บทที่ 1

โครงสร้างข้อมูลกับโปรแกรม

ในบทนี้เราจะมาเรียนว่าทำไมถึงต้องเรียนเกี่ยวกับโครงสร้างข้อมูล จริง ๆ ผมจะสอนตัวโครงสร้างข้อมูลไปเลยก็ได้ แต่ว่าเดี๋ยวพวกเราจะหมดแรงจูงใจที่จะเรียนเพราะไม่รู้ว่าจะเอาไปใช้ทำอะไร

ทำไมต้องเรียนโครงสร้างข้อมูล

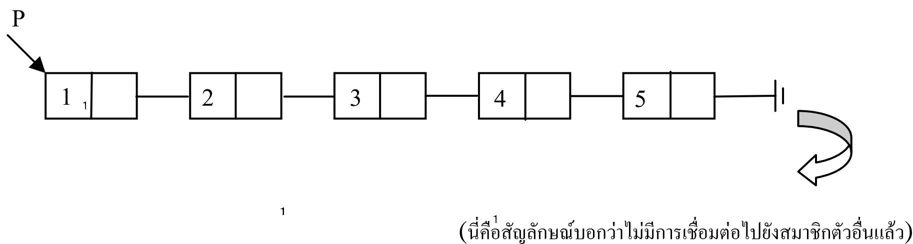
เอาล่ะ ขอผมเริ่มเลยก็แล้วกัน ทำไม เราจึงต้องเรียนโครงสร้างข้อมูล เหตุผลใหญ่ ๆ มีอยู่สองประการ คือ

1. **จะได้รู้จักโครงสร้างข้อมูลที่ใช้กันอยู่ในปัจจุบัน** โครงสร้างข้อมูลที่ใช้กันอยู่ในปัจจุบันล้วนมีพื้นฐานมาจากโครงสร้างข้อมูลที่ผมจะสอนในหนังสือเล่มนี้ทั้งสิ้น ดังนั้น เมื่อเราเรียนรู้โครงสร้างข้อมูลพื้นฐานแล้ว เราก็สามารถเข้าใจ code ต่าง ๆ ได้ง่ายขึ้นเมื่อเราไปทำงานกับคนอื่น และยังสามารถนำโครงสร้างพื้นฐานเหล่านี้มาดัดแปลงใช้ในโปรแกรมของเราเองโดยไม่ต้องเสียเวลาสร้างโครงสร้างข้อมูลใหม่ทั้งหมด

2. **จะได้สามารถเลือกใช้โครงสร้างข้อมูลได้อย่างถูกต้องเหมาะสมกับงาน** เหมาะสมอย่างไรเหมาะสมตรงไหน เดี๋ยวมีตัวอย่างให้ดู ที่ผมว่าเหมาะสมนั้นหมายความว่าเราสามารถหาข้อมูลที่ต้องการได้เร็วจากโครงสร้างข้อมูลที่เราเลือก ในงานหนึ่งเราอาจเลือกใช้โครงสร้างข้อมูลได้จากหลายโครงสร้าง แต่ว่าอาจมีเพียงโครงสร้างข้อมูลชนิดเดียวที่ใช้เวลาในการดึงข้อมูลที่เราต้องการน้อยกว่าโครงสร้างข้อมูลชนิดอื่น ดังตัวอย่างต่อไปนี้

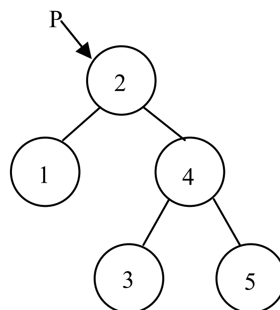
สมมติเราเลือกเก็บจำนวนเต็มห้าจำนวน (1 ถึง 5) ลงในโครงสร้างข้อมูล เราอาจเก็บได้ดังนี้ คือ

- ทางเลือกที่ 1 คือ เรียงเก็บใส่ Sorted Linked List โดยมีจุดเริ่มต้นการค้นข้อมูลที่ P (ขอให้ดูรูป 1.1ไปก่อนยังไม่ต้องคิดมาก)



รูป 1.1: Sorted Linked list ซึ่งเก็บจำนวนเรียงลำดับ 5 จำนวน

- ทางเลือกที่ 2 (รูป 1.2) คือ เราอาจเก็บใส่โครงสร้างข้อมูลแบบต้นไม้ (tree) ต่อไปผมจะขอเรียกว่า tree ไปตลอดเพราะพอพูดถึง “ต้นไม้” ทีไร ผมจะนึกถึงต้นไม้จริง ๆ ทุกที ให้ tree นี้มีจุดเริ่มการค้นหาที่ P เหมือนกัน ในภาพเป็น Binary Search Tree (คือ เรียงข้อมูลจากน้อยไปมาก ให้พวกเราสังเกตลักษณะก็พอ เดี๋ยวค่อยเรียนจริง ๆ อีกที)

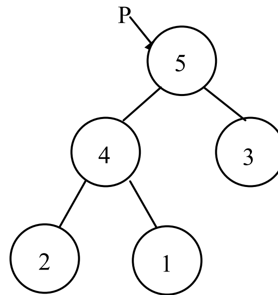


รูป 1.2: Binary Search Tree ซึ่งเก็บจำนวนเรียงลำดับ 5 จำนวน

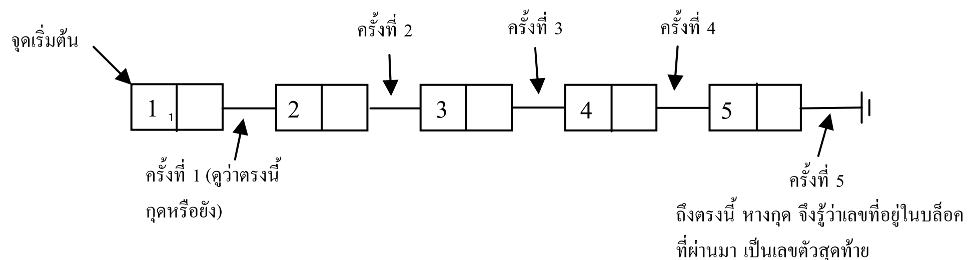
- ทางเลือกที่ 3 (รูป 1.3) คือ เราอาจเก็บใส่ Heap ซึ่งเป็น tree แบบหนึ่ง (ในภาพคือ Maxheap ซึ่งเอาจำนวนที่มากที่สุกไว้บนสุดเสมอ)

จริง ๆ ยังใช้โครงสร้างข้อมูลได้อีกหลายแบบเพื่อเก็บเลข 1 ถึง 5 แต่ผมจะเอาแค่ 3 แบบที่กล่าวมาเปรียบเทียบกันเท่านั้น ถ้างานของเราคือ การดึงเอาข้อมูลตัวที่มากที่สุดมาใช้เรื่อย ๆ เราควรจะเลือกโครงสร้างข้อมูลตัวไหน ทุกคนคงจะเห็นชัดแล้วว่า ถ้าใช้ Sorted Linked List เราต้อง

ทำการตรวจสอบข้อมูลเป็นจำนวน 5 ครั้ง จากซ้ายไปขวา ข้อมูลตัวที่ค่ามากที่สุดจะอยู่ที่ตำแหน่งสุดท้าย (รูป 1.4)



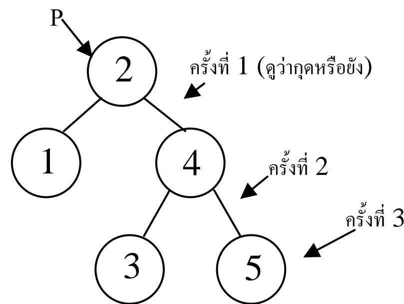
รูป 1.3: Ma xheap ซึ่งเก็บจำนวนเรียงลำดับ 5 จำนวน



รูป 1.4: ขั้นตอนการหาจำนวนที่ค่ามากที่สุดจาก Sorted Linked List

ถ้าเป็น Binary Search Tree จะต้องทำการตรวจสอบข้อมูล 3 ครั้ง ข้อมูลตัวที่ค่ามากที่สุดจะอยู่ที่ตำแหน่งขวาสุดของ tree (รูป 1.5) แต่ถ้าเป็น Maxheap เราเอา 5 มาจากจุดเริ่มต้น P ได้เลย ไม่ต้องเสียเวลาหา

จะเห็นได้ว่าการเลือกโครงสร้างข้อมูลมีผลต่อความเร็วของโปรแกรมจริง ๆ เท่านี้ทุกคนก็รู้ว่าทำไมถึงต้องเรียนแล้วนะ แต่การเปรียบเทียบความเร็วในการหาข้อมูลนั้น ทางทฤษฎีมีวิธีวิเคราะห์อยู่หลายแบบ ไม่ใช่แค่ทำง่าย ๆ อย่างดูรูปแล้วตอบ



รูป 1.5: ขั้นตอนการหาจำนวนที่ค่ามากที่สุดจาก Binary Search Tree

การคำนวณความเร็วของโปรแกรม

ในการหาความเร็วในการทำงานของโปรแกรม เรามักจะไม่ได้เห็นรูป แต่เห็นเป็น code เลยมากกว่า เราจะสามารถมองความเร็วของโปรแกรมจากการดู code เลยได้อย่างไร

ก่อนที่จะทำได้ เราต้องมาเรียนรู้เรื่องทฤษฎีเสียก่อน เราจะเริ่มจากนิยามแรกที่เป็นพื้นฐานในการคิดความเร็วของ code

