The sequence $S_1, S_2, \ldots, S_n$ and the length n of the sequence

Output : large, the largest element in this sequence

```
1. procedure find-large (s, n)
2.     large := S1
3.     for i := 2 to n do
4.         if Si > large then // a larger value was found
5.             large := Si
6.     return (large)
7. end find-large
```

ในการพัฒนาอัลกอริทึมนั้น บ่อยครั้ง ความคิดที่ดี คือ แบ่งปัญหาดั้งเดิม ออกเป็น ปัญหาย่อยๆ ตั้งแต่ สองชุดขึ้นไป โปรซีเดอร์ หนึ่งชุด ถูกพัฒนาขึ้นมา ให้กับ การแก้ปัญหา แต่ละงานย่อย หลังจากนั้น โปรซีเดอร์เหล่านี้ รวมเข้าด้วยกัน เพื่อเป็นผลเฉลยให้กับ ปัญหาดั้งเดิม

สมมติว่า ต้องการอัลกอริทึม หา จำนวนเฉพาะ ที่มีค่าน้อยที่สุด ซึ่ง มากกว่า จำนวนเต็มบวกซึ่งกำหนดให้ (to find the least prime number that exceeds a given positive integer.)

ปัญหา กำหนด จำนวนเต็มบวก n ให้ ต้องการหา จำนวนเฉพาะ p ซึ่งมีค่าน้อยที่สุด และ p > n

เราแบ่งปัญหานี้ออกเป็น สองปัญหาย่อย, ขั้นแรกพัฒนาอัลกอริทึม เพื่อหาว่า จำนวนเต็มบวก หนึ่งตัว เป็นจำนวนเฉพาะหรือไม่ จากนั้น ใช้ อัลกอริทึมนี้ เพื่อหาจำนวนเฉพาะค่าน้อยที่สุด ซึ่ง มากกว่า จำนวนเต็มบวก ซึ่งกำหนดให้

Algorithm 3.2.4 Testing Whether a Positive Integer is Prime

อัลกอริทึมนี้ ทดสอบว่า จำนวนเต็มบวก m เป็นจำนวนเฉพาะหรือไม่ , เอ้าพุท เป็น **true** ถ้า m คือ จำนวนเฉพาะ และเอ้าพุท เป็น **false** ถ้า m ไม่ใช่ จำนวนเฉพาะ

Input : m, a positive integer

Output : **true,** if m is prime

false, if m is not prime

```
procedure is-prime (m)
    for i := 2 to m-l do
        if m mod i = 0 then // i divides m
            return (false)
    return (true)
end is-prime
```

อัลกอริทึม 3.2.5 หา จำนวนเฉพาะค่าน้อยที่สุด ซึ่งมากกว่า จำนวนเต็มบวก n ที่กำหนดให้ การเรียก โปรซีเดอร์ ซึ่งส่งคืน ค่า เช่น ในอัลกอริทึม 3.2.4 ต้องบอกชื่อ โปรซีเดอร์ การเรียก โปรซีเดอร์ ชื่อ proc ซึ่งไม่มีการส่งคืน ค่าใดๆ เขียนดังนี้

call proc $(p_1, p_2, \ldots, p_k)$

เมื่อ $p_1, p_2, \ldots, p_k$ หมายถึง อาร์กิวเมนต์ ส่งไปยัง proc

Algorithm 3.2.5 Finding a Prime Lager Than a Given Integer

อัลกอริทึมนี้ หา จำนวนเฉพาะ เล็กที่สุด ซึ่ง มากกว่า จำนวนเต็มบวก n

Input : n, a positive integer

Output : m, the smallest prime greater than n

```
procedure large-prime (n)
    m := n + 1
    while not is-prime (m) do
        m := m + 1
    return (m)
end large-prime
```

แบบฝึกหัด 3.2

1. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ค้นหาสมาชิกตัวที่มีค่าน้อยที่สุด ในลำดับ ของจำนวน

   S(1), . . . , S(N)

2. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ค้นหาดรรชนี J ของ การเกิดครั้งแรก ของ สมาชิกตัวที่มีค่ามากที่สุด ในลำดับของจำนวน

   S(1), . . . , S(N)

   (ตัวอย่างเช่น ถ้าลำดับคือ

   6.2, 8.9, 4.2, 8.9

   อัลกอริทึม จะส่งคืน ค่า 2)

3. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ซึ่งส่งคืนดรรชนีของการเกิดครั้งแรก ของ ค่า KEY ในลำดับของสายอักขระ

   S(1), . . . , S(N)

   ถ้า KEY ไม่ได้อยู่ในลำดับ อัลกอริทึม จะส่งคืน ค่า 0

   (ตัวอย่างเช่น ลำดับ

   'MARY' 'JOE' 'MARK' 'RUDY'

   และ KEY คือ 'MARK' อัลกอริทึม จะส่งคืน ค่า 3)

4. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ค้นหาดรรชนีของสายอักขระชุดแรก ซึ่งไม่ได้เรียงลำดับตามตัวอักษร ในลำดับของสายอักขระ

   S(1), . . . , S(N)

   ถ้าสายอักขระทั้งหมด เรียงลำดับตามตัวอักษร อัลกอริทึม ส่งคืน ค่า 0

   (ตัวอย่างเช่น ลำดับคือ

   'AMY' 'BRUNO' 'ELIE' 'DAN' 'XEKE'

   อัลกอริทึม ส่งคืน ค่า 4)

5. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ย้อนกลับ (reverses) ลำดับ

   S(1), . . . , S(N)

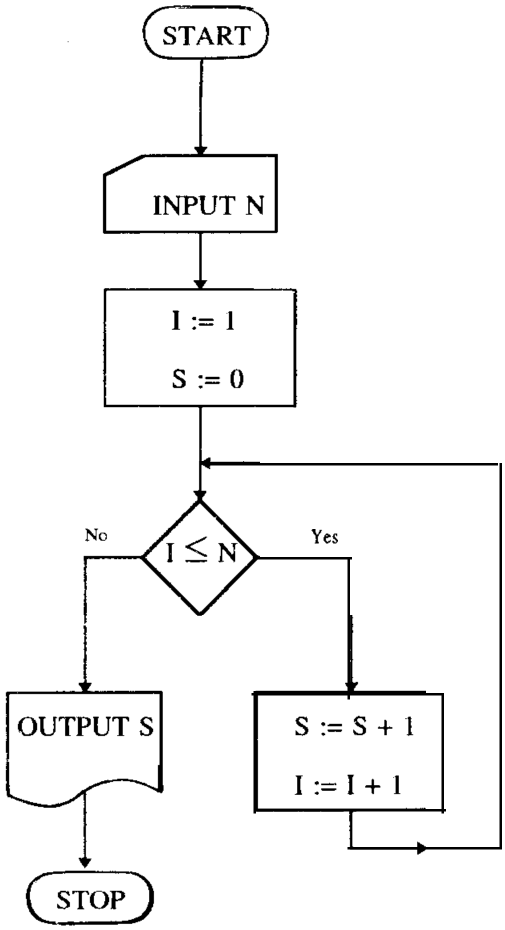
   (ตัวอย่างเช่น ถ้าลำดับคือ

   'AMY' 'BRUNO' 'ELIE'

