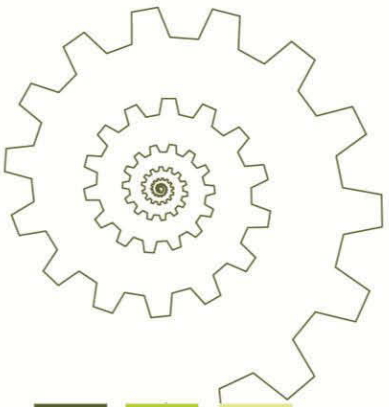
อัลกอริทึม

การออกแบบและวิเคราะห์

สมชาย ประสิทธิ์จูตระกูล

โครงการสนับสนุนการพิมพ์และเผยแพร่ตำราหนังสือวิชาการ
ของคณาจารย์ ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย



คำนำ

ในปัจจุบันใคร ๆ ก็คงเห็นบทบาทที่สำคัญของคอมพิวเตอร์ที่ไปแทรกซึมในแทบทุกกิจกรรม ประจำวันไม่ทางตรงก็ทางอ้อม คอมพิวเตอร์ทำงานได้เร็วขึ้น มีขนาดเล็กลง มีโปรแกรมที่ใช้ง่ายขึ้น ช่วยเสริมกิจกรรมของมนุษย์ได้มากขึ้น ทั้งนี้ก็เพราะว่าเบื้องหลังฮาร์ดแวร์ที่เราจับต้องได้ เบื้องหลังซอฟต์แวร์ที่เรานำมาติดตั้งสั่งงานฮาร์ดแวร์นั้น ก็คือ “กึ๋น” ความคิดของผู้สร้างหรือที่เรียกว่าอัลกอริทึมที่ถูกแปลงมาเป็นฮาร์ดแวร์และโปรแกรมที่ใช้งานได้จริง อัลกอริทึมจึงเป็นศาสตร์ซึ่งมีบทบาทสำคัญมากในการศึกษาทางวิทยาการและวิศวกรรมคอมพิวเตอร์ วิชาอัลกอริทึมเบื้องต้น หรือโดยส่วนใหญ่มักตั้งชื่อว่า การออกแบบและวิเคราะห์อัลกอริทึม จึงถูกบรรจุเป็นวิชาพื้นฐานในสาขาวิชาดังกล่าว

คำว่า “Algorithm” นั้นศัพท์บัญญัติของราชบัณฑิตยสถานใช้คำว่า “ขั้นตอนวิธี” ซึ่งถ้าจะขยายความอย่างไม่เป็นทางการ ก็คือขั้นตอนวิธีการแก้ไขปัญหาเชิงคำนวณด้วยคอมพิวเตอร์ ถึงแม้ว่า “Algorithm” มีความหมายดังกล่าว แต่เป็นคำที่มาจากคำว่า “Al-Khwarizmi” ซึ่งเป็นชื่อนักคณิตศาสตร์ (อ่านรายละเอียดได้ในปลายบทที่ 2) ดังนั้นเพื่อเป็นการให้เกียรตินักคณิตศาสตร์ท่านนี้ ผมจึงขอทับศัพท์ใช้คำว่า “อัลกอริทึม” แทน จึงขอใช้เนื้อที่ตรงนี้ชี้แจงเหตุผลของการเลือกไม่ใช้ศัพท์บัญญัติคำว่า “Algorithm” นี้

วิธีการศึกษาอัลกอริทึมที่ได้ผลวิธีหนึ่ง (ซึ่งผมขอใช้ในที่นี้) ก็คือการศึกษาจากตัวอย่าง ผู้อ่านจะพบตัวอย่างปัญหาแทรกตามกลวิธีการออกแบบมาตรฐานทั่วไป ที่คนในวงการทำๆ กัน นั่นคือก่อนจะออกแบบเก่ง ก็ขอให้ไปศึกษาสิ่งที่ผู้อื่นได้เคยออกแบบกันมาในอดีตเป็นตัวอย่าง วิธีศึกษาแบบนี้ก็มีทั้งข้อดีข้อเสีย ข้อดีก็คือยิ่งรู้ตัวอย่างมาก ก็รู้แนวทางหลากหลายซึ่งพร้อมที่ประยุกต์กับปัญหาใหม่ที่จะพบในอนาคต แต่ก็มีข้อเสียว่าจะยึดติดกับกลวิธีเก่า ๆ ไม่ค่อยยอมคิดอะไรแผลง ๆ ใหม่ ๆ (จึงมักพบบ่อย ๆ ว่าแนวคิดแปลก ๆ ใหม่ ๆ มาจากคนนอกวงการ) จึงขอให้ผู้อ่านระแวงในประเด็นนี้ด้วย