นิยามที่ 1-1

Big O

$T(N) = O(f(N))$ ถ้า $T(N) \le cf(N)$ โดยมี c กับ N_0 เป็นค่าคงที่และ $N \ge N_0$

นิยามนี้เป็นการบอกว่า $T(N)$ มีค่าไม่เกินเท่าใด เมื่อ N มีค่ามาก หรืออีกนัยหนึ่งคือการบอกว่า $T(N)$ มีการเติบโตอย่างไร

ตัวอย่างที่ 1-1

เมื่อ $T(N) = 339N$ และ

$f(N) = N^2$

ถ้าให้ $N_0 = 339$ และ $c = 1$

ก่อนอื่นเราสามารถจะเห็นได้ทันทีว่า $339N_0 \le 1 * N_0^2$

(โดย $339N_0$ คือ $T(N_0)$ และ $1 * N_0^2$ คือ $c * f(N_0)$)

เป็นไปตามนิยามของ Big O เพราะฉะนั้น $T(N) = O(f(N)) = O(N^2)$

แต่ว่านี่ก็ไม่ใช่เพียงคำตอบเดียว

ดู $T(N) = 339N$ ให้ดีจะเห็นว่า $T(N) \le 340N$ ด้วยนี่นา (โดยตอนนี้ให้ $f(N) = N$)

$\therefore T(N)$ ก็เท่ากับ $O(N)$ ด้วย

อย่างนี้ก็มีค่า Big O เกินหนึ่งคำตอบสิ แล้วเราควรตอบตัวไหนดี

ให้ตอบเป็นค่าที่น้อยที่สุด เพราะค่าที่น้อยที่สุด คือค่าที่เราเอามาใช้จริงๆในการคิดความเร็ว ของ

โปรแกรม (อย่าพึ่งงง รอดูตัวอย่างต่อไป) เพราะฉะนั้นคำตอบของข้อนี้คือ $O(N)$

แล้ว $O(N)$ มันเกี่ยวอะไรกับการหาความเร็วในการ execute ของ code ขอให้เราดูตัวอย่างต่อไป

ตัวอย่างที่ 1-2

จงหาเวลาในการ run code ต่อไปนี้

```
sigmaOfSquare(int n)      // calculate ∑_{i=1}^{N} i^2
{
1:         int tempSum;
2:         tempSum = 0;
3:         for (int i=1;i<=n;i++)
4:            tempSum += i*i;
5:         return tempSum;
}
```

จากตัวอย่างนี้ ก่อนอื่นเรามาลองคิดเวลาในการ run เป็น unit ดู ดูทีละบรรทัดไปเลย

บรรทัดที่ 1: int tempSum;

นี่คือการ declare variable ผมสมมติให้การทำอย่างนี้ใช้เวลา 1 unit (เราต้องสมมติเวลา เพราะนี่เป็นการดูจาก code เราไม่ได้ run โปรแกรมแล้วจับเวลาจริง ๆ แต่การสมมติแบบนี้คือแนวทางในการวิเคราะห์โปรแกรมด้วยตัวเราเอง)

บรรทัดที่ 2 และ 5: tempSum = 0; return tempSum;

ในทั้ง 2 บรรทัดนี้ แต่ละบรรทัดมีการกระทำอย่างเดียว ให้ถือว่าใช้เวลาบรรทัดละ 1 unit

บรรทัดที่ 3: for (int i=1;i<=n;i++)

คราวนี้จะต่างกับบรรทัดก่อน ๆ เพราะเป็น loop เรามาดูว่า for loop นี้ทำอะไรบ้าง

1. set i ให้มีค่าเริ่มต้นเป็น 1 - เป็นการ initialize ค่า i ทำแค่ครั้งเดียวเท่านั้น เพราะฉะนั้นเราให้ใช้เวลา 1 unit
2. ทดสอบว่า i <= n หรือไม่ - นี่จะเป็นการทดสอบก่อนเข้า loop นับตั้งแต่แรกเริ่มคือ i มีค่าเป็น 1 ส่วนการ ทดสอบครั้งสุดท้าย คือ ตอนที่จะเข้า loop ไม่ได้เป็นครั้งแรก นั่นคือ เมื่อ i มีค่าเป็น n+1 เพราะฉะนั้น การ ทดสอบทั้งหมดต้องทำ n+1 ครั้ง เราให้เป็นหน่วยเวลา n+1 unit
3. เพิ่มค่า i เป็นค่าที่มากกว่าเดิมอยู่ 1- นี่จะเป็นการเพิ่มค่า i ทุกครั้งก่อนขึ้น loop ใหม่ ครั้งแรกจะเกิดตอนที่ i เท่ากับ 1 ส่วนครั้งสุดท้ายจะเกิดขึ้นตอนที่ i เท่ากับ n และจะเปลี่ยนค่า i เป็น n+1 (พอเป็น n+1 ก็จะเข้า loop ไม่ได้อีก) เพราะฉะนั้น การเพิ่มค่า i นี้จะมี n ครั้ง เราให้เป็นหน่วยเวลาเท่ากับ n unit

จากทุกๆขั้นตอนในบรรทัดนี้ รวมแล้วใช้เวลา 1 + n+1 + n = 2n + 2

บรรทัดที่ 4: tempSum += i*i;

ตัว tempSum += i*i; นี้อยู่ใน loop จากการดูบรรทัดที่ 3 เราทราบแล้วว่าจะมีการเพิ่มค่าทั้งหมด n ครั้ง เพราะฉะนั้น statement นี้ ก็ต้องมีการ execute n ครั้ง เช่นกัน แต่ statement นี้ทำอะไรบ้างในการผ่าน loop แต่ละครั้ง

1. มีการคูณ 1 ครั้ง - ให้เป็นเวลา 1 unit

2. มีการบวก 1 ครั้ง - ให้เป็นเวลา 1 unit
3. มีการทำ assignment 1 ครั้ง -ให้เป็นเวลา 1 unit

ในเมื่อผ่าน loop ทั้งหมด n ครั้ง เวลาทั้งหมดของ statement ในบรรทัดนี้จึงเป็น
(1 + 1 + 1)*n = 3n unit
เมื่อรวมเวลาทุก unit ของทั้งโปรแกรม เราจะได้เวลารวมเท่ากับ
1 + 1 + 1 + (2n + 2) + 3n = 5n + 5 unit

โปรแกรมแค่นี้ยังเสียเวลาในการกะ running time ขนาดนี้ ดังนั้น โปรแกรมใหญ่ ๆ ยิ่งไม่ต้องพูดถึง ดังนั้น ถ้าจะคำนวณ running time ของโปรแกรมให้รวดเร็วเราจำเป็นต้องใช้การประมาณอย่างคร่าว ๆ กว่านี้ นั่นคือต้องใช้ Big O นั่นเอง

สำหรับโปรแกรมนี้ ส่วนที่จะโดน execute มากที่สุดก็คือ ส่วนที่ loop นั่นเอง เวลาในการ run ส่วนอื่นจะน้อยมากเมื่อเทียบกับส่วนที่เป็น loop ซึ่งทำซ้ำไปประมาณ n ครั้ง ดังนั้นเราสามารถประมาณ running time ของ code นี้ในเทอมของ Big O ได้คือ

Running time = $O(n)$

เราสามารถตรวจสอบความถูกต้องของค่าประมาณนี้ได้
จากการที่เรารู้ว่า running time จริง ๆ เท่ากับ 5n + 5 เราจะได้ว่า

$$5n + 5 = O(n)$$

ซึ่งจากนิยามของ Big Oเราจะได้

$$5n + 5 \le c * n \text{ , โดย } c = 6$$

จากตัวอย่าง 1-2 จะเห็นได้ว่า เราสามารถหา Big O ได้จากการดู for loop เท่านั้นเอง ผมจะสรุปหลักของการดู loop ให้ดังต่อไปนี้

For Loop: ค่า Big O คือ จำนวนครั้งที่ loop ทำซ้ำโดยใช้ตัวแปรเป็นหลัก ในตัวอย่าง 1-2 นี้ก็คือ n

Nested Loop: เอาค่า Big O ของ loop แต่ละชั้นมาคูณกัน อย่างเช่นถ้ามีโปรแกรม

```
1:     for (i =1; i <=n; i++)
2:       for (j =1; j <= n; j++)
3:         statements;
```

ค่า Big O จะเป็น $O(n*n)$

ทางทฤษฎี เราสามารถเขียนนิยามออกมาได้ดังต่อไปนี้

นิยามที่ 1-2

ถ้า $T_1(N) = O(f(N))$ และ $T_2(N) = O(g(N))$

$\therefore T_1(N)*T_2(N) = O(f(N)*g(N))$

สำหรับ code ในตัวอย่างนี้นั้น $f(N) = g(N) = N$

Statement ที่เรียงต่อกันแต่ละบรรทัด: เอาค่า Big O ของแต่ละ statement มาเทียบหาตัวที่มากที่สุด ตัวอย่างเช่น ถ้ามีโปรแกรม

```
1:     for (i = 0; i <= n; i++)
2:       statement1;
3:     for (j = 0; j <= n; j++)
4:       for (k = 0; k <= n; k++)
5:         statement2;
```

code นี้เกิดจากการเรียง for loop ที่ 1 และ for loop ที่ 2

Big O รวม คือ $O(n^2)$ ซึ่งเป็นของ loop 2 ที่ไม่คิดเวลาจาก loop แรก ก็เพราะว่าไม่มีนัยสำคัญเลยเมื่อเทียบกับ loop 2 ทางทฤษฎี เราสามารถเขียนนิยามของการหา running time จาก Statement ที่เรียงต่อกันแต่ละบรรทัดออกมาได้ดังต่อไปนี้