อัลกอริทึม จะส่งคืน ลำดับ

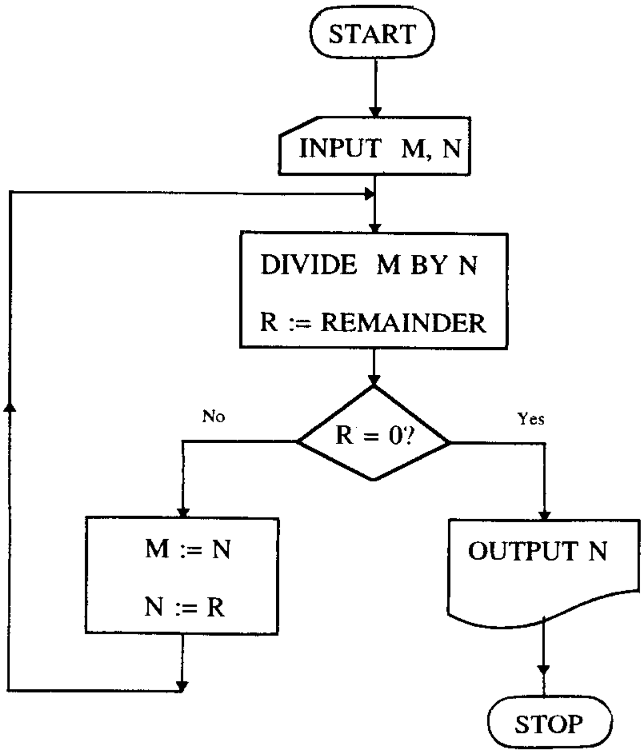
'ELIE' 'BRUNO' 'AMY')

6. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ทดสอบว่า จำนวนเต็มบวก $N > 1$ เป็นจำนวนเฉพาะ (prime) หรือไม่?
7. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 ค้นหาจำนวนเฉพาะค่าน้อยที่สุด ซึ่งมากกว่าจำนวนเต็มบวก N
8. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 จากผังงานข้างล่างนี้



9. จงแสดงให้เห็นว่า อัลกอริทึม ของแบบฝึกหัดข้อ 8 ปฏิบัติการอย่างไร ถ้า $N = 3$
10. อัลกอริทึม ของแบบฝึกหัดข้อ 8 คำนวณอะไร?

11. จงเขียนอัลกอริทึม ในรูปแบบตัวอย่าง 1 (ค่าอินพุต M และ N เป็นจำนวนเต็มบวก)



12. จงแสดงให้เห็นว่า อัลกอริทึมของแบบฝึกหัดข้อ 11 คำนวณอย่างไร ถ้า M = 6 และ N = 10

13. อัลกอริทึม ของแบบฝึกหัดข้อ 11 คำนวณอะไร?

14. จงเขียนอัลกอริทึม รับ อินพุท ซึ่งเป็นเมทริกซ์ ของความสัมพันธ์ R และทดสอบว่า R เป็นการสะท้อนหรือไม่? (Write an algorithm that receives as input the matrix of a relation R and tests whether R is reflexive.)

15. จงเขียนอัลกอริทึม รับ อินพุท ซึ่งเป็น เมทริกซ์ของ ความสัมพันธ์ R และทดสอบว่า R เป็นปฏิสมมาตรหรือไม่?

16. จงเขียนอัลกอริทึม รับ อินพุท ซึ่งเป็น เมทริกซ์ของ ความสัมพันธ์ R และทดสอบว่า R เป็นฟังก์ชันหรือไม่?

17. จงเขียนอัลกอริทึม รับ อินพุท ซึ่งเป็น เมทริกซ์ของ ความสัมพันธ์ R และให้เอ้าพุทเป็นเมทริกซ์ ของ ความสัมพันธ์ผกผัน R^{-1}

3.3 อัลกอริทึมแบบยุคลิด (The Euclidean Algorithm)

อัลกอริทึม ซึ่งเก่าแก่และมีชื่อเสียง ได้แก่ อัลกอริทึมของ ยุคลิด สำหรับ หา ตัวหาร ร่วมมาก ของ จำนวนเต็มสองตัว

ตัวหารร่วมมาก ของ จำนวนเต็ม สองตัว m และ n (ซึ่ง ไม่เท่ากับศูนย์ ทั้งคู่) หมายถึง จำนวนเต็มบวก ใหญ่ที่สุด ซึ่งหาร m และ n ลงตัวทั้งคู่

(The greatest common divisor of two integers m and n (not both zero) is the largest positive integer that divides both m and n.)

ตัวอย่างเช่น ตัวหารร่วมมาก ของ 4 และ 6 คือ 2 ตัวหารร่วมมาก ของ 3 และ 8 คือ 1 เป็นต้น

ถ้า a, b และ q เป็นจำนวนเต็ม, $b \neq 0$ **โดยที่** $a = bq$ เราพูดว่า b หาร a ลงตัว (we say that b **divides** a) และเขียนดังนี้ $b \mid a$

ในกรณีนี้ q เป็น **ผลหาร** (quotient) และเรียก b ว่า **ตัวหาร** (divisor) ของ a

ถ้า b หาร a ไม่ลงตัว เขียนดังนี้ $b \nmid a$

ตัวอย่าง 3.3.1

เนื่องจาก $21 = 3 \cdot 7$

ดังนั้น 3 หาร 21 ลงตัว เขียนดังนี้ $3 \mid 21$

ผลหาร คือ 7

บทนิยาม

ให้ m และ n เป็น จำนวนเต็ม ไม่เท่ากับ ศูนย์ ทั้งคู่ ตัวหารร่วมของ m และ n หมายถึง จำนวนเต็ม ซึ่ง หาร m และ n ลงตัวทั้งคู่ ตัวหารร่วมมาก หมายถึง ตัวหารร่วมที่มีค่าใหญ่ที่สุด ของ m และ n เขียนดังนี้

gcd (m, n)

ตัวอย่าง 3.3.2 จงหา gcd (30, 105)

ตัวหารร่วมของ 30 ได้แก่

1, 2, 3, 5, 6, 10, 15, 30

และตัวหารร่วมของ 105 ได้แก่

1, 3, 5, 7, 15, 21, 35, 105

ดังนั้น ตัวหารร่วม ของ 30 และ 105 ได้แก่

1, 3, 5, 15

ในที่นี้ ตัวที่มีค่า มากที่สุด คือ 15 เพราะฉะนั้น

$$\gcd(30, 105) = 15$$

ทฤษฎีบท 3.3.3

> ถ้า a เป็นจำนวนเต็มบวก ไม่ใช่ค่าลบ, b เป็น จำนวนเต็มบวก
>
> และ $a = bq + r \ , 0 \leq r < b$
>
> จะได้
>
> $$\gcd(a, b) = \gcd(b, r)$$

ตัวอย่าง 3.3.4

ถ้าเราหาร 105 ด้วย 30 , จะได้

$$105 = 30 \cdot 3 + 15$$

เศษ คือ 15 จากทฤษฎีบทข้างต้น

$$\gcd(105, 30) = \gcd(30, 15)$$

ถ้าเราหาร 30 ด้วย 15 จะได้

$$30 = 15 \cdot 2 + 0$$

เศษ คือ 0 จากทฤษฎีบท

$$\gcd(30, 15) = \gcd(15, 0)$$

เพราะว่า $\gcd(15, 0) = 15$

เพราะฉะนั้น $\gcd(105, 30) = \gcd(30, 15) = \gcd(15, 0) = 15$

Algorithm 3.3.5 Euclidean Algorithm

อัลกอริทึมนี้ หา ตัวหารร่วมมาก ของ จำนวนเต็มบวก ไม่ใช่ค่าลบ a และ b เมื่อ a และ b ไม่ใช่ศูนย์ ทั้งคู่

(This algorithm finds the greatest common divisor of the nonnegative integers a and b, where not both a and b are zero.)

