บทที่ 1

อาร์เรย์และพอยน์เตอร์

ในบทนี้เราจะพิจารณาโครงสร้างข้อมูลพื้นฐานสำหรับจัดเก็บและประมวลผลข้อมูลจำนวนมากที่เรียกว่า*อาร์เรย์* (array) รวมไปถึงข้อมูลประเภท*พอยน์เตอร์* (pointer) ซึ่งเก็บตำแหน่งภายในหน่วยความจำ โดยเราจะเริ่มพิจารณาแนวคิดของโครงสร้างข้อมูลและชนิดข้อมูลดังกล่าวโดยไม่ขึ้นกับภาษาโปรแกรมที่ใช้ จากนั้นเราจะศึกษาวิธีการเขียนในภาษา C++ และศึกษาความสัมพันธ์ระหว่างพอยน์เตอร์และอาร์เรย์ซึ่งเป็นคุณลักษณะเฉพาะที่มีในภาษาตระกูล C และ C++

ในบทนี้ เราจะเริ่มศึกษาการวิเคราะห์เวลาการทำงานของอัลกอริทึมอย่างง่าย ก่อนที่จะไปพิจารณาอย่างเป็นทางการในบทที่ 2

1.1 อาร์เรย์

อาร์เรย์เป็นโครงสร้างข้อมูลที่เก็บกลุ่มของข้อมูลเป็นรายการ โดยที่ข้อมูลแต่ละตัวจะถูกเก็บต่อเนื่องกันในหน่วยความจำ และถูกอ้างถึงโดยใช้ดัชนี (index) ตัวอย่างง่าย ๆ ของอาร์เรย์คือรายการข้อมูลด้านล่างนี้

$$2, 3, 5, 7, 11, 13, 17, 19, 23$$

ถ้าเราเรียกรายการดังกล่าวว่ารายการ A และอ้างถึงข้อมูลแต่ละตัวด้วยดัชนีที่เริ่มต้นด้วย 0 ข้อมูลแต่ละตัวในรายการจะถูกอ้างถึงได้ดังตารางในรูปที่ 1.1

▷ **คำถาม 1.1** การคำนวณค่า
จงหาผลลัพธ์ของนิพจน์เหล่านี้ (1) $A[4]$, (2) $A[7]$, (3) $A[A[0]]$, (4) $A[A[A[0]]]$, (5) $A[200]$ ◁

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
2	3	5	7	11	13	17	19	23

รูปที่ 1.1: การอ้างถึงข้อมูลแต่ละตัวในอาร์เรย์ A

TODO: ใส่รูป

รูปที่ 1.2: การเก็บข้อมูลของอาร์เรย์ในหน่วยความจำ

เฉลย:
(1) 11, (2) 19, (3) 5, (4) 13, (5) ไม่มีค่า (ดูอธิบายเพิ่มเติม)

การหาคำตอบของคำถามที่ (3) นั้น จำเป็นต้องเข้าใจขั้นตอนการคำนวณค่าของนิพจน์ เราต้องการหาค่า $A[A[0]]$ ดังนั้นเราต้องหาค่า $A[0]$ ก่อน เมื่อพิจารณาในอาร์เรย์เราพบว่า $A[0]$ คือ 2 ดังนั้น จากนั้นเราจึงพิจารณาข้อมูล $A[2]$ ในอารเรย์ ซึ่งจะได้ค่า 5

ในการทำงานจริง อาร์เรย์จะเก็บในหน่วยความจำที่ต่อเนื่องกัน และมักจะมีขอบเขตที่จำกัดและต้องระบุเมื่อเริ่มใช้ เช่น อาร์เรย์จำนวน 100 ช่อง หรือ 100000 ช่องเป็นต้น รูปที่ 1.2 แสดงตัวอย่างของการเก็บข้อมูลของอาร์เรย์ในหน่วยความจำ

คำถามที่ (5) เป็นการอ้างถึงข้อมูลที่อยู่นอกขอบเขตของอาร์เรย์ ซึ่งผลลัพธ์ที่ได้จะขึ้นกับภาษาโปรแกรมที่ใช้ สำหรับภาษา C หรือ C++ ผลลัพธ์ที่ได้จะขึ้นกับข้อมูลในหน่วยความจำในตำแหน่งที่ $A[200]$ ควรจะอยู่ เราจะได้ศึกษารายละเอียดนี้ต่อไป อย่างไรก็ตาม ปกติแล้ว ในการใช้งานอาร์เรย์ เราจะไม่อ้างถึงข้อมูลที่อยู่นอกขอบเขตของอาร์เรย์

การระบุดัชนีของข้อมูลในอาร์เรย์ในหนังสือเล่มนี้จะอ้างอิงจากภาษาตระกูลภาษา C นั่นคือเริ่มต้นที่ 0 สำหรับบางภาษา เราสามารถระบุค่าเริ่มต้นของดัชนีได้และมักเริ่มที่ 1 เช่นภาษาปาสคาล (Pascal) เป็นต้น อย่างไรก็ตาม แนวคิดในการพัฒนาโปรแกรมนั้นจะไม่ต่างกัน

เมื่อเราสามารถอ้างถึงข้อมูลได้ด้วยดัชนี เราสามารถใช้ตัวแปรเพื่อแทนค่าดัชนีของข้อมูลที่เราต้องการใช้งานได้ ความสามารถนี้ทำให้เราสามารถเขียนโปรแกรมที่มีลักษณะดังด้านล่างได้

ให้ $x \leftarrow 0$ (A1.1)
พิจารณา ตัวแปร $i \leftarrow 0, 1, \ldots, 8$
  ให้ $x \leftarrow x + A[i]$

▷ **คำถาม 1.2**
อัลกอริทึมดังกล่าวคำนวณค่าบางอย่างในตัวแปร x ค่านั้นคืออะไร? ◁

เฉลย:
ผลรวมของข้อมูลทั้งหมดในอาร์เรย์ A

สังเกตว่าอัลกอริทึม A1.1 เขียนให้ทำงานกับอาร์เรย์ A ที่มีดัชนีมากที่สุดคือ 8 เท่านั้น ในการพัฒนาอัลกอริทึมทั่วไปเรามักเขียนให้ทำงานได้กับข้อมูลทั่วไป ซึ่งในกรณีนี้ การจะปรับให้ทำงานได้กับอาร์เรย์ใด ๆ เราจะต้องระบุขนาดของอาร์เรย์ด้วย เราสามารถเขียนอัลกอริทึมดังกล่าวโดยระบุพารามิเตอร์ให้ชัดเจนขึ้นได้ดังด้านล่าง

คำนวณค่าบางอย่างของอาร์เรย์ A ที่มีข้อมูลจำนวน n ตัว (A1.2)

  ให้ $x \leftarrow 0$
  พิจารณา ตัวแปร $i \leftarrow 0, 1, \ldots, n-1$
    ให้ $x \leftarrow x + A[i]$
  คืนค่า x เป็นคำตอบ

1.1.1 เวลาที่ใช้ในการทำงาน

ค่าพารามิเตอร์ n ที่เราส่งให้กับโปรแกรมย่อย ระบุจำนวนรอบของการทำงาน ซึ่งจะเป็นตัวกำหนดเวลาที่โปรแกรมย่อยใช้ในการทำงานด้วย อย่างไรก็ตาม เพียงแค่พิจารณาโปรแกรมย่อยดังกล่าว เราไม่สามารถระบุเวลาจริง ๆ ที่โปรแกรมย่อยจะทำงานได้เนื่องจากเราไม่ทราบปัจจัยหลาย ๆ อย่าง

▷ **คำถาม 1.3** เวลาการทำงานจริงบนคอมพิวเตอร์
ปัจจัยอะไรบ้างที่กำหนดเวลาทำงานบนคอมพิวเตอร์จริง ๆ ของโปรแกรมย่อยข้างต้น ◁

เฉลย:
เวลาในการทำงานจริง ขึ้นกับ (1) โปรแกรมภาษาคอมพิวเตอร์ที่เขียนจากโปรแกรมย่อย (2) คอมไพเลอร์ที่ใช้ (3) เครื่องคอมพิวเตอร์ที่นำโปรแกรมไปทำงาน และสถานะของเครื่องในขณะที่โปรแกรมทำงาน

สังเกตว่าการพิจารณาแค่อัลกอริทึมเพียงอย่างเดียว หรือกระทั่งจะพิจารณาโปรแกรมในภาษาเครื่องที่ถูกคอมไพล์แล้วร่วมด้วย ก็ไม่สามารถทำให้เราระบุเวลาการทำงานบนคอมพิวเตอร์จริงได้อย่างแม่นยำ ยิ่งในปัจจุบันที่คอมพิวเตอร์สามารถทำงานหลาย ๆ งานในเวลาเดียวกัน การทำนายเวลาการทำงานจริงยิ่งกระทำได้ยากขึ้นด้วย

อย่างไรก็ตาม แม้การระบุเวลาการทำงานจริง ๆ ทำได้ยาก การทำนายเวลาการทำงานของอัลกอริทึมก่อนที่จะนำไปพัฒนาเป็นโปรแกรมก็ยังเป็นสิ่งจำเป็นมาก เนื่องจากในหลาย ๆ เราสามารถเลือกใช้อัลกอริทึมได้หลากหลาย และอัลกอริทึมเหล่านั้นก็มีความซับซ้อนในการนำไปพัฒนาเป็นโปรแกรมที่แตกต่างกัน โปรแกรมเมอร์จึงต้องเลือกใช้อัลกอริทึมให้เหมาะสม นั่นคือเป็นอัลกอริทึมที่เมื่อนำไปพัฒนาแล้ว มีประสิทธิภาพพอ (ทำงานได้ทันเวลา) และมีความซับซ้อนในการเขียนในระดับที่โปรแกรมเมอร์สามารถจัดการได้ การเลือกนำอัลกอริทึมที่ทราบว่ามีประสิทธิภาพดีที่สุดไปพัฒนานั้น อาจไม่ใช่ทางเลือกที่ดีที่สุดก็เป็นได้

ดังนั้น เราจะพยายามวิเคราะห์เวลาการทำงานของโปรแกรมย่อย ที่อยู่ในรูปของโปรแกรมลำลองด้านบน ให้ละเอียดเท่าที่เราพอจะทำได้ แน่นอนเราจำเป็นต้องเพิ่มข้อสมมติหลายอย่างเพื่อให้การวิเคราะห์เป็นไปได้

ข้อสมมติข้อแรก (ที่เราจะใช้ตลอดในหนังสือเล่มนี้) คือ เราจะสมมติว่าคอมพิวเตอร์นั้นทำงานทีละคำสั่ง นั่นคือไม่ใช่คอมพิวเตอร์แบบขนาน หรือเป็นคอมพิวเตอร์ที่มีหน่วยประมวลผลหลายตัวทำงานพร้อมกัน[1]

ถ้าพิจารณาต่อไป เราจะพบว่าโปรแกรมย่อย A1.2 ทำงานโดยใช้เวลาในการทำงานที่แปรผันตามค่าพารามิเตอร์ n เพื่อจะให้เราสามารถวิเคราะห์เวลาการทำงานออกมาได้ เราจะสมมติว่าคอมพิวเตอร์เมื่อทำงานตามโปรแกรมดังกล่าวใช้เวลา 1 หน่วยในการประมวลผลคำสั่งแต่ละบรรทัด เราจะสามารถคำนวณเวลาที่โปรแกรมดังกล่าวใช้โดยพิจารณาจำนวนครั้งที่คำสั่งในแต่ละบรรทัดทำงาน ดังด้านล่าง

[1] TODO: ระบุว่าถึงจะเป็นกรณีดังกล่าว การวิเคราะห์ก็ยังเป็นไปได้

```
    ให้ x ← 0      ▷ ▷ ▷ ทำงาน 1 ครั้ง                                              (A1.3)
    พิจารณา ตัวแปร i ← 0, 1, ..., n − 1      ▷ ▷ ▷ ทำงาน n ครั้ง
        ให้ x ← x + A[i]      ▷ ▷ ▷ ทำงาน n ครั้ง
    คืนค่า x เป็นคำตอบ      ▷ ▷ ▷ ทำงาน 1 ครั้ง
```

ดังนั้นเราจะได้ว่าเวลารวมคือ $2n+2$ หน่วย คำถามที่ตามมาก็คือ ผลลัพธ์จากการวิเคราะห์ดังกล่าวมีความแม่นยำและสามารถนำไปใช้วิเคราะห์และตัดสินใจต่อไปได้เพียงใด เราจะพิจารณาผลจากการสมมติและความถูกต้องที่ใช้ได้ในบทที่ 2

1.1.2 การประมวลผลรายการด้วยอาร์เรย์

ในส่วนนี้เราจะพัฒนาโปรแกรมลำลองเพื่อประมวลผลข้อมูลในรายการที่เก็บในอาร์เรย์ พร้อมกับวิเคราะห์เวลาการทำงาน

▷ **คำถาม 1.4**
สมมติว่าเรามีรายการของข้อมูล ลองนึกตัวอย่างการประมวลผลที่เราสามารถกระทำกับข้อมูลในรายการนี้ ◁

ก่อนที่เราจะประมวลผลได้ เราต้องพิจารณาวิธีการจัดเก็บข้อมูลแบบรายการลงในอาร์เรย์ก่อน สังเกตว่าโครงสร้างข้อมูลแบบอาร์เรย์มีลักษณะเป็นรายการอยู่แล้ว อย่างไรก็ตามในการจัดการกับรายการที่มีจำนวนข้อมูลเปลี่ยนแปลงได้ การใช้อาร์เรย์เพียงอย่างเดียวนั้นไม่เพียงพอ

▷ **คำถาม 1.5**
อะไรคือสิ่งที่ขาดหายไป ถ้าเราใช้แค่อาร์เรย์ในการจัดเก็บรายการที่จำนวนข้อมูลในรายการเปลี่ยนแปลงได้ ◁

ดังนั้น เราจะใช้ตัวแปรอีกหนึ่งตัวในการเก็บจำนวนข้อมูลที่มีในอาร์เรย์ โปรแกรมลำลองที่เราจะพัฒนาจะเปลี่ยนค่าของตัวแปรนี้โดยตรงเพื่อปรับให้มีค่าที่ถูกต้องภายหลังการประมวลผล ในการพัฒนาโปรแกรมลำลองให้เป็นโปรแกรมภาษา C++ การทำงานดังกล่าวจะต้องใช้การส่งรับพารามิเตอร์เป็นพอยน์เตอร์หรือส่งแบบ pass by reference ซึ่งเราจะได้พิจารณาในส่วน 1.3 นอกจากนี้ในบทที่ **??** เราจะได้ศึกษาวิธีการที่จะ “ประกอบรวม” อาร์เรย์และตัวแปรที่เก็บจำนวนข้อมูลที่อยู่ในอาร์เรย์เข้าด้วยกัน เพื่อสร้างเป็นชนิดข้อมูลใหม่ที่นำไปใช้งานได้สะดวกต่อไป

เราจะพิจารณาการประมวลผลกับอาร์เรย์ในรูปแบบต่าง ๆ ดังนี้ (1) การค้นข้อมูลในรายการ, (2) การเพิ่มข้อมูลลงไปตอนท้ายของรายการ, (3) การลบข้อมูลในรายการ, และ (4) การแทรกข้อมูลในรายการ

การค้นข้อมูล

สำหรับการค้นข้อมูลในรายการ เป้าหมายของการทำงานคือทราบว่ามีข้อมูลที่เราต้องการหาหรือไม่ และถ้ามีอยู่ที่ตำแหน่งใด ในกรณีนี้เราจะต้องพิจารณาข้อมูลทุกตัวในรายการ โปรแกรมลำลองมีลักษณะไม่ต่างจากที่เราเคยเขียนเท่าใดนัก

```
ค้นหาข้อมูล x ในอาร์เรย์ A ที่มีข้อมูลจำนวน n ตัว                                      (A1.4)
    พิจารณา ตัวแปร i ← 0, 1, ..., n − 1
```

```
    ถ้า A[i] = x
        คืนค่า i เป็นผลลัพธ์
ตอบว่าไม่พบค่าที่ต้องการ
```

ในการพัฒนาโปรแกรมจริง ๆ เราจะต้องจัดการในกรณีที่จะต้องตอบว่าไม่พบค่าที่ต้องการให้ชัดเจนกว่านี้ แต่ในขณะนี้เราจะสมมติว่าโปรแกรมย่อยสามารถตอบแบบนี้ได้

▷ **คำถาม 1.6**
ในกรณีของโปรแกรมย่อยสำหรับหาผลรวม เราพบว่าโปรแกรมทำงานในเวลาที่แปรผันกับค่า n เสมอ เป็นไปได้หรือไม่ ที่โปรแกรมย่อยสำหรับจะทำงานโดยวนรอบเป็นจำนวนครั้งที่น้อยกว่าค่า n มาก? และเป็นในกรณีใด? ◁

▷ **คำถาม 1.7**
สำหรับอาร์เรย์ที่มีข้อมูล n ตัว เมื่อใดที่โปรแกรมย่อยจะทำงานโดยวนรอบมากที่สุด ◁

โปรแกรมย่อยข้างต้นอาจจะทำงานได้รวดเร็วมาก ถ้าข้อมูลที่ต้องการค้นหาอยู่ตอนต้นของอาร์เรย์ โปรแกรมย่อยลักษณะนี้เป็นตัวอย่างที่ดีของโปรแกรมย่อยที่เวลาการทำงานขึ้นกับข้อมูลป้อนเข้า ทำให้ในการวิเคราะห์เวลาการทำงานนั้น เราจำเป็นจะต้องพิจารณาข้อมูลป้อนเข้าด้วย อย่างไรก็ตามเราไม่สามารถที่จะวิเคราะห์เวลาการทำงานของโปรแกรมลำลองบนข้อมูลป้อนเข้าทุกรูปแบบได้ เพราะว่าจำนวนของข้อมูลป้อนเข้านั้นมีไม่จำกัด

ในทางปฏิบัติแล้ว เราจึงจะแบ่งวิเคราะห์เวลาการทำงานเป็นกรณีย่อย ๆ สามกรณีคือ

- การวิเคราะห์ในกรณีที่ดีที่สุด (best-case analysis),
- การวิเคราะห์ในกรณีที่เลวร้ายที่สุด (worst-case analysis), และ
- การวิเคราะห์ในกรณีเฉลี่ย (average-case analysis)

สำหรับการวิเคราะห์ในกรณีเฉลี่ยนั้น เป็นการวิเคราะห์เชิงความน่าจะเป็น เราจำเป็นจะต้องนิยามลักษณะการกระจายของข้อมูลป้อนเข้าให้ชัดเจน จึงจะสามารถกระทำได้ เราจะได้ศึกษาตัวอย่างการวิเคราะห์นี้ในบทที่ 2 (TODO: เพิ่มหรือลบ) ในที่นี้เราจะสนใจเฉพาะการวิเคราะห์กรณีที่ดีที่สุด และการวิเคราะห์ในกรณีที่เลวร้ายที่สุดเท่านั้น

กรณีที่ดีที่สุดคือกรณีที่มีการวนรอบเพียงรอบเดียว นั่นคือเป็นกรณีที่ $A[0] = x$ สังเกตว่าถ้าเราสมมติให้การประมวลผลแต่ละบรรทัดใช้เวลา 1 หน่วย ในกรณีที่ดีที่สุด โปรแกรมลำลองดังกล่าวจะใช้เวลาทำงาน 4 หน่วย

กรณีที่เลวร้ายที่สุดเกิดขึ้นเมื่อไม่พบข้อมูลที่ต้องการหา สังเกตว่าโปรแกรมจะทำงานวนอยู่ที่สองบรรทัดแรกเป็นจำนวน n ครั้ง และคืนคำตอบ ดังนั้นโปรแกรมจะใช้เวลาทำงาน $2n + 1$ หน่วย

เช่นเดียวกับการวิเคราะห์อย่างง่ายในส่วนที่แล้ว เราจะพิจารณาแนวคิดการวิเคราะห์ทั้งสามแบบอย่างละเอียดในบทที่ 2

การเพิ่มข้อมูลลงไปท้ายรายการ

เราจะเพิ่มข้อมูลลงไปตอนท้ายของข้อมูลในอาร์เรย์ นั่นคือใส่ข้อมูลในอาร์เรย์ที่มีดัชนีมากกว่าดัชนีตัวสุดท้าย โปรแกรมลำลองที่น่าจะทำงานได้เขียนดังนี้

เพิ่มข้อมูล x ในตอนท้ายอาร์เรย์ A ที่มีข้อมูล n ตัว (A1.5)

$A[n] \leftarrow x$
$n \leftarrow n + 1$

อย่างไรก็ตาม ในการนำไปใช้จริง โปรแกรมลำลองดังกล่าวอาจจะทำให้เกิดข้อผิดพลาดขึ้นระหว่างการทำงานได้

▷ **คำถาม 1.8**
กรณีใดที่โปรแกรมลำลองข้างต้นอาจทำให้เกิดข้อผิดพลาดขึ้นระหว่างการทำงาน ◁

เฉลย:
จากที่เราได้เคยเกริ่นบ้างแล้วว่า ในการใช้งานอาร์เรย์ โดยมากจะต้องระบุขอบเขตหรือจำนวนข้อมูลมากที่สุดที่เก็บในอาร์เรย์ได้ ในกรณีของโปรแกรมลำลองนี้ถ้าเราเรียกใช้เมื่อ n มีขนาดมากกว่าหรือเท่ากับจำนวนข้อมูลที่อาร์เรย์เก็บได้ คำสั่ง $A[n] \leftarrow x$ ก็อาจจะเขียนข้อมูลลงในหน่วยความจำบริเวณที่อยู่นอกขอบเขตของอาร์เรย์ A ได้

ดังนั้นเพื่อความไม่ประมาท โปรแกรมย่อยควรจะต้องตรวจสอบขนาดของอาร์เรย์เพื่อป้องกันความผิดพลาดนี้ด้วย ในการเขียนต่อไปเราจะให้ $MAXLEN$ เป็นค่าคงที่แทนขนาดมากที่สุดของอาร์เรย์ A เราปรับแก้โปรแกรมย่อยได้ดังด้านล่าง

เพิ่มข้อมูล x ในตอนท้ายอาร์เรย์ A ที่มีข้อมูล n ตัว (แก้ไข) (A1.6)

ถ้า $n < MAXLEN$ แล้ว
  $A[n] \leftarrow x$
  $n \leftarrow n + 1$
ไม่เช่นนั้น
  รายงานว่าไม่สามารถเพิ่มข้อมูลได้

โปรแกรมย่อยนี้ ในการวิเคราะห์เวลาการทำงานมีสองกรณีให้เราพิจารณา สังเกตว่า จะใช้เวลาในการทำงานไม่เกิน 3 หน่วยไม่ว่าในกรณีใด

ในกรณีแรก (กรณีที่ $n < MAXLEN$) โปรแกรมย่อยจะใช้เวลาการทำงาน 3 หน่วย และในอีกกรณีจะใช้เวลาการทำงาน 2 หน่วย อย่างไรก็ตาม ผู้อ่านอย่าเพิ่งรีบสรุปว่ากรณีแรกทำงานจริง ๆ ได้เร็วกว่า เพราะว่าความแตกต่างนี้จริง ๆ แล้วเกิดจากข้อสมมติว่าการทำงานในทุกคำสั่งมีความเร็วเท่ากันคือ 1 หน่วย ดังนั้นประเด็นสำคัญของการวิเคราะห์นี้คือโปรแกรมย่อยนี้ทำงานในเวลาที่ไม่ขึ้นกับค่า n

การลบข้อมูลในรายการและการแทรกข้อมูลในรายการ

สำหรับการลบข้อมูลและแทรกข้อมูลในรายการ โปรแกรมย่อยที่ประมวลผลนั้นจะรับดัชนีของข้อมูลที่ต้องการลบ และดัชนีที่ต้องการให้นำข้อมูลไปแทรกต่อจากตำแหน่งนั้น ถ้าอาร์เรย์เริ่มต้นของเรามีข้อมูล 9 ตัว ดังด้านล่าง

ดัชนี	0	1	2	3	4	5	6	7	8	9
ข้อมูล	2	3	5	7	11	13	17	19	23	?

จำนวนข้อมูล $n = 9$

สังเกตว่าเราละข้อมูลที่ในอาร์เรย์ที่มีดัชนีอยู่นอกขอบเขตของข้อมูลในรายการไป (โดยแสดงด้วยเครื่องหมาย ?) การลบข้อมูลที่มีดัชนีเป็น 3 ให้ผลดังนี้

ดัชนี	0	1	2	3	4	5	6	7	8	9
ข้อมูล	2	3	5	11	13	17	19	23	?	?

จำนวนข้อมูล $n = 8$

จากอาร์เรย์ดังกล่าวส การแทรกข้อมูล 99 เข้าไปหลังข้อมูลที่มีดัชนีเป็น 1 ให้ผลดังนี้

ดัชนี	0	1	2	3	4	5	6	7	8	9
ข้อมูล	2	3	99	5	11	13	17	19	23	?

จำนวนข้อมูล $n = 9$

▷ **คำถาม 1.9**
การประมวลผลทั้งสองแบบมีกระบวนการหนึ่งที่ต้องดำเนินการคล้าย ๆ กัน คืออะไร ◁

การประมวลผลทั้งสองแบบนี้แสดงให้เห็นข้อจำกัดของการเก็บข้อมูลแบบรายการด้วยอาร์เรย์ (ยกเว้นจะมีเทคนิคพิเศษอื่น ๆ ประกอบ) ที่โครงสร้างข้อมูลเช่นลิงก์ลิสต์ที่เราจะพิจารณาในบทที่ **??** สามารถจัดการได้เป็นอย่างดี

โปรแกรมลำลองด้านล่างแสดงการลบข้อมูลที่มีดัชนี i ในรายการในอาร์เรย์

ลบข้อมูลที่มีดัชนี i ในรายการที่เก็บในอาร์เรย์ A ที่มีขนาด n (แก้ไข) (A1.7)

พิจารณาให้ ตัวแปร $i \leftarrow i, i+1, \ldots, n-1$

$A[i] \leftarrow A[i+1]$

$n \leftarrow n-1$

▷ **คำถาม 1.10**
โปรแกรมลำลอง A1.7 อาจทำให้เกิดความผิดพลาดระหว่างการทำงานได้ในบางกรณีเพราะว่าไม่ได้ตรวจสอบเงื่อนไขบางอย่าง เงื่อนไขนั้นคืออะไร? ◁

สังเกตว่าถ้าดัชนีอยู่นอกของเขตของรายการ หรือในกรณีที่ไม่มีข้อมูลโปรแกรมลำลองอาจเกิดปัญหาระหว่างทำงาน โปรแกรมลำลองด้านล่างเพิ่มเงื่อนไขในการตรวจสอบนี้ สำหรับในกรณีเช่นในตัวอย่างนี้ ภาษา C++ จะมีวิธีการจัดการรายงานความผิดพลาดกลับไปยังโปรแกรมหลักที่เรียกใช้อย่างเป็นระบบ โดยใช้แนวคิดที่เรียกว่า exception ซึ่งเราจะได้พิจารณาต่อไปในบทที่ XXXX (TODO: เพิ่มหรือลบ)

ลบข้อมูลที่มีดัชนี i ในรายการที่เก็บในอาร์เรย์ A ที่มีขนาด n (แก้ไข) (A1.8)

ถ้า $i > n-1$ หรือ $i < 0$

  รายงานความผิดพลาดว่าดัชนีอยู่นอกขอบเขต แล้วจบการทำงาน

พิจารณาให้ ตัวแปร $i \leftarrow i, i+1, \ldots, n-1$

  $A[i] \leftarrow A[i+1]$

$n \leftarrow n-1$

สังเกตว่าเวลาที่โปรแกรมลำลองข้างต้นใช้การทำงาน ขึ้นกับตำแหน่งของข้อมูล เช่นเดียวกับการวิเคราะห์ในส่วนของการค้นข้อมูล เราสามารถพิจารณากรณีต่าง ๆ ได้สามแบบ

▷ **คำถาม 1.11**
กรณีใดที่ทำให้โปรแกรมลำลอง A1.8 ใช้เวลาในการทำงานน้อยที่สุด (best case) และใช้เวลาเป็นเท่าใด? ◁

▷ **คำถาม 1.12**
กรณีใดที่ทำให้โปรแกรมลำลอง A1.8 ทำงานโดยใช้เวลามากที่สุด (worst case) และใช้เวลาเป็นเท่าใด ◁

เราจะละการเขียนโปรแกรมลำลองของการแทรกข้อมูลในรายการไว้เป็นแบบฝึกหัดท้ายบท

1.2 การประกาศและใช้งานอาร์เรย์ในภาษา C++

ในส่วนนี้เราจะศึกษาการใช้งานอาร์เรย์ในภาษา C++ เพื่อเป็นพื้นฐานในการเขียนโปรแกรมสำหรับโครงสร้างข้อมูลอื่น ๆ ในทางปฏิบัติจริง ในไลบรารีมาตรฐานของ C++ มีโครงสร้างข้อมูลแบบ `vector` ที่ใช้งานได้สะดวกกว่าอาร์เรย์มาก เราจะได้พิจารณาวิธีการใช้งานเทคโนโลยีที่มีประโยชน์เหล่านี้ในบทที่ **??**

ในภาษา C++ เราสามารถประกาศและจองเนื้อที่สำหรับตัวแปรประเภทอาร์เรย์ได้โดยใช้รูปแบบดังนี้

ชนิดข้อมูล ชื่อตัวแปร[ขนาด];

ตัวอย่างแสดงการประกาศตัวแปรอาร์เรย์ของจำนวนเต็ม (`int`) และอาร์เรย์ของอักขระ (`char`)

```
int a[100];
char buf[1000];
```

ภายหลังการประกาศ ระบบจะจองเนื้อที่ในหน่วยความจำสำหรับอาร์เรย์ที่เราประกาศไว้ ทำให้เราสามารถใช้อาร์เรย์ในการเก็บข้อมูลได้ ขนาดของเนื้อที่ที่จองจะขึ้นกับขนาดของอาร์เรย์และขนาดของข้อมูลชนิดนั้น ยกตัวอย่างเช่น โดยทั่วไปแล้ว ตัวแปรประเภท `int` ใช้เนื้อที่ในหน่วยความจำ 4 ไบท์ อาร์เรย์ `a` ก็จะมีขนาดเท่ากับ 400 ไบท์ หรือในกรณีของตัวแปรประเภท `char` ที่โดยทั่วไปใช้เนื้อที่ 1 ไบท์ในการจัดเก็บ อาร์เรย์ `buf` ก็จะถูกกันเนื้อที่ไว้ 1000 ไบท์[2]

ในการระบุขนาดของอาร์เรย์ เราไม่จำเป็นต้องระบุขนาดของอาร์เรย์เป็นค่าคงที่ก็ได้ ดังตัวอย่างด้านล่าง

[2]ในการทำงานจริง ระบบอาจจะกันที่ไว้เกินกว่าขนาดที่ต้องใช้จริงก็ได้

```
int m = 1000;
int x[m * 2];
```

อย่างไรก็ตาม หลายครั้งเราไม่ทราบขนาดที่แน่นอนของอาร์เรย์ในขณะที่เราต้องประกาศ ตัวอย่างเช่นในกรณีของการจัดเก็บรายการด้วยอาร์เรย์ ในที่นี้เราจะประกาศอาร์เรย์ให้มีขนาดใหญ่ระดับหนึ่งไว้ก่อน และคอยตรวจสอบว่าข้อมูลล้นอาร์เรย์แล้วหรือยัง เราจะพิจารณาและวิเคราะห์วิธีการการขยายขนาดอาร์เรย์ระหว่างการทำงาน ในบทที่ 2 นอกจากนี้ การใช้งานโครงสร้างข้อมูลแบบ vector ก็สามารถแก้ปัญหานี้ได้เช่นกัน

เราสามารถกำหนดค่าเริ่มต้นให้กับอาร์เรย์เมื่อต่อประกาศได้ ดังตัวอย่างด้านล่างนี้

```
  int A[9] = {2,3,5,7,11,13,19,23,29};
  int b[] = {1,10,100,1000};
  int c[100] = {1,2,3,4};
```

สังเกตตัวแปร B และ c ที่แสดงตัวอย่างที่คอมไพเลอร์จะจองข้อมูลตามจำนวนที่ระบุเมื่อประกาศ (B) และการกำหนดค่าเริ่มต้นให้กับข้อมูลบางตัว

เราจะเริ่มโดยเขียนฟังก์ชัน unknown จากอัลกอริทึม A1.2

```
int unknown(int A[], int n)
{
  int x = 0;
  for(int i = 0; i < n; ++i)
    x += A[i];
  return x;
}
```

ตัวอย่างการเรียกใช้งานฟังก์ชันดังกล่าวแสดงในส่วนของโปรแกรมด้านล่าง ซึ่งโปรแกรมจะพิมพ์ค่า 28 ออกมาเป็นผลลัพธ์

```
  int t[] = {1,2,3,4,5,6,7};
  cout << unknown(t,7);
```

สังเกตว่าเราส่งตัวแปรแบบอาร์เรย์โดยไม่ระบุขนาดของอาร์เรย์ (ประกาศ int A[] เป็นพารามิเตอร์ของฟังก์ชัน) แต่เราสามารถใช้งานได้เหมือนตัวแปรอาร์เรย์ปกติ

การประมวลผลรายการ

เราจะเขียนโปรแกรมที่เกี่ยวข้องกับการเก็บรายการในอาร์เรย์ที่ได้กล่าวถึงในส่วนก่อนหน้าของบทนี้ สำหรับตัวอย่างนี้ เราจะเก็บรายการของจำนวนเต็ม (ข้อมูลประเภท int)

เนื่องจากเราจะต้องจองอาร์เรย์ให้มีขนาดใหญ่พอไว้ก่อนเริ่มต้นใช้งาน เราจะประกาศค่าคงที่ max_size เพื่อเก็บขนาดมากสุดของอาร์เรย์ สังเกตว่าเราใช้ keyword const เพื่อระบุว่าค่านี้จะไม่มีการเปลี่ยนแปลง ในการเขียนฟังก์ชันในการประมวลผลรายการ เราจะใช้ค่าคงทีนี้ในลักษณะที่เป็นตัวแปรแบบโกลบอล (global) นั่นคือเราจะพิจารณาว่าทุก ๆ อาร์เรย์ที่จัดเก็บรายการที่เราจะประมวลผลจะมีขนาดมากที่สุดเท่ากันคือ max_size

```
const int max_size = 10000;
```

เราจะประกาศอาร์เรย์และตัวแปรที่ใช้เก็บขนาดของรายการดังด้านล่าง

```
int list[max_size];
int list_size;
```

โปรแกรมที่ 1.2: ตัวอย่างการใช้ตัวแปรแบบ reference ในการส่งค่าไปยังฟังก์ชัน

```
void m1(int a, int b)
{
  b += a;
}
void m2(int a, int& b)
{
  b += a;
}
```

จริง ๆ แล้ว โครงสร้างข้อมูลของเราจะใช้ตัวแปรทั้งสองด้วยกันตลอด การประกาศในลักษณะข้างต้นทำให้เมื่อเราเรียกใช้ฟังก์ชันต่าง ๆ เราต้องส่งตัวแปรสองตัวเสมอ ซึ่งเป็นเรื่องที่ไม่สะดวก ในบทที่ **??** เราจะศึกษาวิธีการจับตัวแปรทั้งสองให้อยู่รวมกันเป็นเสมือนตัวแปรเดียว

▷ **คำถาม 1.13**
ภายหลังที่เราประกาศตัวแปร `list` และ `list_size` ตามตัวอย่างข้างต้นแล้ว เราสามารถนำไปใช้งาน (เช่นเพิ่มข้อมูลลงในรายการ) ได้ทันทีหรือไม่? ◁

สังเกตว่า โครงสร้างข้อมูลที่เราจะใช้เก็บรายการนั้น ต้องมีการกำหนดค่าเริ่มต้นก่อน เพราะว่าเมื่อประกาศ ตัวแปร `list_size` อาจจะมีค่าเป็นอะไรก็ได้[3] ด้านล่างเป็นฟังก์ชันสำหรับกำหนดค่าเริ่มต้นให้กับโครงสร้างข้อมูลของเรา

โปรแกรมที่ 1.1: การกำหนดค่าเริ่มต้นให้กับรายการ

```
void list_init(int list[], int& size)
{
  size = 0;
}
```

ในฟังก์ชันข้างต้นเราใช้ตัวแปรประเภท *reference* ในการประกาศพารามิเตอร์ `size` ตัวแปรประเภทนี้ใช้สำหรับอ้างถึงตัวแปรอื่นและมีประโยชน์มากในการส่งค่าให้กับฟังก์ชัน

การส่งค่าให้กับฟังก์ชันจะกระทำโดยการคัดลอกค่าของอาร์กิวเมนท์จากฝั่งที่เรียกฟังก์ชัน มายังพารามิเตอร์ในฟังก์ชัน ทำให้การแก้ไขค่าของพารามิเตอร์ไม่มีผลกับอาร์กิวเมนท์ต้นทาง อย่างไรก็ตามในกรณีนี้ เราต้องการปรับค่า `size` ให้เป็นศูนย์ การใช้ตัวแปรประเภท reference ทำให้ตัวแปร `size` อ้างถึงตัวแปรที่เป็นอาร์กิวเมนท์ การอ้างถึงพารามิเตอร์ `size` ในฟังก์ชันก็จะเป็นการอ้างถึงตัวแปรที่ส่งมาเป็นอาร์กิวเมนท์ของฟังก์ชันด้วย

เพื่อความเข้าใจที่ชัดเจนขึ้น ลองพิจารณาตัวอย่างฟังก์ชันในโปรแกรม 1.2

ถ้าเราเรียกใช้ฟังก์ชันทั้งสองดังโปรแกรมด้านล่างนี้

```
int x = 10;
int y = 100;
m1(x,y);              // (3)
m2(x,y);              // (4)
```

เมื่อบรรทัดที่ 3 ทำงานเสร็จ ตัวแปร `y` จะมีค่า 100 เท่าเดิม แม้ว่าภายในฟังก์ชัน `m1` ตัวแปร `b` จะมีค่าเป็น 110 ก็ตาม อย่างไรก็ตามในการเรียกฟังก์ชัน `m2` พารามิเตอร์ `b` จะอ้างถึงตัวแปร `y` ทำให้เมื่อมีการเปลี่ยนแปลงค่า

[3]อย่างไรก็ตาม ถ้าเราประกาศตัวแปรแบบโกลบอล ตัวแปรจะถูกกำหนดค่าเริ่มต้นให้โดยอัตโนมัติตามกฎของ C++ (TODO: ใส่ reference??)