นิยามที่ 1-3

ถ้า $T_1(N) = O(f(N))$ และ $T_2(N) = O(g(N))$

$\therefore T_1(N)+T_2(N) = \max(O(f(N),O(g(N)))$

ประโยคแบบมีเงื่อนไข:

ถ้ามี Code

```
1:     if (condition)
2:        Statement1
3:     Else
4:        Statement2
```

เวลาในการ run ของโปรแกรมนี้คือ เวลาในการ run ที่มากที่สุดระหว่างเวลาในการ run ของ statement 1 และ statement 2 เราไม่รู้ว่า statement ไหนจะถูกเลือกกันแน่ ดังนั้นต้องเลือกตัวที่แย่กว่าไว้ก่อน

ตอนนี้ เรารู้วิธีหา Big O ของ code แล้ว ก่อนที่จะมาเรียนนิยามต่อไป เรามาลองดูตัวอย่างการหาเวลาในการ run ของ code ที่เป็น recursion (recursion คือ โปรแกรมที่เรียกตัวเองเรื่อย ๆ แต่ input ที่ใช้ต้องขนาดลดลงเรื่อย ๆ จนถึงขนาดพื้นฐานที่จะไม่เรียกตัวเองอีก ไม่งั้นจะไม่มีวัน run จนจบ)

ตัวอย่างที่ 1-3

จงหาเวลาในการ run ของ code ต่อไปนี้ ทั้งแบบหน่วย unit และ Big O

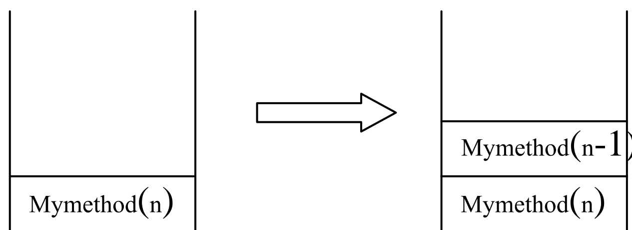
```
1:     My method (int n){
2:        if (n == 1) {
3:           return 1;
4:        } else {
5:           return 2*mymethod(n - 1) + 1;
6:        }
7:     }
```

ก่อนอื่น เรามาดู Big O ก่อน การ run ของโปรแกรมนี้ จำนวนครั้งที่คูณและบวกจะขึ้นอยู่กับค่าของ n (คือ จำนวนครั้งของการเรียกใช้ method นั่นเอง เพราะฉะนั้น Big O ก็คือ $O(n)$) มีข้อสังเกตว่าแม้ว่าเราจะเปลี่ยน code ให้ใช้ iteration แทน recursion แล้วก็ตาม (code ข้างล่างนี้คือแบบที่ใช้ iteration นั่นคือใช้ loopแทนการเรียกใช้ method) โปรแกรมนี้ก็จะยังมีค่า Big O เป็น $O(n)$ เหมือนเดิม

```
1:    mymethod (int n) {
2:      int result = 1;
3:      int i = n;
4:      while (i > 1) {
5:        result = 2 * result + 1;
6:        i = i-1;
7:      }
8:      return result;
9:    }
```

แต่การใช้ recursion จะสิ้นเปลือง memory เพราะการเรียกใช้ method ในแต่ละครั้งต้องเสียเนื้อที่ใน memory เพื่อสร้าง activation record ใหม่ (รูป 1.6)

คราวนี้ มาลองคิดเป็น unit time ดูเล่นๆ (ดูจาก code ตอนทำ recursion นะ) ถ้าเราให้ $A(n)$ คือจำนวน operation ที่เกิดขึ้น เมื่อมี input คือ n (เราจะให้จำนวน operation ทำหน้าที่เป็น unit time ของเรานั่นเอง)



รูป 1.6: สิ่งที่เกิดขึ้นที่ memory เมื่อมีการเรียกใช้ method จากตัวอย่าง 1.4

$\therefore A(1) = 2$ (ค่า 2 นี่มาจาก 2 operations ซึ่ง operation แรกคือการทดสอบค่าว่า $n = 1$ หรือไม่ ส่วน operation ที่สองคือการ return ค่า 1

แล้ว A(2) จะเป็นเท่าไรล่ะ เราสามารถนับจำนวน Operation ได้ดังนี้

$A(2) =$ ทดสอบ if (n = = 1) มาแล้ว (1 operation) + การคูณ (1 operation) + การบวก (1 operation) + $A(1)$ + การ return (1 operation)

$\therefore A(2) = A(1) + 4$

$A(3)$ ก็จะมีรูปแบบเช่นเดียวกันคือ

$A(3) =$ ทดสอบ if(n = = 1) มาแล้ว + การคูณ + การบวก + $A(2)$ + การ return

$\therefore A(3) = A(2) + 4$

จะสังเกตได้ว่า $A(N) = A(N-1) + 4$ โดยมี $A(1) = 2$

นี่คือ สมการของความสัมพันธ์เวียนเกิด ซึ่งเราสามารถทำเป็นรูปแบบปิดได้เพื่อหาค่า unit time ที่แท้จริง จากการสังเกต เราจะได้

$A(5) = A(4) + 4$

$= (A(1) + 4) + 4 + 4 + 4$, ใช้การคลี่

$A(5) = 4(4) + A(1)$

$\therefore A(n) = 4(n-1) + 2 = 4n - 2$

เพื่อความแน่ใจในรูปแบบปิดที่เราหาได้ เราสามารถพิสูจน์ให้เห็นว่า $A(n) = 4n - 2$ มีค่าเท่ากับ $A(n) = A(n-1) + 4, A(1) = 2$ โดยวิธีการอุปนัยทางคณิตศาสตร์

Base case: (n=1)

เราแทนค่าจากสมการรูปแบบปิดของเรา จะได้ว่า

$A(1) = 4(1) - 2 = 2$ ซึ่งผลลัพธ์ตรงกับนิยามแรก

Inductive hypothesis:

ตั้งสมมติฐานว่า $4n - 2$ มีค่าเท่ากับ $A(n-1) + 4$

ต้องพิสูจน์:

ให้ได้ว่า $4(n+1) - 2 = A(n) + 4$ จึงจะถือว่า $4n - 2$ เป็นรูปแบบปิดที่ถูกต้อง

สมการ

$4(n+1) - 2 = A(n) + 4$

นี้ เราสามารถจัดให้เป็น

$4n - 2 + 4 = [A(n-1) + 4] + 4$

ได้ด้วยการกระจาย และนิยามของความสัมพันธ์เวียนเกิด

เรารู้จากข้อสมมติฐานว่า $4n-2$ มีค่าเท่ากับ $A(n-1)+4$
เพราะฉะนั้นจากสมการข้างบนนี้ เราสามารถสรุปได้อย่างชัดเจนว่า
$4(n+1)-2 = A(n)+4$

แสดงว่า $A(n) = 4n-2$ คือจำนวน operation (ในแบบ unit time) ที่ถูกต้อง

ข้อสังเกต: ค่า $4n-2$ ที่ได้นี้ก็สอดคล้องกับค่า $O(n)$ ที่เราได้ตอนแรก เพราะฉะนั้นจะเห็นได้ว่าการหา running time ด้วย Big O มีความสะดวกรวดเร็วกว่าการหาแบบ unit time อยู่มาก

ตอนนี้ ทุกคนคงจะมีความเข้าใจเกี่ยวกับ Big O แล้ว การเลือกใช้โครงสร้างข้อมูลสำหรับงานหนึ่ง ๆ จึงจำเป็นต้องคำนึงถึงค่า Big O ในการ run ด้วย ต่อไปนี้จะเป็นตัวอย่างของปัญหาปัญหาหนึ่งซึ่งเราสามารถใช้การเปรียบเทียบ Big O ของ code ในการเลือกโปรแกรมที่เหมาะสมที่สุดสำหรับการแก้ปัญหานี้

ตัวอย่างที่ 1-4

การแก้ปัญหา Maximum Subsequence Sum มีหลายวิธี เราจะนำเสนอวิธีต่าง ๆ และวิเคราะห์ว่าวิธีไหนเหมาะสมที่สุด โดยเปรียบเทียบ Big O ของแต่ละวิธี (ตัวอย่างนี้เป็นการเปรียบเทียบ Big O ของ algorithms ไม่ได้เกี่ยวกับโครงสร้างข้อมูลโดยตรง แต่อย่างที่แสดงในตัวอย่าง 1.1 โครงสร้างข้อมูลแต่ละชนิดให้ Big O ต่างกัน)

ก่อนอื่น เราต้องรู้ว่า ปัญหา Maximum Subsequence Sum นั้นคืออะไร

ถ้ามีจำนวนเต็ม $A_1, A_2, A_3, \ldots A_n$

Maximum Subsequence Sum คือ ค่า $\sum_{k=i}^{j} A_k$ ที่มีค่ามากที่สุด นั่นก็คือ ผลบวกของจำนวนติดกัน (เริ่มหรือจบตรงไหนก็ได้ ขอให้เป็นจำนวนที่ติดกันก็พอ) ที่มีค่ามากที่สุด อย่างเช่น ถ้ามี
-2, 11, -6, 16, -5, 7