Input : a and b (nonnegative integers, not both zero)

Output : Greatest common divisor of a and b

```
1.  procedure gcd (a, b)
2.     // make a largest
3.     if a < b then
4.         swap (a, b)
           // that is, execute
           // temp := a
           // a := b
           // b := temp
5.     while b ≠ 0 do
6.        begin
7.        divide a by b to obtain a = bq + r, 0 ≤ r < b
8.        a := b
9.        b := r
10.       end
11.    return (a)
12. end gcd
```

ตัวอย่าง จงหา gcd (504, 396) โดยตามรอย อัลกอริทึม 3.3.5

ให้ $a = 504$ และ $b = 396$ เพราะว่า $a > b$ เราย้ายไป บรรทัดที่ 5 เพราะว่า $b \neq 0$ เราประมวลผล บรรทัดที่ 7 เมื่อ หาร a (504) ด้วย b (396) จะได้

$$504 = 396 \cdot 1 + 108$$

จากนั้น ย้ายไป บรรทัดที่ 8 เราให้ $a = 396$ และ $b = 105$ ส่งกลับไปบรรทัดที่ 5

เพราะว่า $b \neq 0$, ประมวลผล บรรทัดที่ 7 เมื่อหาร a (396) ด้วย b (108) จะได้

$$396 = 108 \cdot 3 + 72$$

ต่อไป

$$108 = 72 \cdot 1 + 36$$

$$72 = 36 \cdot 2 + 0$$

จากนั้น มา บรรทัดที่ 8 เราให้ $a = 36$ ละ $b = 0$ ส่งกลับ ไป บรรทัดที่ 5

ขณะนี้ b = 0 เราข้ามไปบรรทัดที่ 11 เมื่อส่งกลับ a (36) เป็น ตัวหารร่วมมาก ของ 396 และ 504

แบบฝึกหัด 3.3

ข้อ 1-6 จงหา จำนวนเต็ม q และ r เพื่อให้ $a = bq + r, 0 \leq r < b$

1. $a = 45, b = 6$
2. $a = 106, b = 12$
3. $a = 66, b = 11$
4. $a = 221, b = 17$
5. $a = 0, b = 31$
6. $a = 0, b = 47$

ข้อ 7-16 จงใช้อัลกอริทึม ของ ยุคลิด คำนวณหา ตัวหารร่วมมาก ของ จำนวนเต็มแต่ละคู่

7. 60, 90
8. 110, 273
9. 220, 1400
10. 315, 825
11. 20, 40
12. 331, 993
13. 2,091; 4,807
14. 2,475; 32,670
15. 67,942; 4,209
16. 490,256; 337

3.4 อัลกอริทึมเรียกซ้ำ (Recursive Algorithms)

โปรซีเดอร์เรียกซ้ำ หมายถึง โปรซีเดอร์ ซึ่งเรียก ตัวมันเอง

(A **recursive procedure** is a procedure that invokes itself.)

อัลกอริทึมเรียกซ้ำ หมายถึง อัลกอริทึม ซึ่ง ประกอบด้วย โปรซีเดอร์เรียกซ้ำ

(A **recursive algorithm** is an algorithm that contains a recursive procedure.)

การเรียกซ้ำ เป็น วิธีแบบธรรมชาติ ใช้ได้ดี สวยงาม เพื่อแก้ปัญหาต่างๆ ขนาดใหญ่ ปัญหาชนิดนี้ แก้ไขได้ โดยใช้ เทคนิคการแบ่งแยกและเอาชนะ (using a divide-and-conquer technique) ซึ่งปัญหา จะถูกแบ่งออกเป็น ปัญหาเล็กๆ ซึ่งเป็นชนิดเดียวกับปัญหาเดิม (original problem) แต่ละปัญหาย่อย ถูกแบ่งย่อยต่อไป จนกระทั่งการประมวลผล ให้ผลลัพธ์เป็น ปัญหาย่อย ซึ่ง สามารถแก้ไขได้ ในวิธีตรงไปตรงมา สุดท้าย ผลเฉลย ให้กับ ปัญหาย่อย นำมารวมกัน จะได้ ผลเฉลยให้กับ ปัญหาเดิม

ตัวอย่าง 1

แฟกทอเรียล n (n factorial) นิยามดังนี้

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)(n-2)\ldots 2\cdot 1 & \text{if } n \geq 1 \end{cases}$$

นั่นคือ ถ้า $n \geq 1$, $n!$ เท่ากับ ผลคูณ ของ จำนวนเต็มทั้งหมด ระหว่าง 1 ถึง n ส่วน $0!$ นิยามให้เป็น 1

ตัวอย่างเช่น

$$3! = 3\cdot 2\cdot 1 = 6$$

$$6! = 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1 = 720$$

โปรดสังเกตว่า แฟกทอเรียล n สามารถเขียน ในเทอมของตัวมันเอง (in terms of itself) นั่นคือ

$$\begin{aligned} n! &= n(n-1)(n-2)\ldots 2\cdot 1 \\ &= n\cdot(n-1)! \end{aligned}$$

ตัวอย่างเช่น

$$\begin{aligned} 5! &= 5\cdot 4\cdot 3\cdot 2\cdot 1 \\ &= 5\cdot 4! \end{aligned}$$

ตาราง 3.4.1 สรุปกระบวนการแก้ปัญหา การคำนวณ 5!