โปรแกรมที่ 1.3: ฟังก์ชัน `list_find` และ `list_append`

```
int list_find(int list[], int size, int q)
{
  for(int i = 0; i < size; ++i)
    if(list[i] == q)
      return i;
  return -1;
}

void list_append(int list[], int& size, int val)
{
  if(size >= max_size)
    throw "List overflow";      // Line 12
  list[size] = val;
  ++size;
}
```

ตัวแปร `b` แล้ว ก็เป็นการเปลี่ยนแปลงค่าในตัวแปร `y` ไปด้วย

การใช้ reference ลดความซับซ้อนในการเขียนฟังก์ชันที่ต้องการส่งผลลัพธ์กลับทางพารามิเตอร์ ตัวแปรแบบ reference มีความสัมพันธ์กับตัวแปรแบบพอยน์เตอร์มาก เราจะอธิบายความสัมพันธ์นี้เพิ่มเติมในส่วน 1.3

ส่วนของโปรแกรม 1.3 แสดงฟังก์ชัน `list_find` ที่ค้นหาข้อมูลในรายการและฟังก์ชัน `list_append` ที่เพิ่มข้อมูลเข้าไปตอนท้ายของรายการ

ถ้าสังเกตบรรทัดที่ 12 จะพบว่าเราแจ้งความผิดพลาดระหว่างการทำงาน เมื่ออาร์เรย์มีขนาดไม่พอ โดยการใช้ exception (ใช้คำสั่ง `throw`)

▷ **คำถาม 1.14** ฟังก์ชันหาค่ามากที่สุด
จงเขียนฟังก์ชัน `list_max` ที่รับรายการและคืนค่าดัชนีของข้อมูลที่มากที่สุดในรายการ ◁

เราจะเขียนฟังก์ชัน `list_delete` ที่ลบข้อมูลดัชนี i ออกจากอาร์เรย์

▷ **คำถาม 1.15** ฟังก์ชัน `list_delete`
จงเขียนฟังก์ชัน `list_delete(int list[], int& size, int i)` ที่รับรายการและดัชนี `i` ของข้อมูลที่ต้องการลบ และลบข้อมูลนั้นออกจากรายการโดยยังรักษาให้ของข้อมูลต่าง ๆ ยังเรียงลำดับกันเหมือนเดิม ◁

เฉลย:
แสดงในโปรแกรม 1.4

▷ **คำถาม 1.16** เวลาการทำงานของ `list_delete`
สมมติให้คำสั่งทุกบรรทัดทำงานโดยใช้เวลา 1 หน่วยเท่ากัน จงวิเคราะห์เวลาการทำงานของ `list_delete` ◁

▷ **คำถาม 1.17** ฟังก์ชัน `list_delete` ที่ไม่รักษาลำดับ
ถ้าเราไม่จำเป็นต้องรักษาลำดับของข้อมูลในรายการที่เหลือให้เรียงกันเหมือนเดิม เราสามารถเขียนฟังก์ชัน `list_delete`

โปรแกรมที่ 1.4: ฟังก์ชัน list_delete

```
void list_delete(int list[], int& size, int i)
{
  if((i < 0) || (i >= size))
    throw "Invalid␣index";
  for(int j = i; j < n-1; ++j)
    list[j] = list[j+1];
  --size;
}
```

โปรแกรมที่ 1.5: ตัวอย่างการเขียนฟังก์ชัน list_delete_all_data แบบหนึ่ง

```
void list_delete_all_data(int list[], int& size, int x)
{
  int os;
  do {
    os = size;
    list_delete_data(list, size, x);
  } while(os != size);
}
```

ขึ้นใหม่ ให้ทำงานโดยใช้เวลาที่ไม่ขึ้นกับขนาดของรายการในพารามิเตอร์ size ได้ จงเขียนฟังก์ชันดังกล่าว ◁

ตัวอย่างฟังก์ชันด้านต้น นำไปพัฒนาฟังก์ชันสำหรับลบข้อมูลในกรณีทั่วไปได้ไม่ยากนัก

▷ **คำถาม 1.18**
จงเขียนฟังก์ชัน list_delete_data(int list[], int& size, int x) ที่รับรายการและข้อมูล x แล้วลบข้อมูลดังกล่าวตัวแรกออกจากรายการ (ลบเฉพาะการปรากฏครั้งแรกในรายการ) ให้เขียนโดยเรียกใช้ฟังก์ชันด้านบนแทนที่จะเขียนใหม่ทั้งหมด ◁

▷ **คำถาม 1.19**
จงเขียนฟังก์ชัน list_delete_all_data(int list[], int& size, int x) ที่รับรายการและข้อมูล x แล้วลบข้อมูลดังกล่าวทั้งหมดออกจากรายการ สำหรับข้อนี้ เพื่อประสิทธิภาพสามารถเขียนฟังก์ชันขึ้นมาใหม่ได้ ◁

สำหรับคำถามด้านบน ผู้อ่านอาจจะสังเกตได้ว่า แทนที่จะเขียนฟังก์ชัน list_delete_all_data ขึ้นมาใหม่ทั้งหมด เราสามารถเขียนโดยเรียกฟังก์ชัน list_delete_data ได้ ดังโปรแกรม 1.5 ในบทที่ 2 เราจะวิเคราะห์เปรียบเทียบประสิทธิภาพวิธีการเขียนทั้งสองแบบ

1.2.1 อาร์เรย์หลายมิติ

TODO: เขียนส่วนนี้

ตำแหน่ง	ข้อมูลเป็นบิต	ข้อมูลถ้าพิจารณาเป็นตัวเลข
0	0000 0000	0
1	0000 1111	15
⋮	⋮	⋮
100	1001 0010	146
101	1111 1111	255
102	0001 0000	16
103	0110 0100	100
⋮	⋮	⋮

รูปที่ 1.3: ตัวอย่างการเก็บข้อมูลในหน่วยความจำพร้อมด้วยที่อยู่

1.3 พอยน์เตอร์และการใช้งานในภาษา C++

เนื่องจากพอยน์เตอร์เป็นชนิดข้อมูลที่ใกล้ชิดกับสถาปัตยกรรมของคอมพิวเตอร์มากที่สุดในภาษา C++ เราจะเริ่มโดยศึกษาโครงสร้างการเก็บข้อมูลในหน่วยความจำคอมพิวเตอร์กันก่อน

คอมพิวเตอร์เก็บข้อมูลแบบดิจิทัล หน่วยย่อยที่สุดของข้อมูลแบบดิจิทัลคือบิต (bit ย่อมาจาก binary digit) ซึ่งเป็นข้อมูลที่มีสองสถานะ โดยมากจะแทนด้วย 0 และ 1 หรือปิดกับเปิด แต่อาจจะเป็นอย่างอื่นก็ได้เช่น มีแสง ไม่มีแสง, ศักย์ไฟฟ้าสูง ศักย์ไฟฟ้าต่ำเป็นต้น

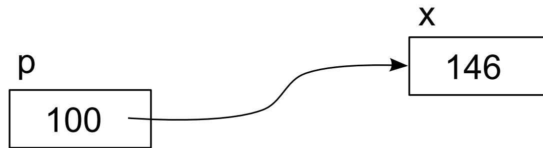
หน่วยความจำคอมพิวเตอร์ในปัจจุบันเก็บข้อมูลหลายล้านบิต การอ้างถึงข้อมูลจำนวนมากเหล่านั้นกระทำผ่านทางระบบเรียกที่อยู่ (address) กล่าวคือข้อมูลที่อ้างถึงได้ทุกหน่วยจะมีที่อยู่เฉพาะใช้สำหรับอ้างถึง ลักษณะการอ้างถึงข้อมูลเช่นนี้ก็ไม่ต่างจากการที่เราใช้ดัชนีในการอ้างถึงข้อมูลในอาร์เรย์นั่นเอง

อย่างไรก็ตาม ในการกำหนดที่อยู่นั้น จะไม่ได้กำหนดให้กับข้อมูลทุกบิตโดยตรง เนื่องจากบิตเป็นหน่วยที่เล็กเกินไปในหลาย ๆ กรณี แต่จะกำหนดให้กับกลุ่มของบิตที่เรียงกัน กลุ่มละ 8 บิต ซึ่งเรียกว่าไบท์ (byte) ซึ่งสามารถพิจารณาเป็นเลขฐานสอง 8 หลักก็ได้ ตัวอย่างของการเก็บข้อมูลแสดงดังรูปที่ 1.3

ข้อมูลที่เก็บในหน่วยความจำหลายชนิดอาจจะมีขนาดใหญ่เกินกว่าจะเก็บได้ใน 1 ไบท์ เช่นข้อมูลประเภท `int` ในภาษา C++ โดยมากข้อมูลเหล่านั้นก็จะถูกจัดเก็บอยู่ในหลายไบท์เรียงต่อกัน รูปแบบและวิธีการจัดเก็บเหล่านี้อยู่นอกขอบเขตของหนังสือเล่มนี้ ผู้อ่านที่สนใจสามารถอ่านเพิ่มเติมได้ในหนังสือสถาปัตยกรรมคอมพิวเตอร์ทั่วไป

ข้อมูลประเภทพอยน์เตอร์เป็นข้อมูลสำหรับเก็บตำแหน่งของหน่วยความจำ เพื่อใช้อ้างถึงข้อมูลที่อยู่ในตำแหน่งดังกล่าว ในบางภาษาเช่น Java หรือ C# ก็มีการใช้งานข้อมูลชนิดนี้ในการอ้างถึงข้อมูลที่เป็นวัตถุแต่เรียกว่าเป็นการอ้างถึง (reference)

สมมติว่าตัวแปร `x` ใช้เนื้อที่ในหน่วยความจำที่ตำแหน่ง 100 และเก็บค่า 146 ถ้าตัวแปรแบบพอยน์เตอร์ `p` ใช้เนื้อที่ในหน่วยความจำที่ตำแหน่ง 103 และ *ชี้* ไปที่ตัวแปร `x` ตัวแปร `p` จะเก็บค่า 100 สำหรับการทำความเข้าใจทั่วไป ข้อมูลประเภทนี้มักเขียนแทนด้วยช่องที่มีลูกศร เพื่อแสดงการชี้ไปยังตำแหน่งของข้อมูลอื่น ๆ ดังตัวอย่างในรูป 1.4



รูปที่ 1.4: แผนภาพแสดงพอยน์เตอร์ p

1.3.1 การประกาศชนิดข้อมูล โอเปอเรเตอร์ * และ &

วิธีการประกาศและใช้ข้อมูลประเภทนี้ขึ้นกับภาษาโปรแกรมที่ใช้ สำหรับภาษา C++ เราใช้เครื่องหมาย * ในการระบุชนิดข้อมูล กล่าวคือเราจะเขียน

ชนิดข้อมูล*

เพื่อระบุว่าเป็นพอยน์เตอร์ไปยังชนิดข้อมูลดังกล่าว ยกตัวอย่างเช่น

```
int* p;
```

เป็นการประกาศตัวแปร p ที่เป็นพอยน์เตอร์ไปยังข้อมูลชนิด `int` ตัวแปรประเภทพอยน์เตอร์จะต้องอ้างถึงตำแหน่งในหน่วยความจำ เราจะใช้โอเปอเรเตอร์แบบนำหน้า & ในการอ้างถึงตำแหน่งในหน่วยความจำของตัวแปรหรือข้อมูลดังแสดงในตัวอย่างด้านล่าง

```
int x = 146;
int* p = &x;    // Line 2
```

หลังการทำงานในบรรทัดที่ 2 ตัวแปร `p` จะเก็บตำแหน่งในหน่วยความจำของตัวแปร `x` หรือเราจะกล่าวว่า `p` ชี้ไปที่ตัวแปร `x`

ต่อไปถ้าเราอ้างถึงตัวแปร p เราจะหมายถึง "ตำแหน่ง" ของข้อมูลในหน่วยความจำ ถ้าเราต้องการอ้างถึง "ข้อมูล" ในหน่วยความจำตำแหน่งนั้น เราจะใช้โอเปอเรเตอร์นำหน้า * ยกตัวอย่างเช่นในโปรแกรมด้านล่าง

```
cout << *p;         // Line 1
*p = 1000;          // Line 2
cout << x;          // Line 3
```

โปรแกรมในบรรทัดที่ 1 จะพิมพ์ค่า 146 เนื่องจากตัวแปร `p` ชี้ไปที่ตำแหน่งของตัวแปร `x` ที่มีค่า 146 เมื่อเราอ้างถึง "ข้อมูล" ที่ตำแหน่งที่ p เก็บอยู่ เราจึงได้ค่าเป็น 146

การอ้าง `*p` หมายถึงข้อมูลในตำแหน่งหน่วยความจำที่ตัวแปร `p` ชี้อยู่ ดังนั้นโปรแกรมในบรรทัดที่ 2 ก็จะเป็นการกำหนดค่าให้กับหน่วยความจำที่เดียวกับตัวแปร `x` ทำให้โปรแกรมในบรรทัดที่ 3 พิมพ์ค่า 1000 ออกมา

สังเกตว่าเครื่องหมาย * ในคำสั่ง `int* p = &x;` กับ `*p = 1000;` มีความหมายต่างกัน ในตัวอย่างแรกเป็นการขยายชนิดข้อมูล `int` เพื่อระบุชนิดข้อมูลว่าเป็นพอยน์เตอร์ไปยังจำนวนเต็ม โดยมีการกำหนดค่าเริ่มต้นด้วยเครื่องหมายเท่ากับ ในตัวอย่างที่สอง เครื่องหมาย * เป็นโอเปอเรเตอร์แบบนำหน้า ที่กระทำกับตัวแปร p ในตัวอย่างแรก ถ้าเราไม่กำหนดค่าเริ่มต้นให้กับพอยน์เตอร์ `p` ทันทีเมื่อประกาศ โปรแกรมจะเขียนได้ดังนี้

```
int* p;
p = &x;
```

โอเปอเรเตอร์ * กับ & เป็นโอเปอเรเตอร์ที่ทำหน้าที่ตรงข้ามกัน ถ้าพิจารณาตามแผนภาพในรูปที่ 1.4 โอเปอเรเตอร์ * จะพาเราไปตามลูกศร ส่วน & จะพาเราย้อนกลับ

▷ **คำถาม 1.20**
นิพจน์ `*(&x) = 1234;` มีความหมายอย่างไร? ◁

▷ **คำถาม 1.21**
สำหรับตัวแปร x ที่ประกาศและกำหนดค่าเริ่มต้นด้วยคำสั่ง `int x = 146;` นิพจน์ &x หมายถึงตำแหน่งในหน่วยความจำของ x นิพจน์ &(&x) หมายถึงอะไร? ◁

เฉลย:
นิพจน์ดังกล่าวไม่มีความหมาย และจะทำให้เกิดความผิดพลาดระหว่างการคอมไพล์ เพื่อความเข้าใจที่ชัดเจนยิ่งขึ้น กรุณาอ่านส่วน 1.3.8

ผู้ที่ใช้งานพอยน์เตอร์นั้นจะต้องระวังเป็นพิเศษในการจัดการหน่วยความจำ โดยเฉพาะเกี่ยวกับสถานะของหน่วยความจำที่พอยน์เตอร์นั้นชี้ไป พิจารณาตัวอย่างในโปรแกรมย่อยด้านล่าง

```
int* crazy(int x)
{
   int y = x + 10;  int* z = &y;
   return z;
}
```

ฟังก์ชันดังกล่าวคืนค่าพอยน์เตอร์ไปยังตัวแปร y ซึ่งเป็นตัวแปรภายในของฟังก์ชัน crazy ตัวแปรประเภทนี้โดยทั่วไปจะถูกเก็บในหน่วยความจำที่กันเนื้อที่ไว้แค่ช่วงที่ฟังก์ชันทำงานเท่านั้น เมื่อฟังก์ชันจบการทำงานลง หน่วยความจำส่วนนี้ก็อาจจะถูกใช้โดยตัวแปรอื่น ๆ อย่างไรก็ตาม พอยน์เตอร์ที่คืนมาก็ยังชี้ไปที่ตำแหน่งเดิมของตัวแปร y อยู่ ทำให้เมื่ออ่านเขียนข้อมูลในตำแหน่งนั้น อาจจะเกิดปัญหาไปเขียนทับข้อมูลของส่วนอื่นของโปรแกรมได้

1.3.2 การจองหน่วยความจำ: โอเปอเรเตอร์ new และโอเปอเรเตอร์ delete

นอกจากที่จะให้พอยน์เตอร์ชี้ไปยังตัวแปรต่าง ๆ แล้ว เรายังสามารถจองหน่วยความจำเพื่อเก็บข้อมูลโดยเฉพาะได้ด้วยโอเปอเรเตอร์ new หน่วยความจำที่ขอจองมาได้นี้จะเป็นหน่วยความจำที่ระบบเชื่อว่ายังไม่มีการใช้ ทำให้เราสามารถใช้เนื้อที่เหล่านี้เก็บข้อมูลได้ตามต้องการ

โอเปอเรเตอร์ new นอกจากจะจองหน่วยความจำสำหรับข้อมูลแล้ว ยังกำหนดค่าเริ่มต้นให้กับข้อมูลดังกล่าวก่อนที่จะคืนพอยน์เตอร์กลับมา

หน่วยความจำที่จองมานี้ เมื่อไม่ใช้แล้วจะต้องถูกปล่อยคืนให้กับระบบด้วยโอเปอรเรเตอร์ delete เพื่อเก็บไว้จัดสรรในครั้งต่อ ๆ ไป พิจารณาตัวอย่างการใช้งานดังนี้

```
int* p = new int;
int* q = p;             // Line 2
p = new int;            // Line 3
// ...                     Line 4
// ... do something
```

```
delete p;
delete q;
```

เมื่อโปรแกรมทำงานผ่านบรรทัดที่ 2 พอยน์เตอร์ p และ q จะชี้ไปยังตำแหน่งที่จองไว้ในหน่วยความจำเดียวกัน เมื่อถึงบรรทัดที่ 4 ตัวแปรทั้งสองจะชี้ไปยังหน่วยความจำคนละที่ เนื่องจากบรรทัดที่ 3 เราได้จองหน่วยความจำใหม่ให้กับ p เมื่อโปรแกรมทำงานเสร็จ เราก็จะสั่ง delete เพื่อให้ระบบคืนค่าหน่วยความจำที่จองไว้ทั้งสองหน่วย

พิจารณาตัวอย่างโปรแกรมด้านล่าง

```
int c = 0;
for(int i = 0; i < N; ++i) {
  int* p = new int;
  p = c + i;
  c = p;
}
```

▷ **คำถาม 1.22**
โปรแกรมข้างต้นทำงานอะไร ถ้าตัวแปร N มีค่ามาก ๆ โปรแกรมดังกล่าวจะทำให้เกิดอะไรขึ้น ◁

ความสามารถในการจองหน่วยความจำเมื่อโปรแกรมทำงานนี้เป็นสิ่งที่จำเป็นมากในการพัฒนาโครงสร้างข้อมูลที่ซับซ้อนขึ้นต่อไป อย่างไรก็ตาม ถ้าเราไม่ระมัดระวังพอและไม่คืนหน่วยความจำที่จองไว้เมื่อเลิกอ้างถึงแล้ว เราอาจจะพบปัญหาโปรแกรมใช้หน่วยความจำมากเกินไปเนื่องจากมีการรั่วไหลของการใช้งานหน่วยความจำก็ได้ (เรียกว่า memory leak)

ในภาษา C++ เราสามารถซ่อนการทำงานของโครงสร้างข้อมูลรวมถึงการจองและคืนหน่วยความจำไว้ภายในคลาส ถ้าพัฒนาอย่างถูกวิธี จะช่วยลดปัญหาหน่วยความจำรั่วไหลได้

1.3.3 พอยน์เตอร์ว่าง

นอกจากเราจะกำหนดค่าให้ตัวแปรพอยน์เตอร์ชี้ไปที่ตำแหน่งต่าง ๆ ในหน่วยความจำโดยใช้โอเปอเรเตอร์ & แล้ว เรายังสามารถกำหนดให้ตัวแปรพอยน์เตอร์มีสถานะเป็น "ไม่ได้ชี้ไปที่ไหน" ได้โดยการกำหนดค่า 0 ให้กับตัวแปรพอยน์เตอร์นั้น[4][5] พอยน์เตอร์ดังกล่าวเรียกว่าเป็น*พอยน์เตอร์ว่าง* (*null pointer*)

พอยน์เตอร์ที่มีค่าเป็นพอยน์เตอร์ว่างนี้ ในภาษา C++ ไม่ได้ทำงานแตกต่างจากพอยน์เตอร์ธรรมดาเท่าใดนัก นั่นคือ โปรแกรมสามารถเรียกหาค่าที่ชี้โดยพอยน์เตอร์ดังกล่าวได้ ผลลัพธ์ที่ได้ขึ้นกับคอมไพเลอร์และระบบปฏิบัติการที่ใช้ แต่โดยทั่วไปการเรียกอ่านค่าตำแหน่งที่ 0 มักจะถูกป้องกันไว้ด้วยระบบปฏิบัติการ ทำให้โปรแกรมที่เรียกใช้ดังกล่าว crash ได้เป็นต้น อย่างไรก็ตาม ภาษา C/C++ รับประกันว่าพอยน์เตอร์ว่างสองตัวจะเปรียบเทียบได้เท่ากัน ถึงแม้จะเป็นพอยน์เตอร์คนละชนิดก็ตาม

1.3.4 ชนิดข้อมูลแบบ void และพอยน์เตอร์ไปยัง void

ชนิดข้อมูล void เป็นชนิดข้อมูลพิเศษที่ไม่มีข้อมูลใดเป็นสมาชิก การประกาศว่าฟังก์ชันคืนค่าเป็นข้อมูลชนิดนี้คือการประกาศว่าฟังก์ชันดังกล่าวไม่คืนค่า นอกจากเราจะใช้ void เพื่อระบุฟังก์ชันแล้ว พอยน์เตอร์ไปยัง void (ชนิด

[4]ในมาตรฐาน C++11 มีการนิยามค่าคงที่ nullptr เพื่อใช้แทนพอยน์เตอร์ว่างเพื่อแก้ปัญหาการใช้ค่าคงที่ซ้ำซ้อน อย่างไรก็ตาม เพื่อให้หนังสือนี้ใช้งานได้กับคอมไพเลอร์ที่ยังไม่ปรับรุ่นให้ทันมาตรฐานใหม่นี้ เราจะยังใช้ค่าคงที่ 0 เพื่อแทนพอยน์เตอร์ว่างอยู่ ถ้าผู้อ่านใช้คอมไพเลอร์รุ่นใหม่แล้ว ขอแนะนำให้ใช้ nullptr แทน 0

[5]ในภาษา C นิยมใช้ค่าคงที่ NULL ที่นิยามไว้ในไฟล์หัวมาตรฐาน อย่างไรก็ตาม เราสามารถใช้ค่าคงที่ 0 ใน C++ ได้เลย

ข้อมูล void*) แทนพอยน์เตอร์ที่ชี้ไปยังตำแหน่งในหน่วยความจำที่เราไม่สนใจว่าจะเป็นข้อมูลชนิดใด เราสามารถกำหนดค่าพอยน์เตอร์ไปยังข้อมูลใด ๆ ให้กับพอยน์เตอร์ประเภท void* ได้ และพอยน์เตอร์ประเภท void* ก็สามารถถูกแปลง (cast) เป็นพอยน์เตอร์ประเภทอื่น ๆ ได้ อย่างไรก็ตามการแปลงในลักษณะนี้ ถ้าไม่ระวังอาจก่อให้เกิดข้อผิดพลาดระหว่างการทำงานได้

1.3.5 พอยน์เตอร์กับการส่งพารามิเตอร์

เราสามารถใช้พอยน์เตอร์เพื่อส่งค่าผ่านพารามิเตอร์ในลักษณะเดียวกับการส่งแบบ pass by reference ได้ พิจารณาฟังก์ชันด้านล่างที่สลับค่าในตัวแปรสองตัว

```
void swap1(int& a, int& b)
{
  int t = a;
  a = b;
  b = t;
}
```

ฟังก์ชันนี้ทำงานได้ตามที่เราต้องการ เนื่องจากชนิดข้อมูลของพารามิเตอร์ a และ b คือ int& ฟังก์ชันดังด้านล่างจะไม่สามารถสลับค่าในตัวแปรที่ส่งมาได้ แม้ว่าค่าในตัวแปร a และ b จะเปลี่ยนในฟังก์ชันก็ตาม เพราะว่าการส่งค่าโดยปกติจะเป็นการส่งแบบ pass by value

```
void swap2(int a, int b)     // broken
{
  int t = a;
  a = b;
  b = t;
}
```

อย่างไรก็ตาม เนื่องจากเราต้องการเปลี่ยนแปลงค่าของอาร์กิวเมนท์ที่ส่งมาให้กับฟังก์ชัน เราจำเป็นต้องอ้างถึงตำแหน่งของอาร์กิวเมนท์นั้น ด้านล่างเป็นฟังก์ชัน swap3 ที่เขียนโดยใช้พอยน์เตอร์

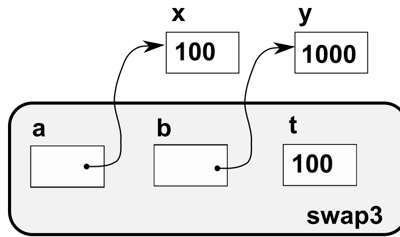
```
void swap3(int* a, int* b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
```

เนื่องจากฟังก์ชัน swap3 รับอาร์กิวเมนท์เป็นพอยน์เตอร์ การเรียกใช้งานก็จะยุ่งยากขึ้นเล็กน้อย ดังแสดงในโปรแกรมด้านล่าง

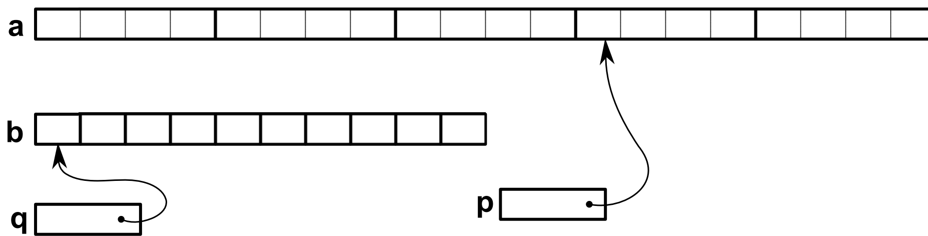
```
int x = 100;
int y = 1000;
swap3(&x, &y);
```

รูป 1.5 แสดงการชี้ของพอยน์เตอร์ในการเรียกใช้งานฟังก์ชัน swap3

▷ **คำถาม 1.23**
ฟังก์ชัน swap1 และ swap3 แม้มีการทำงานที่เหมือนกัน การเรียกใช้งาน swap3 นั้นดูยุ่งยากกว่า อย่างไรก็ตาม มีบางกรณีที่การประกาศพารามิเตอร์ที่ต้องการแก้ไขด้วยพอยน์เตอร์ทำให้เรามีอิสระในการเรียกใช้งานมากกว่า กรณีนั้นคืออะไร? ◁



รูปที่ 1.5: พอยน์เตอร์ต่าง ๆ ในการเรียกใช้ฟังก์ชัน `swap3`



รูปที่ 1.6: พอยน์เตอร์ที่ชี้ที่ตำแหน่งต่าง ๆ ในอาร์เรย์

เฉลย:
ในกรณีของการส่งตัวแปรแบบ pass by reference ทางฝั่งผู้เรียกใช้ฟังก์ชันจำเป็นต้องมีตัวแปรเพื่อส่งมายังพารามิเตอร์นี้ แต่ในกรณีของการส่งแบบพอยน์เตอร์ พารามิเตอร์ดังกล่าวจะเป็นพารามิเตอร์ที่สามารถละเอาไว้ได้โดยส่ง พอยน์เตอร์ว่าง `0` มาแทน ดังนั้น ในการประกาศพารามิเตอร์ ถ้าพารามิเตอร์ใด จำเป็นต้องมีตัวแปรมารับค่า เราควรจะใช้การประกาศแบบ pass by reference ถ้าพารามิเตอร์นั้นไม่จำเป็นต้องมีอยู่จริงในบางกรณี เราก็ควรจะประกาศรับพารามิเตอร์ด้วยพอยน์เตอร์แทน

1.3.6 การคำนวณกับพอยน์เตอร์

เราสามารถดำเนินการเชิงตัวเลขกับพอยน์เตอร์ได้ ปกติพอยน์เตอร์สำหรับชนิดข้อมูลใด ๆ ก็จะใช้ชี้ไปยังตำแหน่งในหน่วยความจำของข้อมูลชนิดนั้น พิจารณาตัวอย่างโปรแกรมด้านล่าง

```
int a[5];
char b[10];
int* p = &a[3];
char* q = &b[0];
```

ถ้าเราสมมติว่าข้อมูลประเภท `int` ใช้เนื้อที่ 4 ไบท์ในการจัดเก็บในหน่วยความจำ และ `char` ใช้เนื้อที่ 1 ไบท์ในการจัดเก็บ ตัวแปรต่าง ๆ จะอยู่ในหน่วยความจำดังในรูปที่ 1.6

เนื่องจากพอยน์เตอร์ถูกนิยามเทียบกับชนิดข้อมูลที่ชี้ไป ดังนั้นจึงไม่แปลกอะไรถ้าเมื่อเราเพิ่มหรือลดค่าของพอยน์เตอร์นั้นด้วยจำนวนเต็ม เราจะหมายถึงตำแหน่งของข้อมูลชนิดนั้นตัวถัด ๆ ไป หรือตัวก่อน ๆ หน้า ยกตัวอย่างเช่น `p+1` เป็นพอยน์เตอร์ที่ชี้ไปที่ `a[4]` และ `q+1` เป็นพอยน์เตอร์ที่ชี้ไปที่ `b[1]`

สังเกตว่าเราบวก 1 เข้ากับ p เพื่อเลื่อนจาก a[3] ไปยัง a[4] ถึงแม้ว่าตำแหน่งในหน่วยความจำของ a[3] และ a[4] จะต่างกัน 4 ตำแหน่งก็ตาม

นอกจากการบวกและลบด้วยจำนวนเต็มแล้ว เรายังสามารถนำพอยน์เตอร์ที่ชี้ไปยังข้อมูลชนิดเดียวกันมาลบกันได้ด้วย

▷ **คำถาม 1.24**
พิจารณาตัวแปร int* r = &a[1] และ int* s = &a[4] นิพจน์ s - r มีค่าเป็นเท่าใด? ◁

1.3.7 อาร์เรย์และพอยน์เตอร์ในภาษา C/C++

สังเกตว่าจากส่วนที่ 1.3.6 เราสามารถใช้พอยน์เตอร์ในการประมวลผลข้อมูลในอาร์เรย์ได้ ยกตัวอย่างเช่นฟังก์ชันในโปรแกรมที่ 1.6 ที่หาผลรวมของข้อมูลในรายการ

โปรแกรมที่ 1.6: ฟังก์ชันที่หาผลรวมของข้อมูลในรายการโดยใช้พอยน์เตอร์วิ่งไปในอาร์เรย์

```
int sum1(int list[], int size)
{
  int s = 0;
  int* p = &list[0];
  for(int i = 0; i < size; ++i) {
    s += *p;
    ++p;
  }
  return s;
}
```

▷ **คำถาม 1.25**
โปรแกรม 1.6 ปรับค่าพอยน์เตอร์ p ให้ชี้ไปยังข้อมูลต่าง ๆ ในอาร์เรย์ ให้แก้ฟังก์ชัน sum1 ให้ทำงานได้เหมือนเดิม โดยให้พอยน์เตอร์ p มีค่าคงที่โดยชี้อยู่ที่ตำแหน่งของ list[0] ตลอดเวลา ◁

สังเกตว่าในโปรแกรม 1.6 เรายังใช้ดัชนี i ในการวนรอบอยู่ อย่างไรก็ตาม ยังมีอีกรูปแบบในการเขียนที่อาจจะดูกระทัดรัดกว่า ดังแสดงในโปรแกรม 1.7

โปรแกรมที่ 1.7: ฟังก์ชันที่หาผลรวมของข้อมูลในรายการโดยใช้พอยน์เตอร์วิ่งไปในอาร์เรย์ อีกแบบหนึ่ง

```
int sum2(int list[], int size)
{
  int s = 0;
  int* p = &list[0];          // Line 4
  int* end = &list[size];     // Line 5
  while(p != end) {
    c += *p;
    ++p;
  }
  return s;
}
```

สังเกตว่าในโปรแกรม 1.7 ตัวแปร end อาจจะชี้ออกไปนอกขอบเขตของอาร์เรย์ได้ ในกรณีนี้ ภาษา C++ รับประกันว่าพอยน์เตอร์ที่ชี้ไปที่ข้อมูลที่เกินอาร์เรย์ไป 1 หน่วย จะยังสามารถใช้งานได้ เพื่อให้การเขียนโปรแกรมในลักษณะนี้

ไม่เกิดข้อผิดพลาด อย่างไรก็ตาม การอ้างพอยน์เตอร์ออกไปยังตำแหน่งที่อยู่ก่อนหน้าข้อมูลตัวแปรของอาร์เรย์ หรือตำแหน่งที่เกินกว่าข้อมูลตัวสุดท้ายมากกว่า 1 ตำแหน่งนั้น ระบบจะไม่รับประกันว่าจะสามารถทำได้

ในภาษา C++ นอกจากที่เราจะนำพอยน์เตอร์ไปชี้ที่อาร์เรย์เพื่อประมวลผลได้แล้ว เรายังสามารถนำอาร์เรย์และพอยน์เตอร์มาใช้สลับกันไปมาได้ในอีกหลาย ๆ กรณี เราจะเริ่มจากกรณีที่ใช้อาร์เรย์ในรูปของพอยน์เตอร์ก่อน

สำหรับอาร์เรย์ใด ๆ ถ้าเราอ้างถึงอาร์เรย์โดยไม่ระบุดัชนี เราจะได้ค่าเป็นตำแหน่งของข้อมูลตัวแรกในอาร์เรย์ ดังนั้น บรรทัดที่ 4 และ 5 ในโปรแกรม 1.7 สามารถแก้เป็นดังด้านล่างได้

```
int* p = list;                  // Line 4
int* end = list + size;         // Line 5
```

อย่างไรก็ตาม ถึงแม้ว่าเราจะอ้างตัวแปรอาร์เรย์เป็นพอยน์เตอร์ได้ เราจะไม่สามารถกำหนดค่าให้กับมันได้ โปรแกรมด้านล่างจะคอมไพล์ไม่ผ่านถ้าตัวแปร `list` เป็นอาร์เรย์ แต่ถ้า `list` เป็นพอยน์เตอร์จะสามารถทำได้ (ในภาษาเชิงเทคนิค เราจะกล่าวว่านิพจน์ `list` ไม่มี L-value ผู้ที่สนใจสามารถ อ่านเพิ่มเติมได้ในส่วน 1.3.8)

```
list = list + 1;               // does not compile
```