ผู้ออกแบบอัลกอริทึมที่ดีจำต้องรู้จักข้อปัญหาที่สนใจ รู้จักเลือกกลวิธีการออกแบบ รู้จักเลือกใช้โครงสร้างข้อมูลที่เหมาะสม และที่สำคัญต้องรู้จักวิเคราะห์ด้วยว่าผลที่ได้ออกแบบไว้ดีหรือเลวเพียงใด ดังนั้นก่อนจะเข้าสู่เรื่องราวทางอัลกอริทึมนั้น ผู้อ่านจะต้องมีพื้นฐานทางการเขียนโปรแกรม โครงสร้างข้อมูล และภินทนคณิตศาสตร์

ผมขอเน้นว่าพื้นฐานทางการเขียนโปรแกรมนั้นจำเป็นอย่างยิ่งยวด เนื่องจากเราจะต้องรู้ว่าแนวคิดที่นำเสนอในอัลกอริทึมนั้นทำให้เห็นจริงได้ด้วยฮาร์ดแวร์หรือซอฟต์แวร์ ดังนั้นจึงต้องมีความสามารถในการเปลี่ยนแนวคิดมาเป็นโปรแกรมที่ทำงานได้จริง ขอให้นักเรียนเขียนโปรแกรมเป็นสักหนึ่งภาษา (โดยส่วนตัวชอบ C C++ หรือ Java) อัลกอริทึมต่างๆ ที่พบในเอกสารนี้เมื่อเขียนโปรแกรมสมบูรณ์แล้วอย่างมากก็อยู่ในหลักเป็นร้อยบรรทัด ซึ่งถือว่าไม่มาก ผมพบว่าทักษะส่วนนี้เป็นสิ่งที่น่าเป็นห่วงที่สุดสำหรับนักเรียนที่จะเรียนอัลกอริทึมให้ได้ผล

จึงอยากขอเน้นอีกประการหนึ่งสำหรับนักเรียนว่า วิชาทางสาขานี้ไม่ใช่มานั่งฟัง จำ แล้วก็สอบ การศึกษาจะได้ผลดี (ต่อตัวนักเรียนเอง) ก็คือต้องอ่าน คิด ทำ และถาม (ส่วนการเข้าเรียนและการเข้าสอบนั้นเป็นเรื่องของระเบียบและกฎเกณฑ์) เราสามารถเจาะเนื้อหาในรายละเอียดจากการอ่านตำรา ฝึกสมองจากการคิดระหว่างการทำแบบฝึกหัด ฝึกฝีมือจากการทำการบ้านการเขียนโปรแกรม และสร้างความกระจ่างในเนื้อหาจากการถามและถกเถียงกับผู้อื่น ผมอยากให้บรรยากาศการเรียนรู้เป็นแบบ “อ่าน+คิด+ทำ+ถาม” มากกว่าแบบ “ฟัง+บ่น+สอบ+ทิ้ง”

หนังสือเล่มนี้ประกอบไปด้วยเนื้อหาห้าส่วนด้วยกันคือ

1. เรื่องพื้นฐาน (บทที่ 1 และ 2)
2. การวิเคราะห์ (บทที่ 3 4 และ 5)
3. โครงสร้างข้อมูล (บทที่ 6 และ 7)
4. การออกแบบ (บทที่ 8 ถึง 12)
5. การจัดกลุ่มปัญหา (บทที่ 13)