Problem	Simplified problem
5!	5·4!
4!	4·3!
3!	3·2!
2!	2·1!
1!	1·0!
0!	None

ตาราง 3.4.2

Problem	Solution
0!	1
1!	1·0! = 1
2!	2·1! = 2
3!	3·2! = 3·2 = 6
4!	4·3! = 4·6 = 24
5!	5·4! = 5·24 = 120

ต่อไป เราเขียนอัลกอริทึมเรียกซ้ำ คำนวณหา ค่า แฟกทอเรียล อัลกอริทึมนี้ แปลจากสมการข้างล่างนี้ โดยตรง

(Next, we write a recursive algorithm that computes factorials. The algorithm is a direct translation of the equation)

$$n! = n \cdot (n - 1)!$$

อัลกอริทึม 3.4.3 การคำนวณหาค่า factorial n

อัลกอริทึมเรียกซ้ำนี้ คำนวณหา n!

```
Input  : n, an integer greater than or equal to 0
Output : n!
1. procedure factorial (n)
2. if n = 0 then
3.     return (1)
4. return (n * factorial (n - 1))
5. end factorial
```

อัลกริทึม 3.4.4 การคำนวณหาตัวหารร่วมมากแบบเรียกซ้ำ

อัลกอริทึมเรียกซ้ำนี้ หา ตัวหารร่วมมาก ของ จำนวนเต็ม ไม่ใช่ ค่าลบ a และ b เมื่อ a และ b ไม่ใช่ค่า ศูนย์ทั้งคู่ (where not both a and b are zero)

```
Input  : a and b (nonnegative integers, not both zero)
Output : greatest common divisor of a and b

   procedure gcd_recurs (a, b)
      // make a largest
1. if a < b then
2.     swap (a, b)
3. if b = 0 then
4.     return (a)
5. divide a by b to obtain a = bq + r ,  0 ≤ r < b
6. return (gcd_recurs (b, r))
   end gcd_recurs
```

แบบฝึกหัด 3.4

1. จง trace อัลกอริทึม 3.4.3 เมื่อ $n = 4$
2. จง trace อัลกอริทึม 3.4.4 เมื่อ $a = 5$ และ $b = 0$
3. จง trace อัลกอริทึม 3.4.4 เมื่อ $a = 55$ และ $b = 20$
4. จงใช้สูตร

$$S_1 = 1,$$

$$S_n = S_{n-1} + n \ , \ n \geq 2$$

เขียนอัลกอริทึมเรียกซ้ำ คำนวณหา

$$S_n = 1 + 2 + 3 + \ldots + n$$

5. จงใช้สูตร

$$S_1 = 2,$$

$$S_n = S_{n-1} + 2n \ , \ n \geq 2$$

เขียนอัลกอริทึม คำนวณหา

$$S_n = 2 + 4 + 6 + \ldots + 2n$$

6. จงเขียนอัลกอริทึม ไม่ใช่การเรียกซ้ำ เพื่อคำนวณหา n!

(Write a nonrecursive algorithm to compute n!)

3.5 ความซับซ้อนของอัลกอริทึม (Complexity of Algorithms)

โปรแกรมคอมพิวเตอร์ ได้มาจาก อัลกอริทึมที่ถูกต้อง แต่มันจะไม่มีประโยชน์ใดๆ สำหรับ อินพุท บางชนิด ถ้าว่า เวลาที่จำเป็น เพื่อวิ่ง (run) โปรแกรม หรือ หน่วยเก็บที่จำเป็น เพื่อเก็บข้อมูล ตัวแปรโปรแกรม และอื่นๆ มีจำนวนมากเกินไป

สมมติว่า กำหนดให้ เซต X มีสมาชิก n ตัว สมาชิกบางตัวมีสีแดง บางตัวมีสีดำ ต้องการหาจำนวนเซตย่อยของ X ซึ่งมีสมาชิกสีแดงอย่างน้อยที่สุดหนึ่งตัว สมมติว่าเราเขียนอัลกอริทึมให้คำนวณเซตย่อย ทั้งหมดของ X แล้วนับ เซตย่อย ซึ่งมีสมาชิกสีแดงอย่างน้อย 1 ตัว จากนั้น implement อัลกอริทึมชุดนี้ ให้เป็นโปรแกรมคอมพิวเตอร์ เนื่องจากเซตที่มีสมาชิก n ตัว จะมีเซตย่อยทั้งหมด 2^n ชุด ดังนั้น โปรแกรม จึงต้องการเวลาอย่างน้อยที่สุด 2^n หน่วย ในการกระทำ (execute) หน่วยของเวลา 2^n จะเพิ่มขึ้นอย่างรวดเร็ว เมื่อ n มีค่ามากขึ้น ยกเว้น เมื่อ n มีค่าน้อย ดังนั้นจึงเป็นไปไม่ได้ที่จะวิ่งโปรแกรม

การคำนวณหา พารามิเตอร์ความสามารถ ของ โปรแกรมคอมพิวเตอร์ เป็นงานยาก และขึ้นอยู่กับปัจจัยหลายอย่าง เช่น เครื่องคอมพิวเตอร์ที่ใช้ วิธีการแทนที่ข้อมูลและโปรแกรมถูกแปลให้เป็นคำสั่งเครื่องได้อย่างไร (Determining the performance parameters of computer program is difficult task and depends on a number of factors such as the computer that is being used, the way the data are represented, and how the program is translated into machine instructions) ถึงแม้ว่าการประมาณค่าแน่นอน ของ ประสิทธิภาพของโปรแกรมต้องนำเอาปัจจัยต่างๆ มาพิจารณาด้วย สารสนเทศที่เป็นประโยชน์ อาจได้มาจาก การวิเคราะห์ความซับซ้อนของอัลกอริทึมนั้น

เวลาเพื่อกระทำการอัลกอริทึม เป็นฟังก์ชันของข้อมูล (The time needed to execute an algorithm is a function of the input.) เป็นการยากที่จะหาสูตรชัดแจ้ง ของ ฟังก์ชัน

การวิเคราะห์อัลกอริทึม หมายถึง กระบวนการของการแปลง เพื่อประมาณค่า เวลา และ เนื้อที่ ซึ่ง จำเป็นต้องใช้เพื่อกระทำการอัลกอริทึม

(**Analysis of an algorithm** refers to the process of deriving estimates for the time and space needed to execute the algorithm.)

ความซับซ้อนของอัลกอริทึม หมายถึง ปริมาณของเวลาและเนื้อที่หน่วยความจำ ซึ่งจำเป็นต้องใช้ เพื่อ กระทำการ อัลกอริทึม

(**Complexity of an algorithm** refers to the amount of time and space required to execute the algorithm.)

เราสามารถ ถามหา เวลาน้อยที่สุด ซึ่งจำเป็นต้องใช้ เพื่อกระทำการ อัลกอริทึม ระหว่าง อินพุททั้งหมด ขนาด n เวลานี้ เรียกว่า เวลา กรณีดีที่สุด สำหรับ อินพุท ขนาด n

(We can ask for the minimum time needed to execute the algorithm among all inputs of size n. This time is called **the the base-case time** for input of size n.)

เราสามารถ ถามหา เวลามากที่สุด ซึ่งจำเป็นต้องใช้ เพื่อกระทำการ อัลกอริทึม ระหว่าง อินพุททั้งหมด ขนาด n เวลานี้ เรียกว่า เวลา กรณีแย่ที่สุด สำหรับ อินพุท ขนาด n

(We can also ask for the maximum time needed to execute the algorithm among all inputs of size n. This time is called **the worst-case time** for inputs of size n.)

กรณีที่สำคัญอีกอย่างหนึ่ง คือ เวลากรณีเฉลี่ย หมายถึง เวลาเฉลี่ยที่จำเป็นต้องใช้เพื่อ กระทำการ อัลกอริทึม ของ เซตจำกัด ของ อินพุททั้งหมดขนาด n

(Another important case is **average-case-time .** the average time needed to execute the algorithm over some finite set of inputs all of size n.)