▷ **คำถาม 1.26**
โปรแกรมเมอร์ที่เขียนคำสั่ง `list = list + 1;` ในตัวอย่างข้างต้น ต้องการจะทำอะไร? ◁

ในกรณีของอาร์เรย์หลายมิติ การอ้างถึงอาร์เรย์ด้วยชื่อก็จะได้ผลลัพธ์เป็นพอยน์เตอร์เช่นกัน โดยพอยน์เตอร์ที่ได้จะชี้ไปยังตำแหน่งของข้อมูลตัวแรกที่สอดคล้องกับการอ้างถึงขณะนั้น และมีชนิดข้อมูลสอดคล้องกับระดับของการอ้างถึง ยกตัวอย่างเช่น ถ้าเราประกาศอาร์เรย์ `int a[100][200];` การอ้างถึง `a` จะเป็นพอยน์เตอร์ชี้ไปที่ตำแหน่งของ `a[0][0]` และมีชนิดข้อมูลเป็น `int**` (พอยน์เตอร์ไปยังพอยน์เตอร์ไปยัง `int`) ส่วน `a[10]` จะเป็นพอยน์เตอร์ชนิด `int*` ชี้ไปที่ตำแหน่งของ `a[10][0]` เป็นต้น

เช่นเดียวกับในตัวอย่างข้างต้น ทั้ง `a` และ `a[10]` เป็นนิพจน์ที่มีค่า แต่ไม่สามารถกำหนดค่าให้ได้ เนื่องจากเราไม่มีตัวแปรในหน่วยความจำจริง ๆ ที่เก็บค่าเหล่านี้ (นั่นคือ `a` และ `a[10]` ไม่มี L-value)

ในมุมกลับกัน เราสามารถใช้พอยน์เตอร์ในลักษณะเดียวกับอาร์เรย์ได้ กล่าวคือ ถ้า `p` เป็นพอยน์เตอร์ การอ้าง `p[0]` จะมีความหมายเดียวกับ `*p` และการอ้าง `p[10]` จะมีความหมายเดียวกับ `*(p + 10)` ดังนั้นหลาย ๆ ครั้งเราจะพบว่าฟังก์ชันที่รับค่าเป็นอาร์เรย์ แต่กลับประกาศว่ารับพารามิเตอร์เป็นพอยน์เตอร์เพื่อความสะดวกในการใช้งาน

▷ **คำถาม 1.27**
ถ้าในโปรแกรม เรามีความจะเป็นต้องเลื่อนดัชนีทั้งหมดของอาร์เรย์ `list` ลงหนึ่ง หลาย ๆ ครั้ง เราจะความสามารถในการสลับเปลี่ยนระหว่างอาร์เรย์และพอยน์เตอร์เพื่อดำเนินการดังกล่าวได้อย่างไร ◁

การใช้งานพอยน์เตอร์ในรูปแบบของอาร์เรย์เพิ่มความสะดวกในการใช้งานอาร์เรย์ที่เราต้องจองหน่วยความจำเอง โดยมากจะเป็นอาร์เรย์ที่เราไม่ทราบขนาดก่อนถึงเวลาใช้งาน หรือกระทั่งเป็นอาร์เรย์ที่มีขนาดเปลี่ยนไปมาได้

โอเปอเรเตอร์ `new` และ `delete` สามารถใช้เพื่อจองหน่วยความจำแบบอาร์เรย์ได้ โดยรูปแบบในการใช้งานเป็นดังนี้

`new` ชนิดข้อมูล[จำนวนข้อมูล]
`delete []` ตัวแปร

สังเกตว่าเราระบุจำนวนช่องของข้อมูลที่ต้องการใช้ในโอเปอเรเตอร์ new และใส่เครื่องหมาย [] เมื่อใช้โอเปอเรเตอร์ delete เพื่อระบุว่าเป็นการทำลายอาร์เรย์ ดังตัวอย่างด้านล่าง

```
int* a = new int[100];

for(int i = 0; i < 100; ++i)
  a[i] = i;
// ...

delete [] a;
```

สังเกตว่าเราใช้งานตัวแปรพอยน์เตอร์ a ในลักษณะเดียวกับอาร์เรย์ ข้อควรระวังก็คือ ถ้าเราเรียกโอเปอเรเตอร์ new แบบอาร์เรย์ เมื่อเราเรียก delete เราจะต้องระบุ [] ด้วยทุกครั้ง ไม่เช่นนั้นโอเปอเรเตอร์ delete อาจทำงานผิดพลาดได้

1.3.8 แนวคิดเกี่ยวกับ L-value และ R-value ⋆

พิจารณาคำสั่งกำหนดค่าด้านล่าง

```
x = x + 1;
```

ในคำสั่งดังกล่าวเรากล่าวถึงตัวแปร x สองครั้ง ถ้าพิจารณาให้ดีเราจะพบว่าความหมายของการกล่าวถึงทั้งสองครั้งนั้นแตกต่างกัน

- เมื่อเรากล่าวถึง x ในด้านซ้ายของเครื่องหมายกำหนดค่า สิ่งที่เราต้องการจากตัวแปร x คือตำแหน่งในหน่วยความจำ เพื่อให้ระบบสามารถเก็บผลลัพธ์ได้
- สำหรับการกล่าวถึงตัวแปร x ในด้านขวาของเครื่องหมายกำหนดค่านั้น สิ่งที่เราต้องการได้คือค่าของข้อมูลที่เก็บในตัวแปร x

เราจะเรียก "ความหมาย" ของนิพจน์ใด ๆ เมื่ออยู่ด้านขวาของเครื่องหมายกำหนดค่า ว่าเป็น R-value ของนิพจน์นั้น ซึ่งความหมายนี้ จะสอดคล้องกับค่าของนิพจน์นั้น ๆ ในทางกลับกัน L-value ก็เป็น "ความหมาย" ของนิพจน์นั้น เมื่ออยู่ด้านซ้ายของเครื่องหมายกำหนดค่า สังเกตว่าบางนิพจน์จะไม่มีค่า L-value นั่นหมายความว่าเราไม่สามารถใช้นิพจน์นั้นในด้านซ้ายของเครื่องหมายกำหนดค่าได้ นั่นคือ กำหนดค่าไม่ได้นั่นเอง

1.4 แบบฝึกหัด

1. เขียนโปรแกรมลำลองสำหรับการแทรกข้อมูลในรายการ

บทที่ 2

การวิเคราะห์เวลาการทำงานและชนิดข้อมูลนามธรรมพื้นฐาน

ในบทนี้เราจะศึกษาการวิเคราะห์เวลาการทำงานอย่างละเอียดขึ้น พร้อมกับแนะนำชนิดข้อมูลนามธรรม (abstract data type) พื้นฐานบางชนิด ซึ่งในบทนี้เราจะอิมพลีเมนท์[1]ด้วยอาร์เรย์และวิเคราะห์เวลาการทำงาน

2.1 การวิเคราะห์เชิงเส้นกำกับ

ในบทนี้เราจะศึกษาข้อจำกัดของการวิเคราะห์ดังกล่าว และทำความเข้าใจกับเทคนิคการวิเคราะห์ที่เราจะใช้ต่อไปตลอดในหนังสือเล่มนี้

2.1.1 ความแม่นยำของการวิเคราะห์

จากตัวอย่างการวิเคราะห์ในบทที่ 1 เราได้วิเคราะห์อัลกอริทึม A1.2 โดยสมมติให้ทุก ๆ คำสั่งทำงานโดยใช้เวลาเท่ากันคือ 1 หน่วย เราได้ว่าอัลกอริทึมทำงานโดยใช้เวลา $2n + 2$ หน่วย

ผล $2n + 2$ ที่ได้จากการวิเคราะห์นั้น มีความแม่นยำเพียงใด? แน่นอนว่าข้อสมมติว่าทุกคำสั่งทำงานโดยใช้เวลาเท่ากันนั้นไม่เป็นความจริง ดังนั้นเราจะลองปรับการวิเคราะห์โดยสมมติค่าคงที่ $c_1, c_2, c_3,$ และ c_4 เป็นเวลาที่แต่ละคำสั่งทำงานเป็นมิลลิวินาที ดังแสดงด้านล่าง

```
ให้ x ← 0    ▷ ▷ ▷ ทำงานใช้เวลา c1 มิลลิวินาที ทำงานรวม 1 ครั้ง                    (A2.1)
พิจารณา ตัวแปร i ← 0, 1, ..., n − 1    ▷ ▷ ▷ ทำงานใช้เวลา c2 มิลลิวินาที ทำงานรวม n ครั้ง
    ให้ x ← x + A[i]    ▷ ▷ ▷ ทำงานใช้เวลา c3 มิลลิวินาที ทำงานรวม n ครั้ง
คืนค่า x เป็นคำตอบ    ▷ ▷ ▷ ทำงานใช้เวลา c4 มิลลิวินาที ทำงานรวม 1 ครั้ง
```

[1]ในหนังสือเล่มนี้ เราจงใจทับศัพท์คำว่า implement ด้วย "อิมพลีเมนท์" และ implementation ด้วย "อิมพลีเมนเทชัน" เนื่องจากไม่มีคำภาษาไทยที่มีความหมายตรงกับคำดังกล่าวในความหมายที่เราใช้

กรณี	c_1	c_2	c_3	c_4	เวลา (มิลลิวินาที)	เวลาประมาณ (ที่เข้าใจง่ายขึ้น)
1	1	1	1	1	202	0.2 วินาที
2	100	1	1	1	301	0.3 วินาที
3	10,000	1	1	1	10,201	10 วินาที
4	100	1	1	100,000	100,300	2 นาที
5	100	1,000	10,000	100	1,100,200	18 นาที
6	1	20,000	1	100	2,000,201	ครึ่งชั่วโมง
7	10	10	100,000	100,000	10,101,010	สามชั่วโมง

รูปที่ 2.1: เวลาการทำงานหลากหลายที่เป็นไปได้เมื่อ $n = 100$

กรณี	100	200	400	1,000	10,000	100,000	1,000,000
1	202	402	802	2,002	20,002	200,002	2,000,002
2	301	501	901	2,101	20,101	200,101	2,000,101
3	10,201	10,401	10,801	12,001	30,001	210,001	2,010,001
4	100,300	100,500	100,900	102,100	120,100	300,100	2,100,100
5	1,100,200	2,200,200	4,400,200	11,000,200	110,000,200	1,100,000,200	11,000,000,200
6	2,000,201	4,000,301	8,000,501	20,001,101	200,010,101	2,000,100,101	20,001,000,101
7	10,101,010	20,102,010	40,104,010	100,110,010	1,000,200,010	10,001,100,010	100,010,100,010

รูปที่ 2.2: เวลาการทำงานในแต่ละกรณีเมื่อปรับค่า n

เมื่อเราวิเคราะห์โดยละเอียดแล้ว เราจะได้ว่าเวลาการทำงานเป็นมิลลิวินาทีคือ

$$c_1 + c_2 \cdot n + c_3 \cdot n + c_4 = c_1 + c_4 + (c_3 + c_4)n$$

ถ้าเราประมวลผลกับข้อมูลจำนวน 100 ตัว (นั่นคือ $n = 100$) ตัวแปรทั้ง 4 ก็ทำให้ได้เวลาในการประมวลผลที่แตกต่างกันมากมาย ดังตารางในรูป 2.1

จากนิพจน์ง่าย ๆ $2n + 2$ เมื่อเราพยายามใส่รายละเอียดแล้วคำนวณเป็นเวลา กลับกลายเป็นเวลาที่แตกต่างกันได้มากมาย เมื่อเราปรับค่าสมมติให้เป็นค่าต่าง ๆ แน่นอนว่าหลาย ๆ กรณีในตารางอาจจะเป็นกรณีที่เป็นไปไม่ได้จริง ๆ แต่ถ้าเราตั้งใจจะสมมติและละทิ้งรายละเอียดบางอย่างแล้ว เราก็ต้องเตรียมตัวรับผลความคลาดเคลื่อนที่อยู่ในกรอบของการสมมติของเราเช่นเดียวกัน

แม้ตัวอย่างจะเป็นอัลกอริทึมง่าย ๆ ผลของการวิเคราะห์แทบจะบอกอะไรเกี่ยวกับเวลาการทำงานสำหรับค่า n คงที่ค่าหนึ่งไม่ได้เลย เพราะความเปลี่ยนแปลงของเครื่องคอมพิวเตอร์นำอัลกอริทึมไปใช้งาน รวมถึงภาษาโปรแกรมที่เขียน

อย่างไรก็ตาม ค่าคงที่ $c_1, \ldots, c_4$ นั้น ไม่เปลี่ยนแปลง ถ้าเราไม่เปลี่ยนเครื่องคอมพิวเตอร์ที่ใช้งาน ดังนั้นเราจะทดลองแทนค่าเวลาที่คำนวณได้ กับค่า n ต่าง ๆ กันแทน เราจะได้ผลดังตารางในรูปที่ 2.2

ถ้าเราสังเกตค่าในตารางให้ดี เราจะเห็นแนวโน้มบางอย่าง สังเกตการเปลี่ยนแปลงของค่าในทุก ๆ มีลักษณะคล้ายกัน เพื่อให้เห็นค่าชัดเจน เราจะคำนวณอัตราส่วนของค่าในแต่ละคอลัมน์กับค่าในคอลัมน์ก่อนหน้า เราจะได้ตาราง

กรณี	200	400	1,000	10,000	100,000	1,000,000
อัตราการเปลี่ยนของ n	2.000	2.000	2.500	10.000	10.000	10.000
1	1.990	1.995	2.496	9.991	9.999	10.000
2	1.664	1.798	2.332	9.567	9.955	9.995
3	1.020	1.038	1.111	2.500	7.000	9.571
4	1.002	1.004	1.012	1.176	2.499	6.998
5	2.000	2.000	2.500	10.000	10.000	10.000
6	2.000	2.000	2.500	10.000	10.000	10.000
7	1.990	1.995	2.496	9.991	9.999	10.000

รูปที่ 2.3: อัตราส่วนของเวลาการทำงานเมื่อปรับค่า n เทียบกับการเปลี่ยนค่า n

ในรูปที่ 2.3

สังเกตว่าในทุก ๆ แถว การเปลี่ยนแปลงของเวลาใกล้เคียงกับการเปลี่ยนแปลงของ n ยกเว้นแถวที่ 4 ที่นิพจน์ของเวลาคือ $2n + 10001$ มิลลิวินาที อย่างไรก็ตาม ถ้าเราเพิ่มค่า n มากขึ้นเรื่อย ๆ ความแตกต่างดังกล่าวก็จะค่อย ๆ ลดลง

$\triangleright$ **คำถาม 2.1** ค่า n ที่มากพอ
ให้คำนวณหาค่า n ที่ทำให้อัตราส่วนของเวลาการทำงานในกรณีที่ 4 เมื่อข้อมูลมีจำนวน n กับเมื่อกรณีที่ข้อมูลมี $2n$ มีค่ามากกว่า 1.99 (นั่นคือใกล้เคียงกับอัตราส่วน $2n/n = 2$) $\triangleleft$

จากการทดลองแทนค่าดังกล่าว เราพบว่าจากนิพจน์ $2n + 2$ ที่เราวิเคราะห์ไป ส่วนที่ใช้ได้จริง ๆ คือพจน์ n เท่านั้น

นอกจากความแตกต่างด้านค่าคงที่ของเวลาการทำงานของอัลกอริทึมระหว่างเครื่องคอมพิวเตอร์ต่าง ๆ แล้ว การวิเคราะห์ของเราที่ผ่านมายังไม่ได้พิจารณาผลที่เกิดจากการอิมพลีเมนท์แต่อย่างใด เราจะทดลองเพิ่มความละเอียดในการวิเคราะห์โดยพิจารณาไปถึงโปรแกรมที่เขียนเป็นภาษา C++ ดังด้านล่าง

```
int x = 0;
for(int i = 0; i < n; ++i)
  x += A[i];
return x;
```

ถ้าเราต้องการจะนับเวลาการทำงานจริง ๆ เราจะพบว่าคำสั่ง `for` จริง ๆ แล้ว จะทำงานรวม $n + 1$ ครั้ง แทนที่จะเป็น n ครั้ง เนื่องจากในรอบสุดท้ายจะต้องมีการเปรียบเทียบค่า `i` กับ `n` อีกหนึ่งครั้ง ดังนั้นถ้าเราสมมติให้ทุกคำสั่งทำงานด้วยเวลาที่เท่ากันเราจะได้ว่าเวลาการทำงานของเราเปลี่ยนไปเป็น $2n+3$ หรือถ้าเราพิจารณาละเอียดขึ้นอีก เราจะพบว่าคำสั่ง `for` จริง ๆ แล้ว ประกอบด้วยคำสั่งย่อย ๆ อีก 3 คำสั่ง คำสั่งแรกที่กำหนดค่าเริ่มต้นให้กับ `i` ทำงานแค่ 1 ครั้ง คำสั่งเปรียบเทียบทำงาน n ครั้ง และคำสั่งปรับค่าทำงาน $n - 1$ ครั้ง ถ้าเราคิดรวมคำสั่งเหล่านี้ไปด้วย เวลาที่ได้จากการวิเคราะห์ของเราจะเป็น $3n + 3$ หรือถ้าเราพยายามคิดเวลาของแต่ละคำสั่งเพิ่มเติมเข้าไป เราจะต้องสมมติตัวแปรเพิ่มอีก 3 ตัวและจะทำให้การคำนวณต่าง ๆ ยุ่งยากมากขึ้น

อย่างไรก็ตาม ไม่ว่าเราจะวิเคราะห์ได้ $2n + 2$, $2n + 3$, หรือ $3n + 3$ หรือจะเป็น $c_1 + c_4 + (c_3 + c_4)n$ ก็ตาม เมื่อเราไม่ทราบค่าคงที่ของเวลาการทำงานต่าง ๆ ที่ขึ้นกับเครื่องคอมพิวเตอร์และสถานะต่าง ๆ ของเครื่องในเวลานั้น ความแม่นยำของการวิเคราะห์ที่เราได้เพิ่มขึ้นมานั้น ก็ไม่ได้ทำให้เราทำนายเวลาการทำงานได้แม่นยำขึ้นแต่

อย่างใด จากการทดลองแทนค่าที่เราได้ทำมา ทุก ๆ นิพจน์แสดงเวลาการทำงานที่เราวิเคราะห์ได้นั้น มีความหมายไม่ต่างจาก n ในแง่ของการเปลี่ยนแปลงของเวลาการทำงานเมื่อขนาดข้อมูลเพิ่มขึ้น นั่นคือ ถ้าขนาดของข้อมูลเพิ่มขึ้น c เท่า เวลาที่ใช้ในการทำงานก็จะเพิ่มขึ้นประมาณ c เท่าด้วยเช่นกัน

หาคู่ข้อมูล

เราจะทดลองวิเคราะห์อีกอัลกอริทึมหนึ่งเพื่อยืนยันข้อสรุปที่เราได้มา พิจารณาโจทย์ปัญหาด้านล่างนี้

โจทย์ปัญหา 2.1 ทำรั้วพอดี
สมศักดิ์พบกับปัญหาน้ำท่วม ทำให้เขาคิดจะซื้อรั้วบ้านสำเร็จมาใช้ รั้วมีทั้งหมด n แบบ แบบที่ i มีความยาว a_i เมตร และราคา c_i บาท ความยาวรั้วที่เขาต้องการคือ K เมตรและเขาไม่ต้องการนำรั้วสำเร็จมาตัด นอกจากนี้ เพื่อลดภาระในการต่อรั้วสำเร็จเข้าด้วยกัน เขาจะซื้อรั้วสำเร็จไม่เกิน 2 อัน (แต่อาจจะซื้ออันเดียวก็ได้) จงหาวิธีการซื้อที่ทำให้สมศักดิ์ประหยัดเงินมากที่สุด

สมมติว่าข้อมูลป้อนเข้าอยู่ในอาร์เรย์ $A = [a_0, a_1, \ldots, a_{n-1}]$ และ $C = [c_0, c_1, \ldots, c_{n-1}]$ เราจะพัฒนาอัลกอริทึมเพื่อแก้ปัญหานี้

▷ **คำถาม 2.2**
ก่อนจะอ่านต่อไป ลองคิดว่าจะแก้ปัญหาดังกล่าวได้อย่างไร พร้อมทั้งเขียนอัลกอริทึมและพยายามวิเคราะห์เวลาการทำงาน เป็นฟังก์ชันที่ขึ้นกับ n ◁

ปัญหาดังกล่าว เราสามารถพิจารณาคำตอบว่ามีสองประเภทคือกรณีที่ใช้รั้วสำเร็จอันเดียว และกรณีที่ใช้รั้วสองอัน เราสามารถเขียนอัลกอริทึมเพื่อแก้ปัญหานี้ได้ดังอัลกอริทึม A2.2 ที่แสดงในรูปที่ 2.4 อัลกอริทึมดังกล่าวทดลองคำตอบที่เป็นไปได้ทุกแบบและเลือกคำตอบที่ใช้เงินน้อยที่สุด

วงรอบ FOR ชุดแรกพิจารณากรณีที่ใช้รั้วสำเร็จอันเดียว อีกชุดสำหรับกรณีที่ใช้รั้วสำเร็จสองอัน สังเกตว่าอัลกอริทึมดังกล่าวจะทำงานโดยมีจำนวนรอบเท่า ๆ กันเสมอ แต่อาจจะใช้เวลาต่างกันเปลี่ยนไปตามข้อมูล เนื่องจากอาจจะเข้าไปทำงานคำสั่งภายในคำสั่ง IF ต่างกัน อย่างไรก็ตาม อย่างที่เราได้กล่าวไว้ในบทที่ 1 เรามักจะสนใจในกรณีที่แย่ที่สุด ซึ่งในกรณีของอัลกอริทึมนี้เราจะสมมติให้อัลกอริทึมเข้าไปทำงานในทุก ๆ คำสั่งภายในคำสั่ง IF

ในกรณีนี้ เราจะเริ่มโดยการวิเคราะห์เวลาในการทำงานของอัลกอริทึม โดยสมมติให้ทุกคำสั่งใช้เวลาในการทำงาน 1 ครั้งเท่ากัน เราจะแบ่งพิจารณาดังนี้

- คำสั่งในบรรทัดที่ 1,2 และ 12 ทำงานรวมกันใช้เวลา 3 หน่วย (เพราะว่าไม่ได้อยู่ภายในวงรอบอะไรเลย)
- คำสั่งในบรรทัดที่ 3 - 6 แต่ละคำสั่งทำงาน n รอบ ดังนั้นใช้เวลารวมทั้งสิ้น $4n$ หน่วย
- เช่นเดียวกับการวิเคราะห์บรรทัดที่ 3, บรรทัดที่ 7 ทำงาน n รอบ ดังนั้นใช้เวลารวมทั้งสิ้น n หน่วย
- สำหรับวงรอบใน (ตัวแปร j ในบรรทัดที่ 8 - 11) เราจะทดลองแทนค่าดูเพื่อทำความเข้าใจกับอัลกอริทึมนี้ก่อน ในรอบแรกที่ $i = 0$ ตัวแปร j จะมีค่าตั้งแต่ 1 ถึง $n - 1$ นั่นคือวงรอบนี้ทำงาน $n - 1$ รอบ ในรอบที่สองที่ $i = 1$ ตัวแปร j จะมีค่าตั้งแต่ 2 ถึง $n - 1$ นั่นคือทำงาน $n - 2$ รอบ อัลกอริทึมจะทำงานเช่นนี้ไปเรื่อย ๆ จนกระทั่ง $i = n - 2$ ตัวแปร j จะมีค่าเดียวคือ $n - 1$ ถ้าเรานับรวมจำนวนรอบทั้งหมด เราจะได้

เลือกรั้วประหยัดที่สุด	(A2.2)
1. $mcost \leftarrow \infty$	(c_1 milliseconds (ms))
2. $sol \leftarrow$ "Impossible"	(c_1 ms)
3. FOR $i \leftarrow 0, 1, \ldots, n-1$	(c_2 ms)
4. IF $a[i] = K$ and $c[i] \leq mcost$ THEN	(c_3 ms)
5. $mcost \leftarrow c[i]$	(c_1 ms)
6. $sol \leftarrow$ "Buy fence i"	(c_1 ms)
7. FOR $i \leftarrow 0, 1, \ldots, n-1$	(c_2 ms)
8. FOR $j \leftarrow i+1, i+2, \ldots, n-1$	(c_2 ms)
9. IF $(A[i] + A[j] = K)$ and $(c[i] + c[j] < mcost$ THEN	(c_3 ms)
10. $mcost \leftarrow c[i] + c[j]$	(c_1 ms)
11. $sol \leftarrow$ "Buy fence i and j"	(c_1 ms)
12. RETURN sol	(c_4 ms)

รูปที่ 2.4: อัลกอริทึม A2.2 สำหรับปัญหาทำรั้วพอดี พร้อมระบุค่าคงที่ของเวลาที่ใช้ในการวิเคราะห์แบบละเอียด

เท่ากับ

$$(n-1) + (n-2) + (n-3) + \cdots + 1 = n(n-1)/2$$

นั่นคือ บรรทัดที่ 8 - 11 แต่ละบรรทัดจะทำงานทั้งสิ้น $n(n-1)/2$ รอบ ดังนั้นคำสั่งในบรรทัดเหล่านี้ทำงานรวมกัน $4 \cdot n(n-1)/2 = 2n(n-1)$ หน่วย

เมื่อรวมทั้งหมด เราจะได้ว่าถ้าทุก ๆ คำสั่งทำงานโดยใช้เวลา 1 หน่วยเท่ากันหมด อัลกอริทึมจะทำงานโดยใช้เวลา

$$3 + 4n + n + 2n(n-1) = 2n^2 + 3n + 3$$

หน่วย

ในรูป 2.4 เราได้ใส่รายละเอียดของเวลาในแต่ละคำสั่ง ในการวิเคราะห์เช่นเดียวกับในกรณีของอัลกอริทึม **??** อย่างไรก็ตาม เพื่อความกระชับเราจะไม่กำหนดให้ทุกคำสั่งทำงานต่างกันหมด แต่จะกำหนดให้เวลาในการทำงานของคำสั่งที่คล้ายกันเท่ากัน

เมื่อเพิ่มค่าคงที่ดังกล่าวในการวิเคราะห์ เราจะได้ว่า เวลาในการทำงานคือ $2c_1 + 4n(c_2 + c_3 + 2c_1) + c_2 n + 2n(n-1)(c_2 + c_3 + 2c_1) + c_4$ หรือมีค่าเท่ากับ

$$2(c_2 + c_3 + 2c_1)n^2 + (3c_2 + 2c_3 + 4c_1)n + 2c_1 + c4$$

เราทดลองเปลี่ยนค่าของ c_1, c_2, c_3 และ c_4 พร้อม ๆ กับดูแนวโน้มของเวลาที่ใช้ที่เปลี่ยนไปเมื่อ n มีค่าเป็น 100, 200, 400, 800, 1,600, 16,000, และ 160,000 ตามลำดับ ผลที่ได้แสดงในรูปที่ 2.5 สังเกตว่าบรรทัดแรกเมื่อ $c_1 = c_2 = c_3 = c_4 = 1$ เวลาการทำงานมีค่าเป็น $2n^2 + 3n + 3$ เท่ากับเวลาที่เราได้วิเคราะห์ไว้กรณีทุกคำสั่งทำงานด้วยเวลาที่เท่ากันหมด

				n						
c1	c2	c3	c4	100	200	400	800	1,600	16,000	160,000
1	1	1	1	80901	321801	1283601	5127201	20494401	2048144001	204801440001
0.1	1	0.1	1	26361	104721	417441	1666881	6661761	665657601	66560576001
0.01	0.01	0.01	100	909	3318	12936	51372	205044	20481540	2048014500
0.01	0.01	0.01	10000	10809	13218	22836	61272	214944	20491440	2048024400
0.001	0.001	0.001	100000	100080.9	100321.8	101283.6	105127.2	120494.4	2148144	204901440

รูปที่ 2.5: เวลาการทำงานในแต่ละชุดของค่าคงที่เมื่อปรับค่า n

กรณี	200	400	800	1,600	16,000	160,000
n	2.000	2.000	2.000	2.000	10.000	10.000
n2	4.000	4.000	4.000	4.000	100.000	100.000
1 (หรือ $2n^2 + 3n + 3$)	3.978	3.989	3.994	3.997	99.937	99.994
2	3.973	3.986	3.993	3.997	99.922	99.992
3	3.650	3.899	3.971	3.991	99.889	99.993
4	1.223	1.728	2.683	3.508	95.334	99.945
5	1.002	1.010	1.038	1.146	17.828	95.385

รูปที่ 2.6: อัตราส่วนของเวลาการทำงานเมื่อปรับค่า n เทียบกับการเปลี่ยนค่า n

เราได้เปรียบเทียบอัตราส่วนของค่าในคอลัมน์ติดกันและแสดงพร้อมอัตราส่วนของการเปลี่ยนแปลงของค่า n, n^2, และ $2n^2 + 2n + 3$ ในตารางในรูปที่ 2.6

สังเกตว่าในกรณีนี้ อัตราการเปลี่ยนแปลงของเวลาที่ใช้ จะใกล้เคียงกับการเปลี่ยนแปลงของฟังก์ชัน n^2 แทนที่จะเป็น n เหมือนกับในกรณีก่อน

▷ **คำถาม 2.3**
ทำไมในกรณีนี้อัตราการเปลี่ยนแปลงคล้ายกับ n^2 มากกว่า n? ◁

ดังนั้น สิ่งที่เราได้แน่นอนจากการวิเคราะห์ แทนที่จะเป็นเวลาที่สามารถประมาณได้ กลับเป็นข้อมูลของอัตราการเปลี่ยนแปลงของเวลาที่อัลกอริทึมใช้ในการทำงาน ในกรณีนี้เราทราบว่าเวลาการทำงานเมื่อเทียบกับขนาดของข้อมูลแปรผันใกล้เคียงกับฟังก์ชัน n^2

การที่วิธีการนี้ใช้เวลาแปรผันกับขนาดข้อมูลเป็น n^2 หมายความว่า ถ้าข้อมูลมีขนาดใหญ่มาก ๆ อัลกอริทึมดังกล่าวจะใช้เวลามากขึ้นไปด้วย และอาจจะใช้เวลานานเกินกว่าที่เราจะรอผลลัพธ์ได้ (เช่น นานหลายปี) เมื่อเราศึกษาเทคนิคการค้นข้อมูลมากขึ้นแล้ว เราจะสามารถแก้ปัญหานี้ได้โดยใช้เวลาที่แปรผันไม่ต่างจากฟังก์ชัน n มากนัก

จากตัวอย่างที่กล่าวมาแล้ว เราพบว่าแม้ว่าเราจะวิเคราะห์เวลาการทำงานเป็นได้ $2n + 2$, $2n + 3$, $3n + 3$ สิ่งที่เราสามารถกล่าวได้อย่างชัดเจนเกี่ยวกับเวลาการทำงาน ก็คือเวลาที่ใช้มีอัตราการเปลี่ยนแปลงเหมือนกับฟังก์ชัน n และในตัวอย่างนี้ ไม่ว่าเราจะวิเคราะห์เวลาการทำงานแบบคร่าว ๆ ได้เป็น $2n^2 + 3n + 3$ หรือจะพยายามวิเคราะห์ให้ละเอียดขึ้นเป็น $2(c_2 + c_3 + 2c_1)n^2 + (3c_2 + 2c_3 + 4c_1)n + 2c_1 + c4$ สิ่งที่เรารู้อย่างชัดเจนจากการวิเคราะห์ก็คือเวลาที่ใช้มีลักษณะการเปลี่ยนแปลงเหมือนกับฟังก์ชัน n^2

โปรแกรมที่ 2.1: ฟังก์ชัน calA และ calB ที่คำนวณผลรวมของข้อมูลในอาร์เรย์สองมิติ

```
int calA()
{
  int x = 0;
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
      x += ar[i][j];
  return x;
}
int calB()
{
  int x = 0;
  for(int j = 0; j < n; ++j)
    for(int i = 0; i < n; ++i)
      x += ar[i][j];
  return x;
}
```

อย่างไรก็ตาม สิ่งที่เรากล่าวมานั้นยังไม่ชัดเจนทางคณิตศาสตร์มากพอ แม้ว่าเราจะพอนึกภาพได้ แต่เรายังไม่สามารถกล่าวอย่างชัดเจนว่า เมื่อเรากล่าวว่าฟังก์ชันหนึ่งมีอัตราการเปลี่ยนแปลงเหมือนกับอีกฟังก์ชันหนึ่ง แท้จริงแล้วมีความหมายอย่างไร ในส่วน 2.1.2 เราจะศึกษานิยามเกี่ยวกับอัตราการโตของฟังก์ชันที่จะทำให้เรากล่าวถึงแนวคิดนี้ได้อย่างเป็นรูปแบบ

ในส่วนที่เหลือนี้ เราจะยกตัวอย่างความคลาดเคลื่อนในการวิเคราะห์เวลาการทำงาน ที่เป็นผลจากสถาปัตยกรรมของคอมพิวเตอร์ที่โปรแกรมทำงาน

ตัวอย่าง: ผลจากการเปลี่ยนรูปแบบการอ่านค่าในหน่วยความจำ

พิจารณาฟังก์ชันสองฟังก์ชันที่คำนวณผลรวมของข้อมูลในอาร์เรย์สองมิติ ในโปรแกรม 2.1

▷ **คำถาม 2.4**
ฟังก์ชัน calA กับ calB ต่างกันอย่างไร? เมื่อนำไปทำงาน เวลาที่ใช้ทำงานจะแตกต่างกันหรือไม่? ถ้าคิดว่าแตกต่าง ฟังก์ชันใดน่าจะทำงานเร็วกว่า ทำไมถึงน่าจะเป็นเช่นนั้น? ◁

เมื่อนำฟังก์ชันไปทดลอง เนื่องจากต้องมีการกำหนดค่าเริ่มต้นให้กับอาร์เรย์ เราจึงให้โปรแกรมกำหนดค่าเริ่มต้นให้อาร์เรย์ จากนั้นเรียกฟังก์ชันดังกล่าว 10 รอบ เพื่อลดผลของเวลาจากการกำหนดค่าเริ่มต้น

ค่าเฉลี่ยของเวลาการทำงาน เมื่อ n มีค่าเท่ากับ 10,000 ของโปรแกรมที่ใช้ฟังก์ชัน calA คือ 4.55 วินาที ในขณะที่โปรแกรมที่ใช้ฟังก์ชัน calB คือ 11.54 วินาที คิดเป็นประมาณ 2.5 เท่าของเวลาของโปรแกรมที่ใช้ฟังก์ชัน calA

ความแตกต่างดังกล่าว เกิดขึ้นเนื่องจากรูปแบบของการอ่านข้อมูลในหน่วยความจำที่ต่างกัน มีผลต่อประสิทธิภาพของระบบหน่วยความจำ (ในที่นี้คือระบบ cache) ซึ่งการที่จะวิเคราะห์ให้ละเอียดระดับนี้ได้ จะต้องใช้ข้อมูลและความรู้เกี่ยวกับเครื่องคอมพิวเตอร์ที่จะนำโปรแกรมไปทำงาน และการปรับโปรแกรมให้ทำงานได้ดีในลักษณะนี้ แม้จะเป็นงานเชิงวิศวกรรมที่น่าสนใจ แต่ก็อยู่นอกขอบเขตของหนังสือเล่มนี้

สังเกตว่า ในการวิเคราะห์เวลาการทำงานโดยใช้วิธีที่เรากล่าวมาแล้ว เวลาการทำงานของทั้งสองฟังก์ชันจะมีค่าเท่า

กัน แต่เวลาที่ใช้งานจริง กลับแตกต่างกันมาก ฟังก์ชันดังกล่าวเป็นตัวอย่างหนึ่งของความคลาดเคลื่อนของการวิเคราะห์ที่เป็นผลมาจากสถาปัตยกรรมของคอมพิวเตอร์และรูปแบบการเก็บข้อมูลของภาษาโปรแกรมที่ใช้

2.1.2 สัญกรณ์โอใหญ่ (Big-O)

เป้าหมายหลักของการวิเคราะห์เวลาการทำงานของอัลกอริทึม นอกจากเพื่อจะพิจารณาความเป็นไปได้ที่อัลกอริทึมจะมีประสิทธิภาพพอแล้ว ยังทำไปเพื่อเลือกอัลกอริทึมที่เหมาะสมที่สุดที่จะนำมาอิมพลีเมนท์ด้วย ดังนั้นในการวิเคราะห์เราต้องการให้อัลกอริทึมที่มีประสิทธิภาพแตกต่างกัน มีเวลาที่วิเคราะห์ได้แตกต่างกันและสามารถเปรียบเทียบกันได้