เนื้อหาทั้งหมดสามารถใช้สอนได้ในหนึ่งภาคการศึกษา แต่อาจไม่สามารถครอบคลุมได้ทุกตัวอย่าง ถ้าสนใจเฉพาะองค์ความรู้พื้นฐานทางอัลกอริทึม ผู้สอนอาจข้ามเนื้อหาในบทที่ 5 (เรื่องการวิเคราะห์ถัวเฉลี่ย) บทที่ 6 (ในกรณีที่ไม่อยากทวนเรื่องโครงสร้างข้อมูล) และบทที่ 7 (เรื่องโครงสร้างข้อมูลแบบยุ่ง) ได้ เพื่อจะได้มีโอกาสครอบคลุมได้ครบทุกตัวอย่าง และลงลึกในรายละเอียดของการทำงานของอัลกอริทึมและการวิเคราะห์

ต้องขอบอกว่า ผมเองไม่เคยลงทะเบียนเรียนวิชานี้มาก่อน และก็ทำวิจัยทางด้านการออกแบบวงจรรวมความจุสูงระหว่างการศึกษา แต่พบว่า งานวิจัยทั้งหลายในสาขาที่สนใจ (ซึ่งก็คือสายฮาร์ดแวร์) นั้นมีแต่เรื่องของอัลกอริทึม เมื่อเริ่มอาชีพอาจารย์ก็พบว่ามีวิชานี้ปรากฏอยู่ในหลักสูตร แต่ไม่เคยเปิดสอนมาเป็นเวลาสิบกว่าปี ก็เลยเปิดเป็นวิชาเลือก สอนและเรียนรู้สิ่งใหม่ๆ ไปพร้อม ๆ กับนักเรียนมาเป็นเวลา 6 ปี จากนั้นจึงผลักดันให้เป็นวิชาบังคับในหลักสูตร สอนมาได้อีก 3 ปี อยู่มาวันหนึ่งได้ยืมสมุดจดการเรียนของนักเรียนคนหนึ่ง (คุณปภาณิน) มาดูพบว่าเธอเขียนสรุปคำสอนของผมได้ดีกว่าเศษกระดาษที่ผมใช้เตรียมการสอนเสียอีก ก็เลยรู้สึกว่าน่าจะมีหนังสือทางด้านนี้ให้อ่านกันเป็นภาษาไทยเสียที เพราะผมพบว่านักเรียนไม่ค่อยอ่านเนื้อหาในตำราภาษาอังกฤษกันเลย อีกทั้งตำราก็มีราคาแพง จึงใช้เวลาวันละประมาณ 2 ชั่วโมงหลังลูกเข้านอนแล้ว เป็นเวลาประมาณครึ่งปีเขียนได้เป็นต้นฉบับใช้ประกอบการสอนอีกหนึ่งปี ดำเนินการแก้ไขอีกสองรอบ ก่อนที่จะถูกจัดพิมพ์เป็นหนังสือเล่มนี้ (ผู้สนใจสามารถอ่านเพิ่มเติมได้ที่ http://www.cp.eng.chula.ac.th/~somchai/books)

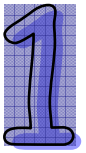
สุดท้ายนี้ต้องขอขอบคุณภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย ที่สนับสนุนวัสดุครุภัณฑ์และโอกาส ให้ผมได้ผลิตต้นฉบับที่ใช้เป็นเอกสารประกอบคำสอนวิชา 2110427 และขอขอบคุณศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ (NECTEC) ที่รับจัดพิมพ์เป็นหนังสือ[1] เพื่อเผยแพร่ได้อย่างราบรื่น

สมชาย ประสิทธิ์จูตระกูล
ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย
somchaip@chula.ac.th
๓๐ พฤษภาคม ๒๕๔๔

[1] สำหรับการพิมพ์ครั้งที่ 1 ถึง 3

สารบัญ

บทนำ



วัตถุประสงค์หลักที่ต้องการให้ผู้อ่านหนังสือเล่มนี้ได้รับคือ ความสามารถในการออกแบบอัลกอริทึมที่มีประสิทธิภาพเพื่อแก้ไขปัญหาที่ได้รับ ความสามารถนี้จะถ่ายทอดกันได้อย่างไร ? วิธีหนึ่งคือ การยกตัวอย่างกลวิธีมาตรฐานต่าง ๆ ในการแก้ไขปัญหา พร้อมทั้งยกตัวอย่างปัญหาและอัลกอริทึมเพื่อแก้ไขปัญหาเหล่านั้น ปัญหาหนึ่งมักมีขั้นตอนวิธีแก้ไขหลายวิธี จึงจำเป็นอย่างยิ่งที่เราต้องมีความรู้ในการวิเคราะห์เพื่อเปรียบเทียบข้อดีและข้อเสียของแต่ละวิธี ดังนั้น สิ่งที่จะศึกษากันประกอบด้วย