ผลรวมของ 11, -6, 16 คือ 21 ผลรวมนี้ก็ดูมีตัวเลขมาก (ตามสัญชาติญาณ) แต่ที่มากที่สุดจริงๆ ก็คือ จาก 11, -6, 16, -5, 7 ซึ่งรวมแล้วได้ 23 ค่านี้แหละคือ Maximum Subsequence Sum

ตอนนี้ เมื่อเรารู้นิยามของปัญหานี้แล้ว มาดูวิธีแก้ปัญหานี้ในแบบต่าง ๆ

วิธีที่ 1

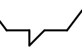
```
1:     int maxSubSum01 ( int [] a) {
2:      int maxSum = 0;
3:      for (int i = 0; i < a.length; i++) {
4:         for (int j = i; j < a.length; j++) {
5:            int theSum = 0;
6:            for (int k = i; k <= j; k++) {
7:               theSum += a[k];
8:            }
9:            if (theSum > maxSum) {
10:              maxSum = theSum;
11:           }
12:        }
13:        return maxSum;
14:     }
15:    }
```

ค่า Big O เมื่อดูจากตัวโปรแกรมแล้ว จะมีค่าเท่ากับ $O(N^3)$ เพราะค่า max ของ loop 3 ชั้น คือ N^3 แต่ทำไมถึงนานขนาดนี้ เรามาดูขั้นตอนของวิธีนี้กัน วิธีนี้คือการดู array แล้วเก็บผลบวกของตำแหน่งติดกันบน array โดย i เป็น index ของจำนวนแรก และ j เป็น index ของจำนวนสุดท้าย ค่า i กับ j จะเปลี่ยนไปเรื่อยๆจนครอบคลุมทุกความเป็นไปได้ สมมุติเรามี array ของจำนวนเต็ม 4 หลักคือ

-2, 11, -6 และ 4

เมื่อ i = 0, j = 0, k = 0:

-2	11	-6	4

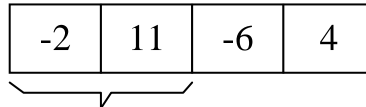


theSum = -2 **maxSum** ยังคงเป็น **0**

ในการ loop หา **theSum** ครั้งต่อไป j ได้กลายเป็น 1 เพราะฉะนั้นค่าที่ k เป็นได้จะมี 2 ค่า คือ 0 กับ 1

เมื่อ i = 0, j = 1, k = 0: ค่าต่างๆยังคงเหมือนเดิม

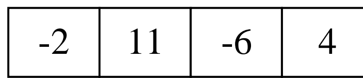
เมื่อ i = 0, j = 1, k = 1:



theSum = -2 + 11 = 9 ส่วน **maxSum = 9**

ในการ loop หา theSum ครั้งต่อไป j จะกลายเป็น 2 เพราะฉะนั้น ค่า k ที่เป็นได้ก็จะมีได้ตั้งแต่ 0 ถึง 2

เมื่อ i = 0, j = 2, k = 0:



theSum = -2

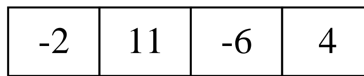
เมื่อ i = 0, j = 2, k = 1**: theSum = -2+11 =9**

เมื่อ i = 0, j = 2, k = 2: **theSum = 9 – 6 = 3** **maxSum** ยังคงเป็น **9**

ในการ loop หา **theSum** ครั้งต่อไป j จะกลายเป็น 3 และต่อไปก็ 4 (ขอข้าม เพราะวิธีการเหมือนเดิม)

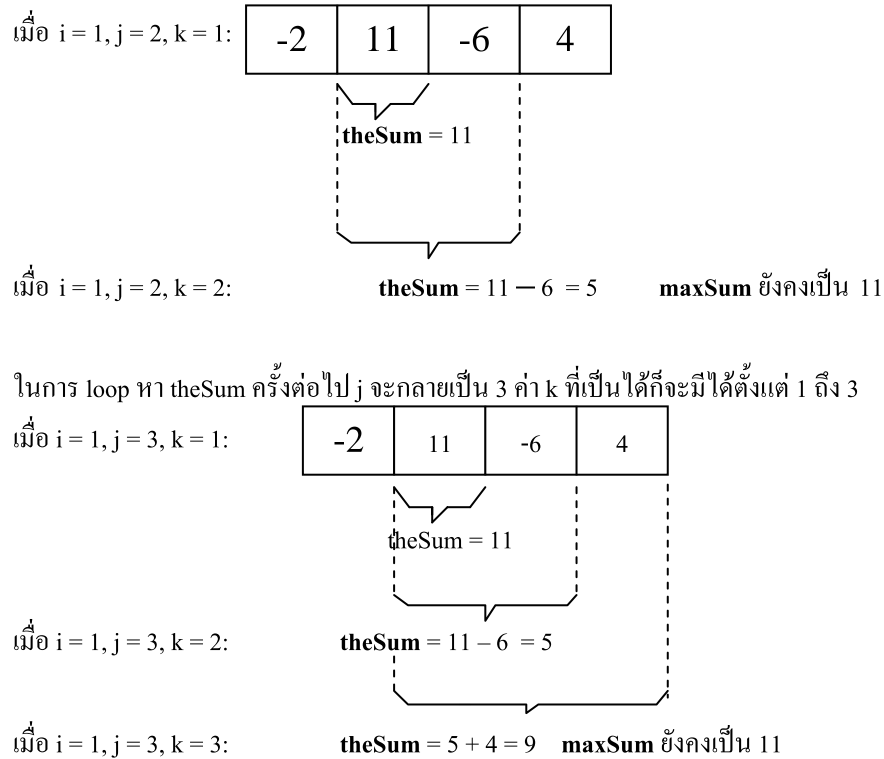
เมื่อ i เปลี่ยนเป็น 1 การ run ก็จะเหมือนเมื่อ i = 0 เพียงแต่เริ่มจาก array element index ที่ 1 แทนที่จะเป็น 0

เมื่อ i = 1, j = 1, k = 1:



theSum = 11 **maxSum** เปลี่ยนเป็น 11

ในการ loop หา theSum ครั้งต่อไป j ได้กลายเป็น 2 ค่าที่ k เป็นได้จะมี 2 ค่า คือ 1 กับ 2



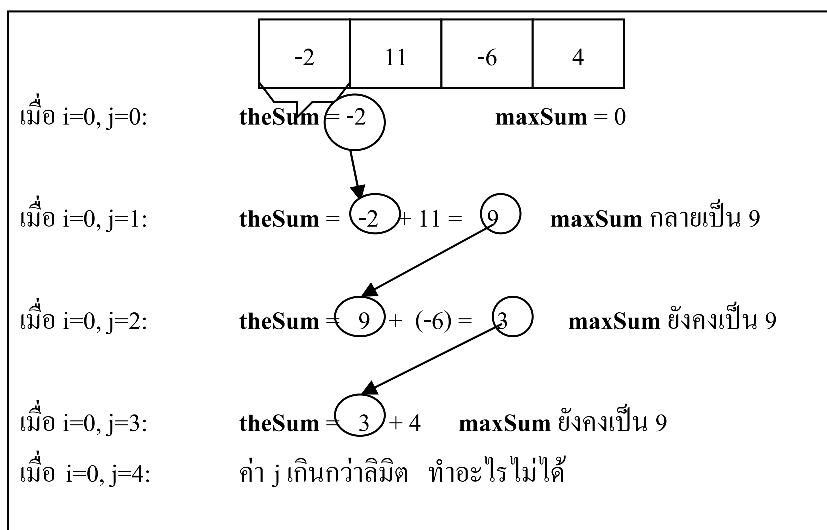
ไม่ทำให้ดูต่อแล้วนะ จากตัวอย่างจะเห็นได้ว่าวิธีนี้เป็นการลองทุกกรณี และวิธีนี้ก็มีจุดอ่อนที่เห็นได้ชัดเจนมาก นั่นคือในการ loop หา **theSum** แต่ละครั้ง โปรแกรมต้องเริ่มเก็บค่าใหม่ตั้งแต่ต้น แทนที่จะบวกต่อจากค่าที่หาได้ในการ loop ครั้งก่อน อย่างเช่น ถ้าบวกจากตำแหน่ง 0 ถึง2 ไปแล้ว ตอนที่บวกจากตำแหน่งที่ 0 ถึง 3 กลับต้องเริ่มบวกใหม่หมด แทนที่จะใช้ผลจากการบวกจากตำแหน่ง 0 ถึง2 ได้

วิธีที่ 2 เรามาดูตัวโปรแกรมกันก่อน ในรูปข้างล่าง

ถ้าดูจาก Big O เราจะเห็นว่ามี for loop อยู่สองลูป ซึ่งแต่ละลูปมีจำนวนการ run ขึ้นตรงกับขนาดของข้อมูลเข้า ดังนั้น โปรแกรมนี้จึงมีค่า Big O เป็น $O(N^2)$ ซึ่งน้อยกว่าวิธีแรก ดังนั้นโปรแกรมนี้เร็วกว่าแน่นอน ถ้าเลือกระหว่างวิธีที่หนึ่งกับวิธีนี้ เราก็ควรจะเลือกวิธีนี้

```
1:    int   maxSubSum02(int [] a) {
2:     int maxSum = 0;
3:     for (int i = 0; i < a.length; i++) {
4:       int theSum = 0;
5:       for (int j = i; j < a.length; j++) {
6:             theSum += a[j];
7:             if (theSum > maxSum) {
8:                maxSum = theSum;
9:             }
10:      }
11:    }
12:    return maxSum;
13:   }
```