▷ **คำถาม 2.5**
จากฟังก์ชันของเวลาที่วิเคราะห์ได้ด้านล่าง พิจารณาว่าฟังก์ชันใดเปรียบเทียบกันได้อย่างมีความหมายในแง่ของประสิทธิภาพ

$$2n, \quad n^3 + 4n, \quad 10n^2, \quad 0.5n + 100, \quad 3n^2 + 1000n$$

◁

จากตัวอย่างในเนื้อหาส่วนที่แล้ว เราพบว่าฟังก์ชันของเวลาที่เราวิเคราะห์ได้นั้น ค่าคงที่ที่ติดมาในนิพจน์นั้นไม่มีผลชัดเจนในการระบุความเร็วหรือช้าของอัลกอริทึม นอกจากนี้ ถ้าฟังก์ชันที่ได้มีหลายพจน์ พจน์ที่มีกำลังต่ำกว่าจะมีผลในเวลาการทำงานน้อยลง ถ้าข้อมูลมีขนาดใหญ่ขึ้นเรื่อย ๆ จากข้อสังเกตทั้งสอง ในการเปรียบเทียบเราสามารถตัดส่วนของนิพจน์ที่ไม่สำคัญออกได้ โดยฟังก์ชันตัวอย่างในคำถามข้างต้น เมื่อตัดค่าคงที่และพจน์ที่มีกำลังต่ำออกไปแล้ว จะเป็นดังด้านล่าง

$$n, \quad n^3, \quad n^2, \quad n, \quad n^2$$

ซึ่งฟังก์ชันเหล่านี้จะบอกแนวโน้มของการโตของเวลาการทำงาน

ก่อนที่เราจะเสนอนิยามของการโตอย่างเป็นทางการ เราจำเป็นต้องย้ำอีกทีว่า การที่เราละทิ้งหรือตัดค่าคงที่ออกไปนั้นไม่ได้หมายความว่าค่าคงที่ต่าง ๆ ไม่มีผลต่อเวลาการทำงาน หลาย ๆ อัลกอริทึมที่วิเคราะห์ออกมาพบว่าประสิทธิภาพดี แต่เมื่อทำงานจริงกลับช้ากว่าอัลกอริทึมที่วิเคราะห์ออกมาได้ว่ามีประสิทธิภาพต่ำกว่า ทั้งนี้เนื่องจากค่าคงที่ต่าง ๆ ที่ผู้วิเคราะห์ได้ละทิ้งออกจากการพิจารณา ทั้งนี้โดยมากเป็นเพราะว่าขนาดของข้อมูลป้อนเข้ามีขนาดไม่ใหญ่พอ ที่ผลของอัตราการโตจะชนะผลของค่าคงที่ เราจะได้เห็นตัวอย่างที่ชัดเจนขึ้นต่อไป

▷ **คำถาม 2.6**
พิจารณาคู่ของฟังก์ชันต่อไปนี้ ให้หาค่า n ที่รับประกันว่าฟังก์ชันแรกจะมีค่าน้อยกว่าฟังก์ชันที่สอง (1) 10000 กับ n, (2) $1000n$ กับ n^2, (3) $5000n$ กับ $2n^2 - 100n$, และ (4) $10000n$ กับ $0.01n^3$ ◁

ในการที่เราจะบอกว่าฟังก์ชันหนึ่ง มีขอบเขตบนของการโตเป็นอะไรเราจะใช้สัญกรณ์โอใหญ่ (big-O notation) โดยตัวอย่างของการใช้สัญกรณ์ดังกล่าว เช่น $n^2 + 100n$ เป็น $O(n^2)$ หรือ $30n + 5$ เป็น $O(n)$

เราจะหาคำนิยามที่สะท้อนสิ่งที่เราได้ทดลองมาในการที่จะระบุว่าฟังก์ชัน $f(n)$ เป็น $O(g(n))$ (หรือมีอัตราการโตไม่เกินฟังก์ชัน $g(n)$)

- ข้อสังเกตข้อแรกก็คือ ค่าคงที่ไม่มีผล นั่นคือฟังก์ชัน $100n$ หรือ $1000n$ หรือกระทั่ง $0.0001n$ ก็เป็น $O(n)$ เหมือนกัน ดังนั้นในการจะกล่าวว่า $f(n)$ เป็น $O(g(n))$ เราต้องยอมให้มีการใช้ค่าคงที่มา "ช่วย" ฟังก์ชัน $g(n)$ ให้มีค่ามากขึ้นจนเปรียบเทียบกับ $f(n)$ ได้

- ข้อสังเกตข้อที่สองคือ พจน์ที่มีกำลังต่ำ ๆ จะไม่มีผล การที่จะทำให้พจน์ดังกล่าวลดความสำคัญลงไปนั้น เราจะต้องวิเคราะห์ฟังก์ชันเมื่อ n มีค่ามาก ๆ เพื่อให้พจน์ที่สำคัญน้อยมีผลต่อค่าของฟังก์ชันลดลง

▷ **คำถาม 2.7** ทดลองสร้างคำนิยาม
จากข้อสังเกตสองข้อข้างต้น ทดลองเขียนคำนิยามของสัญกรณ์โอใหญ่:
เราจะกล่าวว่า $f(n)$ เป็น $O(g(n))$ เมื่อ .... ◁

พิจารณากรณีที่เราต้องการแสดงว่า $3n + 100$ เป็น $O(n)$ (นั่นคือในที่นี้ $f(n) = 3n + 100$ และ $g(n) = n$ และเราต้องการแสดงว่า $f(n)$ เป็น $O(g(n))$) สังเกตว่า

$$3n + 100 > n,$$

สำหรับทุก ๆ $n \geq 0$ ดังนั้นเราจำเป็นจะต้องเพิ่มค่าคงที่ให้กับฟังก์ชัน $g(n)$ อย่างไรก็ตามถ้าเราคูณ $g(n)$ ด้วย 3 ก็ยังไม่เพียงพอเพราะว่า

$$3n + 100 > 3 \cdot n,$$

อยู่ดี ดังนั้นเราจะคูณ $g(n)$ ด้วย 4 อย่างไรก็ตาม อสมการ

$$3n + 100 \leq 4 \cdot n$$

ก็ยังไม่เป็นจริงสำหรับทุก ๆ ค่า n ที่มากกว่าหรือเท่ากับศูนย์

▷ **คำถาม 2.8** ขีดจำกัดล่าง
ค่า n จะต้องมีค่าอย่างน้อยเท่าใด อสมการ $3n + 100 \leq 4n$ จึงจะเป็นจริง? ◁

จากตัวอย่างข้างต้น เราใช้ค่าคงที่สองตัวคือ c เป็นค่าคงที่สำหรับเป็นตัวคูณของฟังก์ชัน $g(n)$ และ n_0 เพื่อระบุขอบเขตล่างของขนาดข้อมูลที่เราสนใจ เราจะนิยามสัญกรณ์โอใหญ่อย่างเป็นทางการดังนี้:

เราจะกล่าวว่าฟังก์ชัน $f(n)$ เป็น $O(g(n))$ ก็ต่อเมื่อ มีค่าคงที่ c และ n_0 ที่ $f(n) \leq c \cdot g(n)$ เมื่อ $n \geq n_0$

ในกรณีดังกล่าว บางครั้งเราก็จะกล่าวว่า $f(n)$ มีการโตเป็น $O(g(n))$ หรือเขียนว่า $f(n) = O(g(n))$ ซึ่งในกรณีหลังนี้ เครื่องหมายเท่ากับที่ใช้ ไม่ได้มีความหมายว่าเท่ากัน แต่เป็นการบิดสัญกรณ์เท่ากับให้เป็นความหมายตามที่เรานิยามมาแล้ว[2]

สำหรับตัวอย่างข้างต้นที่เราแสดงว่า $3n + 100$ เป็น $O(n)$ นั้น เราใช้ $c = 4$ และ n_0 ที่เป็นไปได้ก็คือ 100 อย่างไรก็ตาม เราสามารถเลือกคู่ของค่าคงที่ c และ n_0 แบบอื่น ๆ ได้อีก เช่น ถ้าเราให้ $c = 13$ เราอาจใช้ $n_0 = 10$ ก็เพียงพอ เพราะว่า

$$3n + 100 \leq 13n$$

เมื่อ $n \leq 10 = n_0$

คำถามทั้ง 3 คำถามต่อไปสำคัญต่อการทำความเข้าใจต่อไปในบทนี้

---

[2]ซึ่งถ้าจะให้ถูกต้องตามนิยามจริง ๆ เราจะต้องนิยามสัญกรณ์ $O(\cdot)$ ว่าเป็นเซตของฟังก์ชัน แล้วใช้เครื่องหมาย $\in$ แทนเครื่องหมายเท่ากับ เช่น เราจะเขียนว่า $3n + 100 \in O(n)$ แทนที่จะเขียน $3n + 100 = O(n)$ แต่ในหนังสือเล่มนี้เรายินดีจะบิดพริ้วนิยามของสัญกรณ์เท่ากับเพื่อความเรียบง่าย

▷ **คำถาม 2.9** ฟังก์ชันกำลังสอง
จงแสดงว่า $5n^2 + 100n + 1000$ เป็น $O(n^2)$ ◁

▷ **คำถาม 2.10** ฟังก์ชันที่ใหญ่กว่า
จงแสดงว่า $100n + 1000$ เป็น $O(n^2)$ ◁

▷ **คำถาม 2.11** ค่าคงที่
จงแสดงว่า สำหรับค่าคงที่ c ใด ๆ $c = O(1)$ ◁

สำหรับฟังก์ชันที่เป็นฟังก์ชันพหุนาม เรามีวิธีการที่จะระบุการโตได้โดยพิจารณาพจน์ที่มีกำลังมากที่สุด เราจะพิสูจน์ว่าวิธีดังกล่าวถูกต้อง

ทฤษฎีบท 1 *สำหรับฟังก์ชัน* $f(n) = a_d n^d + a_{d-1}n^{d-1} + a_{d-2}n^{d-2} + \cdots + a_1 n + a_0$ *เราจะได้ว่า* $f(n) = O(n^d)$

บทพิสูจน์: สังเกตว่าพจน์ที่มีสัมประสิทธิ์เป็นลบ จะไม่มีผลในการเปรียบเทียบกับ $c \cdot n^d$ ดังนั้นเราจะตัดออกไป โดยพิจารณาอีกฟังก์ชันหนึ่งที่ทุกสัมประสิทธิ์ไม่เป็นลบ

นิยามฟังก์ชัน $f'(n)$ โดยให้ $f'(n) = \sum_{i=0}^{d} a'_i n^i$ เมื่อ $a'_i = \max\{a_i, 0\}$ สังเกตว่า $f(n) \leq f'(n)$ เมื่อ $n \geq 0$ เสมอ

เราต้องเพิ่มสัมประสิทธิ์ให้กับ n^d เพื่อจะให้ฟังก์ชันนี้ใหญ่กว่าทุก ๆ พจน์ สังเกตว่า

$$a'_i n^i \leq a'_i n^d$$

เมื่อ $n \geq 0$, นั่นคือ

$$\begin{aligned} a'_d n^d + a'_{d-1} n^{d-1} + \cdots + a'_1 n + a'_0 &\leq a'_d n^d + a'_{d-1} n^d + \cdots + a'_1 n^d + a'_0 n^d \\ &= \left(a'_d + a'_{d-1} + \cdots + a'_1 + a'_0\right) n^d, \end{aligned}$$

เมื่อ $n \geq 0$ ดังนั้นเราจะเลือกให้ $n_0 = 0$ และ

$$c = \left(a'_d + a'_{d-1} + \cdots + a'_1 + a'_0\right)$$

เราจะได้ว่า

$$f(n) \leq f'(n) \leq c \cdot n^d,$$

เมื่อ $n \geq n_0 = 0$, นั่นคือ $f(n) = O(n^d)$ ตามต้องการ ∎

▷ **คำถาม 2.12** กำลังที่มากกว่า
ให้ $f(n) = O(n^d)$ จงแสดงว่า $f(n) = O(n^e)$ เมื่อ $e \geq d$ ◁

เนื่องจากโดยมากฟังก์ชันของเราจะเป็นฟังก์ชันที่ขึ้นกับ n เพื่อความสะดวกเราจะละอาร์กิวเมนท์ n ไว้ โดยเราจะเขียน $f(n)$ ด้วย f นอกจากนี้เรายังเขียนแทน $f(n) + g(n)$ ด้วย $f + g$

ด้านล่างเป็นคุณสมบัติอื่น ๆ ที่มีประโยชน์ในการนำสัญลักษณ์โอใหญ่ไปใช้ในการวิเคราะห์เวลาการทำงานของอัลกอริทึม

ทฤษฎีบท 2 *ถ้า* $f = O(h)$ *และ* $g = O(h)$ *แล้ว* $f + g = O(h)$

บทพิสูจน์: เนื่องจาก $f = O(h)$ และ $g = O(h)$ จึงมีค่าคงที่ c_1 และ n_1 ที่

$$f(n) \leq c_1 \cdot h(n)$$

เมื่อ $n \geq n_1$ และค่าคงที่ c_2 กับ n_2 ที่

$$g(n) \leq c_2 \cdot h(n)$$

เมื่อ $n \geq n_2$

ดังนั้น ถ้าให้ $c = c_1 + c_2$ และ $n_0 = \max\{n_1, n_2\}$ เราจะได้ว่า

$$f(n) + g(n) \leq c_1 \cdot h(n) + c_2 \cdot h(n) = c \cdot h(n)$$

เมื่อ $n \geq n_0$, นั่นคือ $f + g = O(h)$ ∎

▷ **คำถาม 2.13**
จงพิสูจน์ว่า ถ้า $f = O(g)$ และ $g = O(h)$ แล้ว $f = O(h)$ ◁

▷ **คำถาม 2.14**
เราสามารถคุณสมบัติสองคุณสมบัติข้างต้น ไปใช้ในการวิเคราะห์อัลกอริทึมในลักษณะใด? ◁

2.1.3 การวิเคราะห์เวลาการทำงานโดยใช้สัญกรณ์โอใหญ่

ที่ผ่านมาเราวิเคราะห์เวลาการทำงานอย่างละเอียดและได้ผลลัพธ์เช่น $2n + 2$ หรือเป็น $2n^2 + 2n + 4$ จากนั้นเราจึงใช้สัญกรณ์โอใหญ่ในการวิเคราะห์การโตและสรุปว่าเวลาการทำงานเป็น $O(n)$ หรือ $O(n^2)$ อย่างไรก็ตามถ้าเราต้องการวิเคราะห์แค่อัตราการโตของเวลาการทำงานในสัญกรณ์โอใหญ่ เราสามารถใช้สัญกรณ์โอใหญ่ระหว่างการนับเวลาการทำงานของแต่ละคำสั่งในอัลกอริทึมได้ด้วย

เราจะกลับมาวิเคราะห์เวลาการทำงานของอัลกอริทึม A2.2 (ในรูปที่ 2.4) โดยใช้สัญกรณ์โอใหญ่ตั้งแต่เริ่มต้น สังเกตว่าอัลกอริทึมนี้มีการทำงาน 4 ส่วนใหญ่ ๆ คือส่วนตอนแรกที่กำหนดค่าเริ่มต้น (บรรทัดที่ 1 และ 2), ส่วนคำนวณรั้วเดี่ยว, ส่วนคำนวณรั้วคู่, และส่วนคืนคำตอบ ในส่วนแรกและส่วนสุดท้าย ทำงานในเวลาคงที่ ดังนั้นจะได้ว่าทำงานในเวลา $O(1)$

สำหรับส่วนรั้วเดี่ยว การทำงานในบรรทัดที่ 4-6 นั้นแต่ละคำสั่งทำงานในเวลาคงที่ ดังนั้นใช้เวลา $O(1)$ การทำงานดังกล่าว ทำซ้ำจำนวน n รอบ ดังนั้นจะใช้เวลารวมเป็น $O(n)$

ในส่วนรั้วคู่นั้น ถ้าเราพิจารณาเฉพาะวงรอบภายใน เราพบว่าไม่ว่าค่า i จะเป็นเท่าใดจะทำงานไม่เกิน n รอบ นั่นคือการทำงานวนรอบ FOR ในบรรทัดที่ 7 แต่ละรอบ จะใช้เวลา $O(n)$ เราจะทำงานทั้งสิ้น n รอบ ดังนั้นจะใช้เวลา $O(n^2)$

ดังนั้นเวลารวมคือ $O(1) + O(n) + O(n^2)$ ซึ่งสามารถแสดงได้ไม่ยากว่าคือ $O(n^2)$

เราจะวิเคราะห์เวลาการทำงานของอีกอัลกอริทึมหนึ่ง

TODO: เขียนส่วนนี้ ให้มีการเปลี่ยนเทียบเวลาการทำงานของอัลกอริทึมสองอันที่เวลาไม่เท่ากัน แสดงกราฟ

ในส่วนถัดไปของบทนี้เราจะได้วิเคราะห์เวลาการทำงานของโครงสร้างข้อมูลหลายแบบที่พัฒนาบนอาร์เรย์ และในส่วนสุดท้ายเราจะได้พิจารณาตัวอย่างอัลกอริทึมที่มีเวลาการทำงานอื่น ๆ

2.2 แบบชนิดข้อมูลนามธรรมและการอิมพลีเมนท์ด้วยอาร์เรย์

แบบชนิดข้อมูลนามธรรม (abstract data types) ระบุวิธีการที่เราจะเข้าถึงและจัดการกับข้อมูลผ่านทางแบบชนิดข้อมูลนั้น ๆ โดยไม่จำเป็นต้องระบุว่าขั้นตอนดำเนินการจริง ๆ ของวิธีการเหล่านั้นเป็นอย่างไร

ในส่วนที่ 1.1.2 เราได้ศึกษาวิธีการนำอาร์เรย์ไปใช้เก็บข้อมูลที่มีลักษณะเป็นรายการ ในที่นี้ เมื่อเรากล่าวถึง "รายการ" เราจะมีภาพอยู่ว่าเราจะต้องดำเนินการอะไรกับรายการได้บ้าง นั่นคือวิธีการที่เราจะเข้าถึงและจัดการกับข้อมูลในรายการ ซึ่งจริง ๆ แล้ว เราอาจจะใช้โครงสร้างข้อมูลแบบอื่นในการอิมพลีเมนท์ไม่จำเป็นต้องเป็นอาร์เรย์ก็ได้

เราจะเรียกกระบวนการทั้งหมดที่เราสามารถเรียกใช้งานกับแบบชนิดข้อมูลนามธรรมว่าอินเทอร์เฟส (interface) ของแบบชนิดข้อมูลนั้น สำหรับข้อมูลแบบรายการที่เราได้พิจารณามาแล้ว จะมีอินเทอร์เฟสโดยทั่วไปดังนี้

- IsEmpty(L) ตรวจสอบว่ารายการ L ว่างเปล่าหรือไม่
- Find(L,x) - ค้นหาข้อมูล x ใน L แล้วคืนผลลัพธ์เป็นข้อมูลที่ทำให้อ้างถึง x ในรายการได้ (ซึ่งจะถูกใช้ในคำสั่ง Delete และ Insert ได้)
- Append(L,x) - เพิ่มข้อมูล x ท้ายรายการ L
- Delete(L,p) - ลบข้อมูลที่ถูกอ้างถึงโดย p
- InsertAfter(L,p,x) - เพิ่มข้อมูล x ต่อจากข้อมูลที่อ้างถึงโดย p

สังเกตว่า อินเทอร์เฟสดังกล่าว มีความแตกต่างจากรูปแบบที่เราใช้ในการประกาศฟังก์ชันที่ทำงานดังกล่าวเล็กน้อย ความแตกต่างนี้ไม่ใช่สิ่งสลักสำคัญมากนักเมื่อเราออกแบบอัลกอริทึม และการนำแบบชนิดข้อมูลไปใช้จริงก็มักจะต้องอ้างอิงกับการอิมพลีเมนท์ตลอดอยู่ดี อย่างไรก็ตาม ถ้าอินเทอร์เฟสของแบบชนิดข้อมูลนามธรรมนี้เหมือนกัน ไม่ว่าจะอิมพลีเมนท์ด้วยโครงสร้างข้อมูลใด ก็จะทำให้การเปลี่ยนประเภทโครงสร้างข้อมูลในโปรแกรมทำได้สะดวก ในบทที่ **??** เราจะศึกษาวิธีการพัฒนาโปรแกรมภาษา C++ เพื่อทำให้บรรลุจุดประสงค์นี้

ผู้อ่านที่สนใจอาจสงสัยว่าเมื่อมี InsertAfter ทำไมไม่มีการระบุกระบวนการ

- InsertBefore(L,p,x) - เพิ่มข้อมูล x ก่อนหน้าข้อมูลที่อ้างถึงโดย p

อยู่ในรายการอินเทอร์เฟส ในบทต่อ ๆ ไป เราจะพบว่าโครงสร้างข้อมูลที่จะรองรับทั้ง InsertAfter และ InsertBefore ได้พร้อม ๆ กันจะต้องมีภาระมากกว่าโครงสร้างข้อมูลที่รองรับแค่รูปแบบเดียวและเราจะได้พัฒนาโครงสร้างข้อมูลดังกล่าวด้วย อย่างไรก็ตามในบทนี้ เราจะยังไม่พิจารณากระบวนการดังกล่าว

ตารางในรูปที่ 2.7 แสดงเวลาที่เราเคยได้วิเคราะห์ไว้ในรูปของอัตราการโต เมื่อรายการมีข้อมูล n จำนวน

สำหรับรายการโดยทั่วไปแล้ว เราอาจจะมีกระบวนการอื่น ๆ อีก เช่น นำรายการที่สองไปต่อท้ายรายการแรก หรือนำรายการหนึ่งไปแทรกภายในอีกรายการหนึ่ง กระบวนการเหล่านี้สามารถอิมพลีเมนท์ด้วยอาร์เรย์ได้ไม่ยากนัก และโดยมากจะใช้เวลา $O(n)$ เมื่อ n แทนจำนวนข้อมูลที่อยู่ในรายการ

การวิ่งไปในชุดของข้อมูล

อินเทอร์เฟสที่เรากล่าวไปแล้วนั้นยังขาดกลุ่มกระบวนการเข้าถึงข้อมูลทุกตัวในรายการไปทีละตัว รูปแบบที่เราจะใช้จะสอดคล้องกับการใช้งานตัววิ่ง (iterator) ในไลบรารีมาตรฐานของ C++

กระบวนการ	เวลาการทำงานเมื่อพัฒนาด้วยอาร์เรย์
IsEmpty	$O(1)$
Find	$O(n)$
Append	$O(1)$
Delete	$O(n)$
InsertAfter	$O(n)$

รูปที่ 2.7: เวลาการทำงานของกระบวนการต่าง ๆ ของรายการ ที่พัฒนาด้วยอาร์เรย์

โปรแกรมที่ 2.2: โปรแกรมที่หาผลรวมโดยการใช้ตัววิ่งผ่านทางอินเทอร์เฟสเทียบกับการใช้พอยน์เตอร์

```
int list_sum_itr(int list[],           | int list_sum_ptr(int list[],
                 int size)              |                  int size)
{                                       | {
  int s = 0;                            |   int s = 0;
  for(list_itr i =                      |   for(int* i = &list[0];
        list_begin(list,size);          |
      i != list_end(list,size);         |       i != &list[size];
      i = list_itr_next(i)) {           |       i = i + 1) {
    s += list_itr_value(i);             |     s += *i;
  }                                     |   }
  return s;                             |   return s;
}                                       | }
```

ในการอ้างถึงข้อมูลทุกตัวในรายการ เราจะใช้ตัววิ่งที่มีลักษณะการทำงานเหมือนกับการใช้พอยน์เตอร์วิ่งไปในอาร์เรย์ สังเกตว่าในกรณีของอาร์เรย์ เรายังสามารถใช้ดัชนีในการไล่พิจารณาข้อมูลทุกตัวได้ แต่กับแบบชนิดข้อมูลอื่น ๆ การใช้ดัชนีอาจจะไม่ใช้วิธีที่เป็นธรรมชาตินัก

ตัววิ่งที่ใช้ในการไล่พิจารณาข้อมูลนี้ มักนิยมใช้ในการอ้างถึงข้อมูลในการใช้งานกระบวนการเช่น Delete หรือ InsertAfter ด้วย

เราจะนิยามอินเทอร์เฟสเพิ่มเติมดังนี้

- Begin(L) คืนตัววิ่งที่อ้างถึงข้อมูลตัวแรกในรายการ L
- End(L) คืนตัววิ่งที่อ้างถึงตำแหน่งที่ไม่มีข้อมูลแล้วของรายการ L (เราจะใช้ตัววิ่งนี้ในการระบุจุดสิ้นสุดของข้อมูล)
- Next(p) คืนตัววิ่งที่อ้างถึงข้อมูลถัดจากตัววิ่ง p
- Value(p) คือข้อมูลที่อ้างถึงโดยตัววิ่ง p

การใช้งานตัววิ่งจะมีลักษณะเดียวกันกับการใช้พอยน์เตอร์วิ่งไปในอาร์เรย์ ดังแสดงในโปรแกรมที่ 2.2

เราจะประกาศแบบชนิดข้อมูล `list_itr` เพื่อใช้เป็นชนิดของข้อมูลตัววิ่ง โดยจะประกาศด้วยคีย์เวิร์ด `typedef` ดังด้านล่าง

```
typedef int* list_itr;
```

โปรแกรมที่ 2.3: การอิมพลีเมนท์บางกระบวนการของรายการด้วยอาร์เรย์

```
// iterator functions
inline list_itr list_begin(int list[],int size) {return &list[0];}
inline list_itr list_end(int list[],int size) {return &list[size];}
inline list_itr list_itr_next(list_itr p) { return p+1; }
inline int list_itr_value(list_itr p) { return *p; }

// other interesting list functions
void list_delete(int list[], int size, list_itr p)
{
  list_itr end = list_end(list,size);
  list_itr np = list_itr_next(p);
  while(np != end) {
    *p = *np;
    p = np;
    np = list_itr_next(np;
  }
}
```

ในบทนี้เราจะเขียนการดำเนินการดังกล่าวในลักษณะของฟังก์ชันธรรมดา ซึ่งจะทำให้การเรียกใช้ค่อนข้างยุ่งยากไม่สะดวกเหมือนกับการใช้งานพอยน์เตอร์ ในบทที่ **??** เราจะใช้การเขียนฟังก์ชันของตัวดำเนินการ (operator overloading) ทำให้กระบวนการดังกล่าวมีรูปแบบการใช้งานเหมือนกับพอยน์เตอร์ดังด้านขวา

โปรแกรมที่ 2.3 แสดงการอิมพลีเมนท์บางกระบวนการในอินเทอร์เฟสของรายการด้วยอาร์เรย์ตามที่ได้กล่าวมาแล้ว สังเกตว่าโดยมากฟังก์ชันที่เกี่ยวข้องกับตัววิ่งเป็นฟังก์ชันที่สั้น การประกาศให้เป็น `inline` จะทำให้คอมไพเลอร์เลือกที่จะแทรกฟังก์ชันเหล่านี้ลงไปยังจุดที่เรียกใช้งานเลย แทนที่จะมีการเรียกฟังก์ชันจริง ๆ เพื่อลดเวลาการทำงาน

ในการใช้งาน `list_itr` เป็นตัววิ่งในการคืนค่าจากฟังก์ชันเช่น `list_find` เราจำเป็นจะต้องกำหนดว่าถ้าไม่พบข้อมูล ตัววิ่งจะมีค่าเป็นเช่นใด เราจะใช้รูปแบบเช่นเดียวกับไลบรารีมาตรฐานคือเราจะคืนค่าตัววิ่งที่อ้างถึงตำแหน่งที่ไม่มีข้อมูล (นั่นคือจะมีค่าเท่ากับค่าที่ได้เมื่อเรียกฟังก์ชัน End)

▷ **คำถาม 2.15**
ทดลองเขียนฟังก์ชันทั้งหมดในอินเทอร์เฟสรายการด้วยอาร์เรย์ ◁

เวลาการทำงานของฟังก์ชันที่เกี่ยวกับตัววิ่งทั้งหมดเป็นค่าคงที่ นั่นคือใช้เวลา $O(1)$

2.2.1 พจนานุกรม

พิจารณาปัญหาต่อไปนี้

โจทย์ปัญหา 2.2

TODO: เพิ่มปัญหาที่มีการใช้พจนานุกรม

ในส่วนนี้เราจะทำความรู้จักกับแบบชนิดข้อมูลนามธรรมที่มีประโยชน์มากที่เรียกว่า *พจนานุกรม* หรือ *dictionary* แบบชนิดข้อมูลนี้รองรับการเก็บข้อมูลที่ประกอบด้วยข้อมูลและกุญแจ จากนั้นสามารถค้นหาข้อมูลได้ด้วยกุญแจ ยกตัวอย่างเช่น เราสามารถเก็บข้อมูลเกี่ยวกับพนักงานบริษัทไว้ โดยใช้กุญแจเป็นหมายเลขประจำตัวพนักงาน เมื่อเราต้องการหาข้อมูล เราก็จะระบุหมายเลขประจำตัวพนักงาน แบบชนิดข้อมูลนี้ก็จะคืนข้อมูลทั้งหมดของพนักงานคนนั้น

โปรแกรมที่ 2.4: โปรแกรมการ Insert ที่เรียกใช้ `list_append` ที่ทำงานผิดพลาด

```
void dict_insert(int keys[], int values[], int& size,   // BUGGY
                 int k, int v)                          // BUGGY
{                                                       // BUGGY
  list_append(keys, size, k);                           // BUGGY
  list_append(values, size, v);                         // BUGGY
}                                                       // BUGGY
```

ให้

ในมุมหนึ่งโครงสร้างข้อมูลแบบอาร์เรย์ ก็เป็นรูปแบบหนึ่งของการเก็บข้อมูลแบบพจนานุกรม แต่กุญแจที่ใช้จะต้องเป็นจำนวนเต็มเท่านั้น นอกจากนี้ขนาดของอาร์เรย์ก็จะขึ้นกับค่าที่มากที่สุดของกุญแจ ซึ่งอาจจะมีค่ามากกว่าจำนวนข้อมูลที่ต้องการเก็บลงในพจนานุกรมเป็นจำนวนมากก็ได้

อินเทอร์เฟสของพจนานุกรมมีรูปแบบทั่วไปดังนี้

- IsEmpty(D) - ตรวจสอบว่าพจนานุกรม D ว่างเปล่าหรือไม่
- Insert(D,(k,v)) - เพิ่มข้อมูล v ลงในพจนานุกรม D โดยให้ k เป็นกุญแจ
- Find(D,k) - ค้นหาข้อมูลในพจนานุกรม D ที่มีกุญแจเป็น k แล้วคืนตัววิ่งที่อ้างถึงข้อมูลดังกล่าว
- Delete(D,p) - ลบข้อมูลในพจนานุกรม D ที่ถูกอ้างถึงโดยตัววิ่ง p

พจนานุกรมยังมีประเภทย่อย ๆ อีกหลายแบบ โดยขึ้นกับเงื่อนไขที่พจนานุกรมแบบนั้นรองรับการใช้งาน เช่น การรองรับกรณีที่ข้อมูลหลายตัวมีกุญแจซ้ำกัน เป็นต้น พจนานุกรมบางประเภทยังสามารถค้นหาข้อมูลที่มีกุญแจน้อยที่สุด (หรือมากที่สุดได้ด้วย)

ในที่นี้ เพื่อความง่าย เราจะสมมติว่าไม่มีกรณีที่ข้อมูลสองจำนวนใช้กุญแจเดียวกัน ถ้ามีการจัดเก็บข้อมูลสองชุดด้วยกุญแจเดียวกัน ผลลัพธ์ที่ได้จะขึ้นกับรูปแบบที่เราจะอิมพลีเมนท์

การอิมพลีเมนท์ด้วยอาร์เรย์แบบแรก

เราจะอิมพลีเมนท์กระบวนการต่าง ๆ โดยใช้อาร์เรย์สำหรับเก็บรายการสองชุด ชุดแรกเป็นกุญแจ อีกชุดเป็นข้อมูล เราจะสมมติว่าทั้งกุญแจและข้อมูลที่เราต้องการจะเก็บเป็นจำนวนเต็ม `int` ในบทที่ **??** เราจะศึกษาวิธีที่ทำให้โครงสร้างข้อมูลที่เราเขียนทำงานกับข้อมูลประเภทใดก็ได้

ในที่นี้เราจะประกาศอาร์เรย์สำหรับรายการดังนี้

```
int keys[max_size];
int values[max_size];
int dict_size;
```

การสร้างพจนานุกรมเปล่าทำโดยกำหนดให้ตัวแปร `dict_size` มีค่าเท่ากับศูนย์ การทดสอบ IsEmpty ทำได้โดยการตรวจสอบตัวแปรดังกล่าว

กระบวนการ Insert สามารถจัดการได้โดยใช้กระบวนการ Append ของรายการที่เขียนโดยอาร์เรย์

โปรแกรมที่ 2.5: การค้นหาข้อมูลในพจนานุกรมแบบแรก

```
int dict_find(int keys[], int values[], int size, int x)
{
  for(int i = 0; i != size; ++i)
    if(keys[i] == x)
      return i;
  return -1;                  // ****
}
```

▷ **คำถาม 2.16**
พิจารณาฟังก์ชันเพิ่มข้อมูลในรายการดังโปรแกรมที่ 2.4 ฟังก์ชันดังกล่าวเรียกใช้การ Append จากแบบชนิดข้อมูลรายการที่พัฒนาด้วยอาร์เรย์ โปรแกรมทำงานผิดพลาดเพราะว่าอะไร? ◁

ตัวอย่างข้างต้นแสดงให้เห็นความผิดพลาดที่เกิดจากการใช้รายการเขียนพัฒนาบนอาร์เรย์สองรายการ แต่มีการเปิดใช้ตัวแปร `list_size` ร่วมกัน การเปิดให้มีการจัดการข้อมูลภายในโดยตรงอาจส่งให้เกิดผลข้างเคียงได้ ในการพัฒนาโครงสร้างข้อมูลด้วยภาษา C++ เราสามารถป้องกันปัญหาดังกล่าวได้โดยการกำหนดขอบเขตการเข้าถึงข้อมูล (จะได้ศึกษาในบทที่ **??**)

▷ **คำถาม 2.17**
ถ้ามีการสั่ง Insert ด้วยข้อมูลสองจำนวนที่มีกุญแจเดียวกัน จะมีผลต่อการทำงานของแบบชนิดข้อมูลที่อิมพลีเมนท์ด้วยวิธีนี้อย่างไร ◁

กระบวนการ Find และ Delete ต้องการตัววิ่งที่อ้างถึงข้อมูลในพจนานุกรม สำหรับตัววิ่งในกรณีนี้เราไม่สามารถใช้พอยน์เตอร์ไปยังข้อมูลในอาร์เรย์ได้เนื่องจากเรามีอาร์เรย์สองอาร์เรย์

▷ **คำถาม 2.18**
ถ้าเราต้องการให้ตัววิ่งนั้นสมบูรณ์ในตัวเอง คือสามารถอ้างถึงข้อมูลทั้งกุญแจและข้อมูลได้โดยใช้เฉพาะข้อมูลที่มีอยู่ในตัววิ่งเท่านั้น (กล่าวคือ ไม่ต้องอ้างถึงตัวแปร `keys` และ `values`) ชนิดข้อมของตัววิ่งจะต้องเก็บข้อมูลใดบ้าง? ◁

เราจะศึกษาวิธีการเขียนคลาสเพื่อที่จะจัดการตามคำตอบของคำถามข้างต้นในบทต่อ ๆ ไป ในบทนี้เราจะใช้แค่ข้อมูลดัชนีเพื่อแทนตัววิ่ง (ที่ไม่สมบูรณ์) ไปก่อน

กระบวนการ Find และ Delete ทำงานได้ไม่ต่างจากในการอิมพลีเมนท์รายการเท่าใดนัก เพียงแต่ในการค้นหาเราจะเทียบกุญแจใน `keys` คืนดัชนี และในการลบ จะต้องย้ายข้อมูลจากทั้งอาร์เรย์ `keys` และอาร์เรย์ `values` ตัวอย่างของฟังก์ชัน `dict_find` แสดงในโปรแกรมที่ 2.5

สังเกตว่าเราคืนค่า `-1` เมื่อกุญแจนั้นไม่อยู่ในข้อมูลที่เก็บไว้ การใช้ค่า `-1` นี้ เป็นไปตามความสะดวกที่เราสามารถกำหนดขึ้นได้ เมื่อเราเขียนต่อ ๆ ไป เรจะเปลี่ยนไปคืนค่าเป็นตัววิ่งที่แสดงถึงสถานะของข้อมูลที่ไม่มี (เช่นที่คืนจาก `list_itr_end` เป็นต้น) ตามรูปแบบของไลบรารีมาตรฐาน

เวลาการทำงานของฟังก์ชั่นต่าง ๆ แสดงในตารางในรูปที่ 2.8

▷ **คำถาม 2.19**
เขียนอัลกอริทึมสำหรับกระบวนการ Delete ข้อมูลในรายการ และวิเคราะห์เวลาการทำงาน ◁

การอิมพลีเมนท์ด้วยอาร์เรย์อีกรูปแบบหนึ่ง

วิธีการอิมพลีเมนท์ที่แล้วเก็บข้อมูลในพจนานุกรมลงในอาร์เรย์โดยไม่มีการจัดระบบใด ๆ เลย ทำให้การหาข้อมูลจำเป็นต้องพิจารณาข้อมูลทุกตัวในอาร์เรย์

▷ **คำถาม 2.20**
ลองเสนอวิธีการต่าง ๆ ที่เป็นไปได้ในการเก็บข้อมูลให้เป็นระบบมากขึ้น เพื่อทำให้การค้นข้อมูลทำได้ง่าย ◁

เราจะทดลองวิธีง่าย ๆ ที่เราก็มักใช้ในชีวิตประจำวัน นั่นคือการเรียงข้อมูลในรายการที่เก็บในอาร์เรย์ตามลำดับของกุญแจ ข้อเสียของเงื่อนไขนี้คือการเพิ่มข้อมูลในพจนานุกรมจะใช้เวลามากขึ้น เนื่องจากเราจะต้องแทรกข้อมูลลงไปในตำแหน่งที่เหมาะสม อย่างไรก็ตาม เงื่อนไขนี้จะทำให้เราจัดการกรณีที่มีการเพิ่มข้อมูลที่ใช้กุญแจซ้ำกันได้

จากข้อกำหนดดังกล่าวทำให้เราสามารถพัฒนากระบวนการค้นหาข้อมูลให้เร็วขึ้นได้ กล่าวคือ ถ้าเราต้องการหาข้อมูลที่มีกุญแจ k โดยการไล่พิจารณากุญแจที่มีค่าน้อยที่สุด จนกระทั่งเราพบกุญแจที่มีค่ามากกว่า k เราจะทราบได้ทันทีว่า ข้อมูลที่เราต้องการหาไม่อยู่ในพจนานุกรมอย่างแน่นอน เราเขียนฟังก์ชัน `dict_find` สำหรับกรณีนี้ได้ดังนี้

```
int dict_find(int keys[], int values[], int size, int x)
{
  for(int i = 0; i != size; ++i) {
    if(keys[i] == x)
      return i;
    if(keys[i] > x)
      return -1;
  }
  return -1;
}
```

▷ **คำถาม 2.21**
กรณีใดบ้างที่ฟังก์ชัน `dict_find` ทำงานได้เร็วกว่าฟังก์ชันเดียวกันในกรณีที่ข้อมูลไม่มีการเรียงลำดับ กรณีใดบ้างที่ไม่มีประโยชน์เลย ◁

ในฟังก์ชันดังกล่าว เราใช้ประโยชน์จากการที่ข้อมูลเรียงกันในการจบการทำงานเร็วกว่ากำหนด อย่างไรก็ตามเรามีวิธีการที่จะค้นหาได้เร็วกว่านั้นในแทบจะทุก ๆ กรณี วิธีที่เราจะใช้นั้นมีลักษณะเดียวกับเทคนิคที่เราใช้ในการเล่นเกมทายเลข

▷ **คำถาม 2.22** เกมทายเลข
เกมทายเลขเป็นเกมระหว่างผู้เล่นสองคน ผู้เล่นคนแรกนึกตัวเลขระหว่าง 1 - 100 จากนั้นผู้เล่นคนที่สองจะทายตัวเลขตัวนั้น ผู้เล่นคนแรกจะใบ้โดยการบอกว่าตัวเลขที่ทายมีค่าเท่ากับ มากกว่า หรือน้อยกว่าตัวเลขที่นึกไว้ ผู้เล่นคนที่สองจะใช้ข้อมูลดังกล่าวในการเลือกเลขที่จะทายตัวต่อ ๆ ไป

ให้หาวิธีการถามของผู้เล่นคนที่สอง เพื่อให้ได้คำตอบเร็วที่สุดสำหรับทุก ๆ ตัวเลขที่ผู้เล่นคนแรกนึกไว้

(คำใบ้: เกมดังกล่าวผู้เล่นคนที่สองจะสามารถทายคำตอบโดยทายเลขไม่เกิน 7 ครั้ง) ◁

▷ **คำถาม 2.23**
ทำไมการถามที่ตอบมาจึงสามารถหาคำตอบได้เร็วกว่าการไล่ถามจาก $1, 2, 3, \ldots$?