- กลวิธีการออกแบบอัลกอริทึม
- การวิเคราะห์อัลกอริทึม
- ความซับซ้อนของปัญหา

อันเป็นเรื่องหลักที่จะได้ศึกษากัน สำหรับในบทนำนี้จะขอยกตัวอย่างเล็ก ๆ ดังต่อไปนี้

การหาค่าน้อยสุดในอาเรย์

ปัญหาที่สนใจในหัวข้อนี้คือ การหาข้อมูลที่มีค่าน้อยสุดในอาเรย์ A อ่านแล้วก็จะรู้สึกได้ทันทีว่า “หมูมาก” เรียนกันมาตั้งแต่วิชาเริ่มเรียนเขียนโปรแกรมแล้ว (ผมมีสมมติฐานว่า ผู้อ่านต้องมีความรู้สึกเช่นนี้จริง ๆ หากใครยังไม่รู้ว่าจะเขียนโปรแกรมแก้ปัญหานี้อย่างไร แสดงว่า หยิบหนังสือผิดเล่มแล้ว !!!) ขอเสนอวิธีหาค่าน้อยสุดให้ดูสามแบบ

เริ่มด้วยวิธีที่หนึ่ง ตรงไปตรงมาคือ การไล่เปรียบเทียบข้อมูลไป n–1 ครั้ง ตั้งแต่ตัวที่สองถึงตัวสุดท้าย โดยจำเฉพาะตัวที่น้อยกว่าจากการเปรียบเทียบ ได้โปรแกรมดังนี้ *

---

* ขอละเลยเรื่องจุกจิกอาทิเช่น ถ้า A ไม่มีข้อมูลเลยจะเกิดอะไรขึ้น เพราะจะทำให้โปรแกรมแลดูรุ่มร่าม

```
01: findMin1( A[1..n] )
02: {
03:   m = A[1]
04:
05:   for ( i = 2 to n )
06:     if ( A[i] < m ) m = A[i]
07:   return m
08: }
```

ผู้อ่านหลายคนอาจมีวิธีของตัวเองในใจที่ไม่ใช่แบบนี้ เช่น บางคนอาจใช้วงวน `while` แทนวงวน `for` บางคนอาจเลือกวิ่งย้อนจากตัวท้ายมาตัวหน้า เป็นต้น

วิธีที่สอง เขียนจากความคิดในการหาค่าน้อยที่สุดที่คิดจากความสัมพันธ์เวียนเกิดข้างล่างนี้

$$m_n = \min(m_{n-1}\,,\, A_n\,) \text{ สำหรับ } n > 1\,,\ m_1 = A_1$$

หมายความว่า ถ้าต้องการหาค่าน้อยสุดในอาเรย์ที่มี n ตัว (m_n) ให้หาค่าน้อยสุดของอาเรย์ n–1 ตัวแรก (m_{n-1}) ก่อน แล้วนำผลมาเปรียบเทียบกับตัวที่ n (A_n) เขียนโปรแกรมแบบเรียกซ้ำ จากนิยามของปัญหาได้ตรงไปตรงมาดังนี้

```
09: findMin2( A[1..n] )
10: {
11:   if ( n == 1 ) return A[1]
12:   m = findMin2( A[1..n-1] )
13:   return ( A[n] < m ? A[n] : m )
14: }
```

บางคนอาจคิดว่า แบบนี้ไม่เห็นมีอะไรเลย ในทางปฏิบัติก็ยังเป็นการเปรียบเทียบแบบลำดับจากตัวแรกไปตัวท้ายอยู่ดี (ไปลองคิดดูเองว่า ทำไม ?)