วิธีนี้เร็วกว่าวิธีแรกยังไง จากตัวโปรแกรม เราจะเห็นได้ว่า โปรแกรมนี้ทำการเก็บค่า theSum ไปเรื่อย ๆ ตามขนาดของ array แต่จะต่างจากวิธีแรก คือ จะมีการเก็บสะสมค่า theSum ไว้ใช้ในการคำนวณครั้งต่อไป โดยมี i เป็นตัวกำหนดตำแหน่งที่จะเริ่มบวก ต่อจากนั้น ก็จะเริ่มบวกจากตำแหน่งที่ i นั้นๆ โดยใช้ j เป็นตัวทำการลูป ดังนั้นลูปในสุดของโปรแกรมนี้จะทำหน้าที่เหมือนลูปสองลูปด้านในของวิธีแรก และการลูปนี้ ก็จะเป็นการ update ค่า maxSum ไปในตัวเลย นี่คือเหตุผลที่ทำให้ประหยัดเวลาได้มาก รูป 1.7 เป็นการแสดงการทำงานของโปรแกรมนี้ เมื่อ i มีค่าเป็น 0 จะเห็นได้ว่ามีการสะสมค่า theSum ไว้ใช้ใน iteration ต่อไป



รูป 1.7 ขั้นตอนการหา max sub sum วิธีที่สอง

เมื่อ i มีค่าเป็น 1 กระบวนการคิดยังคงเหมือนเดิม เพราะฉะนั้น ถ้าขนาดของอาร์เรย์คือ N เราจะมีจุดเริ่มต้นนับอยู่ N จุด โดยในแต่ละจุด มีการบวกสะสมค่าอย่างมากที่สุด N ครั้ง ซึ่งนี่ก็คือที่มาของ $O(N^2)$ นั่นเอง

วิธีที่ 3

ต่อไปมาดูวิธีที่สามกัน วิธีที่สามนี้ใช้ความรู้จากความสัมพันธ์เวียนเกิด [หนังสืออ. สมชาย] มาเป็นหลักในการเขียนโปรแกรม หลักการที่ใช้ คือ การแบ่งแยกและเอาชนะ (divide and conquer) นั่นคือ แบ่งปัญหาออกเป็นส่วนเล็ก ๆ สองส่วนเท่า ๆ กัน แล้วก็แบ่งไปเรื่อย ๆ จนถึงปัญหาพื้นฐาน (ซึ่งแก้ได้ง่าย) หลังจากนั้นก็รวมคำตอบส่วนต่าง ๆ เข้าด้วยกัน
สำหรับในการแก้ปัญหา maximum subsequence sum นั้น ตัว sequence ที่เป็นคำตอบอาจจะอยู่ใน

- ครึ่งซ้ายของการแบ่งปัญหา หรือ
- ครึ่งขวาของการแบ่งปัญหา หรือ
- อยู่ทั้งสองข้าง (ต้องมีตัวสุดท้ายจากครึ่งซ้ายและตัวแรกจากครึ่งขวา)

อย่างในตัวอย่างข้างล่างนี้ ซึ่งเป็น array ที่มีสมาชิก 8 ตัว

1	-2	7	-6	2	8	-5	4

- ครึ่งซ้าย ค่า maxSubSum = 7
- ครึ่งขวา ค่า maxSubSum = 10
- maxSubSum ของครึ่งซ้ายที่มีตัวสุดท้าย (-6) คือ 1
- maxSubSum ของครึ่งขวาที่มีตัวแรก (2) คือ 10

เพราะฉะนั้น maxSubSum ของช่วงที่อยู่ทั้งครึ่งซ้ายและขวา = 10 + 1 = 11
เพราะฉะนั้นค่า maxSubSum ของตัวอย่างนี้คือ max ของ 7, 10 และ 11 นั่นก็คือ 11 นั่นเอง

ดังนั้นเราสามารถใช้การแบ่งข้างซ้ายขวาช่วยในการหาคำตอบได้ ตัวโปรแกรมเป็นดังรูป 1.8

ในการหาค่า Big O ของโปรแกรมในรูป 1.8 นี้ เราสามารถหาได้จากการประมาณ unit running time จากตัวโปรแกรมเอง ถ้าให้เวลาในการ run maxSumDivideConquer สำหรับ array ขนาด n

เป็น T(n) จะเห็นได้ว่าการเรียก maxSumDivideConquer เพื่อหาค่า maxsumleft และ maxsumright จะมีเวลาในการ run = T(n/2)

ส่วนลูปที่หา maxlefthalfSum กับ maxrighthalfSum แต่ละลูปจะ run ไปได้ n/2 ครั้ง เพราะตรวจดูแค่ครึ่งหนึ่งของ input

ส่วนอื่น ๆ ของโปรแกรมนี้จะ run ด้วยเวลาคงที่ ฉะนั้นสามารถตัดพวกนั้นทิ้งไปได้

จะได้ว่า T(n) = 2T(n/2) + 2O(n/2)

T(n) = 2T(n/2) + O(n)

เพราะว่า O(n) <= c*n ตามนิยาม เราสามารถแทน O(n) ด้วย c*n ได้ในการประมาณค่าเวลา เพราะฉะนั้น ค่าประมาณคือ T(n) = 2T(n/2) + cn

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

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

เพราะฉะนั้น

$$\frac{T(\frac{n}{2})}{\frac{n}{2}} = \frac{T(\frac{n}{4})}{\frac{n}{4}} + c \quad (2)$$

$$\frac{T(\frac{n}{4})}{\frac{n}{4}} = \frac{T(\frac{n}{8})}{\frac{n}{8}} + c \quad (3)$$

....

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

เมื่อเอา (1) + (2) + (3) +…..+ (x) จะได้

T(n)/n = T(1)/1 + c * log (n)

T(n) = n* T(1) + c*nlog(n)

เพราะว่า T(1) เป็นค่าคงที่ ดังนั้นเราจึงสามารถตอบได้ทันทีว่า T(n) มี Big O เป็น O(n log(n) ซึ่งค่า Big O นี้น้อยกว่าค่า Big O ของวิธีที่ 2 วิธีนี้จึงน่าจะดีกว่า (แต่อย่าลืมว่าการใช้ recursion ต้องเปลืองหน่วยความจำในการสร้าง stack)

วิธีที่ 4

คือการเอาวิธีที่ 2 มาปรับปรุงให้เร็วขึ้น โดยอาศัยข้อสังเกตที่ว่า

- ตัวแรกของ maximum subsequence sum จะเป็นลบไม่ได้ อย่างเช่น

3	-5	1	4	7	-4		

ค่า -5 จะเป็นตัวแรกของ sequence ของคำตอบไม่ได้ แค่เราให้ตัวแรกเป็นสมาชิกของ array ตัวต่อไปแทน (ในที่นี้คือเลข 1) เราก็จะได้ค่ารวมมากขึ้นแล้ว นั่นคือ ถ้ามีจำนวนบวกอยู่บ้าง ยังไงถ้าเราเริ่มคิด sequence จากจำนวนบวก เราก็จะได้ ค่ามากกว่าเมื่อเริ่มด้วยจำนวนลบแน่ๆ (อย่าลืมว่าเราใช้สมมติฐานว่า maximum subsequence sum มีค่าน้อยที่สุดคือ 0 ดังนั้นแม้ทุกจำนวนจะเป็นลบ เราก็ยังสามารถตอบได้ว่าค่า maximum subsequence sum นั้นเป็น 0)

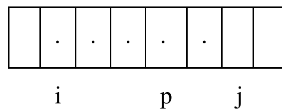
```
1:    int maxSumDivideConquer (int [] array, int
leftindex, int rightindex {
2:    // ต้องถือว่าขนาดของอาร์เรย์หารสองลงตัวพอดีจะได้แบ่งได้เท่ากัน ไม่เช่นนั้นยาก
3:     if (leftindex == rightindex) { // Base Case
4:       if (array[leftindex] > 0)
5:         return array[leftindex];
6:       else
7:         return 0; // maxSubSum มีค่าต่ำสุดคือ 0 เท่านั้น
8:     }
9:
10:    int centerindex =(leftindex + rightindex)/2;
11:    int maxsumleft = maxSumDivideConquer(array,
leftindex, centerindex);
12:    int maxsumright = maxSumDivideConquer ( array,
centerindex + 1, right);

13:    int maxlefthalfSum = 0, lefthalfSum = 0;
14:    // เก็บผลรวมที่มากที่สุดนับจากตัวสุดท้ายของครึ่งซ้ายลงไปถึงตัวแรก
15:    for (int i = center; i >= leftindex; i--) {
16:      lefthalfSum = lefthalfSum + array[i];
17:      if (lefthalfSum > maxlefthalfSum) {
18:        maxlefthalfSum = lefthalfSum;
19:      }
20:    }
21:    int maxrighthalfSum = 0, righthalfSum = 0;
22:    // เก็บผลรวมที่มากที่สุด นับจากตัวแรกของครึ่งขวาไปถึงตัวสุดท้าย
23:    for (int i = centerindex + 1; i <= rightindex; i++) {
24:      righthalfSum = righthalfSum + array [i];
25:      if (righthalfSum > maxrighthalfSum) {
26:        maxrighthalfSum = righthalfSum;
27:      }
28:    }
29:    // จากนั้นหา max ของทั้ง 3 กรณี คือ ของครึ่งซ้าย, ของครึ่งขวา, และของที่อยู่ทั้ง 2
ครึ่ง
30:   return max3 (maxsumleft, maxsumright, maxlefthalfSum
+ maxrighthalfSum)
31:   }
```