คำใบ้: ให้พิจารณาจำนวนคำตอบที่เป็นไปได้ในเวลาหนึ่ง ๆ ◁

ถ้าเราไม่มีข้อมูลอะไรเลย นั่นหมายความว่าเราเชื่อว่าผู้เล่นคนแรกอาจจะนึกตัวเลขใด ๆ ก็ได้ตั้งแต่ 1 - 100 ดังนั้น โดยทั่วไปแล้ว เรามักจะถามเลขตัวแรกเป็น 50 หรือ 49

ตัวเลขนี้มีความน่าสนใจเป็นพิเศษ

▷ **คำถาม 2.24** เลข 50
ทำไมการเลือกทายค่า 50 จึงน่าจะ "มีประโยชน์" มากกว่าเริ่มทายด้วย 10 หรือ 5 หรือ 1
คำใบ้: พิจารณาผลที่ได้จากการทาย 50 เมื่อได้รับคำบอกใบ้จากผู้เล่นคนแรก ◁

ถ้าเราใช้หลักคิดดังกล่าว ซ้ำไปเรื่อย ๆ กับผลลัพธ์ที่ได้ เราจะทายค่าตรงกลางของขอบเขตที่เป็นไปได้ที่เหลือตลอด จนกว่าจะไม่มีค่าที่เป็นไปได้อีก

▷ **คำถาม 2.25** กรณีแย่สุด
ถ้าใช้วิธีดังกล่าว เราจะต้องทายอย่างมากที่สุดกี่ครั้ง? อธิบายเหตุผลประกอบ ◁

เฉลย:
เราจะพิจารณาให้ผู้เล่นคนแรกพยายามเล่นให้ผู้เล่นคนที่สองต้องทายให้มากครั้งที่สุดเท่าที่จะทำได้ เมื่อเขาทาย ค่า 50 ผู้เล่นคนแรกสามารถตอบว่า ผลลัพธ์ถูกต้อง (จบเลยไม่ทายต่อ), ค่าที่ทายมีค่ามากกว่าค่าที่สมมติ (ค่าที่เป็นไปได้ต่อไปคือ 1-49), หรือ ค่าที่ทำนายมีค่าน้อยกว่าค่าที่สมมติ (ค่าที่เป็นไปได้ต่อไปคือ 51 - 100) สังเกตว่า ในกรณีที่ผู้เล่นคนแรกตอบว่าค่าที่ทำนานมีค่าน้อยกว่าค่าที่สมมติ ทำให้ตัวเลือกที่เป็นไปได้เหลือ 50 ตัวเลือก ซึ่งมากที่สุดเท่าที่จะเป็นไปได้แล้ว ถ้าผู้เล่นคนที่สองเริ่มถามด้วยค่า 50

จะตอบว่าผู้เล่นคนแรกควรตอบอะไร เราต้องตอบคำถามให้ได้ว่า ในรอบต่อไป กรณีที่ 2 (1-49) หรือ กรณีที่ 3 (51-100) จะเป็นกรณีที่แย่ที่สุดสำหรับผู้เล่นคนที่สองที่ต้องทายตัวเลขมากกว่ากัน สังเกตว่าขอบเขตบนหรือล่างของช่วงคำตอบที่เป็นไปได้นั้นไม่มีผลต่อจำนวนครั้งในการถาม (จะเป็น 1-49 หรือ 1001 - 1049 วิธีการถามที่ดีที่สุดและวิธีการตอบที่ทำร้ายผู้เล่นคนที่สองมากที่สุด ก็สามารถนำมาใช้ได้ผลเป็นจำนวนครั้งของการถามที่เท่ากัน) อย่างไรก็ตามสิ่งที่สำคัญกว่า ก็คือจำนวนคำตอบที่เป็นไปได้ เพราะว่าถ้าช่วงที่เล็กกว่าต้องใช้จำนวนคำถามที่เป็นไปได้มากกว่าช่วงที่ใหญ่กว่า เราก็สามารถสมมติให้ช่วงที่เล็กมีขนาดเท่ากับช่วงที่ใหญ่กว่าได้โดยเพิ่มขอบเขตบนไปให้จำนวนเท่ากัน

ดังนั้น ทางเลือกที่ผู้เล่นคนแรก จะทำให้ผู้เล่นคนที่สองต้องทายเป็นจำนวนมากครั้งที่สุด ก็คือเลือกให้คำตอบที่เป็นไปได้หลังจากรู้คำตอบแล้ว มีจำนวนมากที่สุดเท่าที่จะทำได้

ถ้าเราไล่พิจารณาไปเรื่อย ๆ ในรอบต่อไปเราจะเลือกคำตอบที่เป็นไปได้คือ $\lfloor 50/2 \rfloor = 25$, $\lfloor 25/2 \rfloor = 12$, $\lfloor 12/2 \rfloor = 6$, $\lfloor 6/2 \rfloor = 3$, $\lfloor 3/2 \rfloor = 1$, รอบสุดท้าย (รอบที่ 7) เราถามอีกหนึ่งครั้ง ก็จะเป็นคำตอบที่ถูกต้องแน่นอน

หลักการดังกล่าว สามารถประยุกต์ใช้กับการค้นหาข้อมูลในพจนานุกรมได้

▷ **คำถาม 2.26** การหาข้อมูลในพจนานุกรม
ถ้าเราทราบว่ากุญแจที่เก็บในอาร์เรย์จำนวน n ตัวเรียงตามลำดับจากน้อยไปหามาก จงออกแบบอัลกอริทึมในการค้นข้อมูลที่มีประสิทธิภาพ โดยใช้แนวคิดที่ได้จากเกมทายเลข
(คำใบ้: อาจจะต้องมีการปรับมุมมองเล็กน้อย) ◁

เราอาจจะต้องมีการปรับมุมมองเล็กน้อย เพื่อนำแนวคิดจากเกมทายเลขมาใช้ได้ ในกรณีนี้ เรามีกุญแจ `k` ที่ต้องการค้นหาในอาร์เรย์ `keys` เราอยากทราบว่าดัชนี `i` ใดในอาร์เรย์ ที่ `keys[i]` มีค่าเท่ากับ `k`

สังเกตว่าในกรณีนี้ ถ้ากุญแจอยู่ในอาร์เรย์ คำตอบที่เป็นไปได้คือ $0, 1, \ldots, n-1$ เราจะสมมติให้ผู้เล่นคนที่หนึ่งทราบคำตอบนี้ จากนั้นผู้เล่นคนที่สอง (นั่นคืออัลกอริทึมของเรา) จะเริ่มทาย เมื่อผู้เล่นทายค่า `x` แล้ว เราสามารถตรวจสอบผลลัพธ์ของการทายและคำใบ้ได้โดยการดูค่าในอาร์เรย์ `keys[x]`

▷ **คำถาม 2.27**
ค่าของ `keys[x]` มีความหมายเป็นคำใบ้ของการทายเลขได้อย่างไร? ◁

จากแนวทางคร่าว ๆ เราจะพัฒนาอัลกอริทึมขึ้นมา โดยอัลกอริทึมจะทายดัชนีจากขอบเขตของดัชนีที่เป็นไปได้ จากนั้นจะดูค่าในอาร์เรย์เพื่อปรับขอบเขตต่อไป

▷ **คำถาม 2.28**
ในการดำเนินการตามอัลกอริทึมดังกล่าว จะต้องใช้ตัวแปรใดเก็บข้อมูลการทำงานในแต่ละรอบบ้าง ◁

เราจะต้องเก็บขอบเขตของดัชนีที่สามารถเป็นดัชนีที่เราต้องการหา เราจะใช้ตัวแปร s และ t ในการเก็บค่าดัชนีที่น้อยที่สุดและดัชนีที่มากที่สุด ที่สามารถเป็นคำตอบได้ เริ่มต้น $s = 0$ และ $t = n - 1$ ในแต่ละรอบ เราจะทายค่าตรงกลางระหว่างสองค่านี้

▷ **คำถาม 2.29** ค่าตรงกลาง
ค่าตรงกลางระหว่าง s และ t ที่เราจะทาย คือค่าใด? ◁

เนื่องจากค่าดัชนีต้องเป็นจำนวนเต็มเท่านั้น เราจำต้องทำการปัดค่ากลางที่ได้ ดังนั้นในแต่ละรอบอัลกอริทึมจะทายค่า $\lfloor (s+t)/2 \rfloor$ ให้ตัวแปร m เก็บค่ากลางนี้

▷ **คำถาม 2.30** การดำเนินการหลังการทาย
ถ้าเราต้องการค้นหากุญแจ x เราจะปรับค่าของ s และ t อย่างไร พิจารณากรณีที่ (1) $keys[m] < x$ และ (2) $keys[m] > x$ ◁

ในการปรับค่าดังกล่าว มีรายละเอียดเล็กน้อยซึ่งถ้าเราไม่ระวัง อาจทำให้เกิดปัญหาได้

▷ **คำถาม 2.31** พิจารณาเงื่อนไขในการปรับค่าดังนี้: (1) ถ้า $keys[m] < x$ ปรับค่า $s \leftarrow m$ และ (2) ถ้า $keys[m] > x$ ปรับค่า $t \rightarrow m$

พิจารณาการทำงานของอัลกอริทึมที่ใช้การปรับค่าดังกล่าว มีกรณีใดบ้างที่เงื่อนไขในการปรับค่านี้ ทำให้อัลกอริทึมไม่สามารถหาข้อมูลที่ต้องการเจอได้

(ตำใบ้: พิจารณกรณีที่เหลือขอบเขตของดัชนีที่เป็นไปได้ขนาดเล็ก ๆ) ◁

▷ **คำถาม 2.32**
เราจะแก้ปัญหาดังกล่าวได้อย่างไร? ◁

เราจะปรับค่าอย่างระมัดระวังขึ้น นั่นคือในกรณี (1) เราจะให้ $s \leftarrow m + 1$ และในกรณีที่ (2) เราจะให้ $t \leftarrow m - 1$ การปรับค่าดังกล่าวมีผลเกี่ยวข้องกับอีกกรณีหนึ่งซึ่งไม่เกิดขึ้นในเกมทายเลขที่เราต้องพิจารณา นั่นคือกรณี

ค้นหากุญแจ k จากอาร์เรย์ $keys$ ขนาด n ที่ข้อมูลเรียงลำดับจากน้อยไปมาก (A2.3)

1. $s \leftarrow 0$, $t \leftarrow n-1$
2. WHILE $s \leq t$ DO
3. $\quad m \leftarrow \lfloor (s+t)/2 \rfloor$
4. $\quad$ IF $keys[m] = k$ THEN
5. $\qquad$ RETURN m and EXIT
6. $\quad$ IF $keys[m] < k$ THEN
7. $\qquad s \leftarrow m+1$
8. $\quad$ IF $keys[m] > k$ THEN
9. $\qquad t \leftarrow m-1$
10. RETURN -1 and EXIT // not found

ที่ค้นหาข้อมูลไม่เจอ

▷ **คำถาม 2.33**
เราจะทราบได้อย่างไรว่ากุญแจที่เราค้นหาไม่อยู่ในอาร์เรย์
(คำใบ้: ดูจากขอบเขตที่เป็นไปได้) ◁

เรานำแนวทางดังกล่าวมาเขียนเป็นอัลกอริทึมได้ดังอัลกอริทึม A2.3

เราจะวิเคราะห์เวลาการทำงานของอัลกอริทึมดังกล่าว โดยทั่วไปแล้ว ในการวิเคราะห์เวลาการทำงานของอัลกอริทึมใด เราจำเป็นจะต้องให้เหตุผลว่าอัลกอริทึมนั้นเมื่อทำงานแล้วก่อให้เกิด "ความก้าวหน้า" และใช้ความก้าวหน้านั้นในการให้เหตุผลต่อไป

เราจะลองใช้เหตุผลเกี่ยวกับความก้าวหน้าในการพิจารณาอัลกอริทึมการค้นหาแบบตามลำดับ ในวิธีอิมพลีเมนท์พจนานุกรมด้วยอาร์เรย์แบบแรกเพื่อแสดงเป็นตัวอย่าง

▷ **คำถาม 2.34** ความก้าวหน้าของการค้นหาแบบตามลำดับ
การค้นหาตามโปรแกรมที่ 2.5 เป็นการค้นหาแบบตามลำดับ ให้พิจารณาโปรแกรมดังกล่าว และลองหาว่าเราสามารถใช้สถานะอะไรของโปรแกรม (เช่น ตัวแปรหรือค่าอื่น ๆ) เพื่อบอกว่าโปรแกรมมีความก้าวหน้าในการทำงานได้บ้าง? ◁

ในการค้นหาข้อมูลจากอาร์เรย์ที่เก็บกุญแจใด ๆ ก็ตาม ถ้าเราไล่พิจารณาตามลำดับจากกุญแจแรก ไปจนถึงกุญแจสุดท้ายในอาร์เรย์ ทุก ๆ ครั้งที่เราพิจารณากุญแจหนึ่ง มีผลลัพธ์ที่เป็นไปได้สองแบบ คือ เราพบกุญแจที่เราต้องการหา หรือถ้าเราไม่พบ เราจะสามารถ "ทิ้ง" กุญแจที่เราพิจารณาอยู่ไปได้ (เนื่องจากไม่ใช่เป้าหมายที่เราต้องการแล้ว) ในกรณีนี้ ความก้าวหน้าของงานที่เราทำก็คือ จำนวนกุญแจที่เราสามารถทิ้งไปได้ ดังนั้นถ้าเรามีกุญแจทั้งสิ้น n กุญแจ และในแต่ละรอบของการทำงาน เราใช้เวลา $O(1)$ เพื่อจะทิ้งกุญแจที่ไม่ถูกต้องได้ 1 อัน ดังนั้นในกรณีที่แย่ที่สุด ก็คือเราจะต้องใช้เวลา $O(n)$

▷ **คำถาม 2.35** ความก้าวหน้าของการค้นหาที่ดีขึ้น

ถ้าเราใช้จำนวนกุญแจที่เราจะต้องพิจารณาแทนปริมาณงานทั้งหมด (เริ่มต้นที่ n หน่วย) ในแต่ละรอบของอัลกอริทึม A2.3 เราจัดการกับงานดังกล่าวไปได้เท่าใด?

(คำใบ้: พิจาณารอบที่ 1, รอบที่ 2, แล้วค่อยพยายามพิจารณาในรอบใด ๆ) $\triangleleft$

เราสามารถพิสูจน์ได้ว่า ในทุก ๆ รอบการทำงานของวนรอบ WHILE จำนวนดัชนีของกุญแจที่เป็นไปได้จะมีค่าลดลงอย่างน้อย 2 เท่า สังเกตว่าเมื่อเริ่มต้นเข้าทำงานในวนรอบ จำนวนกุญแจที่เป็นไปได้คือ $t - s + 1$ ถ้าเราพบคำตอบ อัลกอริทึมจะจบการทำงาน ถ้าไม่เช่นนั้น มีสองกรณีที่เราต้องพิจารณา

- ถ้าเราผ่านเงื่อนไขในบรรทัดที่ 6 เข้าไปทำงานในบรรทัดที่ 7 เราจะได้ว่า $s \leftarrow m+1$ ดังนั้นจำนวนกุญแจใหม่คือ

$$
\begin{aligned}
t - (\lfloor (s+t)/2 \rfloor + 1) + 1 &= \lceil t - s/2 - t/2 \rceil \\
&= \lceil t/2 - s/2 \rceil \\
&\leq t/2 - s/2 + 1/2 = (t - s + 1)/2
\end{aligned}
$$

- ในอีกกรณีหนึ่ง ถ้าเงื่อนไขบรรทัดที่ 8 เป็นจริง แล้วเราเข้าไปทำงานในบรรทัดที่ 9 ในส่วนนี้สามารถพิสูจน์ได้ด้วยวิธีที่คล้ายกัน

$\triangleright$ **คำถาม 2.36** ก้าวหน้าอย่างน้อย...

จงพิสูจน์ว่า ถ้าโปรแกรมทำงานเข้าไปในคำสั่งบรรทัดที่ 9 จำนวนกุญแจที่จะต้องพิจารณาจะมีค่าลดลงอย่างน้อย 2 เท่า $\triangleleft$

เราจะวิเคราะห์เวลาการทำงาน สังเกตว่าในแต่ละรอบการทำงานของวนรอบ WHILE ใช้เวลา $O(1)$ เมื่อทำงานเสร็จในแต่ละรอบ ถ้าโปรแกรมไม่หยุดการทำงาน จากการพิจารณาข้างต้น เราจะพบว่าจำนวนกุญแจที่เหลือต้องพิจารณาจะลดลงอย่างน้อย 2 เท่า นั่นคือเราทราบว่าหลังการทำงานรอบแรก จะเหลือกุญแจไม่เกิน $n/2$ หลังรอบที่สองจะเหลือกุญแจไม่เกิน $n/2/2 = n/4$ และหลังรอบที่ x จะเหลือกุญแจไม่เกิน $n/2^x$ อัน

เราทราบว่าโปรแกรมจะหยุดทำงานแน่นอนถ้าไม่มีกุญแจให้พิจารณาอีก เนื่องจากเมื่อเริ่มต้นเรามีกุญแจต้องพิจารณาจำนวน n กุญแจ ถ้าให้ x แทนจำนวนรอบการทำงานของอัลกอริทึมที่ไม่ใช่รอบสุดท้าย เราจะได้ว่า $\frac{n}{2^x} > 1$ เมื่อแก้สมการเราจะได้ว่า $x < \log_2 n$ นั่นคือโปรแกรมจะทำงานไม่เกิน $\log n$ รอบ[3]

ดังนั้นเวลาการทำงานของการค้นหาดังกล่าวคือ $O(\log n)$ การค้นหาโดยการเลือกค่าที่แบ่งข้อมูลออกเป็นสองส่วนแบบนี้เป็นเทคนิคสำคัญในการออกแบบอัลกอริทึม การค้นหาแบบนี้มีชื่อว่า *การค้นหาแบบทวิภาค* (*binary searching*)

เราสรุปเวลาของกระบวนการต่างเมื่ออิมพลีเมนท์พจนานุกรมด้วยอาร์เรย์ทั้งสองแบบเป็นตารางในรูป 2.8

$\triangleright$ **คำถาม 2.37**

จงอิมพลีเมนท์ฟังก์ชัน `dict_find` ที่มีประสิทธิภาพมากขึ้นเมื่อกุญแจในอาร์เรย์ `keys` เรียงตามลำดับจากน้อยไปหามาก $\triangleleft$

[3]ในหนังสือเล่มนี้ ถ้าไม่ระบุฐาน เราจะถือว่าเราใช้ลอการิทึมฐาน 2 เสมอ

กระบวนการ	เก็บตามลำดับการใส่ข้อมูล	เก็บเรียงตามกุญแจ
IsEmpty	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$
Find	$O(n)$	$O(n)$ หรือ $O(\log n)$ ถ้าค้นหาแบบทวิภาค
Delete	$O(n)$	$O(n)$

รูปที่ 2.8: เวลาการทำงานของกระบวนการต่าง ๆ ของพจนานุกรม ที่พัฒนาด้วยอาร์เรย์ทั้งสองแบบ โดยที่ n แทนจำนวนข้อมูลในพจนานุกรม

ความแตกต่างระหว่าง $O(n)$ และ $O(\log n)$

TODO: เขียนส่วนนี้

ควรจะเลือกใช้วิธีการอิมพลีเมนท์แบบใด?

TODO: เขียนส่วนนี้

บทที่ 3

การซ่อนรายละเอียด: คลาสและเท็มเพลต

ในบทนี้เราจะศึกษาเครื่องมือของภาษา C++ ที่เราจะใช้พัฒนาโครงสร้างข้อมูล เราจะศึกษาบางส่วนของแนวคิดการโปรแกรมเชิงวัตถุ (object-oriented programming) และการโปรแกรมเชิงเจนเนอริก (generic programming) ผ่านทางการปรับปรุงโครงสร้างข้อมูลสำหรับจัดเก็บรายการด้วยอาร์เรย์ที่เราได้อิมพลีเมนท์ในบทก่อน ๆ

เป้าหมายของเนื้อหาในบทนี้เพื่อที่จะปูพื้นฐานภาษา C++ ให้กับผู้อ่านให้มากพอที่จะศึกษาการเขียนโครงสร้างข้อมูลในบทต่อ ๆ ไปได้เท่านั้น ดังนั้นรายละเอียดหลาย ๆ อย่างของการเขียนจะถูกละไว้หรือเอ่ยถึงคร่าว ๆ ผู้อ่านที่สนใจจึงควรหาแหล่งอ้างอิงเพิ่มเติมด้วย

3.1 การรวมกลุ่มข้อมูลและกระบวนการจัดการ

ในส่วนนี้ เราจะปรับปรุงรูปแบบการใช้งานโครงสร้างข้อมูลให้สะดวกขึ้น โดยการรวมตัวแปรที่เกี่ยวข้องหลายตัวเข้าด้วยกันเป็นชนิดข้อมูลใหม่ จากนั้นเราจะรวมกระบวนการจัดการเข้าไว้ด้วย

3.1.1 การรวมกลุ่มข้อมูล: `struct`

ถ้าเราพิจารณาฟังก์ชันต่าง ๆ ที่เราอิมพลีเมนท์ เราจะพบว่าทุกครั้งในการเรียกใช้เราจำเป็นจะต้องส่งพารามิเตอร์ `list` ที่เก็บข้อมูล และ `size` ที่ระบุจำนวนข้อมูลในรายการไปพร้อม ๆ กันตลอด ข้อสังเกตนี้แสดงว่าข้อมูลทั้งสองมีความเกี่ยวข้องกันมาก ในกรณีนี้เพราะว่าเป็นข้อมูลพื้นฐานของการจัดการเกี่ยวกับรายการของเรา

สิ่งแรกที่เราสามารถปรับปรุงได้คือการรวมกลุ่มข้อมูลนี้เข้าด้วยกัน ในภาษา C++ เราสามารถประกาศชนิดข้อมูลแบบโครงสร้างโดยใช้คีย์เวิร์ด `struct` เพื่อรวมกลุ่มข้อมูลหลาย ๆ ชนิดเข้าด้วยกัน

```
const int max_list_size = 1000;

struct list {
  int items[max_list_size];
  int size;
};
```

ในการประกาศนั้นเราจะระบุสมาชิก (member) ของโครงสร้างที่เราประกาศ เราสามารถประกาศชนิดข้อมูล `list` ที่มีสมาชิกคืออาร์เรย์ `items` และจำนวนเต็ม `size` ดังโปรแกรมด้านล่าง (สังเกตว่าเราประกาศขนาดของอาร์เรย์ไว้ที่ค่าคงที่ `max_list_size` โดยให้มีขนาด 1000 ช่อง)

โปรแกรมที่ 3.1: บางฟังก์ชันที่ทำงานกับ list

```
typedef int* list_itr;

void list_init(list& l)
{
  l.size = 0;
}

list_itr list_itr_end(list& l)
{
  return &l.items[l.size];
}

list_itr list_find(list& l, int x)
{
  for(int i = 0; i < l.size; ++i)
    if(l.items[i] == x)
      return &l.items[i];
  return list_itr_end(l);
}

list_itr list_append(list& l, int x)
{
  if(l.size >= max_list_size)
    throw "List␣overflow";
  l.items[l.size] = x;
  ++l.size;
  return &l.items[l.size-1];
}
```

สังเกตว่าเราจะต้องใช้เครื่องหมาย ; ในการปิดการประกาศ struct ด้วย

เราสามารถประกาศตัวแปรที่มีชนิดข้อมูลเป็น list ได้ ตัวแปรนี้จะมีสมาชิกชื่อ items และ size ซึ่งสามารถอ้างถึงได้โดยใช้โอเปอเรเตอร์ . ยกตัวอย่างเช่น ถ้าเราประกาศ

```
list lst;
```

เราจะสามารถอ้างถึง lst.items และ lst.size ได้

เราสามารถปรับฟังก์ชันต่าง ๆ ที่เคยต้องรับพารามิเตอร์สองตัวให้รับโครงสร้าง list เพียงพารามิเตอร์เดียวได้ บางฟังก์ชันที่สำคัญแสดงในโปรแกรมที่ 3.1

สังเกตว่าฟังก์ชัน list_append คืนค่าเป็นตัววิ่งที่ชี้ไปที่ข้อมูลที่เพิ่งเพิ่มเข้าไปในรายการ ค่านี้โดยมากจะไม่ได้ใช้ อย่างไรก็ตามอาจจะเป็นประโยชน์ในการอ้างถึงต่อไปได้ เราจึงให้ฟังก์ชันดังกล่าวคืนค่ามา

อย่างไรก็ตาม เราต้องระวังวิธีการผ่านพารามิเตอร์ เนื่องจากโดยปกติ C++ จะส่งค่าพารามิเตอร์แบบ pass by value ดังนั้น ถ้าเราส่งค่าโครงสร้าง list โดยตรง จะทำให้ระบบสร้างตัวแปรประเภท list อีกตัวหนึ่งและคัดลอกข้อมูลจากอาร์กิวเมนท์ที่เราส่งไปยังตัวแปรชั่วคราวนี้ ซึ่งในกรณีของโครงสร้าง list นอกจากจะทำให้โปรแกรมทำงานไม่ถูกต้องแล้ว (เช่นในกรณีของฟังก์ชัน list_init) ยังเป็นการทำให้โปรแกรมมีประสิทธิภาพการทำงานต่ำลงโดยไม่จำเป็นด้วย

ตัวอย่างการเรียกใช้งานฟังก์ชันดังกล่าวแสดงในโปรแกรมต่อไปนี้

โปรแกรมที่ 3.2: การประกาศโครงสร้าง `list` ที่ระบุฟังก์ชันสมาชิก

```
struct list {
  // members
  int items[max_list_size];
  int size;

  typedef int* iterator;

  // member functions
  void init();
  iterator begin();
  iterator end();
  iterator find(int x);
  iterator append(int x);
  void del(iterator p);
  void insert_after(iterator p, int x);
};
```

```
  list lst;
  list_init(lst);
  list_append(lst,10);  list_append(lst,30);  list_append(lst,20);
  if(list_find(lst,5) != list_itr_end(lst))  // find 5 in list
    cout << "found" << endl;
  else
    cout << "not␣found" << endl;
```

3.1.2 การรวมกระบวนการเข้าเป็นฟังก์ชันสมาชิก

สังเกตว่าฟังก์ชันต่าง ๆ ที่ทำงานกับรายการที่เราเขียนนั้น โดยมากจะทำงานกับโครงสร้าง `list` ที่รับพารามิเตอร์มา เนื่องจากฟังก์ชันเหล่านี้ประมวลผลกับข้อมูลโดยตรง เราจะรวมฟังก์ชันเหล่านั้นเข้าเป็นสมาชิกของโครงสร้าง `list` เพื่อให้การใช้งานทำได้สะดวกขึ้น ตัวอย่างการใช้งานฟังก์ชันเหล่านี้ผ่านทางตัวแปร `list` แสดงในโปรแกรมถัดไป เราจะสังเกตว่าการเรียกใช้งานดูกระชับขึ้น

```
  list lst;
  lst.init();
  lst.append(10);  lst.append(30);  lst.append(20);
  if(lst.find(5) != lst.end())  // find 5 in list
    cout << "found" << endl;
  else
    cout << "not␣found" << endl;
```

เราจะเพิ่มการประกาศฟังก์ชันสมาชิกเข้าไปในการประกาศ `struct` ดังแสดงในโปรแกรมที่ 3.2 ฟังก์ชันเหล่านี้เรียกว่า*ฟังก์ชันสมาชิก* (*member function*)

นอกจากนี้ เรายังนำชนิดข้อมูล `list_itr` ย้ายไปประกาศไว้ภายในโครงสร้าง `list` ด้วย โดยใช้ชื่อชนิดข้อมูลเป็น `iterator` (สังเกตว่าเราตัดคำขึ้นต้น `list` ทิ้งไปหมด เพื่อไม่ให้ชื่อซ้ำซ้อน) นอกจากนี้เรายังหลีกเลี่ยงการตั้งชื่อฟังก์ชันสมาชิกว่า `delete` โดยใช้ชื่อว่า `del` แทน เพราะจะซ้ำกับชื่อโอเปอเรเตอร์ `delete`

ฟังก์ชันสมาชิกเหล่านี้ จะทำงานกับตัวแปรโครงสร้าง `list` และสามารถอ้างถึงสมาชิกที่เป็นตัวแปรเช่น `size` และ `items` ได้โดยตรง รูปที่ 3.1

TODO: add this

รูปที่ 3.1: แสดงลักษณะการทำงานของฟังก์ชันสมาชิก

```
void list::init()
{
  size = 0;
}

list::iterator list::end()
{
  return &items[size];
}

list::iterator list::find(int x)
{
  for(int i = 0; i < size; ++i)
    if(items[i] == x)
      return &items[i];
  return end(l);
}
```

หลังจากประกาศฟังก์ชันสมาชิกต่าง ๆ เหล่านี้ ในโครงสร้าง `list` แล้ว เราจะนิยามฟังก์ชันดังกล่าว โปรแกรมที่ **??** แสดงการนิยามฟังก์ชัน `list::init`, `list::end`, และ `list:find`

เช่นเดียวกับการอ้างถึงสมาชิก `size` และ `items` ของตัวแปร `l` ด้วย `l.size` และ `l.items` ในการอ้างถึงฟังก์ชันสมาชิกของตัวแปรต่าง ๆ เราจะใช้โอเปอเรเตอร์จุด (`.`) ส่วนในกรณีที่เราจะอ้างถึงสมาชิกของโครงสร้างเอง เราจะใช้โอเปอเรเตอร์ `::` เช่น `list::init` และ `list::iterator`

สังเกตว่าภายในฟังก์ชันสมาชิก `list::init` เราสามารถอ้างถึงสมาชิก `size` ได้โดยตรง ตัวแปรดังกล่าวจะอ้างถึงสมาชิก `size` ของตัวแปรที่เราเรียกฟังก์ชันเหล่านี้ผ่าน เช่น ถ้าเราเรียก `lst.init()` ตัวแปร `size` ในฟังก์ชันสมาชิกที่เราเขียนจะหมายถึง `lst.size`

▷ **คำถาม 3.1** ฟังก์ชันสมาชิกอื่น ๆ
เขียนฟังก์ชันสมาชิกอื่น ๆ ของโครงสร้าง `list` และทดสอบกับโปรแกรมตัวอย่าง ◁

3.2 การซ่อนรายละเอียด: `class`

ในส่วนที่แล้วเราพัฒนาชนิดข้อมูล `list` ที่รวมข้อมูลและกระบวนการที่ทำงานกับข้อมูลไว้ด้วยกัน ทำให้การใช้งานต่าง ๆ สะดวกขึ้นและเขียนโปรแกรมได้อย่างกระชับขึ้น การสร้างชนิดข้อมูล `list` ที่เราทำนี้ เป็นส่วนหนึ่งของการจัดการกับความซับซ้อนของซอฟต์แวร์ โดยการแบ่งงานเป็นส่วนย่อย ๆ และพัฒนาโปรแกรมเพื่อรับผิดชอบงานแต่ละส่วนแยกกันไป

อย่างไรก็ตาม สังเกตว่าในการเขียนที่ผ่านมา เราไม่สามารถรับประกันว่างานที่แบ่งออกมาพัฒนาเป็นส่วน ๆ นั้นจะถูกแบ่งไปดูแลแยกขาดจากกัน ยกตัวอย่างเช่น อาจมีส่วนหนึ่งของระบบที่ต้องการแทรกข้อมูลในรายการเพื่อเป็นรายการแรก ซึ่งโครงสร้าง `list` ที่เราได้พัฒนาไว้ ยังไม่ได้รองรับ โปรแกรมเมอร์ผู้รับผิดชอบในส่วนนั้นก็อาจจะเปิดดูส่วนของโปรแกรมที่เราเขียนขึ้น และเขียนโปรแกรมในลักษณะด้านล่างขึ้นมาเพื่อทำงานดังกล่าว

```
for(int i = lst.size-1; i >= 0; --i)
  lst.items[i+1] = lst.items[i];
lst.items[0] = x;
++lst.size;
```