วิธีที่สาม เสนอให้พิจารณาข้อมูลทีละครึ่งคือ หาตัวน้อยสุดครึ่งซ้าย หาตัวน้อยสุดครึ่งขวา แล้วค่อยมาเปรียบเทียบกัน ได้โปรแกรมแบบเรียกซ้ำข้างล่างนี้

```
15: findMin3( A[1..n] )
16: {
17:   return findMin3R( A[1..n] )
18: }
19:
20: findMin3R( A[a..b] )
21: {
22:   if ( a == b ) return A[a]
23:   mid = (a + b) / 2
24:   m1 = findMin3R( A[a..mid] )
25:   m2 = findMin3R( A[mid+1..b] )
26:   return ( m1 < m2 ? m1 : m2 )
27: }
```

อยากทราบว่า สามวิธีที่ได้นำเสนอมา วิธีใดน่าใช้ที่สุด ?

น่าใช้ในแง่ไหน ? ในแง่ของอัลกอริทึมแล้วเขาพิจารณากันสองประเด็นคือ ประสิทธิภาพเชิงเวลา หรือพูดง่าย ๆ ว่า วิธีใดทำงานเร็วกว่ากัน และเรื่องของปริมาณหน่วยความจำที่ใช้ระหว่างการทำงาน แน่นอนว่า มีประเด็นอื่น ๆ ที่ควรนำมาพิจารณาด้วย เช่น ความซับซ้อนของการเขียนโปรแกรม (นั่นคือ โปรแกรมใด*เขียนง่ายกว่ากัน อ่านเข้าใจ*ง่ายกว่ากัน) เป็นต้น อย่างไรก็ตาม จะขอยึดเรื่องเวลาการทำงานเป็นประเด็นสำคัญในการพิจารณา

จากวิธีการหาค่าน้อยสุดสามวิธีข้างต้น วิธีใดทำงานเร็วสุด ? คำตอบง่าย ๆ ก็คือ ลองเขียนเป็นโปรแกรม สั่งแปล ลองทดสอบกับข้อมูลจริง แล้วก็จับเวลาเพื่อเปรียบเทียบ ทำแบบนี้ออกจะดูทื่อไปหน่อย เหมือนกับบอกว่า จะส่งดาวเทียมแบบไหนดี ก็บอกว่า ลองส่งไปสักสามดวงสามแบบ แล้วก็จะรู้เองว่าแบบไหนดี เราต้องมีความสามารถที่จะประเมินประสิทธิภาพของอัลกอริทึมที่ได้ออกแบบไว้โดยไม่ต้องเขียน+แปล+สั่งงานโปรแกรมจริง อันนี้เป็นเรื่องของการวิเคราะห์

จากวิธีที่เขียนให้ดูข้างต้นนี้ เพื่อความง่ายในการอ้างอิง ขอกำหนดให้ เวลาการทำงานของคำสั่งในบรรทัดที่ k (ดูหมายเลขกำกับบรรทัด) คือ t_k วิธีแรกเป็นการทำงานแบบลำดับคือไล่เปรียบเทียบตัวที่สองไปถึงตัวท้าย ดังนั้น ถ้ามีข้อมูล n ตัว ก็ต้องทำงานที่คำสั่ง `for` n ครั้ง แต่ทำงานในวงวน `for` n–1 ครั้ง (รอบสุดท้าย เงื่อนไขใน `for` จะไม่เป็นจริง ทำให้หลุดจากวงวน) บวกกับเวลานอกวงวนอีกเล็กน้อย รวมเป็นเวลา

$$T_1(n) = t_3 + t_5 n + t_6(n–1) + t_7$$

สำหรับเวลาการทำงานของวิธีที่สอง $T_2(n)$ เท่ากับเวลาในการหาตัวน้อยสุดของข้อมูล n–1 ตัวแรก ตามด้วยภาระอีกเล็กน้อย แต่ถ้ามีข้อมูลตัวเดียวก็คืนค่าได้เลย (ใช้เวลา t_{11}) เขียนเป็นความสัมพันธ์เวียนเกิดของเวลาการทำงาน พร้อมทั้งหาผลเฉลยได้ดังนี้

$$\begin{aligned}
T_2(n) &= T_2(n–1) + t_{11} + t_{12} + t_{13} \quad \text{สำหรับ } n > 1,\ T_2(1) = t_{11} \\
&= T_2(n–2) + 2(t_{11} + t_{12} + t_{13}) \\
&= T_2(n–3) + 3(t_{11} + t_{12} + t_{13}) \\
&\cdots \\
&= T_2(n–(n–1)) + (n–1)(t_{11} + t_{12} + t_{13}) \\
&= t_{11} + (n–1)(t_{11} + t_{12} + t_{13}) \\
&= t_{11} n + (t_{12} + t_{13})(n–1)
\end{aligned}$$