รูป 1.8 โค้ดการหา maximum subsequence sum ด้วย divide/ conquer

- subsequence ที่มีผลรวมเป็นลบ ไม่สามารถเป็นส่วนเริ่มต้นของ maximum subsequence sum ได้ด้วยเหตุผลเช่นเดียวกัน

ถ้าให้เราอยู่ระหว่างการทำลูป ลูปหนึ่ง โดยที่ i เป็น index ของตัวแรกของ subsequence และ j เป็น index ของตัวสุดท้ายของ subsequence (ซึ่งตัวสุดท้ายนี้ทำให้ subsequence มีค่าเป็นลบ) และ p เป็น index ใด ๆ ระหว่าง i+1 กับ j ดังรูป

array a



ขั้นต่อไปของลูปก็น่าจะเป็นการเลื่อน j ออกไป ซึ่ง

- ถ้า a[j] เป็นลบ เราก็ไม่ได้ maximum subsequence sum ที่ดีขึ้น ค่าของ maximum subsequence sum ก็จะยังคงไม่เปลี่ยน
- ถ้า a[j] เป็นบวก เราก็จะได้ค่า a[i]+…+a[j] ที่มากขึ้นกว่า a[i]+…+a[j ตัวเก่า] แต่เพราะ a[i]+…+a[j ตัวเก่า] เป็นลบ ค่าผลรวมจึงไม่มีทางสู้ค่าของ maximum subsequence sum ที่เก็บสะสมมาตอนที่ผลรวมยังเป็นบวกได้ หรือแม้แต่จะสู้ค่า a[j ตัวใหม่] เพียวๆก็ไม่ได้ ดังนั้นจึงสรุปได้ว่า ถ้ามี subsequence เป็นลบแล้วละก็ เราไม่ต้องทำการเลื่อน j ให้เสียเวลา เราควรไปเลื่อน i เลย

แต่ควรจะเลื่อน i ไปแค่หนึ่งช่องเท่านั้นหรือ จากสมมติฐานของข้อนี้ เรารู้ว่า a[j] เท่านั้นที่ทำให้ a[i]+…+a[j] มีค่าเป็นลบ เพราะฉะนั้น การเลื่อน i เพิ่มไปหนึ่งตำแหน่ง ก็มีแต่ทำให้ค่าผลรวมจาก a[i] ถึง a[p] (โดย p เป็นค่าที่อยู่ระหว่าง i ถึง j) ลดลง นั่นคือ ถ้าเลื่อน j จาก i ตัวนี้ ยังไงก็ไม่มีทางเพิ่ม max sub sum ได้ เพราะฉะนั้นถ้าจะให้ sequence มีค่ามากขึ้นได้อีก มีทางเดียวคือเราต้องเริ่มคิดจากตำแหน่งที่ j+1 นั่นคือเลื่อน i ไปที่ j+1 เลย การทำแบบนี้ทำให้การตรวจสอบเร็วขึ้นมาก

เรามาดูตัวโปรแกรมกันจากรูป 1.9 จากตัวโปรแกรมจะเห็นว่ามีการตรวจสอบหนึ่งรอบ โดยเรียงตรวจสอบทุกสมาชิก แต่ว่าจะ reset ค่า theSum (สะสมมาจากตำแหน่ง i ตามข้อสังเกตของเรา) ที่สะสมไว้ให้เป็น 0 เมื่อเจอตัวที่ทำให้ sequence เป็นลบ(ตัวนี้อยู่ที่ตำแหน่ง j) ดังนั้น theSum ต่อไปจะถูกสร้างจากการรวมเลขตำแหน่งที่ j+1 เป็นต้นไป ซึ่งเป็นไปตามข้อสังเกต

ของวิธีที่ 4 นี้ และจะเห็นได้ชัดเจนว่า Big O ลดลงมาเหลือแค่ O(n) เท่านั้น เพราะมีลูปเพียงลูปเดียว

```
1:    int maxsubsumOptimum (int[] array) {
2:    int maxSum = 0, theSum = 0;
3:    for (int j = 0; j < a.length; j++) {
4:       theSum = theSum + array [j];
5:       if ( theSum > maxSum) {
6:          maxSum = theSum;
7:                } else if (theSum < 0) {// ถ้าเจอ a[j] ที่ทำให้ทั้ง
8:                 //sequence เป็นลบ
9:
10:      theSum = 0;        // นี่คือจะมาเริ่มคิดใหม่จากตำแหน่ง j+1
11:     }
12:   }
13:   return maxSum;
14:   }
```

รูป 1.9 โค้ดของวิธีที่สี่ในการหา maximum subsequence sum

การมี log ใน Big O

จากตัวอย่างวิธีที่ 3 ของการแก้ปัญหา maximum subsequence sum จะเห็นว่ามี log เข้ามาเกี่ยวข้อง เรามีหลักง่าย ๆ ในการดูว่าโปรแกรมมี Big O เป็น log คือถ้าเราสามารถใช้เวลาคงที่ (O(1)) ในการแบ่งปัญหาปัญหาหนึ่งออกเป็นส่วน ๆ เท่า ๆ กัน (อย่างในตัวอย่าง maximum subsequence sum เราแบ่งปัญหาเป็นสองส่วนเท่า ๆ กัน) ปัญหานั้นจะมี Big O เป็น O(log n)ในการคิดปัญหาประเภทนี้เรามักสมมติให้มีการอ่านข้อมูลเข้าเรียบร้อยแล้ว มิฉะนั้นแค่การอ่านข้อมูลเข้าก็เป็น O(n) แล้ว

ตัวอย่างที่ 1-5

มีสมาชิก อาร์เรย์ a[0] ถึง a[n-1] ซึ่งเรียกจากน้อยไปหามากอยู่ในอาร์เรย์ a เรียบร้อยแล้ว เราต้องการจะหา x ว่าอยู่ในช่องของ index ไหน (ถ้าไม่อยู่เลยให้แสดงผลเป็น -1)

ถ้าใช้การไล่หาจากสมาชิกอาร์เรย์ตัวแรก (Big O คือ O(n)) จะเสียเวลาได้ ในเมื่อเรารู้ว่าอาร์เรย์นี้เรียงสมาชิกไว้แล้ว ก็ควรใช้ความรู้นี้ให้เป็นประโยชน์ โดยดูสมาชิกตัวกลางของอาร์เรย์ ถ้ามันน้อยกว่า x แสดงว่า เหลือครึ่งขวาของ array ให้หาอย่างเดียว ถ้ามันมากกว่า x ก็จะเหลือครึ่งซ้ายให้หาอย่างเดียว ทำอย่างนี้จะหาได้เร็วขึ้นมาก วิธีนี้เรียกว่า Binary search

มาดูตัวโปรแกรมกัน

```
1:     int binarySearch (int[] a, int x) {
2:       int left = 0, right = a.length - 1;
3:       while (left <=right) {
4:         int mid = (left + right)/2;
5:         if (a[mid] < x ) {
6:           left = mid + 1;
7:         } else if (a[mid] > x) {
8:           right = mid - 1;
9:         } else {
10:          return mid;
11:        }
12:      }
13:      return -1; // ถึงตรงนี้แสดงว่าไม่เจอเลย
14:    }
```

จะเห็นได้ว่าในแต่ละครั้งที่ลูป ขนาดของอาร์เรย์ที่เราต้องตรวจสอบจะลดไปครึ่งหนึ่งทุกครั้ง ฉะนั้นค่า Big O คือ O(log(n)) การหาข้อมูลด้วยวิธีนี้ใช้ได้ดีกับตารางที่ไม่ค่อยมีการเปลี่ยนแปลง (ไม่งั้นต้องจัดเรียงข้อมูลบ่อยเกินไป)

ตัวอย่างที่ 1-6

ลองดูโปรแกรมที่หาตัวหารร่วมมากของ n กับ m

```
1:     long gcd (long m , long n) {
2:       while (n!=0) {
3:         long rem = m%n;
4:         m = n;
5:         n = rem;
6:       }
7:       return m;
8:     }
```

นี่คือการหาเศษไปเรื่อย ๆ จนเศษเป็น 0 ค่าเศษที่เกิดก่อน 0 จะเป็นคำตอบ (รูป 1.10 แสดงขั้นตอนการหา gcd (2564, 1988)) จากโปรแกรม ค่าที่กำหนดว่าจะมีการวนลูปหรือไม่ ก็คือค่า n ซึ่งมาจาก rem (เศษ) อีกที เพราะฉะนั้น การลดลงของเศษจะบอกว่า Big O เป็นเท่าไร แต่ถ้าเราดูจากรูป 1.10 จะเห็นว่า ในการเข้าลูปครั้งหนึ่งไปการเข้าลูปถัดไป ค่าเศษลดลงอย่างไม่เป็นสัดส่วนเลย (ช่วงแรกจะลดลงมาก จาก 576 เป็น 260 และ 56 แต่ช่วงหลังจะไม่ค่อยลด จาก 56 ไป 36 และ 20) ในการหาค่า Big O ของตัวอย่างนี้เราต้องใช้ประโยชน์จากทฤษฎีต่อไปนี้