โปรแกรมดังกล่าวสามารถทำงานได้ถูกต้อง อย่างไรก็ตามโปรแกรมไม่มีการตรวจสอบขอบเขตของอาร์เรย์เลย ทำให้โปรแกรมอาจจะเกิดความผิดพลาดได้ในภายหลัง สังเกตว่า ถ้าโปรแกรมเมอร์คนที่เขียนโปรแกรมดังกล่าว เขียนผิดพลาด (เช่น ลืมเพิ่มค่า `lst.size` ก็อาจจะมีผลกระทบไปกับทุกส่วนของระบบที่มีการใช้งานรายการนี้)

อย่างไรก็ตาม ความยากของการพัฒนาซอฟต์แวร์ขนาดใหญ่ไม่ได้เกิดจากความซับซ้อนเพียงอย่างเดียว แต่ระบบขนาดใหญ่ยังมีความต้องการในการปรับเปลี่ยนตลอดเวลา เพื่อให้สามารถรองรับความต้องการของผู้ใช้ได้

สมมติว่าในอนาคต มีความจำเป็นต้องเปลี่ยนรูปแบบในการเก็บข้อมูลรายการใหม่ จากที่เคยใช้อาร์เรย์ มาใช้โครงสร้างข้อมูลแบบอื่นที่มีประสิทธิภาพมากกว่า ถ้าเรายอมให้โปรแกรมในส่วนต่าง ๆ พัฒนากระบวนการจัดการรายการได้โดยเป็นอิสระ เราจะต้องเข้าไปแก้ไขโปรแกรมต่าง ๆ เหล่านั้นเพื่อให้การเปลี่ยนแปลงโครงสร้างภายในของรายการเป็นไปได้

ยิ่งไปกว่านั้น ถ้าเราเปลี่ยนแปลงโปรแกรมโดยที่ยังใช้ตัวแปรสมาชิก `items` หรือ `size` อยู่แต่ใช้ในความหมายอื่น โปรแกรมในส่วนอื่น ๆ ที่เรียกใช้ข้อมูลจากรายการผ่านทางสมาชิกดังกล่าวโดยตรงก็จะยังสามารถคอมไพล์ได้ รวมถึงอาจจะทำงานได้โดยที่เราไม่ทราบว่ามีความผิดพลาดซ่อนอยู่ ความผิดพลาดพวกนี้ก็เป็นเสมือนระเบิดเวลารอการระเบิดเท่านั้นเอง

ภาษา C++ ได้พัฒนามาจากภาษา C โดยมีเป้าหมายที่จะทำให้ภาษาโปรแกรมช่วยจัดการกับความซับซ้อนได้ดีขึ้น หนึ่งในความสามารถที่เพิ่มเข้ามาคือการรวมกระบวนการเข้ากับข้อมูล ดังที่เราได้ทดลองใช้แล้ว อีกหนึ่งความสามารถที่สำคัญไม่แพ้กัน ก็คือความสามารถในการกำหนดขอบเขตการเข้าใช้สมาชิกของชนิดข้อมูลที่เราได้สร้างขึ้น เช่น ในกรณีนี้ เราสามารถระบุได้ว่าสมาชิก `items` และ `size` เป็นสมาชิกภายในของส่วนนี้ และไม่ยอมให้โปรแกรมในส่วนอื่น ๆ เข้าถึงได้โดยตรง

เราจะศึกษาความสามารถดังกล่าวโดยการพัฒนาคลาส (`class`) สำหรับจัดการกับรายการ

ในภาษา C++ คลาส เป็นการสร้างชนิดข้อมูลใหม่ ในลักษณะเดียวกันกับ `struct` จริง ๆ แล้ว โครงสร้างสองแบบนี้มีการใช้งานเหมือนกันทุกประการ ยกเว้นเกณฑ์มาตรฐานในการควบคุมการอ้างถึงสมาชิก สำหรับตัวแปรที่ประกาศขึ้นจากคลาสหรือโครงสร้าง เรานิยมเรียกว่า *วัตถุ*

เราจะนิยามคลาสได้ในลักษณะเดียวกับการนิยาม `struct` แต่สมาชิกต่าง ๆ ที่เรานิยามในคลาสจะไม่สามารถถูกอ้างถึงได้จากโปรแกรมที่อยู่ “ภายนอก” คลาส ยกเว้นจะเป็นส่วนที่เราระบุไว้ พิจารณาตัวอย่างโปรแกรมด้านล่างเพื่อเปรียบเทียบ

```
struct test1 { int a;  int f() { return a+1; } };
class test2 { int a; int f() { return a+1; } };

test1 s1;  test2 s2;
cout << s1.a << endl;        // works
cout << s2.a << endl;        // compilation error
cout << s1.f() << endl;      // works
cout << s2.f() << endl;      // compilation error
```

สังเกตว่าฟังก์ชันสมาชิก `test2::f` สามารถอ้างถึงสมาชิก `a` ได้ แต่โปรแกรมภายนอกไม่สามารถอ้างถึงได้

ในการระบุขอบเขตการอ้างถึงสมาชิกในคลาส เราจะใช้คีย์เวิร์ด `private` และ `public` สมาชิกที่อยู่หลักการระบุ `public` เป็นส่วนสาธารณะจะถูกอ้างถึงจากส่วนใด ๆ ของโปรแกรมก็ได้ ส่วนที่ถูกระบุว่าเป็น `private` จะ

เป็นสมาชิกส่วนตัวและถูกอ้างถึงได้จากสมาชิกของคลาสเท่านั้น ในกรณีที่เราไม่ได้ระบุอะไร สมาชิกของคลาสจะเป็นสมาชิกส่วนตัว ในทางกลับกัน สำหรับการนิยามโครงสร้างนั้น สมาชิกที่ไม่ได้ระบุขอบเขตการเข้าถึง จะถูกพิจารณาเป็นสมาชิกสาธารณะ

พิจารณาคลาส test2 ที่แก้ไขแล้วด้านล่างที่เปิดให้ฟังก์ชัน test2::f สามารถเข้าถึงได้จากทุก ๆ ที่ สังเกตว่าเราเพิ่มฟังก์ชัน set_a เพื่อให้เราสามารถกำหนดค่าให้กับ a ได้ด้วย (ไม่เช่นนั้นเราจะไม่มีทางกำหนดค่าให้กับ a ได้เลย)

```
class test2 {
  int a;
public:
  void set_a(int aa) { a = aa; }
  int f() { return a+1; }
};
```

3.2.1 คลาสสำหรับตัวนับ

เราจะเขียนคลาส counter สำหรับใช้เป็นตัวนับ คลาสดังกล่าวไม่ได้มีความซับซ้อนในการเขียนจึงเป็นตัวอย่างที่ดีในการเริ่มต้น

ในการออกแบบคลาสนั้น เราจำเป็นจะต้องพิจารณาส่วนที่ผู้ใช้จะเรียกใช้งานก่อน ฟังก์ชันสมาชิกในกลุ่มนี้จะเป็นอินเทอร์เฟสของวัตถุในคลาส ที่ผู้ใช้งานจะเรียกใช้ อินเทอร์เฟสของตัวนับนั้นไม่ซับซ้อน ดังแสดงในรายการด้านล่าง

- Init(z) – กำหนดค่าเริ่มต้นให้กับตัวนับ โดยให้มีค่าเท่ากับ z
- Inc() – เพิ่มค่าตัวนับขึ้น 1
- Get() – คืนค่าปัจจุบันของตัวนับ

สังเกตว่าโดยทั่วไปแล้ว เรามักกำหนดให้ตัวนับมีค่าเริ่มต้นเป็น 0 จึงเป็นการเหมาะสมที่จะเพิ่มฟังก์ชันสมาชิกที่ทำงานดังกล่าวไว้ด้วย

- Init — กำหนดค่าเริ่มต้นให้กับตัวนับ โดยให้มีค่าเท่ากับ 0

เราจะนิยามคลาสดังกล่าวในโปรแกรมที่ 3.3 สมาชิกข้อมูลของคลาส counter มีเพียงแค่ค่าตัวนับที่นับได้ c เนื่องจากฟังก์ชันสมาชิกของคลาสนี้สั้น เราจึงนิยามฟังก์ชันสมาชิกเหล่านี้ไว้ภายในการนิยามคลาส สำหรับฟังก์ชันที่นิยามภายในคลาสเหล่านี้ คอมไพลเลอร์ภาษา C++ จะแปลงให้เป็นฟังก์ชันแบบ inline โดยอัตโนมัติ

สังเกตการประกาศอาร์กิวเมนท์ปริยายในฟังก์ชันสมาชิก counter::init ถ้ามีการเรียกคำสั่งที่ไม่ได้ระบุค่าพารามิเตอร์ z จะเปรียบเสมือนส่งค่า 0 เป็นอาร์กิวเมนท์

ในคลาส เรานิยมประกาศสมาชิกที่เป็นสาธารณะก่อน เพราะข้อมูลส่วนอินเทอร์เฟสจะเป็นสิ่งสำคัญกว่าสำหรับผู้ที่นำคลาสของเราไปใช้

โปรแกรมที่ 3.3: นิยามของคลาส counter

```
class counter {
public:
  void init(int z=0) { c = z; }
  void inc() { ++c; }
  int get() { return c; }
```

```
private:
  int c;
};
```

3.2.2 ตัวสร้าง: constructor

ฟังก์ชันสมาชิก counter::init ทำหน้าที่เฉพาะ คือกำหนดค่าเริ่มต้นให้กับวัตถุในคลาส counter

▷ **คำถาม 3.2**
ถ้าเราลืมเรียกฟังก์ชัน init ก่อนการใช้งานตัวนับจะเกิดอะไรขึ้น? ◁

สังเกตว่า ระหว่างที่เราประกาศตัวแปร จนถึงก่อนที่จะเรียกใช้ฟังก์ชัน init นั้น สถานะของวัตถุนั้นเป็นสถานะที่ไม่มีความหมาย เนื่องจากยังไม่มีการกำหนดค่า ในภาษา C++ เราสามารถกำหนดฟังก์ชันสมาชิกพิเศษให้ทำงานทันที เมื่อมีการประกาศใช้งานวัตถุของคลาสนั้น ๆ ฟังก์ชันนี้จะทำหน้าที่เหมือนกับฟังก์ชัน init แต่เราจะสามารถรับประกันได้ว่าเมื่อเรานิยามวัตถุแล้ว วัตถุนั้นจะอยู่ในสภาพที่ "พร้อมใช้งาน" ทันที

ฟังก์ชันดังกล่าว เรียกว่า*ตัวสร้าง* (*constructor*) โดยเราสามารถระบุฟังก์ชันดังกล่าวโดยตั้งชื่อให้เหมือนกับชื่อคลาส ตัวอย่างของการประกาศตัวสร้าง และการใช้งานแสดงในโปรแกรมที่ 3.4 ในโปรแกรมดังกล่าว เมื่อเราประกาศวัตถุ a และ b ฟังก์ชันสมาชิก counter::counter จะถูกเรียกใช้งาน

โปรแกรมที่ 3.4: นิยามของคลาส counter ที่ใช้ตัวสร้าง และตัวอย่างการใช้งาน

```
class counter {
public:
  counter(int z=0) { c = z; }
  void inc() { ++c; }
  int get() { return c; }
private:
  int c;
};

  //...
  counter a, b(10);
  cout << a.get() << endl;     // outputs 0
  b.inc(); a.inc();
  cout << b.get() << endl;     // outputs 11
  cout << a.get() << endl;     // outputs 1
```

สังเกตว่าภายในตัวสร้าง counter::counter จริง ๆ แล้วสิ่งที่เราทำก็เป็นการสร้างตัวแปร c นั่นเอง เราสามารถระบุความตั้งใจดังกล่าวให้ชัดเจนโดยระบุคำสั่งดังกล่าวให้เป็นการเรียกตัวสร้างของ c ดังส่วนของโปรแกรมด้านล่างนี้ เราจะได้เห็นการใช้งานในลักษณะนี้ต่อ ๆ ไป

```
  //...
  counter(z=0) : c(z) {}
```

ข้อควรระวังในการใช้การกำหนดค่าลักษณะนี้คือ สมาชิกจะได้รับการกำหนดค่า**ตามลำดับที่นิยาม**ในคลาส ไม่ใช่ตามลำดับที่เราเขียนในรายการกำหนดค่าเริ่มต้น

โปรแกรมที่ 3.5: นิยามคลาส list

```
const int max_list_size = 1000;

class list {
public:
  typedef int* iterator;

  list() : size_(0) {}

  int size() { return size_; }
  iterator begin() { return &items_[0]; }
  iterator end() { return &items_[size_]; }

  iterator find(int x);
  iterator append(int x);
  void del(iterator p);
  void insert_after(iterator p, int x);

private:
  int items_[max_list_size];
  int size_;
};
```

3.2.3 คลาส list

เราสามารถปรับการนิยามโครงสร้าง list ให้ซ่อนสมาชิก items และ size ได้ดังโปรแกรมที่ 3.6 สังเกตว่าเราเปลี่ยนไปนิยามคลาส และประกาศฟังก์ชันสมาชิกต่าง ๆ แบบ public

เราต้องการให้ผู้ใช้สามารถเข้าถึงขนาดของรายการได้ เราจึงสร้างฟังก์ชันสมาชิก list::size สังเกตว่าชื่อดังกล่าวตรงกับชื่อของตัวแปรที่เราเคยใช้ เพื่อแก้ปัญหาดังกล่าวเราอาจจะใช้ฟังก์ชันสมาชิกเป็น list::get_size อย่างไรก็ตาม เราต้องการทำให้อินเทอร์เฟสเรียบง่าย ดังนั้นเราจะคงอินเทอร์เฟสเป็น list::size ไว้ แต่จะเปลี่ยนชื่อสมาชิกส่วนตัวแทน

หนังสือเล่มนี้จะทำตามคำแนะนำในการตั้งชื่อสมาชิกส่วนตัว โดยจะตามชื่อด้วยเครื่องหมายขีดล่าง นั่นคือเราจะเปลี่ยนชื่อจาก items และ size เป็น items_ และ size_

สังเกตการใช้งานตัวสร้างเพื่อกำหนดค่าเริ่มต้นให้กับตัวแปร size_ เราได้ย้ายนิยามของฟังก์ชันสมาชิกที่เกี่ยวกับตัววิ่งมานิยามภายในการนิยามคลาสเพราะว่าเป็นฟังก์ชันที่สั้น ส่วนการนิยามฟังก์ชันสมาชิกอื่น ๆ ของโครงสร้างนั้นไม่มีการเปลี่ยนแปลง ยกเว้นการอ้างสมาชิก items_ และ size_

3.2.4 ค่าคงที่ภายในคลาส

ในการประกาศคลาส list เราใช้ค่าคงที่ max_list_size ที่ประกาศไว้ก่อนการนิยามคลาส

▷ คำถาม 3.3

สังเกตว่าเรากำหนดค่าคงที่ max_list_size ไว้ก่อนการนิยามคลาส ถ้าเราประกาศค่าคงที่ดังกล่าวไว้ภายในคลาสจะเป็นอย่างไร? ทดลองปรับโปรแกรมและพิจารณาข้อผิดพลาดที่คอมไพเลอร์แจ้ง ทำไมจึงเกิดปัญหานั้น? ◁

การประกาศค่าคงที่ max_list_size อย่างที่เราทำ จะทำให้มีชื่อ max_list_size ปรากฏขึ้นในรายการของ

ชื่อทั้งหมดในโปรแกรม ผลก็คือโปรแกรมในส่วนอื่น ๆ ไม่สามารถตั้งชื่อตัวแปรหรือค่าคงที่ด้วยชื่อนี้ได้ อย่างไรก็ตาม เนื่องจากค่าคงที่ดังกล่าวเป็นค่าที่ใช้กับคลาส `list` เท่านั้น เราจึงควรกำหนดค่าคงที่ดังกล่าวให้อยู่ในขอบเขตของคลาส `list`

ในการกำหนดค่าคงที่ดังกล่าว เราไม่สามารถประกาศเป็นตัวแปรสมาชิกในคลาสได้ (ดังที่ได้ทดลองตามคำถาม 3.3) เพราะว่าตัวแปรดังกล่าวจะต้องมีการกำหนดค่าก่อน แต่การใช้ค่าคงที่เพื่อกำหนดขนาดตัวแปรนั้นจำเป็นต้องทราบขนาดเมื่อเริ่มสร้างวัตถุ ซึ่งในขณะนั้นตัวแปรค่าคงที่ที่เราต้องการใช้ยังไม่เรียบร้อย

มีวิธีการกำหนดค่าคงที่ในคลาสหลายรูปแบบ เราจะเลือกวิธีที่สะดวกที่สุดคือการนิยามค่าคงที่ภายในการประกาศชนิดข้อมูล `enum` ที่ไม่กำหนดชื่อ ดังแสดงด้านล่าง สังเกตว่าเราประกาศค่าคงที่ดังกล่าวในส่วนประกาศสมาชิกส่วนตัวของคลาส

โปรแกรมที่ 3.6: นิยามคลาส `list`

```
class list {
public:
  // ...

private:
  enum { max_list_size = 1000 };

  int items_[max_list_size];
  int size_;
};
```

สำหรับค่าคงที่ที่เราต้องการให้อ้างถึงได้จากภายนอกคลาส เราสามารถประกาศในส่วนสาธารณะได้ เช่นตัวอย่างโปรแกรมด้านล่าง

```
class demo {
public:
  enum { my_constant = 12345 };
};

  //...
  cout << demo::my_constant << endl;   // accessing the constant
```

นอกจากวิธีนี้ ยังอีกวิธีหนึ่งคือการกำหนดค่าคงที่แบบสถิตย์ ซึ่งจะกล่าวในส่วน 3.5.2

3.3 รายการบนอาร์เรย์ที่เปลี่ยนขนาดได้

คลาส `list` ที่เราพัฒนาขึ้นเก็บข้อมูลบนอาร์เรย์ที่มีขนาดเท่ากับค่าคงที่ `max_list_size` ในส่วนนี้เราจะปรับปรุงคลาสดังกล่าว ให้ผู้ใช้สามารถระบุขนาดของอาร์เรย์ที่ต้องการใช้ได้เมื่อเริ่มสร้าง การปรับปรุงนี้จะทำให้เราได้ศึกษาความสามารถอื่น ๆ ของภาษา C++

▷ **คำถาม 3.4**

เราจะต้องปรับตัวสร้างของคลาส list อย่างไร? ◁

แทนที่เราจะใช้อาร์เรย์ เราจะใช้พอยน์เตอร์ไปยัง `int` ชี้ไปยังอาร์เรย์ที่เราจะจองให้มีขนาดตามที่ผู้ใช้ระบุ เราจะปรับสมาชิกส่วนตัวของคลาสและตัวสร้างเป็นดังโปรแกรมที่ 3.7 สังเกตว่าฟังก์ชัน `list:list` ที่เป็นตัวสร้างจะรับพารามิเตอร์ `max_size` เพื่อนำมาจองอาร์เรย์ เราใส่ค่าปริยายของขนาดของอาร์เรย์ไว้เท่ากับค่าคงที่ `default_max_list_size`

โปรแกรมที่ 3.7: นิยามสมาชิกส่วนตัว และตัวสร้างของรายการที่เลือกขนาดอาร์เรย์ได้

```
class list {
public:
  list(int max_size=default_max_list_size);
  // ...
private:
  enum { default_max_list_size = 1000 };

  int* items_;
  int size_;
  int max_size_;
};

list::~list(int max_size)
  : size_(0), max_size_(max_size)
{
  items_ = new int[max_size_];
}
```

โปรแกรมที่ 3.8: ตัวทำลายของคลาส `list`

```
class list {
public:
  //...
  list(int max_size=default_max_list_size);
  ~list();
  //...
};

list::~list()
{
  delete [] items_;
}
```

ในโปรแกรมที่เราเขียนขึ้น เมื่อเราประกาศวัตถุของคลาส `list` ตัวสร้างจะจองเนื้อที่หน่วยความจำเป็นเนื้อที่อาร์เรย์และกำหนดให้พอยน์เตอร์ `items_` ชี้ไปที่อาร์เรย์นั้น อย่างไรก็ตามเมื่อวัตถุนั้นหมดขอบเขตการใช้งานแล้ว ถ้าเราไม่ได้ดำเนินการอะไร จะไม่มีการคืนหน่วยความจำคืนกลับให้ระบบ ทำให้เกิดสภาวะที่เรียกว่าหน่วยความจำรั่ว (memory leak) เพื่อหลีกเลี่ยงปัญหาดังกล่าว ถ้าเราจองหน่วยความจำเมื่อเราสร้างวัตถุ เราจะต้องคืนหน่วยความจำก่อนที่วัตถุนั้นถูกทำลาย เราจะทำโดยเขียนฟังก์ชันพิเศษที่เรียกว่าตัวทำลาย

3.3.1 ตัวทำลาย (destructor), ตัวสร้างสำหรับการคัดลอก (copy constructor), และฟังก์ชันกำหนดค่า (assignment operator)

คลาสในภาษา C++ สามารถระบุฟังก์ชันพิเศษเรียกว่า*ตัวทำลาย* ที่ทำหน้าที่เก็บกวาด เช่น คือทรัพยากรที่ได้จองไว้ ฟังก์ชันดังกล่าวจะถูกเรียกโดยอัตโนมัติก่อนที่วัตถุจะถูกทำลาย ตัวทำลาย จะมีชื่อเหมือนกับคลาสแต่ขึ้นต้นด้วย ~ (เครื่องหมาย tilde $\sim$) เช่น `list::~list`

เราจะประกาศฟังก์ชันดังกล่าวในนิยามคลาส และเขียนฟังก์ชันดังกล่าวได้ดังโปรแกรม 3.8

อย่างไรก็ตาม คลาส `list` ที่เราเขียนขึ้น ยังมีการทำงานที่ผิดพลาดอยู่ ทั้งนี้เนื่องมาจากพฤติกรรมมาตรฐานของการคัดลอกวัตถุ เพื่อให้เห็นการทำงานชัดเจน เราจะปรับตัวสร้างและตัวทำลายให้พิมพ์ข้อความว่า `"Constructor called"` และ `"Destructor called"` ตามด้วยตำแหน่งในหน่วยความจำของสมาชิก `items`

พิจารณาโปรแกรมด้านล่างนี้

```
int main()
{
  list a, b;

  a.append(1);
  b = a;
  b.append(2);
}
```

เมื่อโปรแกรมทำงาน ให้ผลลัพธ์ดังด้านล่าง

```
Constructor called 0x9786008
Constructor called 0x9786fb0
Destructor called 0x9786008
Destructor called 0x9786008
```

ในหลาย ๆ ระบบปฏิบัติการโปรแกรมจะถูกปิดลงเพราะเกิดข้อผิดพลาด

▷ คำถาม 3.5
ปัญหาดังกล่าวเกิดขึ้นเนื่องจากอะไร?
(คำใบ้: พิจารณาตำแหน่งของหน่วยความจำของ `items_` เมื่อมีการเรียกตัวทำลาย ◁

คลาสใน C++ มีฟังก์ชันพิเศษสำหรับจัดการกับคลาสมากมาย เช่น การกำหนดค่าเริ่มต้น การทำลาย และการกำหนดค่าผ่านทางเครื่องหมายเท่ากับ เป็นต้น ถ้าเราไม่ระบุฟังก์ชันเล่านี้ C++ จะจัดการดำเนินการให้เราอัตโนมัติ ซึ่งโดยมากแล้วก็จะทำงานได้ถูกต้อง อย่างไรก็ตาม ผลที่ได้ก็ไม่ตรงกับที่เราต้องการ

โดยทั่วไปแล้ว ถ้าคลาสของเรามีสมาชิกที่จองทรัพยากร เช่นจองหน่วยความจำ หรือเปิดปิดแฟ้มข้อมูล เมื่อมีการสร้างหรือใช้งาน และในตัวทำลายมีการคืนทรัพยากรเหล่านั้น มีฟังก์ชันพิเศษสองฟังก์ชันที่เราต้องเขียน เพื่อให้คลาสทำงานได้ถูกต้อง

พิจารณากรณีของคลาส `counter` สมมติเรามีตัวแปร `a` และ `b` เป็นข้อมูลประเภทดังกล่าว เมื่อเราสั่ง

```
a = b;
```

สิ่งที่เกิดขึ้นก็คือมีความพยายามจะกำหนดค่าให้กับวัตถุ `a` ด้วยวัตถุ `b` เนื่องจากไม่มีข้อมูลระบุว่าต้องดำเนินการเช่นใด ระบบจะคัดลอกข้อมูลของทุก ๆ สมาชิกของ `b` ไปยังวัตถุ `a` สิ่งที่เกิดขึ้นเหมือนกับเราสั่ง `a.c = b.c`

▷ คำถาม 3.6
พิจารณาการสั่ง `a = b;` ในกรณีที่ตัวแปรทั้งสองเป็นวัตถุของคลาส `list` อะไรจะเกิดขึ้น? ◁

เราสามารถเปลี่ยนการทำงานของคำสั่งดังกล่าวให้คัดลอกข้อมูลให้ถูกต้อง ได้โดยเขียนฟังก์ชันกำหนดค่า (assignment) ฟังก์ชันดังกล่าวคือฟังก์ชันโอเปอเรเตอร์เท่ากับ `operator=`) นี่คือฟังก์ชันสมาชิกแรกที่เราต้องเขียน

พิจารณาโปรแกรมตัวอย่างด้านล่างนี้

```
void m(list a) { cout << a.size() << endl; }

main()
{
  list b;
  m(b);
  cout << "endding" << endl;
}
```

ผลลัพธ์ของโปรแกรมดังกล่าวเป็นดังด้านล่าง

```
Constructor called0x839b008
0
Destructor called0x839b008
endding
Destructor called0x839b008
```

จากนั้นโปรแกรมก็เกิดปัญหาและถูกระบบปฏิบัติการปิดไปโดยอัตโนมัติ (crash)

▷ **คำถาม 3.7** ความผิดพลาดในการส่งพารามิเตอร์
อธิบายคร่าว ๆ ว่าความผิดพลาดดังกล่าวเกิดขึ้นได้เพราะอะไร? ◁

เมื่อเราเรียกฟังก์ชัน `m` เราส่งตัวแปร `b` เป็นอาร์กิวเมนท์ อย่างไรก็ตาม เนื่องจากพารามิเตอร์ `a` เป็นการส่งพารามิเตอร์แบบ by value ทำให้มีการ "สร้าง" วัตถุ `a` ขึ้นมาใหม่จากวัตถุ `b` โดยใช้ตัวสร้างที่นิยมเรียกว่าตัวสร้างสำหรับการคัดลอก (copy constructor)

เนื่องจากเราไม่ได้เขียนวิธีการสร้างดังกล่าวไว้ ระบบจึงสร้าง `list` ใหม่โดยการคัดลอกข้อมูลจากทุก ๆ สมาชิกของ `b` ไปให้กับวัตถุใหม่ ซึ่งจะทำให้เกิดปัญหาเช่นเดียวกับกรณีฟังก์ชันโอเปอเรเตอร์เท่ากับ

ดังนั้นเพื่อให้การคัดลอกเป็นไปได้ เราจะต้องเขียนฟังก์ชันสมาชิก

```
list& list::list(const list& lst)
```

เพื่อสร้าง `list` ใหม่จาก `lst` สังเกตว่าการประกาศของตัวสร้างสำหรับการคัดลอกจะต้องอยู่ในรูปแบบเช่นนี้ นั่นคือ รับพารามิเตอร์แบบ reference (เพื่อไม่ให้เกิดการคัดลอกซ้อน) และเป็นข้อมูลชนิดค่าคงที่ เพื่อรับประกันว่าจะเกิดการคัดลอกอย่างเดียวเท่านั้นไม่มีการแก้ไขข้อมูลของ `lst` นี่คือฟังก์ชันอีกฟังก์ชันที่เราต้องเขียน

ในโปรแกรมที่ 3.9 เราจะประกาศฟังก์ชันสมาชิกทั้งสอง ซึ่งจะเรียกใช้งานฟังก์ชันส่วนตัว `list::copy_from` พร้อมกับนิยามฟังก์ชันตัวสร้างสำหรับการคัดลอก

สำหรับฟังก์ชันกำหนดค่านั้นจะซับซ้อนกว่าเล็กน้อย ทั้งนี้เพราะว่าผู้ใช้สามารถสั่งงานคำสั่งเช่น `a=a;` ได้ เราจึงต้องระวังไม่คัดลอกข้อมูลในกรณีดังกล่าว

▷ **คำถาม 3.8**
สังเกตว่าฟังก์ชันกำหนดค่าจะต้องคืนหน่วยความจำของอาร์เรย์ที่จองไว้ก่อน ถ้าเขียนโอเปอเรเตอร์เท่ากับให้ทำงานในลักษณะเดียวกับตัวสร้างสำหรับการค้ดลอก จะมีผลอย่างไรถ้าผู้ใช้สั่ง `a=a;` ? ◁

เราจะตรวจสอบได้อย่างไร? คลาสในภาษา C++ จะมีสมาชิกพิเศษชื่อ `this` ซึ่งจะเป็นพอยน์เตอร์ชี้ไปที่วัตถุที่ฟังก์ชันกำลังทำงานอยู่ สมาชิกนี้มีประโยชน์มากในหลาย ๆ กรณี แต่ในที่นี้เราจะใช้เพื่อตรวจสอบว่าพารามิเตอร์

โปรแกรมที่ 3.9: การประกาศฟังก์ชันสมาชิกที่เกี่ยวกับการคัดลอก

```
class list {
public:
  // ...
  list(const list& lst);
  list& operator=(const list& lst);
  // ...
private:
  // ...
  void copy_from(const list& lst);
};

void list::copy_from(const list& lst)
{
  size_ = lst.size_;
  max_size_ = lst.max_size_;
  items_ = new int[max_size_];
  for(int i=0; i<size_; ++i)
    items_[i] = lst.items_[i];
}

list::list(const list& lst)
{
  copy_from(lst);
}
```

`lst` เป็นวัตถุเดียวกับ `this` หรือเปล่า นอกจากนี้เรายังใช้ `this` ในการคืนตัววัตถุเองเป็นผลลัพธ์ของโอเปอเรเตอร์ด้วย โปรแกรมแสดงในโปรแกรมที่ 3.10

สังเกตว่าฟังก์ชันกำหนดค่าคืนค่าเป็นข้อมูลชนิด `list&` และเราคืนค่า `*this` แทนวัตถุที่ได้รับค่าออกไป เพื่อให้เราสั่งคำสั่งเช่น `(a=b).append(10);` ได้

3.4 โครงสร้างข้อมูลสำหรับข้อมูลชนิดใดก็ได้: แม่แบบ

ที่ผ่านมาเราได้พัฒนาโครงสร้างข้อมูลสำหรับจัดการกับรายการที่มีข้อมูลเป็นจำนวนเต็ม โดยเก็บข้อมูลบนอาร์เรย์ อย่างไรก็ตามถ้าพิจารณาให้ดีเราจะพบว่าโปรแกรมที่เราเขียน ไม่จำเป็นต้องทำงานกับ `int` เท่านั้น แต่สามารถทำงานกับข้อมูล

โปรแกรมที่ 3.10: ฟังก์ชันสำหรับโอเปอเรเตอร์เท่ากับ

```
list& list::operator=(const list& l)
{
  if(this == &l)
    return *this;

  delete [] items_;
  copy_from(l);
  return *this;
}
```

ชนิดอื่น ๆ ได้ด้วย

จากโปรแกรมที่เราเขียน เราจะพิจารณาว่าเราดำเนินการอะไรกับข้อมูลประเภท int ที่เราเก็บในรายการบ้าง

- เราสามารถอาร์เรย์ของ int
- เรากำหนดค่าให้กับ items_
- เราเปรียบเทียบข้อมูลด้วยโอเปอเรเตอร์ == (ในฟังก์ชัน list::find

กระบวนการเหล่านี้เป็นอินเทอร์เฟสของวัตถุที่เราจะเก็บใน list ที่โปรแกรมของ list เรียกใช้ ดังนั้น ถ้าเรามีคลาสใด ๆ ที่รองรับกระบวนการเหล่านี้ โปรแกรมที่เราเขียนก็ควรจะทำงานด้วยได้โดยไม่เกิดข้อผิดพลาดระหว่างการคอมไพล์และการทำงาน แต่ผลของการทำงานจะถูกต้องหรือไม่ก็ขึ้นกับว่ากระบวนการต่าง ๆ ข้างต้นนี้ของคลาสนั้น ๆ ทำงานด้วยความหมายตรงตามที่เราคาดไว้หรือไม่

ภาษา C++ รองรับการเขียนโปรแกรมที่ทำงานได้กับชนิดข้อมูลใด ๆ ก็ได้ผ่านทาง*แม่แบบ* (*template*) การเขียนโปรแกรมในลักษณะดังกล่าวนิยมเรียกว่าการโปรแกรมเชิงเจเนอริค (generic programming)

พิจารณาตัวอย่างฟังก์ชัน my_swap ที่ทำหน้าที่สลับค่าของ int แสดงดังด้านล่างนี้[1]

```
void my_swap(int& a, int& b)
{
  int temp = a;  a = b;  b = temp;
}
```

เราสามารถเขียนใหม่โดยใช้แม่แบบ เพื่อให้ฟังก์ชันดังกล่าวทำงานกับข้อมูลคลาสใด ๆ ก็ได้ โดยเพิ่มการระบุให้คลาสของข้อมูลเป็นพารามิเตอร์ของแม่แบบ ดังแสดงด้านล่าง

```
template <class T>
void my_swap(T& a, T& b)
{
  T temp = a;  a = b;  b = temp;
}
```

ฟังก์ชันดังกล่าวสามารถทำงานกับชนิดข้อมูลใด ๆ ก็ได้ ที่รองรับฟังก์ชันกำหนดค่า (นั่นรวมไปถึงคลาส list ที่เราเขียนไว้ในส่วนที่แล้ว) สังเกตว่าฟังก์ชันดังกล่าวถูกนิยามโดยมีชนิดข้อมูล T เป็นพารามิเตอร์ของแม่แบบ ในการเรียกใช้งาน C++ จะเลือกพารามิเตอร์ T ให้โดยอัตโนมัติ ดังแสดงด้านล่าง

```
  int a = 10; int b = 100;
  my_swap(a,b);
  double ad = 10.7; double bd = 112.233;
  my_swap(ad,bd);
```

สิ่งที่เกิดขึ้นเมื่อเราเรียกใช้งานแม่แบบก็คือคอมไพล์เลอร์จะสร้างคลาสหรือฟังก์ชันเฉพาะเจาะจง สำหรับชนิดข้อมูลที่เราระบุเป็นพารามิเตอร์ จากตัวอย่างข้างต้นคอมไพเลอร์จะสร้างฟังก์ชัน my_swap(int&, int&) และ my_swap(double&, double&) เพื่อรองรับการเรียกใช้งานฟังก์ชันแม่แบบแต่ละครั้ง

คลาส list สามารถปรับเปลี่ยนให้เป็นคลาสแม่แบบ list ที่มีชนิดข้อมูลของข้อมูลในรายการเป็นพารามิเตอร์ได้ไม่ยาก ส่วนนิยามแม่แบบแสดงในโปรแกรมที่ 3.11 ฟังก์ชันสมาชิกบางส่วนแสดงในโปรแกรมที่ 3.12

[1]เราตั้งชื่อฟังก์ชันว่า my_swap เพื่อจะได้ไม่ซ้ำกับฟังก์ชัน std::swap ในไลบรารีมาตรฐาน

โปรแกรมที่ 3.11: คลาสแม่แบบ list

```
template<class T>
class list {
public:
  typedef T* iterator;

  list(int max_size=default_max_list_size);
  ~list();

  list(const list& lst);
  list& operator=(const list& lst);

  int size() { return size_; }
  iterator begin() { return &items_[0]; }
  iterator end() { return &items_[size_]; }

  iterator find(T x);
  iterator append(T x);
  void del(iterator p);
  void insert_after(iterator p, T x);

private:
  enum { default_max_list_size = 1000 };

  T* items_;
  int size_;
  int max_size_;

  void copy_from(const list& lst);
};
```

โปรแกรมที่ 3.12: นิยามของสมาชิกบางฟังก์ชันของคลาสแม่แบบ list

```
template<class T>
list<T>::list(int max_size)
  : size_(0), max_size_(max_size)
{
  items_ = new T[max_size_];
}

template<class T>
list<T>::~list()
{
  delete [] items_;
}

template<class T>
typename list<T>::iterator list<T>::append(T v)
{
  if(size_ >= max_size_)
    throw "List␣overflow";
  items_[size_] = v;
  ++size_;
  return &items_[size_ - 1];
}
```