สำหรับวิธีที่สามนั้น เราสามารถเขียนความสัมพันธ์เวียนเกิดของเวลาการทำงานได้ดังนี้

$$T_3(n) = T_3(\lceil n/2 \rceil) + T_3(\lfloor n/2 \rfloor) + t_{22}+t_{23}+t_{24}+t_{25}+t_{26} \quad \text{สำหรับ } n > 1,\ T_3(1) = t_{22}$$

กำหนดให้ $c = t_{22}+t_{23}+t_{24}+t_{25}+t_{26}$ และให้ $n = 2^k$ สามารถวิเคราะห์หาผลเฉลยได้ง่ายขึ้นดังนี้

$$\begin{aligned}
T_3(n) &= 2T_3(n/2) + c \\
&= 2(\ 2T_3(n/2^2) + c\ ) + c = 2^2\ T_3(n/2^2) + 2c + c \\
&= 2^2(\ 2T_3(n/2^3) + c\ ) + 2c + c\ \ = 2^3\ T_3(n/2^3) + 2^2c + 2c + c \\
&\cdots \\
&= 2^k\ T_3(n/2^k) + 2^{k-1}c + 2^{k-2}c + \ldots + 2^1c + 2^0c \\
&\cdots \\
&= 2^{\lg n}\ T_3(n/2^{\lg n}) + 2^{\lg n-1}\ c + 2^{\lg n-2}c + \ldots + 2^1c + 2^0c \\
&= 2^{\lg n}\ t_{22} + 2^{\lg n-1}\ c + 2^{\lg n-2}c + \ldots + 2^1c + 2^0c \\
&= n\ t_{22} + c(n-1) \\
&= t_{22}\ n + (t_{22}+t_{23}+t_{24}+t_{25}+t_{26})(n-1)
\end{aligned}$$

สรุปเวลาการหาค่าน้อยสุดของทั้งสามวิธีดังนี้

วิธีที่ 1 $T_1(n) = t_5n + t_6(n-1) + t_3 + t_7$

วิธีที่ 2 $T_2(n) = t_{11}n + (t_{12} + t_{13})(n-1)$

วิธีที่ 3 $T_3(n) = t_{22}n + (t_{22}+t_{23}+t_{24}+t_{25}+t_{26})(n-1)$

เวลาทำงานจริงจะเป็นเท่าใด คงขึ้นกับเวลาการทำงานจริงของแต่ละคำสั่ง ขึ้นกับว่าใช้ภาษาอะไร ตัวแปลภาษาตัวไหน ทำงานบนเครื่องอะไร แต่ถ้าพิจารณาละเอียดสักเล็กน้อยจะพบว่าคำสั่งที่เขียนส่วนใหญ่ก็เป็นคำสั่งพื้น ๆ จะมีก็แต่คำสั่งเรียกฟังก์ชันในบรรทัดที่ 12, 24 และ 25 ที่เป็นคำสั่งเรียกซ้ำ มีภาระมาก เพราะการเรียกฟังก์ชันมีภาระในการจัดการ stack frame ส่งค่าพารามิเตอร์ และย้ายการทำงานไปยังฟังก์ชันอื่น ในเชิงประสิทธิภาพการทำงาน จึงสรุปได้ว่า

- ทั้งสามวิธีทำงานใช้เวลาแปรตามค่าของ n เป็นฟังก์ชันเชิงเส้นทั้งสิ้น
- วิธีที่ 1 น่าจะทำงานได้เร็วสุด เนื่องจากคำสั่งที่ทำงานซ้ำ (คำสั่ง `if` และ `for`) ใช้เวลาน้อยกว่าของโปรแกรมอื่น (ซึ่งมีการเรียกฟังก์ชัน)