เราสามารถ วัด เวลา ที่จำเป็นต้องใช้ ของ อัลกอริทึม โดยการนับ จำนวน ของ คำสั่ง ซึ่ง จะถูกกระทำการ อีกทางเลือกหนึ่งคือ เราอาจใช้เวลาหยาบๆ ประมาณค่า เช่น จำนวน เวลา ซึ่งแต่ละลูป ถูก กระทำการ แต่ถ้ากิจกรรมหลักของอัลกอริทึมนั้น คือ ทำการเปรียบเทียบ เช่นที่เกิดขึ้น ใน รูทีนการเรียงลำดับข้อมูล เราอาจนับ จำนวนครั้ง ของการเปรียบเทียบ ปกติ เราสนใจการประมาณค่าโดยทั่วไป เพราะว่า จากที่ได้ตั้งข้อสังเกตไว้ว่า ความสามารถจริง (actual performance) ของการ implement โปรแกรม ของ อัลกอริทึม จะขึ้นอยู่กับปัจจัยหลายอย่าง

TABLE 3.5.1 Time to Execute an Algorithm If One Step Takes 1 Microsecond to Execute

Number of Steps to Termination for Input of Size n	Time to Execute If n =								
	3	**6**	**9**	**12**	**50**	**100**	**1000**	**10^5**	**10^6**
1	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec
$\lg \lg n$	10^{-6} sec	10^{-6} sec	2×10^{-6} sec	2×10^{-6} sec	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	4×10^{-6} sec
$\lg n$	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	6×10^{-6} sec	7×10^{-6} sec	10^{-5} sec	2×10^{-5} sec	2×10^{-5} sec
n	3×10^{-6} sec	6×10^{-6} sec	9×10^{-6} sec	10^{-5} sec	5×10^{-5} sec	10^{-4} sec	10^{-3} sec	0.1 sec	1 sec
$n \lg n$	5×10^{-6} sec	2×10^{-5} sec	3×10^{-5} sec	4×10^{-5} sec	3×10^{-4} sec	7×10^{-4} sec	10^{-2} sec	2 sec	20 sec
n^2	9×10^{-6} sec	4×10^{-5} sec	8×10^{-5} sec	10^{-4} sec	3×10^{-3} sec	0.01 sec	1 sec	3 hr	12 days
n^3	3×10^{-5} sec	2×10^{-4} sec	7×10^{-4} sec	2×10^{-3} sec	0.13 sec	1 sec	16.7 min	32 yr	31,710 yr
2^n	8×10^{-6} sec	6×10^{-5} sec	5×10^{-4} sec	4×10^{-3} sec	36 yr	4×10^{16} yr	3×10^{287} yr	3×10^{30089}	3×10^{301016} yr

ตัวอย่าง 1 การค้นหาค่ามากที่สุด ใน ลำดับจำกัด, บทนิยามที่เป็นเหตุ เป็นผลของเวลากระทำการ คือ จำนวนของการทำซ้ำ (iterations) ของ while ลูป ท้ายบทนิยามนี้, worst-case, best-case และ average-case สำหรับอินพุทขนาด n คือ n-1 เพราะว่า ลูปกระทำการ n-1 ครั้งเสมอ บ่อยครั้ง เราสนใจ กรณีดีที่สุด หรือ กรณีแย่ที่สุด จริง ของเวลา ที่ใช้กระทำการ ของอัลกอริทึมน้อยกว่า เวลาดีที่สุดหรือแย่ที่สุด เป็นอย่างไร เมื่อขนาดของอินพุทเพิ่มขึ้น

สมมติว่า worst-case time ของ อัลกอริทึมชุดหนึ่ง คือ

$$t(n) = 60n^2 + 5n + 1 \qquad (1)$$

สำหรับอินพุท ขนาด n, สำหรับ n ขนาดใหญ่ เทอม $60n^2$ โดยประมาณ จะเท่ากับ t(n) (ดู ตาราง 3.5.2)

ตาราง 3.5.2

n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6,051	6,000
100	600,501	600,000
1,000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

จะเห็นว่า การเพิ่มขึ้นของ t(n) คล้ายกับ $60n^2$ จากตัวอย่างนี้ ถ้า วัด worst-case time สำหรับ อินพุท ขนาด n ด้วยหน่วย วินาที (ii seconds) จะได้ว่า

$$T(n) = n^2 + \frac{5}{60}n + \frac{1}{60}$$

คือการวัด worst-case times สำหรับอินพุท ขนาด n มีหน่วยเป็น นาที (minutes) ขณะนี้มีการเปลี่ยนแปลง หน่วย (units) ของการวัด ซึ่ง ไม่มีผลใดๆ ต่อ การเพิ่มขึ้น ของ worst-case time ดังนั้น เมื่อเราไม่สนใจ สัมประสิทธิ์ ซึ่งเป็นค่าคงที่ ภายใต้ ข้อสมมติฐานนี้ t(n) เพิ่มขึ้น เหมือน n^2 เมื่อ n เพิ่มขึ้น เราพูดว่า t(n) คือ อันดับ ของ n^2 เขียนดังนี้

$$t(n) = \Theta(n^2)$$

อ่านว่า "t(n) is theta of n^2"

ความคิดพื้นฐาน นี้คือ การแทนที่ นิพจน์ เช่น $t(n) = 60n^2 + 5n + 1$ ด้วยนิพจน์ ที่ง่ายกว่า เช่น n^2 ซึ่งเติบโต ด้วย อัตราเหมือนกันกับ t(n)

สำหรับวัตถุประสงค์ ของ การประมาณค่าเวลา ทีต้องการใช้ ของอัลกอริทึม ซึ่งมีการเปรียบเทียบ เมื่อกระทำหน้าที่ เหมือนกัน บ่อยครั้ง มันมีประโยชน์ที่จะมีการประมาณค่า ให้สูงกว่า ของ พารามิเตอร์ที่มี ความสามารถ มากกว่า ค่าจริงๆ ของ พารามิเตอร์เหล่านี้ การประมาณค่าเช่นนี้ ใช้สัญลักษณ์ โอตัวใหญ่ ("big oh" notation)

บทนิยาม 1 ให้ f และ g เป็นฟังก์ชันบน {1, 2, 3, . . . }

เราเขียน

$$f(n) = O(g(n))$$

และพูดว่า f(n) เป็น **อันดับอย่างมากที่สุด** g(n) ถ้ามีค่าคงที่บวก C_1 อยู่จริง โดยที่

$$|f(n)| \leq C_1|g(n)|$$

สำหรับทุกค่า และ n เป็นจำนวนเต็มบวกมากอย่างจำกัด

เราเขียน

$$f(n) = \Omega(g(n))$$

และพูดว่า f(n) เป็นอันดับน้อยที่สุด g(n) ถ้ามีค่าคงที่บวก C_2 อยู่จริง โดยที่

$$|f(n)| \geq C_2|g(n)|$$

สำหรับ จำนวนเต็มบวก n มากจำกัด

เราเขียน

f(n) = Ⓗ (g(n))

และพูดว่า f(n) เป็นอันดับ g(n) ถ้า f(n) = O(g(n)) และ f(n) = !&g(n))

(Let f and g be functions on {1, 2, 3, . . . }

We Write

$$f(n) = O(g(n))$$

and say that f(n) is of **order at most** g(n) if there exists a positive constant C, such that

$$|f(n)| \leq C_1|g(n)|$$

for all but finitely many positive integers n.