ฟังก์ชันสมาชิกของคลาสแม่แบบสามารถนิยามได้ไม่ต่างจากฟังก์ชันแม่แบบ my_swap ที่เราได้เขียนมาแล้ว สังเกตว่าเราจะนิยามฟังก์ชันไว้ภายในขอบเขตของ `template<class T>` และภายในนั้นก็จะใช้ `T` ได้เหมือนเป็นคลาสทั่วไป นิยามของฟังก์ชันสมาชิกบางฟังก์ชันแสดงในโปรแกรมที่ 3.12

ในการเขียนดังกล่าว มีรายละเอียดเชิงเทคนิคเกี่ยวกับ การใช้งานชนิดข้อมูล `iterator` ที่เรานิยามขึ้นภายในแม่แบบ `list` ในการอ้างถึง `list<T>::iterator` นั้น เราจะต้องระบุคีย์เวิร์ด `typename` นำหน้าเพื่อบอกกับคอมไพเลอร์ว่าสิ่งที่เราอ้างถึงนั้นคือแบบชนิดข้อมูล ดังแสดงในฟังก์ชันสมาชิก `append` ของแม่แบบ `list` ผู้ที่สนใจสามารถอ่านรายละเอียดเพิ่มเติมได้ในส่วนที่ 3.5.7

3.5 รายละเอียดเพิ่มเติมเกี่ยวกับภาษา C++

ภาษา C++ มีรายละเอียดปลีกย่อยเชิงเทคนิคมากพอสมควร ในส่วนนี้เราจะสรุปบางประเด็นที่สำคัญไว้

3.5.1 สรุปฟังก์ชันสมาชิกพิเศษ

คลาสในภาษา C++ มีฟังก์ชันสมาชิกพิเศษหลายฟังก์ชันที่ทำงานพิเศษ ในบทนี้เราได้พิจารณาฟังก์ชันดังรายการต่อไปนี้ (เราจะเขียนชื่อฟังก์ชันโดยสมมติว่าคลาสชื่อ `myclass`)

- `myclass::myclass(...)`
  ตัวสร้าง (constructor) ทำหน้าที่ในการกำหนดค่าเริ่มต้นให้กับวัตถุ จะถูกเรียกเมื่อมีการประกาศวัตถุ ตัวสร้างอาจจะรับพารามิเตอร์ที่จำเป็น แต่จะไม่คืนค่าใด ๆ

- `myclass::~myclass()`
  ตัวทำลาย (destructor) ทำหน้าที่ในการจัดการก่อนที่วัตถุจะถูกทำลาย เช่น คืนทรัพยากรที่วัตถุขอมาจากระบบ เป็นต้น

- `myclass::myclass(const myclass& obj)`
  ตัวสร้างสำหรับคัดลอก ตัวสร้างนี้จะรับพารามิเตอร์แบบอ้างอิงไปยังวัตถุของคลาสเดียวกัน ตัวสร้างนี้สามารถเรียกใช้ได้โดยตรงหรืออาจจะโดนเรียกเมื่อมีการส่งค่าไปยังฟังก์ชันแบบ pass by value โดยทั่วไปแล้ว คลาสที่มีการจองและคืนทรัพยากรเมื่อถูกทำลาย เช่น มีการจองหน่วยความจำ และต้องการรองรับการส่งค่าไปยังฟังก์ชันแบบ pass by value จะต้องเขียนฟังก์ชันนี้

- `myclass& myclass::operator(const myclass& obj)`
  ฟังก์ชันโอเปอเรเตอร์กำหนดค่า ถูกเรียกใช้เมื่อมีการให้ค่าวัตถุของคลาสดังกล่าว เช่นเดียวกับตัวสร้างสำหรับคัดลอก คลาสที่มีการจองทรัพยากรและคืนทรัพยากรมักจำเป็นต้องเขียนฟังก์ชันเหล่านี้

อย่างไรก็ตาม ถ้าโดยมากเราใช้งานวัตถุโดยไม่มีการคัดลอก และเมื่อมีการส่งค่าไปยังฟังก์ชันจะใช้การส่งค่าแบบอ้างอิงหรือส่งด้วยพอยน์เตอร์เท่านั้น เช่น ในกรณีของคลาสที่ใช้งานแบบอ้างอิงเท่านั้น (ดูส่วน 3.5.4) เราก็ไม่จำเป็นต้องเขียนตัวสร้างสำหรับคัดลอก และโอเปอเรเตอร์กำหนดค่า อย่างไรก็ตาม เพื่อป้องกันการคัดลอกโดยไม่ได้ตั้งใจ เราอาจจะประกาศฟังก์ชันทั้งสองเป็นสมาชิกส่วนตัว โดยไม่ต้องนิยามฟังก์ชัน เพื่อให้คอมไพเลอร์ตรวจสอบว่าไม่มีการคัดลอกวัตถุเกิดขึ้นได้ ยกตัวอย่างเช่นดังโปรแกรมด้านล่างนี้

```
class myclass {
  // ...
private:
  myclass(const myclass& obj);
  myclass& operator=(const myclass& obj);
};
```

3.5.2 สมาชิกสถิตย์

3.5.3 การแยกแฟ้มโปรแกรม

ในภาษา C++ เราสามารถแบ่งโปรแกรมเป็นส่วน ๆ เพื่อความสะดวกในการจัดการพัฒนาและแก้ไข รวมถึงช่วยลดเวลาในการคอมไพล์โปรแกรมขนาดใหญ่ เมื่อมีโปรแกรมแค่เพียงบางส่วนที่มีการเปลี่ยนแปลง

3.5.4 การใช้วัตถุแบบ value และแบบ reference

3.5.5 การประกาศฟังก์ชันหลายฟังก์ชันโดยใช้ชื่อเดียวกัน: การโอเวอร์โหลด

3.5.6 คลาสสืบทอด

3.5.7 ความจำเป็นที่ต้องระบุ `typename` เมื่อใช้ชนิดข้อมูลที่ขึ้นกับตัวแปรแม่แบบ

ส่วนนี้อธิบายและยกตัวอย่างแสดงความจำเป็นต้องระบุคีย์เวิร์ด `typename` แต่ไม่จำเป็นมากต่อเนื้อหาหลัก ผู้อ่านสามารถข้ามไปได้[2]

[2]เนื้อหาและตัวอย่างนำมาจากส่วน C.13.5 ของหนังสือของ Stroustrup

เมื่อถูกนิยาม เราไม่ทราบว่าชนิดข้อมูลที่ถูกส่งเป็นพารามิเตอร์เป็นชนิดข้อมูลใด ทำให้เมื่อมีการอ้างถึงสมาชิกของแบบชนิดข้อมูลนั้น คอมไพเลอร์จะไม่สามารถแยกแยะได้ว่าสมาชิกนั้นคืออะไร

พิจารณาตัวอย่างเช่นโปรแกรมถัดไป

```
template <class C> void f()
{
  C::t(a);
}
```

ถ้าพิจารณาเพียงผ่าน ๆ เราอาจจะคิดว่า t เป็นฟังก์ชันแบบสถิตย์ของคลาส (static function) ซึ่งเราจะได้กล่าวถึงต่อไป แต่ก็เป็นไปได้ที่ t จะเป็นชนิดข้อมูล และการนิยามดังกล่าวก็เหมือนกับการนิยามตัวแปรแต่มีการใส่วงเล็บเกิน เช่น int (a);

ดังนั้น ภาษา C++ จึงได้กำหนดว่าถ้ามีการอ้างถึงสมาชิกของคลาสแม่แบบ คอมไพเลอร์จะพิจารณาให้สมาชิกนั้นไม่เป็นชนิดข้อมูล นอกเสียจากเราจะระบุว่าสมาชิกนั้นเป็นชนิดของข้อมูล โดยการเขียนคีย์เวิร์ด typename นำหน้า

ข้อกำหนดดังกล่าวเป็นจริงสำหรับการอ้างถึง list<T>::iterator ด้วย ดังนั้น เราจึงต้องระบุว่าชื่อดังกล่าวหมายถึงชนิดข้อมูลด้วย typename

บทที่ 4

การเรียกตัวเอง

การเรียกตัวเองเป็นแนวคิดที่ทรงพลังมาก เราจะทำความเข้าใจกับแนวคิดดังกล่าวผ่านทางตัวอย่างและคำถาม เราจะเริ่มจากปัญหาที่ง่ายและตรงไปตรงมาซึ่งสามารถแก้ไขได้ด้วยอัลกอริทึมแบบวนซ้ำทั่วไป เราจะพิจารณาปัญหาที่ยากขึ้นในตอนท้ายของบทนี้ อย่างไรก็ตามแนวคิดของการเรียกตัวเองจะเป็นแนวคิดพื้นฐานในการทำความเข้าใจโครงสร้างข้อมูลที่เราจะศึกษาในบทอื่น ๆ ด้วย

ในการทำความเข้าใจกับการเรียกตัวเองนั้น ต้องใช้ความรู้พื้นฐานเกี่ยวกับโปรแกรมย่อย (function ในภาษา C++) ผู้อ่านสามารถทบทวนได้ที่ภาคผนวก **??**

เราจะเริ่มจากปัญหาเกี่ยวกับการคำนวณที่ข้อมูลป้อนเข้าเป็นจำนวนเต็ม จากนั้นจะพิจารณาปัญหาที่ข้อมูลป้อนเข้ามีลักษณะเป็นรายการ เช่น ปัญหาการหาค่ามากที่สุด และปัญหาการจัดเรียงข้อมูล

4.1 การคำนวณทางพีชคณิต

เราจะเขียนโปรแกรมย่อยที่บวกจำนวนธรรมชาติสองจำนวน ตัวอย่างของโปรแกรมย่อยนี้แสดงดังด้านล่าง

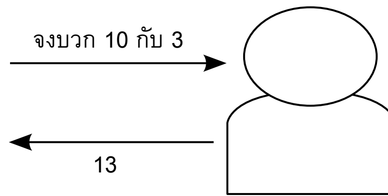
บวกจำนวนธรรมชาติ A **กับ** B (A4.1)
 คำนวณค่า $A + B$ แล้วคืนผลลัพธ์

เมื่อเราเรียกใช้โปรแกรมย่อยดังกล่าวให้บวก 10 กับ 3 เราจะได้ผลลัพธ์เป็น 13 ลักษณะของการเรียกใช้ แสดงในรูป 4.1

เราจะปรับโปรแกรมย่อยดังกล่าว ให้เป็นโปรแกรมย่อยแบบเรียกตัวเอง สมมติว่าเราทราบวิธีการเพิ่มค่าจำนวนธรรมชาติขึ้น 1 และลดค่าจำนวนธรรมชาติลง 1 พิจารณาวิธีการบวกจำนวนธรรมชาติ A เข้ากับ B ดังอัลกอริทึมที่ A4.2

นิยามข้างต้นมีลักษณะเหมือนงูกินหาง เพราะว่าเรากำลังนิยามการบวกจำนวนธรรมชาติด้วยการบวกจำนวนธรรมชาติ อย่างไรก็ตาม เราจะละความสงสัยดังกล่าวไว้ก่อนแล้วทดลองบวก 10 กับ 3 ดังนี้

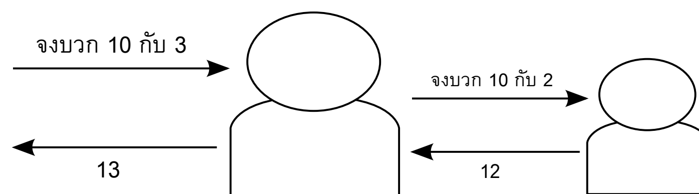
- เนื่องจาก 3 ไม่เท่ากับ 0 เราจึงคำนวณค่า $3 - 1 = 2$ จากนั้นเราต้องการคำนวณหาผลบวกของ 10 กับ 2 เมื่อได้ผลบวกแล้ว เราจะเพิ่มค่าขึ้น 1 เพื่อได้ผลบวกของ 10 กับ 3 ตามต้องการ



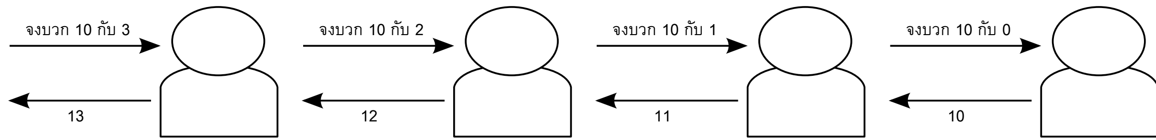
รูปที่ 4.1: ตัวอย่างการเรียกใช้โปรแกรมย่อย

บวกจำนวนธรรมชาติ A กับ B (A4.2)

IF $B = 0$ THEN RETURN A and EXIT
ELSE
LET $C \leftarrow B - 1$
ให้ D เท่ากับผลบวกของ A กับ C
RETURN $D + 1$



รูปที่ 4.2: ตัวอย่างการเรียกใช้โปรแกรมย่อยที่เรียกใช้โปรแกรมย่อยเพื่อคำนวณผลบวกของ 10 กับ 2



รูปที่ 4.3: ตัวอย่างการเรียกใช้โปรแกรมย่อยที่เรียกตัวเอง

จากขั้นตอนด้านบน ถ้าเราทราบว่าผลบวกของ 10 กับ 2 คือ 12 เมื่อเพิ่มค่าขึ้น 1 เราจะได้ผลลัพธ์ของ 10 กับ 3 ซึ่งมีค่าเท่ากับ 13 รูป 4.2 แสดงการทำงานของโปรแกรมย่อยดังกล่าว

▷ **คำถาม 4.1** 10 + 2
เราจะหาผลลัพธ์ของการบวก 10 กับ 2 ได้อย่างไร? ◁

เราก็จะหาผลลัพธ์ด้วยวิธีเดียวกัน ซึ่งในการผลบวก เราจะต้องหาผลลัพธ์ของการบวก 10 กับ 1 และจะเป็นเช่นนี้เป็นเรื่อย ๆ ดังตัวอย่างด้านล่าง (รูปที่ 4.3 แสดงลักษณะการคำนวณ)

- เนื่องจาก 3 ไม่เท่ากับ 0 เราจึงคำนวณค่า $3 - 1 = 2$ จากนั้นเราต้องการคำนวณหาผลบวกของ 10 กับ 2 ในการคำนวณผลบวกดังกล่าว เราจะใช้วิธีการเดิม
  - เนื่องจาก 2 ไม่เท่ากับ 0 เราจึงคำนวณค่า $2 - 1 = 1$ จากนั้นเราต้องการคำนวณหาผลบวกของ 10 กับ 1 ในการคำนวณผลบวกดังกล่าว เราจะใช้วิธีการเดิม
    - ∗ เนื่องจาก 1 ไม่เท่ากับ 0 เราจึงคำนวณค่า $1 - 1 = 0$ จากนั้นเราต้องการคำนวณหาผลบวกของ 10 กับ 0 ในการคำนวณผลบวกดังกล่าว เราจะใช้วิธีการเดิม
      - · เนื่องจาก 0 เท่ากับ 0 ผลลัพธ์ของการบวก 10 กับ 0 คือ 10
    - ∗ เมื่อเราได้ผลลัพธ์ของการบวก 10 กับ 0 แล้ว (คือ 10) เราคำนวณ $10 + 1$ ได้ผลลัพธ์ 11 ซึ่งเป็นผลลัพธ์ของการบวก 10 กับ 1
  - เมื่อเราได้ผลลัพธ์ของการบวก 10 กับ 1 แล้ว (คือ 11) เราคำนวณ $11+1$ ได้ผลลัพธ์ 12 ซึ่งเป็นผลลัพธ์ของการบวก 10 กับ 2
- เมื่อเราได้ผลลัพธ์ของการบวก 10 กับ 2 แล้ว (คือ 12) เราคำนวณ $12 + 1$ ได้ผลลัพธ์ 13 ซึ่งเป็นผลลัพธ์ของการบวก 10 กับ 3

▷ **คำถาม 4.2** การลบ
เขียนขั้นตอนการลบจำนวนธรรมชาติ A กับ B ในรูปแบบเดียวกับการคำนวณผลบวก ◁

▷ **คำถาม 4.3** การคูณ
เขียนขั้นตอนการคูณจำนวนธรรมชาติ A กับ B ในรูปแบบเดียวกับการคำนวณผลบวก ◁

▷ **คำถาม 4.4** ศูนย์

ถ้าเราตัดบรรทัดแรกที่ระบุเงื่อนไขที่ทำงานเมื่อ $B = 0$ ออก เมื่อเราดำเนินการตามขั้นตอนวิธีดังกล่าว ผลลัพธ์จะเป็นเช่นใด ◁

เราสามารถเขียนอัลกอริทึมดังกล่าวเป็นโปรแกรมได้ไม่ยากดังนี้

```
int add(int a, int b)
{
  if(b==0)
    return a;

  int c = b-1;
  return 1 + add(a,c);
}
```

4.2 ค่าสูงสุด

เราจะออกแบบอัลกอริทึมแบบเรียกตัวเองสำหรับการคำนวณค่าสูงสุด กล่าวคือ ให้อาร์เรย์ A ของจำนวนเต็ม n จำนวน เราต้องการคำนวณค่าสูงสุด

กรณีที่เราสามารถตอบคำถามได้ง่าย คือกรณีที่ $n = 1$ กล่าวคือเราสามารถตอบได้ทันทีว่าค่าสูงสุดเท่ากับ $A[0]$

การคำนวนค่าสูงสุดของอาร์เรย์ A ที่มีสมาชิก n ตัว (ขั้นฐาน) (A4.3)

IF $n = 1$ THEN RETURN $A[0]$ and EXIT

ELSE

จะต้องออกแบบต่อไป

สำหรับบทนี้เพื่อความสะดวกในการเขียน แทนที่เราจะเรียกข้อมูลในอาร์เรย์โดยเขียนในวงเล็บเหลี่ยม เช่น $A[4]$ หรือ $A[i]$ เราจะเขียนเป็นตัวห้อย เช่น A_4 หรือ A_i และจะเขียนอาร์เรย์โดยระบุข้อมูลในอาร์เรย์ในวงเล็บเหลี่ยม เช่น อาร์เรย์ $[2, 3, 5, 7, 11]$ เป็นต้น

สำหรับกรณีทั่วไป เราจะเริ่มโดยพิจารณาปัญหาเมื่อข้อมูลนำเข้ามีขนาดเล็กลง โดยเราจะให้

$$A' = [A_0, A_1, \ldots, A_{n-2}]$$

นั่นคือ A' คืออาร์เรย์ A ที่ตัดข้อมูลตัวสุดท้ายทิ้งไป

ปัญหานี้เราจะเรียกว่า *ปัญหาย่อย* (subproblem) เราจะสมมติว่าเราสามารถหาคำตอบของปัญหาได้ กล่าวคือ ให้ M คือค่าสูงสุดของอาร์เรย์ A'

▷ **คำถาม 4.5** ค่าสูงที่สุดจาก M

สมมติว่าเราทราบว่า M คือค่าสูงสุดของอาร์เรย์ $A' = [A_0, A_1, \ldots, A_{n-2}]$ เราสามารถคำนวณค่าสูงสุดของอาร์เรย์ $A = [A_0, A_1, \ldots, A_{n-1}]$ ได้อย่างไร?

(คำใบ้: ถ้า M ไม่ใช่ข้อมูลที่มีค่าสูงสุดของอาร์เรย์ ค่าอื่นที่เป็นไปได้คือค่าใด?) ◁

เมื่อเราพิจารณาข้อมูลสูงสุดของอาร์เรย์ A มีความเป็นไปได้สองกรณีคือ กรณีที่ข้อมูลสูงที่สุดอยู่ในอาร์เรย์ A' ในอีกกรณีหนึ่งคือข้อมูลสูงที่สุดคือ x_n ถ้าเป็นในกรณีแรกค่าสูงสุดคือ M ในอีกกรณีหนึ่งคือ ข้อมูลตัวสุดท้าย A_{n-1} ใน A ซึ่งเราสามารถเทียบข้อมูลทั้งสองเพื่อหาค่าสูงที่สุดได้ ดังอัลกอริทึมด้านล่าง

การคำนวนค่าสูงสุดของอาร์เรย์ A ที่มีสมาชิก n ตัว (A4.4)

1. IF $n = 1$ THEN RETURN $A[0]$ and EXIT
2. ELSE
3. ให้ M คือค่าสูงสุดของอาร์เรย์ A' ที่มีสมาชิก $n-1$ ตัว เมื่อ $A' = [A_0, A_1, \ldots, A_{n-2}]$
4. IF $A_{n-1} > M$ THEN
5. RETURN A_{n-1}
6. ELSE
7. RETURN M

คำถามก็คือ ในขั้นตอนที่ 3 เราจะคำนวณหาค่า M ได้อย่างไร? สังเกตว่าปัญหาดังกล่าวก็คือปัญหาเดียวกับปัญหาเริ่มต้นที่เราต้องการจะแก้ แต่มีข้อมูลป้อนเข้าที่แตกต่างออกไป ดังนั้น ในการแก้ปัญหาดังกล่าว เราก็จะเรียกฟังก์ชันที่เราเตรียมไว้สำหรับแก้ปัญหานั้น นั่นก็คือฟังก์ชันที่เรากำลังเขียนอยู่นั่นเอง

จากแนวคิดดังกล่าว เราสามารถพัฒนาโปรแกรมที่คำนวณค่าสูงสุดได้ดังนี้

```
int arraymax(int ar[], int n)
{
  if(n==1)
    return ar[0];
  else {
    int m = arraymax(ar,n-1);    // Line 6
    if(m > xn)
      return m;
    else
      return ar[n-1];
  }
}
```

สังเกตว่าในโปรแกรมดังกล่าว เราไม่ได้สร้างอาร์เรย์ A' ขึ้นมาใหม่แต่อย่างใด เนื่องจากวิธีการเราสามารถพิจารณาให้ A' เป็นแค่ส่วนต้นของอาร์เรย์ A สิ่งที่เราทำเพื่อป้อนเป็นข้อมูลป้อนเข้าให้กับฟังก์ชัน `arraymax` คือปรับจำนวนของข้อมูลในอาร์เรย์เท่านั้นเอง

▷ **คำถาม 4.6** ผลบวกของอาร์เรย์
ออกแบบอัลกอริทึมที่รับอาร์เรย์ A ของจำนวนเต็ม n จำนวน แล้วคำนวณผลรวมของข้อมูลในรายการ ◁

▷ **คำถาม 4.7** เปลี่ยนทิศทาง
แทนที่เราจะนิยามให้ A' เป็นอาร์เรย์ A ที่ลบข้อมูลตัวสุดท้ายออกไป เราอาจจะนิยาม

$$A' = [A_1, A_2, \ldots, A_{n-1}]$$

นั่นคือ ให้เป็นอาร์เรย์ A ที่ลบข้อมูลตัวแรกออก จงพัฒนาอัลกอริทึมหาค่ามากที่สุดโดยใช้การเรียกตัวเองบนอาร์เรย์ A' ที่นิยามใหม่นี้ และเขียนโปรแกรม

(คำแนะนำเพิ่มเติม: ในการส่งค่า A' ในการเรียกตัวเองอาจจะดูซับซ้อนกว่าวิธีเก่าเล็กน้อย แต่อย่าลืมว่าเราสามารถบวกพอยน์เตอร์ได้) ◁

4.2.1 ฟังก์ชันเรียกตัวเอง

อัลกอริทึมดังกล่าวสามารถพิจารณาว่าเป็นการคำนวณค่าฟังก์ชัน f ที่มีนิยามดังต่อไปนี้

$$f([A_0, A_1, \ldots, A_{n-1}]) = \begin{cases} A_0, & \text{ถ้า } n = 1 \\ \max\{A_{n-1}, f([A_0, A_1, \ldots, A_{n-2}])\}, & \text{ในกรณีอื่น ๆ} \end{cases}$$

4.2.2 การทำซ้ำกับการเรียกตัวเอง

การคำนวณค่าสูงสุดในรายการเป็นปัญหาพื้นฐานที่เหมาะกับอัลกอริทึมแบบทำซ้ำ ด้านล่างแสดงส่วนของโปรแกรมดังกล่าว

```
int arraymax(int ar[], int n)
{
  int m = ar[0];
  for(int i=0; i<n; ++i)
    if(ar[i] > m)
      m = ar[i];
  return m;
}
```

ผู้อ่านที่สนใจอาจเริ่มสงสัยว่าการพัฒนาโปรแกรมแบบเรียกตัวเองมีประโยชน์อย่างไร?

TODO: อธิบายเพิ่มเติม

อย่างไรก็ตาม ภาษาโปรแกรมภายใต้กรอบคิดที่ไม่ใช่ภาษาเชิง imperative อาจจะไม่มีโครงสร้างควบคุมที่เป็นการวนรอบ เช่น ภาษาเชิงฟังก์ชัน เช่น Haskell หรือ ML หรือภาษาเชิงตรรก เช่น Prolog โปรแกรมที่เขียนบนภาษาในกลุ่มนี้ จะไม่มีแนวคิดเกี่ยวกับการเปลี่ยนแปลงค่าของตัวแปร จึงทำให้โปรแกรมที่เขียนปราศจากผลข้างเคียง (side effect) ทั้งหมดนี้ทำให้สามารถทดสอบโปรแกรมสะดวกขึ้น ลดข้อผิดพลาด และทำให้โปรแกรมสามารถทำงานแบบขนานได้ง่ายขึ้นด้วย

4.3 การจัดเรียงข้อมูล

ในส่วนนี้เราจะพิจารณาตัวอย่างการออกแบบอัลกอริทึมโดยการคิดแบบเรียกตัวเอง ปัญหาที่เราสนใจคือปัญหาการจัดเรียงข้อมูล ซึ่งเป็นปัญหาเกี่ยวกับการประมวลผลข้อมูลที่สำคัญมาก

เรามีจำนวนเต็ม n จำนวน อยู่ในอาร์เรย์ x (นั่นคือข้อมูลคือ $[x_0, x_1, \ldots, x_{n-1}]$) เราต้องการเรียงข้อมูลในอาร์เรย์ดังกล่าวจากน้อยไปมาก

▷ **คำถาม 4.8** กรณีง่าย

มีกรณีใดบ้างที่เราสามารถเรียงข้อมูลในอาร์เรย์ x ได้ง่ายมาก ◁

กรณีที่อาร์เรย์มีข้อมูลเพียง 1 จำนวน การจัดเรียงอาร์เรย์ดังกล่าวสามารถทำได้โดยง่าย นั่นคือ ไม่ต้องทำอะไรเลย ถ้าไม่ใช่กรณีดังกล่าว เราจะเรียงข้อมูลได้อย่างไร?

แทนที่จะเริ่มแบบไม่มีอะไรเลย เราจะสมมติว่าในการพัฒนาโปรแกรมในการจัดการเรียงข้อมูล n ตัว เราสามารถเรียงข้อมูลในอาร์เรย์ที่มีข้อมูลจำนวน m ตัวได้

สังเกตว่าเรากำลังแก้ปัญหาการจัดเรียงข้อมูล แต่เราสมมติว่าเราแก้ปัญหานั้นได้แล้ว! ถ้าสมมติกันได้ง่าย ๆ แบบนี้โปรแกรมที่ใช้เรียงข้อมูลของเราคงมีลักษณะดังนี้

เรียงข้อมูลในอาร์เรย์ x ที่มีข้อมูล n ตัว (A4.5)
  เรียงข้อมูลในอาร์เรย์ x ที่มีข้อมูล n ตัว
  คืนผลลัพธ์

▷ **คำถาม 4.9** อัลกอริทึมสมมติ
อัลกอริทึมดังกล่าวทำงานได้จริงหรือไม่? เพราะเหตุใด? ◁

อัลกอริทึมข้างต้นนั้นทำงานไม่ได้จริง เพราะว่าการทำงานของอัลกอริทึมจะเป็นการเรียกตัวเองไปเรื่อย ๆ ไม่มีวันสิ้นสุด

ดังนั้นการสมมติว่าเราสามารถเรียงข้อมูลในอาร์เรย์ได้นั้น จึงต้องมีขอบเขตที่ก่อให้เกิด "ความก้าวหน้า" ของการทำงาน ในที่นี้เราจะใส่เงื่อนไขเพิ่มเติมว่าขนาดของอาร์เรย์ที่เราสามารถสมมติว่าเรียงได้นั้นมีค่าไม่เกิน $n-1$ หรือให้ $m < n$

ดังนั้นเป้าหมายของเราจะเปลี่ยนจากการพยายามเรียงข้อมูล n ตัว ไปเป็นการพยายามทำงานบางอย่าง เพื่อทำให้งานที่เหลือกลายเป็นปัญหาการเรียงข้อมูล m ตัว โดยที่ $m < n$

▷ **คำถาม 4.10** แนวทางการแก้ปัญหา
แนวทางที่เราระบุในย่อหน้าข้างต้นไม่ใช่แนวทางเดียวในการคิดแบบเรียกตัวเอง มีแนวทางอื่นอีกหรือไม่?
(คำใบ้: สลับ) ◁

เราจะได้พิจารณาแนวทางอื่นในการคิดแบบเรียกตัวเองในตัวอย่างถัด ๆ ไป

สำหรับปัญหาการจัดเรียงข้อมูลนี้ เราจะเริ่มโดยการพยายามหาคำตอบบางส่วนให้ได้ เพื่อที่จะได้ทำให้งานที่เหลือลดลง

▷ **คำถาม 4.11** คำตอบบางส่วน
คำตอบบางส่วนของปัญหาการเรียงข้อมูลมีได้หลายแบบ ลองยกตัวอย่างสัก 2 แบบ ◁

ในที่นี้เราจะพยายามทำให้ข้อมูลบางตัวในอาร์เรย์ อยู่ในตำแหน่ง "ที่ถูกต้อง" นั่นคือ ตำแหน่งที่ข้อมูลนั้นจะอยู่เมื่ออาร์เรย์ดังกล่าวถูกเรียงแล้ว

▷ **คำถาม 4.12** ตำแหน่งที่ถูกต้อง
สมมติเราพิจารณาข้อมูล x_0 จะหาตำแหน่งที่ถูกต้องของ x_0 ได้อย่างไร? ◁

แทนที่เราจะเริ่มต้นด้วยข้อมูลบางตัว เช่น x_0 แล้วหาตำแหน่งที่ถูกต้อง เราอาจจะมองกลับกัน คือเริ่มที่ตำแหน่งบางตำแหน่ง แล้วหาข้อมูลที่ควรจะอยู่ในตำแหน่งดังกล่าว

▷ **คำถาม 4.13** ข้อมูลถูกต้อง
(1) เราจะหาข้อมูลที่ควรจะมีดัชนีเท่ากับ 0 ในอาร์เรย์ที่เรียงแล้วได้อย่างไร? (2) เราจะหาข้อมูลที่ควรจะมีดัชนีเท่ากับ $n-1$ ในอาร์เรย์ที่เรียงแล้วได้อย่างไร? ◁

อัลกอริทึมในการเรียงอาร์เรย์ที่เราจะพัฒนาขึ้นนั้น ขึ้นกับว่าเราอยากจะตอบคำถาม (1) หรือ (2) ในคำถามที่ 4.13 ข้างต้นมากกว่ากัน เพื่อความง่ายในการส่งค่าพารามิเตอร์ เราจะเลือกแนวทางจากคำถาม (2) (ดูคำถามที่ 4.7 ประกอบ)

แผนการของอัลกอริทึมเราจะเป็นดังด้านล่าง

> **เรียงข้อมูลในอาร์เรย์ x ที่มีข้อมูล n ตัว** (A4.6)
> ย้ายข้อมูลที่มีค่ามากที่สุดไปยังตำแหน่ง $n-1$ โดยใช้การสลับกับข้อมูลอื่น ๆ
> เรียงข้อมูลในอาร์เรย์ x' ที่มีข้อมูล $n-1$ ตัว โดยที่ $x' = [x_0, x_1, \ldots, x_{n-2}]$

เมื่อได้แผนการแล้ว การพัฒนาอัลกอริทึมที่สมบูรณ์และโปรแกรมก็ไม่ใช่เรื่องยากแต่อย่างใด โปรแกรมที่ 4.1 แสดงอัลกอริทึมการจัดเรียงดังกล่าว โดยละฟังก์ชัน `find_max_index` ที่หาดัชนีของข้อมูลที่มากที่สุด และฟังก์ชัน `swap` ไว้

อัลกอริทึมนี้ถ้าไม่เขียนโดยการเรียกตัวเอง นิยมเรียกว่าการจัดเรียงแบบเลือก (selection sort)

โปรแกรมที่ 4.1: การจัดเรียงข้อมูลด้วยการจัดเรียงแบบเลือก

```
int find_max_index(int x[], int n) { // ... }

void sort(int x[], int n)
{
  if(n<=1)
    return;
  int maxi = find_max_index(x, n);
  swap(x[n-1], x[maxi]);
  sort(x, n-1);
}
```

4.4 การค้นหาแบบทวิภาค: นิยามแบบเรียกตัวเอง

ในส่วนนี้เราจะพิจารณาอัลกอริทึมการค้นหาแบบทวิภาคในส่วนที่ 2.2.1 ซ้ำอีกครั้งหนึ่ง โดยเราจะเขียนฟังก์ชันการค้นหาใหม่ในรูปแบบของอัลกอริทึมแบบเรียกตัวเอง

สังเกตว่าในแต่ละครั้งที่เราเปรียบเทียบค่าที่ต้องการค้นหากับข้อมูลในอาร์เรย์ เราจะสามารถปรับขอบเขตของดัชนีที่เป็นไปได้ จากนั้นเราก็จะพยายามแก้ปัญหาเดิม คือ ค้นหาข้อมูลในอาร์เรย์ภายใต้ของเขตที่เปลี่ยนไป

ดังนั้นถ้าเราพิจารณาให้ดี ปัญหาที่เราต้องการจะแก้คือ:

> ค้นหาข้อมูล q จากอาร์เรย์ A ขนาด n ที่ข้อมูลเรียงลำดับจากน้อยไปมาก โดยที่ดัชนีที่เป็นไปได้มีค่าระหว่าง s ถึง t

โดยในการค้นครั้งแรก เราจะให้ $s = 0$ และ $t = n-1$ นั่นเอง

เราปรับอัลกอริทึม A2.3 ใหม่ให้เป็นอัลกอริทึมแบบเรียกตัวเองได้ดังอัลกอริทึม A4.7 ในการเรียกใช้อัลกอริทึมดังกล่าว เราจะเรียกผ่านอัลกอริทึม A4.8

▷ **คำถาม 4.14**
อิมพลีเมนท์อัลกอริทึม A4.7 และทดสอบ ◁

การออกแบบอัลกอริทึมโดยคิดเชิงเรียกตัวเองทำให้เรามองปัญหาแบบเฉพาะเจาะจงมากขึ้น อย่างไรก็ตามอัลกอริทึมที่ออกแบบได้มักดูเสมือนว่าทำงานไม่ครบถ้วนสมบูรณ์ เราจะทราบได้อย่างไรว่าอัลกอริทึมของเราทำงานได้ถูกต้องแล้ว?

```
BinarySearch(q, A, s, t)                                        (A4.7)
1. IF s > t THEN RETURN −1 and EXIT
2. m ← ⌊(s + t)/2⌋
3. IF A[m] = q THEN
4.     RETURN m and EXIT
5. IF A[m] < q THEN
6.     RETURN BinarySearch(q, A, m + 1, t)
7. ELSE
8.     RETURN BinarySearch(q, A, t, m − 1)
```

```
Find(q, A, n)                                                   (A4.8)
1. RETURN BinarySearch(q, A, 0, n − 1)
```

4.5 การอุปนัยเชิงคณิตศาสตร์

ในการพัฒนาโปรแกรมใด ๆ สิ่งที่เราต้องสนใจก่อนที่จะพิจารณาถึงประสิทธิภาพของโปรแกรม ก็คือความถูกต้องของโปรแกรมนั้น ๆ อย่างไรก็ตาม การพิสูจน์ความถูกต้องของโปรแกรมย่อยแบบเรียกตัวเองนั้นมีความซับซ้อนเป็นพิเศษ ขั้นตอนการทำงานทั้งหมดมักไม่ได้ถูกระบุออกมาอย่างชัดเจนภายในโปรแกรมย่อยนั้น แต่จะเป็นการโยนภาระการทำงานให้กับโปรแกรมย่อยนั้นเอง

ในส่วนนี้เราจะศึกษาเกี่ยวกับเทคนิคการพิสูจน์ที่เรียกว่า *การอุปนัยทางคณิตศาสตร์* (mathematical induction) ซึ่งเป็นเครื่องมือสำคัญในการพิสูจน์ความถูกต้องของโปรแกรมย่อยแบบเรียกตัวเอง