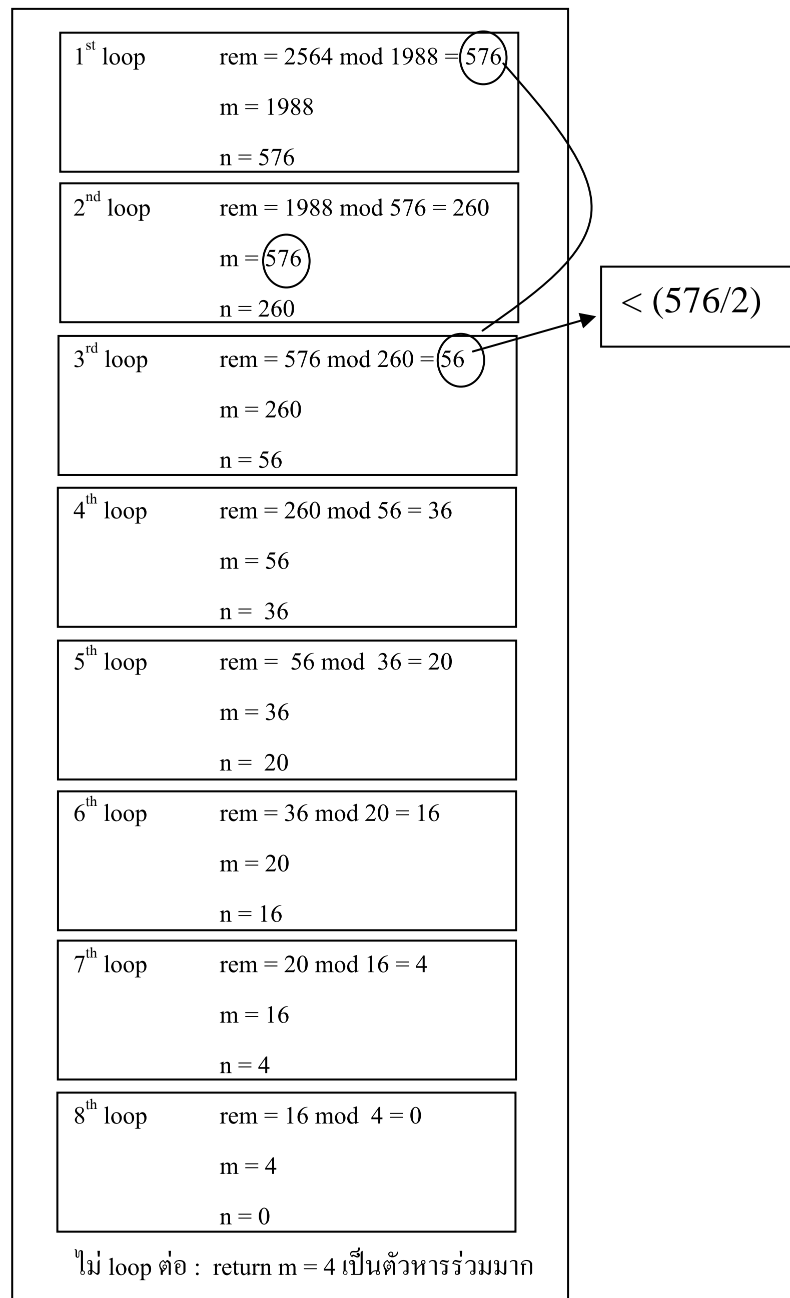
นิยามที่ 1-4

ถ้า $M > N$ แล้ว $M \bmod N < M/2$

เราสามารถพิสูจน์นิยามนี้ได้ไม่ยาก

- ถ้า $N \le M/2$: เนื่องจากเศษต้องน้อยกว่า N เพราะฉะนั้น เศษก็มีค่าน้อยกว่า M/2 โดยปริยาย
- ถ้า $N > M/2$: M หาร N จะได้ 1 กับเศษ M-N นั่นคือ 1 กับเศษ M – (> M/2) เพราะฉะนั้น เศษจะน้อยกว่า M/2 โดยปริยาย

จากนิยามนี้ เราดูรูป 1.10 อีกที จะเห็นว่า เศษของลูป ที่ 1 ถูกนำมาใช้เป็น m เริ่มต้น(ก่อนหาเศษ)ของลูปที่ 3 (และเศษของลูปที่ x จะถูกนำมาใช้เป็น m เริ่มต้นของลูปที่ (x+2)) เพราะฉะนั้น เศษจากลูปที่ (x+2) จะต้องมีค่าน้อยกว่าครึ่งหนึ่งของเศษจากลูปที่ x นี่แสดงว่าเมื่อลูปผ่านไปสองครั้ง ขนาดของ เศษลดลงเกินหรือเท่ากับครึ่งหนึ่งแน่ ๆ นี่ทำให้เราประมาณจำนวนครั้งของการลูป ได้ คือ 2 log n ครั้ง ซึ่งหมายความว่า Big O คือ O (log n)



รูป 1.10 ขั้นตอนการหาตัวหารร่วมมากของ 2564 กับ 1988

ตัวอย่างที่ 1-7

การยกกำลังเลข อันนี้เกี่ยวกับ log n ยังไง? ตามปกติ ถ้าให้หา x^n เราก็คงจะเอา x มาคูณกัน n-1 ครั้ง แต่ถ้าให้ดูดี เราสามารถแก้ปัญหานี้ในรูปแบบของการแบ่งแยกและเอาชนะได้ ดังนี้

- ถ้า n เป็นเลขคู่ ก็แบ่งเป็น $x^{n/2}$ สองตัวแล้วเอาผลลัพธ์มาคูณกัน $(x^{n/2} * x^{n/2}) = x^n$
- ถ้า n เป็นเลขคี่ ก็แบ่งเป็น $x^{(n-1)/2}$ สองตัวกับ x หนึ่งตัวแล้วเอาผลลัพธ์มาคูณกันหมด $(x^{(n-1)/2} * x^{(n-1)/2} * x) = x^n$

จะสังเกตได้ว่า ปัญหาถูกแบ่งเป็น 2 ครึ่ง (ประมาณ) เท่ากัน โดยใช้การคูณอย่างมากแค่ 2 ครั้ง นี่แสดงว่าต้องใช้การคูณอย่างมากแค่ 2 log n ครั้ง (ค่า Big O คือ O (log n))

ตัวโปรแกรมเป็นดังนี้

```
1:    long power (long x, int n){
2:        if (n==0)
3:          return 1;
4:        if (isEven (n))
5:          return power (x*x, n/2);
6:        else
7:          return power (x*x, n/2)*x;
8:    }
```

สำหรับเรื่องของ log นี้มีนิยามที่สำคัญคือ

นิยามที่ 1-5

$\log^k n = O(n)$ เมื่อ k เป็นค่าคงที่

(นิยามนี้บอกเราว่า logarithm function มีอัตราการเพิ่มต่ำมาก)

นิยามที่ 1-6

ฟังก์ชั่น $f(n) = \log_a n$ มีค่า big O เป็น $O(\log_b n)$ (a และ b เป็นจำนวนบวกที่ไม่ใช่ 1)

(สรุปคือ log function มี growth rate เท่ากัน)

พิสูจน์นิยาม 1-6:

ให้ $\log_a n = x$ และ $\log_b n = y$

$\therefore a^x = n, b^y = n$

$\therefore x \ln a = y \ln b = \ln n$

$x \ln a = y \ln b$

$\log_a n * \ln a = \log_b n * \ln b$

$$\log_a n = \log_b n * \frac{\ln b}{\ln a} = (\log_b n) * c$$

$\therefore \log_a n = O(\log_b n)$

รูป 1.11 แสดง runtime ของ big O ต่างๆ จากน้อยไปมาก

runtime
c
log n
$\log^k n$
n
n log n
n^2
n^3
2^n

รูป 1.11 runtime ของ big O

นิยามอื่น ๆ นอกเหนือจาก Big O

นิยามที่ 1-7

Big Omega (Ω)

T(N) = Ω (g(N)) ถ้ามีค่าคงที่ C และ N_0 ซึ่ง

T(N) >= C g(N) โดย N>=N_0

จากนิยาม ถ้า $f(N) = \Omega (N^2)$ แล้วล่ะก็ $f(N) = \Omega (N) = \Omega (N^{1/2})$
แต่เราควรเลือกอันที่ค่าใกล้ความจริงที่สุด

นิยามที่ 1-8

Big Theta (Θ)

$T(N) = \Theta (h(N))$ ก็ต่อเมื่อ $T(N) = O(h(N))$ และ $T(N) = \Omega (h(N))$

(คือมี c1, c2, N_0 ที่ $c1*h(N) <= T(N) <= c2*h(N)$ สำหรับ $N >= N_0$

นิยามที่ 1-9

small O

$T(N) = o(p(N))$ ถ้า $T(N) = O(p(N))$ แต่ $T(N) \neq \Theta (p(N))$

- $T(N) = O(f(N))$ มีความหมายเหมือนกับ $f(N) = \Omega (T(N))$ เราสามารถเรียกได้ว่า f(N) เป็น “upper bound” ของ T(N) และ T(N) ก็ถือได้ว่าเป็น lower bound ของ f(N)
- $f(N) = N^2$ และ $g(N) = 2N^2$ มีค่า Big O และ Big Ω เท่ากัน นั่นคือ $f(N) = \Theta (g(N))$
- $f(N) = N^2$ มี Big O ได้หลายค่า (เช่น $O(N^3)$, $O(N^4)$) แต่ค่าที่ดีที่สุดคือ $O(N^2)$ เราสามารถใช้สมการ $f(N) = \Theta (N^2)$ เป็นการบอกว่า ค่า Big O -คือ N^2 นี้เป็น Big O ที่ดีที่สุด