We write

$$f(n) = \Omega(g(n))$$

and say that f(n) is order at least g(n) if there exists a positive constant C_2 such that

$$|f(n)| \geq C_2|g(n)|$$

for all but finitely many positive integers n.

We write

$$f(n) = \Theta(g(n))$$

and say that f(n) is of order g(n) if f(n) = $O(g(n))$ and f(n) = $\Omega(g(n))$

นิพจน์ ในรูปแบบ f(n) = $O(g(n))$ เรียกว่า big oh notation ของ f

ในทำนองเดียวกัน f(n) = $\Omega(g(n))$ เรียกว่า omega notation ของ f

และ f(n) = @(g(n)) เรียกว่า theta notation ของ f

ตัวอย่าง 2 เนื่องจาก

$$n^2 + n + 3 \leq 3n^2 + 3n^2 + 3n^2 = 9n^2$$

ให้ $C_1 = 9$ จากบทนิยาม 1 จะได้ว่า

$$n^2 + n + 3 = O(n^2)$$

ตัวอย่าง 3 ถ้าเราแทน จำนวนเต็มแต่ละตัว 1, 2, ..., n ด้วย n ใน ผลบวก 1 + 2 + ... + n, ผลบวกจะไม่ลดลง และเราได้

$$1 + 2 + \ldots + n \leq n + n + n + \ldots + n = n \cdot n = n^2$$

จากบทนิยาม จะได้ว่า

$$1 + 2 + \ldots + n = O(n^2)$$

ในการหา lower bound เราเริ่มต้นกระบวนการข้างต้น และแทนจำนวนเต็มแต่ละตัว 1, 2, ..., n ด้วย 1 ในผลบวก 1 + 2 + ... +n จะได้

$$1 + 2 + \ldots + n \geq 1 + 1 + \ldots + 1 = n$$

แสดงว่า $1 + 2 + \ldots + n = \Omega(n)$

ตัวอย่าง 4 ถ้า k เป็นจำนวนเต็มบวก และแทนจำนวนเต็มแต่ละตัว 1, 2, ..., n ด้วย n

$$1^k + 2^k + \ldots + n^k \leq n^k + n^k + \ldots + n^k = n^{k+1}$$

ดังนั้น สำหรับ $n \geq 1$

$$1^k + 2^k + \ldots + n^k < O(n^{k+1})$$

ในการหา lower bound

$$\begin{aligned} 1^k + 2^k + \ldots + n^k &\geq \lceil n/2 \rceil^k + \ldots + (n-1)^k + n^k \\ &\geq \lceil n/2 \rceil^k + \ldots + \lceil n/2 \rceil^k + \lceil n/2 \rceil^k \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil^k \\ &\geq (n/2)(n/2) \\ &= n^{k+1}/2^{k+1} \end{aligned}$$

สรุปว่า

$$1^k + 2^k + \ldots + n^k = \Omega(n^{k+1})$$

และ

$$1^k + 2^k + \ldots + n^k = \Theta(n^{k+1})$$

ตัวอย่าง 5 เพราะว่า

$$\begin{aligned} |3n^3 + 6n^2 - 4n + 2| &\leq 3n^3 + 6n^2 + 4n + 2 \\ &\leq 6n^3 + 6n^3 + 6n^3 + 6n^3 \\ &\leq 24n^3 \end{aligned}$$

จะได้ว่า

$$3n^3 + 6n^2 - 4n + 2 = O(n^3)$$

จากตัวอย่างข้างต้นนี้ จะนำไปใช้ แสดงว่า พหุนาม (polynomial) ใน n ของ องศา k คือ $O(n^k)$

ทฤษฎีบท 1 ให้

$$a_n n^k + a_{n-1} n^{k-1} + \ldots + a_1 n + a_0$$

เป็น พหุนาม ใน n ของ องศา k แล้ว

$$a_n n^k + a_{n-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$$

พิสูจน์ ให้

$$C = \max \{|a_k|, |a_{k-1}|, \ldots, |a_1|, |a_0|\}$$

แล้ว

$$
\begin{aligned}
|a_n n^k + a_{n-1} n^{k-1} + \ldots + a_1 n + a_0| &\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \ldots + |a_1| n + |a_0| \\
&\leq Cn^k + Cn^{k-1} + \ldots + Cn + C \\
&\leq Cn^k + Cn^k + Cn^k + Cn^k \\
&= (k+1)Cn^k
\end{aligned}
$$

ดังนั้น

$$a_n n^k + a_{n-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$$

บทนิยาม 2 ถ้าอัลกอริทึม ต้องใช้ f(n) หน่วยของเวลา เพื่อจบ กรณีดีที่สุด สำหรับ อินพุท ขนาด n และ

$$t(n) = O(g(n))$$

เราพูดว่า เวลาดีที่สุดซึ่งต้องใช้ โดยอัลกอริทึม เป็น อันดับอย่างมากที่สุด g(n) หรือ เวลาดีที่สุดซึ่งต้องการใช้ โดย อัลกอริทึม คือ O(g(n)) ในทำนองเดียวกัน ถ้า อัลกอริทึม ต้องใช้ t(n) หน่วยของเวลา เพื่อจบ กรณีแย่ที่สุด สำหรับ อินพุท ขนาด n และ

$$t(n) = O(g(n))$$

เราพูดว่า เวลาแย่ที่สุดซึ่งใช้ โดยอัลกอริทึม เป็นอันดับ อย่างมากที่สุด g(n) หรือ เวลาแย่ที่สุดซึ่งต้องใช้ โดย อัลกอริทึม คือ O(g(n))

ถ้าอัลกอริทึม ต้องใช้ t(n) หน่วยของเวลา เพื่อจบ กรณีเฉลี่ย สำหรับ อินพุท ขนาด n และ

$$t(n) = O(g(n))$$

เราพูดว่า เวลากรณีเฉลี่ย ต้องใช้ โดย อัลกอริทึม เป็นอันดับ อย่างมากที่สุด g(n) หรือ เวลาเฉลี่ยซึ่งต้องใช้ โดย อัลกอริทึม คือ O(g(n))

(If an algorithm requires t(n) units of time to terminated in the base case for an input of size n and

$$t(n) = O(g(n)),$$

we say that **the best-case time required by the algorithm** is of order at most g(n) or that **the best-case time required by the algorithm is** $O(g(n))$. Similarly, if an algorithm requires t(n) units of time in the worst case for an input of size n and

$$t(n) = O(g(n)).$$

We say that **the worst-case required by the algorithm** is of order at most g(n) or that the **worst-case time required by the algorithm** is $O(g(n))$.

If an algorithm requires t(n) units of time to terminate in the average case for an input of size n and

$$t(n) = O(g(n)).$$

We say that **the average-case time required by the algorithm is** of order at most g(n) or that the **average-case time required by the algorithm** is $O(g(n))$.