เราจะเริ่มจากตัวอย่าง พิจารณาผลรวม $1 + 2 + 3 + \cdots + n$ ถ้ายังพอจำได้ ผลรวมดังกล่าวมีค่าเท่ากับ $\frac{n(n+1)}{2}$ อย่างไรก็ตามเราจะพิสูจน์ได้อย่างไรว่าผลรวมดังกล่าวมีค่าเท่ากับนิพจน์ที่อ้างมาจริง

เราอาจทดลองได้โดยการแทนค่า n ด้วยค่าต่าง ๆ เช่น

- เมื่อ $n = 1$ เราจะได้ว่าผลรวมข้างต้นมีค่าเท่ากับ 1 ในขณะที่นิพจน์ $\frac{n(n+1)}{2} = \frac{1\cdot 2}{2} = 1$ เช่นกัน
- เมื่อ $n = 2$ เราจะได้ว่า ผลรวมมีค่า $1 + 2 = 3$ ในขณะที่ $\frac{n(n+1)}{2} = \frac{2\cdot 3}{2} = 3$ เช่นกัน

เราสามารถไล่แทนค่าไปได้เรื่อย ๆ ... อย่างไรก็ตามวิธีดังกล่าวไม่สามารถพิสูจน์ได้ว่าผลรวมมีค่าเท่ากับนิพจน์ที่อ้างมาได้ เนื่องจากอาจมีค่าบางค่าที่เราไม่ได้แทนลงไปที่ผลรวมมีค่าไม่เท่ากับนิพจน์ดังกล่าว

ในการพิสูจน์โดยทั่วไป หลายครั้งเราจะเริ่มจากสิ่งที่เราทราบ จากนั้นจะใช้เหตุผลเพื่อเชื่อมโยงให้ได้ผลลัพธ์ตามที่เราต้องการ และในบางครั้งเราก็จะเริ่มจากผลที่เราต้องการ แล้วจึงพยายามโยงสิ่งที่เราทราบมาหาผลดังกล่าวโดยใช้ลำดับการให้เหตุผลที่ถูกต้อง

การอุปนัยทางคณิตศาสตร์ นั้นมีลักษณะที่ดูผิวเผินแล้วมีลักษณะแตกต่างออกไป เราจะเริ่มจากตัวอย่างการพิสูจน์ว่านิพจน์ดังกล่าวข้างต้นมีค่าเท่ากับผลรวมตามที่เราอ้างไว้

ขั้นที่ 1. เราจะเริ่มโดยการตรวจสอบว่านิพจน์ดังกล่าวมีค่าเท่ากับผลรวมเมื่อ $n = 1$ ซึ่งขั้นตอนนี้เราได้ทำไปแล้วตอนต้น

ขั้นที่ 2a. จากนั้นเราสมมติว่านิพจน์ดังกล่าวมีค่าเท่ากับผลรวมเมื่อ $n = k$ สำหรับจำนวนนับ k ใด ๆ ที่มีค่ามากกว่าหรือเท่ากับ 1 นั่นคือ

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

ขั้นที่ 2b. จากข้อสมมติดังกล่าว เราจะแสดงว่านิพจน์ดังกล่าวมีค่าเท่ากับผลรวมเมื่อ $n = k+1$ ด้วย นั่นคือเราจะพิสูจน์ว่า:

$$1 + 2 + \cdots + k + (k+1) = \frac{(k+1)(k+2)}{2}$$

จากข้อสมมติที่ $n = k$ เราสามารถพิสูจน์สมการเป้าหมายได้ไม่ยากนัก ดังนี้

$$\begin{aligned}
1 + 2 + \cdots + k + (k+1) &= (1 + 2 + \cdots + k) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1) \quad \text{(โดยการแทนค่าจากข้อสมมติ)} \\
&= \frac{k(k+1) + 2(k+1)}{2} \quad \text{(จัดพจน์)} \\
&= \frac{(k+1)(k+2)}{2} \quad \text{(ดึงพจน์ย่อยร่วม)}
\end{aligned}$$

ก่อนจะพิจารณาต่อไปเราจะทบทวน (อย่างไม่เป็นรูปแบบ) ว่าในขั้นตอนทั้ง 2 ขั้น เราได้แสดงอะไรไปบ้าง

- เราแสดงว่านิพจน์ดังกล่าวเท่ากับผลรวมจริงเมื่อ $n = 1$
- เราสมมติให้นิพจน์ดังกล่าวเท่ากับผลรวมเมื่อ $n = k$ สำหรับ $k \geq 1$ ใด ๆ จากนั้นแสดงว่านิพจน์ดังกล่าวเท่ากับผลรวมเมื่อ $n = k + 1$ ขั้นตอนนี้กล่าวในอีกทางหนึ่งก็คือการพิสูจน์ว่า:

  ถ้า นิพจน์ดังกล่าวเท่ากับผลรวมเมื่อ $n = k$ สำหรับ $k \geq 1$ ใด ๆ แล้ว นิพจน์ดังกล่าวเท่ากับผลรวมเมื่อ $n = k + 1$ ด้วย

ความจริงทั้งสองข้อเมื่อนำมารวมกันกลับมีพลังในการพิสูจน์มากมาย กล่าวคือ เมื่อเราทราบว่านิพจน์เท่ากับผลรวมเมื่อ $n = 1$ (จากข้อ 1.) เราสามารถนำความจริงที่พิสูจน์ในข้อ 2 มาใช้ได้ โดยพิจารณากรณีที่ $k = 1$ ดังนั้นความจริงข้อ 2 ทำให้เราสรุปได้ว่า นิพจน์เท่ากับผลรวมเมื่อ $n = k + 1 = 2$

ถ้าเราเอาความจริงข้อ 2 มาใช้อีกรอบ โดยครั้งนี้ให้ $k = 2$ (สังเกตว่าเราสามารถความจริงข้อ 2 มาใช้ได้เนื่องจากเราทราบว่าผลรวมเท่ากับนิพจน์เมื่อ $n = 2$ แล้ว) เราจะสามารถสรุปได้ว่า นิพจน์เท่ากับผลรวมเมื่อ $n = k + 1 = 3$ ด้วย

เราสามารถนำความจริงข้อ 2 มาใช้ไปเรื่อย ๆ ทำให้เราสามารถอ้างไปได้เรื่อย ๆ ว่านิพจน์นั้นเท่ากับผลรวมเมื่อ $n = 4, 5, 6, 7, \ldots$ นั่นคือเราสามารถสรุปได้ว่านิพจน์ดังกล่าวมีค่าเท่ากับผลรวมสำหรับทุก ๆ จำนวนเต็ม n ที่ $n \geq 1$.

การอ้างความจริงสองข้อเพื่อสรุปในย่อหน้าก่อนนั้นค่อนข้างจะเป็นการสรุปที่ไม่เป็นรูปแบบทางการนัก ก่อนที่จะเขียนอย่างเป็นทางการ เราจะแนะนำหลักการอุปนัยเชิงคณิตศาสตร์

หลักการอุปนัยทางคณิตศาสตร์
สำหรับเพรดิเคต $P(n)$ ที่ขึ้นกับจำนวนนับ n ถ้าเราสามารถพิสูจน์ได้ว่า
1. $P(1)$ จริง และ
2. สำหรับจำนวนนับ $k \geq 1$ ใด ๆ $P(k) \Rightarrow P(k+1)$
แล้ว $P(n)$ เป็นจริงสำหรับทุก ๆ จำนวนนับ n

เราเรียกขั้นที่ 1 ว่า*ขั้นฐาน* (basis) และเรียกส่วนที่สองว่า*ขั้นอุปนัย* (inductive step) ในการพิสูจน์ขั้นอุปนัย เราเริ่มโดยการสมมติว่า $P(i)$ เป็นจริงสำหรับจำนวนเต็ม i ใด ๆ ข้อสมมติดังกล่าวเรียกว่า*สมมติฐานการอุปนัย* (induction hypothesis) เนื่องจากในประโยค $P(n)$ เราใช้ n เป็นตัวแปรของการอุปนัย เราจะกล่าวว่า การอุปนัยเชิงคณิตศาสตร์ดังกล่าว เป็น*การอุปนัยบนตัวแปร* n

สังเกตว่าในตัวอย่างที่ยังไม่สมบูรณ์ของเรานั้น เราได้พิสูจน์ขั้นที่ 1 และขั้นที่ 2 ไปแล้ว โดยเพรดิเคตที่เราพิสูจน์ $P(n)$ คือ

$$\text{``}1 + 2 + \cdots + n = \frac{n(n+1)}{2}\text{''}$$

ดังนั้นจากหลักการอุปนัยทางคณิตศาสตร์ เราสามารถสรุปได้ว่า $P(n)$ จริงสำหรับทุก ๆ จำนวนนับ n

ในส่วนต่อไปของบทนี้เราจะได้ดูตัวอย่างการพิสูจน์ด้วยการอุปนัยทางคณิตศาสตร์อีกหลาย ๆ ตัวอย่าง

TODO: เพิ่มตัวอย่าง

4.5.1 ความถูกต้องของโปรแกรม

ในการแสดงว่าอัลกอริทึมทำงานถูกต้องนั้น เราจำเป็นต้องระบุสิ่งที่เราต้องการจากอัลกอริทึมให้ชัดเจน วิธีการหนึ่งที่นิยมใช้ก็คือการระบุ เงื่อนไขก่อนหน้า (precondition) และเงื่อนไขตามหลัง (postcondition) กล่าวคือ

- **เงื่อนไขก่อนหน้า (precondition)** จะระบุเงื่อนไขของข้อมูลป้อนเข้าของอัลกอริทึม ยกตัวอย่างเช่น ในอัลกอริทึมสำหรับการค้นหาแบบทวิภาคในส่วนที่ 4.4 เงื่อนไขก่อนหน้าคือข้อมูลในอาร์เรย์จะต้องเรียงลำดับจากน้อยไปหามาก ส่วนในกรณีของอัลกอริทึมการเรียงลำดับที่เราพิจารณาในส่วนที่ 4.3 นั้นไม่ได้มีการกำหนดเงื่อนไขเริ่มต้นพิเศษแต่อย่างใด แค่เงื่อนไขความถูกต้องของข้อมูล เช่น อาร์เรย์มีข้อมูล n ตัว เป็นต้น

- **เงื่อนไขตามหลัง (postcondition)** จะเป็นเงื่อนไขที่เราต้องการได้รับจากอัลกอริทึม ยกตัวอย่างเช่นในกรณีของอัลกอริทึมค้นหาแบบทวิภาคเราต้องการเงื่อนไขว่า ถ้าข้อมูลที่ต้องการค้นอยู่ในอาร์เรย์ อัลกอริทึมจะคืนดัชนีของข้อมูลนั้น และถ้าข้อมูลไม่อยู่ในอาร์เรย์อัลกอริทึมจะต้องคืนค่า -1 ส่วนกรณีของอัลกอริทึมจัดเรียงข้อมูล เราต้องการเงื่อนไขว่าอาร์เรย์ผลลัพธ์มีข้อมูลของอาร์เรย์เดิมทั้งหมด และข้อมูลในอาร์เรย์นั้นเรียงตามลำดับจากน้อยไปมาก

การแสดงความอัลกอริทึมทำงานถูกต้องก็คือการพิสูจน์ว่า ถ้าข้อมูลป้อนเข้าสอดคล้องกับเงื่อนไขก่อนหน้า หลังจากที่อัลกอริทึมทำงานเสร็จแล้ว ผลลัพธ์และข้อมูลต่าง ๆ สอดคล้องกับเงื่อนไขตามหลัง

เนื่องจากเงื่อนไขทั้งสองถูกระบุขึ้นเพื่อใช้ในการแสดงว่าอัลกอริทึมทำงานถูกต้องหรือไม่ ดังนั้นเงื่อนไขทั้งสองนี้จะถูกระบุขึ้นโดยพิจารณาจากผลลัพธ์ที่เราต้องการจากอัลกอริทึมเป็นหลัก ไม่ใช่จากการทำงานของอัลกอริทึมเอง

หนังสือเล่มนี้จะไม่ลงรายละเอียดในการพิสูจน์ความถูกต้องอย่างเต็มรูปแบบ แต่จะใช้แนวคิดหลัก ๆ นี้ในการแสดงว่าอัลกอริทึมทำงานถูกต้อง นอกจากนี้ ในการพัฒนาโปรแกรมในชีวิตจริง แม้ว่าเราจะไม่ได้พิสูจน์ความถูกต้องของโปรแกรมตลอดเวลา แต่การใส่ใจกับเงื่อนไขก่อนหน้าและเงื่อนไขตามหลังอย่างสม่ำเสมอ จะช่วยลดความผิดพลาดในโปรแกรม หรือช่วยทำให้เราพบข้อผิดพลาดได้รวดเร็วขึ้น

ในส่วนนี้ ต่อไปเราจะใช้การอุปนัยทางคณิตศาสตร์ในการพิสูจน์ความถูกต้องของอัลกอริทึม โดยการแสดงว่าเงื่อนไขตามหลังเป็นจริง เมื่ออัลกอริทึมทำงานเสร็จสิ้น

อัลกอริทึมเรียงข้อมูลแบบการเลือก

แม้ว่าอัลกอริทึม A4.6 จะทำงานบนอาร์เรย์ป้อนเข้าโดยตรง เพื่อความสะดวกในการพิสูจน์ เราจะแยกอาร์เรย์ป้อนเข้าและอาร์เรยผลลัพธ์ออกจากกัน โดยเราจะเรียกอาร์เรย์ป้อนเข้าว่าอาร์เรย์ A และ อาร์เรย์ผลลัพธ์ว่าอาร์เรย์ B

$\triangleright$ **คำถาม 4.15** เงื่อนไขความถูกต้อง
ก่อนจะอ่านต่อไป ให้ลองระบุเงื่อนไขตามหลังที่เราต้องการจากอัลกอริทึมการจัดเรียงข้อมูล $\triangleleft$

เงื่อนไขความถูกต้องของการจัดเรียงมีดังนี้

- เงื่อนไขก่อนหน้า: ไม่มี
- เงื่อนไขตามหลัง: อาร์เรย์ B เป็นการเรียงสับเปลี่ยน (permutation) ของอาร์เรย์ A และสำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < n-1$ เราจะได้ว่า $B[i] \leq B[i+1]$

สังเกตว่าเงื่อนไขตามหลังประกอบด้วยเงื่อนไขย่อยสองเงื่อนไข คือ ($P1$) อาร์เรย์ B เป็นการเรียงสับเปลี่ยน (permutation) ของอาร์เรย์ A เพื่อรับประกันว่าข้อมูลป้อนเข้าจะอยู่ครบ และ ($P2$) สำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < n-1$ เราจะได้ว่า $B[i] \leq B[i+1]$ เพื่อรับประกันการเรียงลำดับของข้อมูล

ดังนั้นในการพิสูจน์ เราไม่จำเป็นที่จะต้องพิสูจน์เงื่อนไขทั้งสองนี้ไปพร้อมกันก็ได้ จากโปรแกรม สังเกตว่าสิ่งเดียวที่โปรแกรมเปลี่ยนแปลงอาร์เรย์ A เพื่อให้ได้อาร์เรย์ B คือการสลับค่าข้อมูลในอาร์เรย์ด้วยฟังก์ชัน `swap` เพราะฉะนั้นเราสามารถสรุปได้ว่าอาร์เรย์ B เป็นการเรียงสับเปลี่ยนของอาร์เรย์ A นั่นคือเงื่อนไข ($P1$) เป็นจริง

เมื่อเราทราบว่าเงื่อนไข ($P1$) จริงแล้ว เราจะแสดงเงื่อนไข ($P2$) ด้วยการอุปนัยเชิงคณิตศาสตร์ (ในการพิสูจน์ เราจำเป็นจะต้องใช้ความจริงว่า ($P1$) เป็นจริงด้วย)

$\triangleright$ **คำถาม 4.16** โครงสร้างของการอุปนัยเชิงคณิตศาสตร์กับการเรียกตัวเอง
ลองพิจารณาโครงสร้างของการพิสูจน์โดยใช้การอุปนัยเชิงคณิตศาสตร์ เทียบกับอัลกอริทึมแบบเรียกตัวเอง ว่ามีโครงสร้างในรูปแบบเดียวกันอย่างไรบ้าง? $\triangleleft$

การพิสูจน์โดยใช้การอุปนัยเชิงคณิตศาสตร์ใช้จำนวนนับ n ในการแบ่งประโยคที่ต้องการพิสูจน์เป็นส่วนย่อย ๆ เราจะใช้ n แทนจำนวนข้อมูลในอาร์เรย์ที่เราต้องการจะเรียง เราจะให้ประโยค $P(n)$ คือ

> $P(n)$: อัลกอริทึม A4.6 เมื่อรับอาร์เรย์ขนาด n ให้ผลลัพธ์เป็นอาร์เรย์ B ที่สำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < n-1$ เราจะได้ว่า $B[i] \leq B[i+1]$

ขั้นฐาน: เราจะเริ่มพิสูจน์ในกรณีฐาน นั่นคือ เมื่อ $n = 1$: สังเกตว่าในกรณีดังกล่าว ดัชนี i ที่สอดคล้องกับเงื่อนไข $0 \leq i < n - 1 = 0$ นั้นไม่มี ดังนั้นประโยค $P(1)$ จึงเป็นจริง[1]

ขั้นอุปนัย: เราจะสมมติให้ $P(n)$ เป็นจริงสำหรับกรณีที่ $n = k$ สำหรับจำนวนนับ k ใด ๆ เราจะพิสูจน์ว่า $P(n)$ เป็นจริงสำหรับกรณีที่ $n = k + 1$

สำหรับขั้นตอนนี้ ถ้าเราเขียนอย่างรวบรัดเรามักเขียนว่า สำหรับ k ใด ๆ เราสมมติให้ $P(k)$ เป็นจริง และจะพิสูจน์ว่า $P(k + 1)$ เป็นจริง

พิจารณาการทำงานของอัลกอริทึมเมื่ออาร์เรย์มีขนาด $k + 1$

สังเกตว่าอัลกอริทึมทำงานโดยการย้ายข้อมูลที่มีค่ามากที่สุดไปไว้ในอาร์เรย์ตำแหน่งที่ k นั่นคือเราทราบว่า $B[i] \leq B[k]$ สำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < k$

จากนั้นอัลกอริทึมเรียกตัวเองเพื่อทำงานกับตอนต้นของอาร์เรย์ ที่มีขนาด k จากข้อสมมติว่า $P(k)$ เป็นจริง เราจะได้ว่าเมื่ออัลกอริทึมทำงานเสร็จ เราจะได้ว่าอาร์เรย์ B มีคุณสมบัติว่า สำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < k - 1$ เราจะได้ว่า $B[i] \leq B[i + 1]$ (สังเกตขอบเขตของดัชนี i ว่ามีค่าถึงแค่ $k - 2$ เท่านั้น)

เนื่องจากการทำงานในขั้นของการเรียกตัวเอง ไม่มีการเปลี่ยนแปลงค่าของ $B[k]$ นอกจากนี้ เนื่องจาก $(P1)$ เป็นจริง อาร์เรย์ผลลัพธ์จากการเรียกตัวเองในส่วน $B[0] \ldots B[k-1]$ จะเป็นการเรียงสับเปลี่ยนของอาร์เรย์เดิม นั่นคือ หลังการเรียกตัวเอง $B[k - 1]$ จะมีค่าเท่ากับบาง $B[i]$ ก่อนการเรียกตัวเอง

อย่างไรก็ตาม เนื่องจากก่อนการเรียกตัวเอง เราทราบว่า $B[k]$ มีค่ามากกว่าหรือเท่ากับทุก ๆ $B[i]$ ดังนั้นเราสามารถสรุปได้ว่า หลังการเรียกตัวเอง $B[k] \geq B[k - 1]$

ดังนั้น เราจึงได้ว่า สำหรับทุก ๆ ดัชนี i ที่ $0 \leq i < k$ เราจะได้ว่า $B[i] \leq B[i+1]$, ซึ่งหมายความว่า $P(k+1)$ เป็นจริงนั่นเอง

เนื่องจากเราได้พิสูจน์ทั้งขั้นฐานและขั้นอุปนัยแล้ว โดยการอุปนัยเชิงคณิตศาสตร์ เราสามารถสรุปได้ว่า $P(n)$ เป็นจริง สำหรับทุก ๆ จำนวนนับ n นั่นคือ อัลกอริทึมจัดเรียง A4.6 เมื่อทำงานเสร็จแล้วผ่านเงื่อนไขตามหลัง $(P2)$

เนื่องจากเราสามารถแสดงได้ว่าอัลกอริทึมผ่านทั้งเงื่อนไข $(P1)$ และ $(P2)$ เราจึงสรุปว่าอัลกอริทึมทำงานได้ถูกต้อง

ในขั้นอุปนัยนี้ เมื่อเราชำนาญขึ้น เรามักเขียนรวบรัดขึ้นอีกโดยไม่แยกระหว่าง n กับ k กล่าวคือเราจะเขียนว่า สำหรับ n ใด ๆ เราสมมติให้ $P(n)$ เป็นจริง และจะพิสูจน์ว่า $P(n+1)$ เป็นจริง ทั้งนี้สังเกตว่า n ในประโยคข้างต้นกับ n ภายในนิยามของประโยค $P(n)$ เป็นตัวแปรคนละตัวกัน โดยเราสามารถพิจารณาได้ว่า n ภายในนิยามของประโยค $P(n)$ เป็นตัวแปรท้องถิ่น (local variable) ของประโยค P นั่นเอง

อัลกอริทึมการค้นหาแบบทวิภาค

เช่นเดียวกับการวิเคราะห์อัลกอริทึมการจัดเรียง เราจะวิเคราะห์ความถูกต้องของการค้นหาแบบทวิภาค (อัลกอริทึม A4.7 และ A4.8) โดยเริ่มจากการระบุเงื่อนไขก่อนหน้าและเงื่อนไขตามหลัง

▷ คำถาม 4.17

ระบุเงื่อนไขก่อนหน้าและเงื่อนไขตามหลัง (อย่างเป็นทางการมากขึ้น) ที่เราต้องการเพื่อพิสูจน์ความถูกต้องของอัลกอริทึม

◁

[1] ในกรณีนี้ เรานิยมกล่าวว่า $P(1)$ เป็นจริงเนื่องจากสิ่งที่ต้องพิจารณานั้นว่างเปล่า (vacuously true) เพราะว่าเซตของสิ่งที่เราต้องพิสูจน์เงื่อนไขนั้น ไม่มีสมาชิกอยู่เลย

เราต้องการค้นหาข้อมูล q จากอาร์เรย์ A ที่มีข้อมูล n ตัวเรียงลำดับจากน้อยไปหามาก เงื่อนไขความถูกต้องของอัลกอริทึมการค้นหา Find (A4.8) มีดังนี้

- เงื่อนไขก่อนหน้า: อาร์เรย์ A ต้องเรียงลำดับจากน้อยไปหามาก
- เงื่อนไขตามหลัง: ถ้ามีข้อมูล q ในอาร์เรย์ A จะต้องคืนผลลัพธ์เป็นดัชนีของข้อมูลนั้น ถ้าไม่มีจะต้องคืนค่า -1

อย่างไรก็ตาม ในกรณีนี้เรามีอัลกอริทึมสองอัลกอริทึมที่ต้องพิจารณา คือ อัลกอริทึม Find (A4.8) และอัลกอริทึม BinarySearch (A4.7) หน้าที่หลักของอัลกอริทึม Find คือการกำหนดค่าเริ่มต้นให้กับพารามิเตอร์ s และ t ส่วนการทำงานหลักจะอยู่ที่ฟังก์ชัน BinarySearch

ดังนั้นการที่เราจะแสดงว่าอัลกอริทึม Find ทำงานได้ถูกต้องได้นั้น เราจำเป็นจะต้องแสดงว่าอัลกอริทึม BinarySearch ทำงานถูกต้องด้วย เราจะวิเคราะห์หาเงื่อนไขก่อนหน้าที่เราจะรับประกันให้อัลกอริทึม และเงื่อนไขตามหลังที่เราต้องการจากอัลกอริทึมดังกล่าว

สังเกตว่า ถ้าพารามิเตอร์ s และ t มีค่าไม่ถูกต้อง อัลกอริทึม BinarySearch ก็จะไม่สามารถคืนดัชนีของข้อมูลที่เราต้องการออกมาได้ ข้อสังเกตนี้ทำให้เราคาดเดาได้ว่าเงื่อนไขก่อนหน้าจะต้องเกี่ยวข้องกับ s และ t ด้วย

$\triangleright$ **คำถาม 4.18**

ทดลองระบุเงื่อนไขก่อนหน้า และเงื่อนไขตามหลังที่เราต้องการ จากอัลกอริทึม BinarySearch (A4.7) $\triangleleft$

เราสามารถระบุเงื่อนไขสำหรับความถูกต้องของอัลกอริทึม BinarySearch ได้ดังนี้

- เงื่อนไขก่อนหน้า: อาร์เรย์ A ต้องเรียงลำดับจากน้อยไปหามาก และถ้า q อยู่ในอาร์เรย์ A จะมีดัชนี i ที่ $s \leq i \leq t$ และ $A[i] = q$
- เงื่อนไขตามหลัง: ถ้ามีข้อมูล q ในอาร์เรย์ A อัลกอริทึมจะต้องคืนผลลัพธ์เป็นดัชนีของข้อมูลนั้น ถ้าไม่มีจะต้องคืนค่า -1

เราจะพิสูจน์ว่าถ้าเงื่อนไขก่อนหน้าเป็นจริง แล้วภายหลังการทำงานของ BinarySearch ผลลัพธ์ที่ได้จะเป็นไปตามเงื่อนไขตามหลัง เนื่องจากอัลกอริทึมดังกล่าวเป็นอัลกอริทึมแบบเรียกตัวเอง เทคนิคแรกที่เราเลือกใช้พิสูจน์ก็คือการอุปนัยเชิงคณิตศาสตร์

อย่างไรก็ตาม สำหรับอัลกอริทึมนี้เราจะใช้การอุปนัยเชิงคณิตศาสตร์แบบเข้ม (strong induction) ซึ่งมีรูปแบบแตกต่างจากการอุปนัยเชิงคณิตศาสตร์ที่เราได้กล่าวถึงในตอนต้นเล็กน้อย

หลักการอุปนัยทางคณิตศาสตร์แบบเข้ม

สำหรับเพรดิเคต $P(n)$ ที่ขึ้นกับจำนวนนับ n ถ้าเราสามารถพิสูจน์ได้ว่า

1. $P(1)$ จริง และ
2. สำหรับจำนวนนับ $k \geq 1$ ใด ๆ $P(1) \wedge P(2) \cdots \wedge P(k) \Rightarrow P(k+1)$

แล้ว $P(n)$ เป็นจริงสำหรับทุก ๆ จำนวนนับ n

สังเกตว่าในขั้นอุปนัย เรามีเงื่อนไขที่มากขึ้น แทนที่จะมีแค่ $P(k)$ แต่เราสามารถอ้างได้ตั้งแต่ $P(1), P(2)$ จนถึง $P(k)$ อย่างไรก็ตาม การใช้การอุปนัยแบบเข้มมีประโยชน์แค่ความสะดวกในการพิสูจน์เท่านั้น เนื่องจากการ

อุปนัยแบบที่เราแนะนำในตอนต้น (หรือที่นิยมเรียกว่าการอุปนัยแบบอ่อน หรือ week induction) กับการอุปนัยแบบเข้มนั้นสมมูลกัน

ก่อนจะใช้การอุปนัยเชิงคณิตศาสตร์ได้ เราจะต้องระบุประโยคที่เราจะพิสูจน์รวมถึงค่าพารามิเตอร์ของประโยคดังกล่าวให้ได้เสียก่อน หลักการสำคัญในการวิเคราะห์อัลกอริทึมเรียกตัวเอง ก็ไม่ต่างจากการวิเคราะห์อัลกอริทึมทั่วไป นั่นคือ เราจำเป็นจะต้องนิยาม "ความก้าวหน้า" ของอัลกอริทึมให้ชัดเจน ข้อสังเกตที่น่าสนใจก็คือ นิยามของความก้าวหน้าที่เราใช้จะต้องสอดคล้องกับบางขั้นตอนที่เราใช้ในการพิสูจน์โดยการอุปนัยเชิงคณิตศาสตร์

$\triangleright$ **คำถาม 4.19**
ลองพิจารณาการพิสูจน์ความถูกต้องของอัลกอริทึมเรียงข้อมูล นิยามของความก้าวหน้าของอัลกอริทึมคืออะไร? จากนั้นให้พิจารณาบทพิสูจน์และอธิบายว่าที่จุดใดที่เราอ้างถึงและใช้ความก้าวหน้าดังกล่าวในการพิสูจน์ $\triangleleft$

$\triangleright$ **คำถาม 4.20** ความก้าวหน้าของการค้นหาแบบทวิภาค
นิยามของความก้าวหน้าของอัลกอริทึมการค้นหาแบบทวิภาคคืออะไร? (เราเคยพิจารณาเรื่องนี้ไปในส่วน 2.2.1) $\triangleleft$

เนื่องจากเราใช้จำนวนข้อมูลที่เหลือในการแสดงความก้าวหน้าของอัลกอริทึม เราจะทำการอุปนัยบนจำนวนนั้น กล่าวคือเราจะพิสูจน์ว่าประโยค

> $Q(k)$: "อัลกอริทึม BinarySearch เมื่อเรียกใช้งานโดยที่มีจำนวนข้อมูลที่เป็นไปได้ $t - s + 1 = k$ จำนวนและเงื่อนไขก่อนหน้าเป็นจริง จะให้ผลลัพธ์ที่สอดคล้องกับเงื่อนไขตามหลัง"

เป็นจริงสำหรับทุก ๆ จำนวนเต็ม k ที่ $k \geq 0$

สังเกตว่าเราจะเริ่มการอุปนัยที่ $k = 0$ ซึ่งไม่ตรงกับนิยามของหลักการอุปนัยเชิงคณิตศาสตร์เสียทีเดียว อย่างไรก็ตามเราจะดำเนินการพิสูจน์ในลักษณะนี้ไปก่อนและพิจารณาประเด็นนี้ในภายหลัง

ขั้นฐาน: เมื่อ $k = 0$ และเงื่อนไขก่อนหน้าเป็นจริง เราสามารถสรุปได้ว่าข้อมูล q ไม่อยู่ในอาร์เรย์ (ดูคำถาม 4.21) นอกจากนี้เราทราบว่า $t - s + 1 = 0$ หรือ $s = t + 1$ ทำให้เข้าเงื่อนไขของคำสั่ง IF ในบรรทัดที่ 1 และอัลกอริทึม BinarySearch จะคืนค่า -1 ซึ่งสอดคล้องกับเงื่อนไขตามหลัง

$\triangleright$ **คำถาม 4.21**
พิสูจน์ว่า ถ้า $k = 0$ และเงื่อนไขก่อนหน้าเป็นจริง ข้อมูล q จะไม่อยู่ในอาร์เรย์ $\triangleleft$

ขั้นอุปนัย: พิจารณากรณีที่ $k > 0$ เราจะสมมติว่าสำหรับทุก ๆ $k' < k$ ประโยค $Q(k')$ เป็นจริง นั่นคือ ถ้ามีการเรียกอัลกอริทึม BinarySearch ที่จำนวนข้อมูลที่เป็นไปได้คือ k' และเงื่อนไขก่อนหน้าเป็นจริง ผลลัพธ์ที่ได้จะสอดคล้องกับเงื่อนไขตามหลัง

เรามีกรณีที่ต้องพิจารณา 3 กรณี แยกตามการทำงานของอัลกอริทึม BinarySearch

กรณีที่ 1: เมื่อ $A[m] = q$. ในกรณีนี้อัลกอริทึมจะตอบ m ซึ่งทำให้ผลลัพธ์สอดคล้องกับเงื่อนไขตามหลัง

กรณีที่ 2: เมื่อ $A[m] < q$. เนื่องจากข้อมูลในอาร์เรย์เรียงลำดับกัน (จากเงื่อนไขก่อนหน้า) เราจะทราบว่าสำหรับทุก ๆ ดัชนี i ที่ $0 \leq i \leq m$, $A[i] < q$ เมื่อเรานำความจริงนี้มาประกอบกับเงื่อนไขก่อนหน้า นั่นคือ ถ้ามีข้อมูล q ใน A จะมีดัชนี i ที่ $s \leq i \leq t$ ที่ $A[i] = q$ เราจะสามารถสรุปได้ว่า ถ้ามีข้อมูล q ใน A จะต้องมีดัชนี i ที่ $m + 1 \leq i \leq t$ ที่ $A[i] = q$

จากนั้นอัลกอริทึมจะเรียกตัวเองเพื่อค้นหาแบบทวิภาคในอาร์เรย์โดยส่งค่าดัชนีเริ่มต้นเป็น $m+1$ และดัชนีสุดท้ายเป็น t สังเกตว่าในกรณีนี้ ข้อสรุปข้างต้นแสดงว่าเงื่อนไขเริ่มต้นของอัลกอริทึม BinarySearch ที่เรียกตัวเองเป็น

จริง นอกจากนี้จำนวนที่เป็นไปได้ในการเรียกตัวเองคือ $k' = t - \lfloor \frac{s+t}{2} \rfloor - 1 < t - s + 1 < k$ ดังนั้น โดยสมมติฐานการอุปนัยเราจะได้ว่า ผลลัพธ์ที่คืนจากการเรียกตัวเองถูกต้อง นั่นคือ ถ้า q อยู่ในอาร์เรย์ A โดยมีดัชนีตั้งแต่ $m+1$ จนถึง t อัลกอริทึมจะคืนค่าดัชนี i ที่ $A[i] = q$ และถ้า q ไม่อยู่ในอาร์เรย์ในดัชนีตั้งแต่ $m+1$ ถึง t อัลกอริทึมจะคืนค่า -1

ดังนั้น จากเงื่อนไขตามหลังที่ได้หลังการเรียกตัวเองและเงื่อนไขก่อนหน้าที่เรารับประกัน เราสามารถสรุปได้ว่าผลลัพธ์ของ BinarySearch เมื่อจำนวนข้อมูลที่เป็นไปได้คือ k สอดคล้องกับเงื่อนไขตามหลังในกรณีที่ 2 นี้

กรณีที่ 3: เมื่อ $A[m] > q$. สำหรับกรณีนี้ เราจะละไว้เป็นคำถามที่ 4.22

เมื่อเราได้พิจารณาทุกกรณีแล้ว เราสามารถสรุปได้ว่า ถ้าเราสมมติให้ $Q(0), Q(1), \ldots, Q(k-1)$ เป็นจริง แล้ว $Q(k)$ จะเป็นจริงด้วย นั่นคือเราแสดงขั้นอุปนัยได้แล้ว

ดังนั้น จากหลักการอุปนัยเชิงคณิตศาสตร์แบบเข้ม เราสามารถสรุปได้ว่า $Q(k)$ เป็นจริงสำหรับทุก ๆ ค่า k ที่ $k \geq 0$

$\triangleright$ **คำถาม 4.22** กรณีที่ 3
พิสูจน์ขั้นอุปนัยในกรณีที่ 3 $\triangleleft$

หมายเหตุ. สังเกตว่าเราปรับรูปแบบการพิสูจน์แบบอุปนัยเชิงคณิตศาสตร์เล็กน้อย นั่นคือ เราไม่ได้เริ่มค่าพารามิเตอร์ที่ 1 และแทนที่เราจะสมมติ $Q(0) \wedge Q(1) \wedge \cdots \wedge Q(k)$ แล้วพิสูจน์ $Q(k+1)$ เรากลับสมมติ $Q(0) \wedge Q(1) \wedge \cdots \wedge Q(k-1)$ และพยายามพิสูจน์ $Q(k)$ ทั้งหมดนี้เป็นแค่การปรับรูปแบบ แต่ไม่ได้ทำข้อสรุปของการอุปนัยเชิงคณิตศาสตร์เปลี่ยนไปแต่อย่างใด เพราะเราสามารถนิยามประโยค $Q'(i)$ ให้เป็นประโยค $Q(i-1)$ และพิสูจน์ประโยค $Q'(i)$ สำหรับ $i = 1, 2, \ldots$ ได้ด้วยการอุปนัยเชิงคณิตศาสตร์ในรูปแบบมาตรฐาน จากนั้นข้อสรุปว่า $Q'(i)$ เป็นจริงสำหรับ $i = 1, 2, \ldots$ ก็จะสมมูลกับข้อสรุปว่า $Q(k)$ เป็นจริงเมื่อ $k = 0, 1, 2, \ldots$ นั่นเอง