จากข้อสังเกตนี้ทำให้เราได้อีกหนึ่งนิยามคือ

นิยามที่ 1-10

ถ้า T(N) คือ Polynomial degree k แล้ว

$$T(N) = \Theta (N^k)$$

จากนิยามนี้ ถ้า $T(N) = 5N^4 + 4N^3 + N$ เราจะได้ว่า $T(N) = \Theta(N^4)$

Best case Worst case และ Average case

เวลาในการ run แบบ worst case คือ เวลาเมื่อการ run ต้องใช้จำนวนขั้นตอนมากที่สุด ส่วน best case คือ เวลาในการ run ที่น้อยที่สุดที่เป็นไปได้ สองอย่างนี้คงไม่ก่อให้เกิดปัญหาอะไร แต่ว่าแล้ว average case (เวลาเฉลี่ย) ล่ะจะหาอย่างไร ถ้าเราคิดแบบง่าย ๆ ก็จะสามารถหาได้จากการ

1. ดูว่ามีกี่ input (ข้อมูลเข้า) เข้าโปรแกรม
2. สำหรับแต่ละ input ดูว่าโปรแกรมใช้กี่ขั้นตอน (unit time) ในการ run
3. ค่าเฉลี่ย (average case running time) = ผลรวมของจำนวน unit time จากทุก input / จำนวน input

แต่การหาค่าเฉลี่ยอย่างง่ายนี้ ตั้งอยู่บนสมมติฐานว่า input แต่ละแบบมีอัตราการเกิดขึ้นเท่ากัน ถ้าจะให้ไม่ต้องมีสมมติฐานนี้ เราต้องเอาค่าความน่าจะเป็นในการเกิดของ input แบบต่าง ๆ เข้ามาคิดด้วยนั่นคือ

Average case running time = Σi ความน่าจะเป็นของ $input_i$ * ค่า unit time เมื่อใช้ $input_i$

โดย ความน่าจะเป็นมีค่าตั้งแต่ 0 ถึง 1 และความน่าจะเป็นทั้งหมดบวกกันต้องได้ 1 แต่วิธีนี้ก็ยังตั้งอยู่บนสมมติฐานที่ว่า ค่า ความน่าจะเป็นทั้งหมดต้องหาได้

ตัวอย่างที่ 1-8

ถ้าเราต้องการหา x ในอาร์เรย์ขนาด n

Best case เกิดขึ้นเมื่อเจอ x ในช่องแรกของอาร์เรย์เลย ส่วน Worst case เกิดขึ้นเมื่อ x อยู่ช่องสุดท้ายในอาร์เรย์ (หรืออาจไม่อยู่เลย) Average case อาจเป็นได้หลายอย่าง ถ้าเราสมมติว่า x อาจอยู่ในช่องไหนของอาร์เรย์ก็ได้ (คือแต่ละช่องมีโอกาสเท่ากัน)

- โอกาสที่ x จะอยู่ในช่องแรก คือ 1/n
- โอกาสที่ x จะอยู่ในช่องที่ i ใดๆ คือ 1/n

เพราะฉะนั้น Average Case running time = 1/n * (จำนวนขั้นตอนที่ใช้ในการหาเมื่อ x อยู่ช่องที่ 1) + 1/n * (จำนวนขั้นตอนที่ใช้ในการหาเมื่อ x อยู่ช่องที่ 2) + ... + 1/n * (จำนวนขั้นตอนที่ใช้ในการหาเมื่อ x อยู่ช่องที่ n หรืออาจไม่อยู่เลย)

= (1 + 2 +… + n) / n = (n+1)/2

สรุปได้ว่าจะได้เวลาของ average case เป็น (n+1)/2 ซึ่งมี big O เป็น O(n) ซึ่งในที่นี้เป็นค่า big O ที่เท่ากับค่า big O ของ worst case แต่ถ้าค่าความน่าจะเป็นเปลี่ยนไป เช่น

- ให้ความน่าจะเป็นที่จะเจอ x ในช่องแรก = $\frac{1}{4}$
- ความน่าจะเป็นที่จะเจอ x ในช่องที่ 2 = $\frac{1}{8}$

เพราะฉะนั้น ความน่าจะเป็นที่จะเจอ x ในช่องหนึ่งนอกจากช่องแรกและช่องที่ 2

$$= (1 - \frac{1}{4} - \frac{1}{8}) / (n-2)$$

$$= \frac{5}{8(n-2)}$$

ค่าเฉลี่ยของ running time จะกลายเป็น

$$= \frac{1}{4} * 1 + \frac{1}{8} * 2 + \frac{5}{8(n-2)} * 3 + \frac{5}{8(n-2)} * 4 + ... + \frac{5}{8(n-2)} * n$$

$$= \frac{1}{2} + (3 + 4 + ... + n) * \frac{5}{8(n-2)}$$

แบบฝึกหัด

1. ให้ $f(n) = 7n*\log_2 n$ และ $g(n) = n^2$ จงหาค่า n0 (ซึ่ง <= n) ซึ่งทำให้ f(n) <g(n)
2. จงพิสูจน์ว่า ถ้า $T_1(N) = O(f(N))$ และ $T_2(N) = O(g(N))$ แล้ว $T_1(N) + T_2(N) = \max(O(f(N), O(g(N)))$

3. จงพิสูจน์ว่า T(N) = O(f(N)) ก็ต่อเมื่อ f(N) = Ω (T(N))
4. จงแสดงว่า $3^{(n+1)}$ เป็น $O(3^n)$
5. จงแสดงว่า n = O(n log n)
6. จงพิสูจน์ว่า $\log^k n = o(n)$ เมื่อ k เป็นค่าคงที่
7. ถ้า f(n) = 3n เมื่อ n เป็นจำนวนคู่ และ $f(n)=n^2$ เมื่อ n เป็นจำนวนคี่ จงหาค่า big O ของ f(n)
8. ถ้ามีตัวเลขอยู่ n ตัว จงเขียนโปรแกรมเพื่อหาค่าที่มากที่สุดและน้อยที่สุดโดยใช้การเปรียบเทียบไม่เกิน 3n/2 ครั้ง
9. สมมติว่ามีโปรแกรมอยู่สองโปรแกรม โปรแกรมที่หนึ่งมี worst case running time = 230 n $\log_2 n$ ส่วนโปรแกรมที่สองมี worst case running time = n^2 จงหาว่าเมื่อ n เป็นเท่าไร โปรแกรมที่สองจึงจะมี worst case running time มากกว่าโปรแกรมแรก
10. ให้เรามีสมการ $p(x) = \sum_{i=0}^{n} a_i x^i$ เราสามารถนำมาเขียนใหม่เป็น $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + xa_n)\ldots)))$ ซึ่งวิธีการนี้เรียกว่าวิธีการของ Horner จงเขียนโปรแกรมหา p(x) รวมทั้งบอกค่า big O ด้วย
11. จงเขียนโปรแกรมเพื่อคูณเลขสองจำนวน โดยให้ใช้เครื่องหมายบวกเท่านั้น โปรแกรมนี้มีค่า big O เป็นเท่าไร
12. จงพิสูจน์ว่า $\sum_{i=1}^{n}(2i-1) = n^2$
13. จงแสดงว่า $\sum_{i=1}^{n}\lceil \log_2 i \rceil = \Theta(n \log n)$
14. สมมติว่าห้องสมุด"โคตรโหด" ปรับเราเมื่อคืนหนังสือสายวันแรกเป็นจำนวนเงิน 2 บาท ถ้าเราไม่คืน ค่าปรับจะยกกำลังสองในวันต่อไปและต่อไปเรื่อยๆ ถามว่าในวันที่ n ค่าปรับจะเป็นเท่าไร และจะใช้เวลากี่วันค่าปรับจึงจะเท่ากับ x บาท
15. จงดัดแปลงโปรแกรมที่หา maximum subsequence sum ให้เก็บค่า index ของ maximum subsequence sum ไว้ด้วยเพื่อพร้อมพิมพ์ออกมาทุกเวลา
16. จงเขียนโปรแกรมเพื่อหา minimum subsequence sum โปรแกรมนี้มีค่า big O เป็นเท่าไร
17. จงเขียนโปรแกรมเพื่อหา maximum subsequence product โปรแกรมนี้มีค่า big O เป็นเท่าไร

18. จงเขียนโปรแกรมเพื่อหาว่าจำนวนเต็มบวก n เป็นจำนวนเฉพาะหรือไม่ โปรแกรมนี้มีค่า big O เป็นเท่าไร
19. ถ้าใน 0.5 วินาที โปรแกรมหนึ่งสามารถจัดการ input ได้หนึ่งร้อยห้าสิบตัว จงหาว่าในห้านาทีโปรแกรมนี้จะสามารถจัดการกับ input ได้กี่ตัว เมื่อโปรแกรมนี้เป็น O(n log n) และ $O(n^2)$ ตามลำดับ
20. จงหา big O จากโค้ดชุดต่อไปนี้

```
1:    sum=0;
2:    for(i=0;i<n;i++)
3:        for(j=0;j<n*n;j++)
4:              sum++;
```

```
1:    sum=0;
2:    for(i=0;i<n;i++)
3:         for(j=0;j<i;j++)
4:             sum++;
```

```
1:    sum=0;
2:         for(i=0;i<n;i++)
3:              for(j=0;j<i*i;j++)
4:                   for(k=0;k<j;k++)
5:                       sum++;
```

```
1:    sum=0;
2:         for(i=1;i<n;i++)
3:              for(j=1;j<i*i;j++)
4:                   if(j%i = =0)
5:                        for(k=0;k<j;k++)
6:                             sum++;
```