โดยการแทนที่ 0 ด้วย Ω และ "at most)' ด้วย "at least" ใน บทนิยามข้างต้น เราจะได้ บทนิยามของ best-case worst-case หรือ average-case time ของอัลกอริทึม ซึ่งเป็น อันดับน้อยที่สุด g(n)

ตัวอย่าง 6 สมมติว่า อัลกอริทึมชุดหนึ่ง ต้องการ

$$n^2 + n + 3$$

หน่วย ของ หน่วยความจำ สำหรับอินพุท ขนาด n เราได้แสดงให้เห็นแล้วจากตัวอย่าง 1 ว่า

$$n^2 + n + 3 = O(n^2)$$

ดังนั้น อัลกอริทึม ต้องใช้เนื้อที่ เท่ากับ $O(n^2)$

ตัวอย่าง 7

$$60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2 = 66n^2 \text{ for } n \geq 1$$

เราให้ $C_1 = 66$ จากบทนิยาม จะได้ว่า

$$60n^2 + 5n + 1 = O(n^2)$$

เนื่องจาก

$$60n^2 + 5n + 1 \geq 60n^2 \quad \text{for } n \geq 1$$

เราอาจให้ $C_2 = 60$ จากบทนิยาม จะได้

$$60n^2 + 5n + 1 = \Omega(n^2)$$

เพราะว่า $60n^2 + 5n + 1 = O(n^2)$

และ $60n^2 + 5n + 1 = \Omega(n^2)$

$\therefore \quad 60n^2 + 5n + 1 = \Theta(n^2)$

ตัวอย่าง 8

ในหนังสือเล่มนี้ เราให้ lg n แทน $\log_2 n$ (ล็อกการิทึม ของ n ฐาน 2) เพราะว่า lg n < n สำหรับ $n \geq 1$

$$2n + 3 \lg n < 2n + 3n = 5n \text{ for } n \geq 1$$

ดังนั้น

$$2n + 3 \lg n = O(n)$$

และ $2n + 3 \lg n \geq 2n \quad \text{for } n \geq 1$

ดังนั้น

$$2n + 3 \lg n = \Omega(n)$$

เพราะฉะนั้น

$$2n + 3 \lg n = \Theta(n)$$

ตัวอย่าง 9

จงหา theta notation ในเทอมของ n สำหรับ จำนวน ครั้ง ของ การกระทำการ (execute) ข้อความสั่ง x := x + 1

```
1. for i := 1 to n do
2.     for j = 1 to i do
3.         x := x + 1
```

ขั้นแรก i ถูก set ให้เป็น 1, ขณะที่ j วิ่งจาก 1 ไป 1, บรรทัดที่ 3 ถูกกระทำการ 1 ครั้ง
ต่อไป i ถูก set ให้เป็น 2, ขณะที่ j วิ่งจาก 1 ไป 2, บรรทัดที่ 3 ถูกกระทำการ 2 ครั้ง

เช่นนี้เรื่อยไป ดังนั้น จำนวนครั้ง ทั้งหมด ของ บรรทัดที่ 3 จะถูก กระทำการ ดังนี้

$$1 + 2 + \ldots + n = \Theta(n^2)$$

ดังนั้น theta notation สำหรับ จำนวนครั้ง ของ ข้อความสั่ง x := x + 1 ถูกกระทำการ เท่ากับ $\Theta(n^2)$

ตัวอย่าง 10 จงคำนวณหา สัญกรณ์ "big oh" กรณีดีที่สุด กรณีแย่ที่สุด และ กรณีเฉลี่ย ของเวลาที่ต้องใช้ ในการ กระทำการ (execute) อัลกอริทึมข้างล่างนี้ สมมติว่า อินพุท ขนาด N และเวลาดำเนินงาน (run time) ของอัลกอริทึมนี้ คือ จำนวนครั้งของการเปรียบเทียบ กระทำที่ขั้นตอนที่ 3 สมมติด้วว่า ความเป็นไปได้เท่ากับ N + 1 ของ KEY จะอยู่ที่ตำแหน่งใดๆ ในลำดับ หรือ อาจจะไม่มีอยู่ ในลำดับ มีเท่าๆ กัน

Algorithm Searching an Unordered Sequence

กำหนด ลำดับ หนึ่งชุด ดังนี้

$S_1, s, \ldots, S_n, \ldots$

และค่า key, อัลกอริทึมนี้ ค้นหา ตำแหน่ง (location) ของ key ถ้า ไม่พบ key เอ้าพุท ของ อัลกอริทึม จะมีค่าเป็น 0

Input : $S_1, s, \ldots, S_n$, n, and key (the value to search for)

Output : The location of key, or if key is not found, 0

```
1. procedure linear-search (s, n, key)
2.     for i := I to n do
3.         if key = Si then
4.             return (I) // successful search
5      return (0) // unsuccessful search
6. end linear-search
```

ถ้า S_1 = key, บรรทัดที่ 3 ถูกกระทำการหนึ่งครั้ง ดังนั้น

The best-case time of อัลกอริทึม ข้างต้นนี้ คือ $\Theta(1)$

The worst-case time ถ้าไม่มี key อยู่ในลำดับคือ $\Theta(n)$

สุดท้าย พิจารณา average-case time ของอัลกอริทึม ถ้า key ถูกพบ ที่ตำแหน่งที่ i บรรทัดที่ 3 ถูกกระทำการ i ครั้ง และถ้า key ไม่มีอยู่ในลำดับ บรรทัดที่ 3 จะถูก กระทำการ n ครั้ง ดังนั้น จำนวนเวลาเฉลี่ย บรรทัดที่ 3 ซึ่งจะถูกกระทำการ คือ

$$\frac{(1 + 2 + \ . \ . \ . \ + n) + n}{n + 1}$$

ขณะนี้

$$\frac{(1 + 2 + \ . \ . \ . \ + n) + n}{n + 1} \angle \frac{n^2 + n}{n + 1}$$

$$= \frac{n(n + 1)}{n + 1} = n$$

เพราะฉะนั้น เวลาเฉลี่ย ของ อัลกอริทึม คือ

$O(n)$

สำหรับอัลกอริทึมนี้, กรณีเฉลี่ย และกรณีแย่ที่สุด ของ เวลาดำเนินงานเท่ากัน คือ $O(n)$

เมื่อใช้ สัญลักษณ์ "big oh" แสดงความสามารถ (performance) ของอัลกอริทึม สิ่งที่สำคัญต้องระลึกไว้เสมอคือ มันให้เฉพาะ การประมาณค่าที่สูงกว่าเท่านั้น (only an upper estimate) สำหรับค่าจริง ของ พารามิเตอร์ความสามารถ ตัวอย่างเช่น ถ้าเราบอกว่า กรณีเฉลี่ยเวลาดำเนินงาน ของ อัลกอริทึม A และอัลกอริทึม B คือ $O(n^2)$ และ $O(n^3)$ ตามลำดับ เรารู้สึกได้ว่า อัลกอริทึม A ดีที่สุด อย่างไรก็ตาม เพราะว่า เราให้เฉพาะการประมาณค่าสูงกว่า แต่อาจจะไม่เป็นเช่นนั้นแน่นอน แต่โดยปกติ เราจะเลือก ฟังก์ชัน g(n) ดีที่สุด เพื่ออธิบายอันดับ $O(g(n))$ ของอัลกอริทึม

สมมติว่า อัลกอริทึม A และอัลกอริทึม B ต้องการเนื้อที่หน่วยความจำ $O(n)$ และ $O(n^2)$ ตามลำดับ สำหรับอินพุทใดๆ ขนาดของค่าคงที่ อาจจะสำคัญ ตัวอย่างเช่น สมมติว่า อินพุท ขนาด n อัลกอริทึม A ต้องการ 300n หน่วยของความจำ และอัลกอริทึม B ต้องการ

$5n^2$ หน่วยของความจำ สำหรับอินพุท ขนาด ของ $n = 5$ อัลกอริทึม A ต้องการ 1,500 หน่วยของความจำ ส่วนอัลกอริทึม B ต้องการ 125 หน่วยความจำ ในกรณีนี้ อัลกอริทึม B จะมีประสิทธิภาพมากกว่า แต่สำหรับ อินพุทขนาดใหญ่พอเพียง แน่นอน อัลกอริทึม A มีประสิทธิภาพมากกว่า นอกเหนือไปจาก ข้อสังเกตข้างต้นนี้ สัญลักษณ์ "big oh" มีประโยชน์มาก

รูปแบบต่างๆ เกิดขึ้นบ่อยมาก จึงมีชื่อพิเศษกำหนดให้ ดังที่แสดง ในตาราง 3.5.2 ซึ่งมีความหมายต่างๆ ดังนี้ lg หมายถึง ล็อกการิทึม ฐานสอง, รูปแบบต่างๆ ในตาราง 3.5.2ยกเว้น $O(n^m)$ ได้จัดเรียงไว้เพื่อว่า ถ้า $O(f(n))$ อยู่ข้างบน $O(g(n))$ แล้ว $f(n) \leq g(n)$ สำหรับทุกค่าแต่ n เป็นจำนวนเต็มบวกมากอย่างจำกัด ดังนั้น ถ้าอัลกอริทึม A และอัลกอริทึม B มีเวลาดำเนินงาน เท่ากับ $O(f(n))$ และ $O(g(n))$ ตามลำดับ อัลกอริทึม A และอัลกอริทึม B ต้องการ $C_1f(n)$ และ $C_2g(n)$ หน่วยของเวลา ตามลำดับ และ $O(f(n))$ ซึ่งอยู่เหนือ $O(g(n))$ ในตาราง 3.5.2 แสดงว่า อัลกอริทึม A จะมีประสิทธิภาพ มากกว่า อัลกอริทึม B สำหรับ อินพุทขนาดใหญ่พอเพียง

ตาราง 3.5.2

"Big Oh" form	Name
$O(1)$	Constant
$O(\lg \lg n)$	Log log
$O(\lg n)$	Logarithmic
$O(n)$	Linear
$O(n \lg n)$	n log n
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^m)$	Polynomial
$O(m^n)$, $m \geq 2$	Exponential
$O(n!)$	Factorial

หมายเหตุ lg = log to the base 2

m is a fixed nonnegative integer

มันเป็นสิ่งสำคัญ ที่จะพัฒนาความรู้สึก สำหรับ ขนาดสัมพัทธ์ ของฟังก์ชัน ในตาราง 3.5.2 ในรูป 3.5.1 เรามีกราฟของฟังก์ชันเหล่านี้บางชุด อีกวิธีหนึ่งในการพัฒนาข้อดีบางอย่างสำหรับขนาดสัมพัทธ์ ของ ฟังก์ชัน f(n) ในตาราง 3.5.2 คือ คำนวณหาว่า อัลกอริทึมนั้น ใช้เวลานานเท่าไหร่ ในการจบ (terminate) ซึ่ง เวลาดำเนินงานเท่ากับ t(n) สำหรับวัตถุประสงค์นี้ สมมติว่า เครื่องคอมพิวเตอร์ คำนวณ หนึ่งขั้นตอน ใช้เวลา 1 ไมโครวินาที (10^{-6} sec) ตาราง 3.5.1 แสดงเวลากะทำการ ภายใต้ข้อสมมตินี้ สำหรับ อินพุทขนาดต่างๆ กัน โปรดสังเกตว่า มันเป็นไปได้ ที่จะทำให้เกิดผล (implement) สำหรับ n^2 หรือ n^3 ขั้นตอน แทบจะเนไปไม่ได้ แต่เป็นไปได้สำหรับอินพุทขนาดใหญ่ โปรดสังเกตด้วยว่า ผลลัพธ์จะดีขึ้น เมื่อเราย้าย จากขั้นตอน n^2 ไปยังขั้นตอน n lg n

แบบฝึกหัด 3.5

ข้อ l-32 **จงเลือก** theta notation **จาก ตาราง** 3.5.3 **สำหรับ นิพจน์ แต่ละชุด**

1. $6n + I$
2. $2n^2 + 1$
3. $6n^3 + 12n^2 + 1$
4. $3n^2 + 2n \lg n$
5. $2 \lg n + 4n + 3n \lg n$
6. $6n^6 + n + 4$
7. $2 + 4 + 6 + \ldots + 2n$
8. $(6n + 1)^2$
9. $(6n + 4)(1 + \lg n)$
10. $\frac{(n + 1)(n + 3)}{n + 2}$
11. $\frac{(n^2 + \lg n)(n + 1)}{n + n^2}$
12. $2 + 4 + X + 16 + \ldots + 2^n$

ข้อ 13-15 **จงเลือก** theta notation **สำหรับ** $f(n) + g(n)$

13. $f(n) = \Theta(1)$, $g(n) = \Theta(n^2)$
14. $f(n) = 6n^3 + 2n^2 + 4$, $g(n) = \Theta(n \lg n)$
15. $f(n) = \Theta(n^{3/2})$, $g(n) = \Theta(n^{5/2})$

ข้อ 16-26 จงเลือก theta notation **จาก** $\Theta(1)$, $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n \lg n)$, $\Theta(n^2)$, $\Theta(n^3)$, Θ(2”) **หรือ** $\Theta(n!)$ **สำหรับ จำนวนครั้งที่ ข้อความสั่ง** $x := x + 1$ **จะถูกกระทำการ**

16. for i := 1 to 2n do

    x : - x + l

17. i := 1

```
while i ≤ 2n do,
    begin
        x := x + 1
        i := i + 2
    end
```

18.

```
for i := 1 to n do
    for j := 1 to n do
        x := x + 1
```

19.

```
for i := 1 to 2n do
    for j := 1 to n do
        x := x + 1
```

20.

```
for i := 1 to n do
    for j := 1 to ⌊i/2⌋ do
        x := x + 1
```

21.

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            x:=x+ 1
```

22.

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to i do
            x := x + 1
```

23.

```
for i := 1 to n do
    for j := 1 to i do
        for k := 1 to j do
            x := x + 1
```

24. **j := n**

```
while j ≥ 1 do
    begin
        for i := 1 to j do
            x := x + 1
        j := ⌊j/3⌋
    end
```

25. **i := n**

```
while i ≥ 1 do
    begin
        x := x + 1
        i := ⌊i/2⌋
    end
```

26. **i := n**

```
while i ≥ 1 do
    begin
        for j := 1 to n do
            x := x + 1
        i := ⌊i/2⌋
    end
```