หลายคนคงรู้สึกตอนนี้ว่า การวิเคราะห์ตัวอย่างอัลกอริทึมเล็ก ๆ ที่ได้แสดงมา ยุ่งและละเอียดเหลือเกิน (ว่าไปแล้ว นี่เป็นโปรแกรมที่เล็กที่สุดที่อธิบายกันในหนังสือเล่มนี้) ในแง่ของศาสตร์ทางอัลกอริทึมที่จะศึกษากัน เราคงไม่ลงไปวิเคราะห์กันในรายละเอียดของแต่ละคำสั่งแบบที่ได้ทำมาหรอก เพราะพอถึงที่สุด เราก็ยังไม่รู้ค่าของ t_k ต่าง ๆ ที่ติดอยู่ในฟังก์ชัน (เพราะมันขึ้นกับปัจจัยหลายอย่างที่กล่าวมาเช่น ภาษา ตัวแปล เครื่อง) สิ่งที่เรา

สนใจมากกว่าคือ แนวโน้มการเพิ่มของเวลาการทำงานเมื่อขนาดของข้อมูลเพิ่มขึ้น จากวิธีการหาค่าน้อยสุดทั้งสาม พบว่า เวลาการทำงานแปรตามจำนวนข้อมูลในลักษณะเชิงเส้นทั้งสิ้น นั่นคือ ถ้าจำนวนข้อมูลเพิ่มขึ้น 100 เท่า เวลาการทำงานก็เพิ่มขึ้น 100 เท่าตาม การเติบโตของฟังก์ชันที่แทนประสิทธิภาพการทำงานนี่เองคือสิ่งที่เราสนใจวิเคราะห์กัน ซึ่งเราจะศึกษาเครื่องมือที่ช่วยวิเคราะห์อัลกอริทึมที่ซับซ้อนกันต่อไป

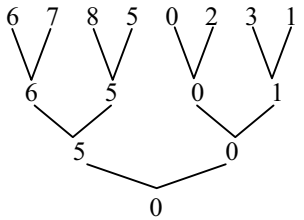
การหาค่าน้อยสุดอันดับที่สอง

ตัวอย่างที่แล้วบอกเราว่า ต้องการหาค่าน้อยสุดในอาเรย์ ให้เขียนโปรแกรมแบบธรรมดาใช้วงวนวิ่งไล่เปรียบเทียบข้อมูลในอาเรย์ ทำงานเร็วกว่าแบบใช้ลูกเล่นเขียนเป็นแบบเรียกซ้ำที่ค่อนข้างยุ่งยาก คราวนี้ขอเพิ่มความต้องการของปัญหาให้มากขึ้นคือ นอกจากต้องการค่าน้อยสุดแล้ว ยังต้องการค่าน้อยสุดอันดับสองด้วย จะทำอย่างไรดี ?

ก็ “หมู” อีก เพียงเขียนอีกวงวนหนึ่งหลังการหาค่าน้อยสุด แล้วหาตัวน้อยสุดในอาเรย์ที่ไม่พิจารณาช่องที่เป็นตัวน้อยสุดที่หาได้ในวงวนแรก เท่านี้ก็เสร็จ (ผมคงไม่ต้องเขียนโปรแกรมให้ดูนะ) ด้วยขั้นตอนวิธีนี้ แน่นอนว่า การหาตัวน้อยสุดอันดับสองจะต้องเปรียบเทียบข้อมูลเพิ่มอีก $n - 2$ ครั้ง ใช้เวลาเป็นฟังก์ชันเชิงเส้นกับจำนวนข้อมูลเช่นเดียวกับการหาตัวน้อยสุด

มีวิธีอื่นหรือไม่ ที่จะได้ตัวน้อยสุดอันดับที่สอง ด้วยจำนวนการเปรียบเทียบข้อมูลที่น้อยกว่า

คำตอบคือมี โดยใช้ผลที่ได้จากการหาตัวน้อยสุดในวิธีที่ 3 ที่นำเสนอในหัวข้อที่แล้ว ถ้าลองแจกแจงขั้นตอนการเปรียบเทียบข้อมูลในวิธีที่ 3 พบว่า เป็นการเปรียบเทียบข้อมูลทีละคู่ ๆ แสดงได้ด้วยต้นไม้แบบทวิภาคดังตัวอย่างในรูปที่ 1–1 ถ้าลองไล่การทำงานของวิธีที่ 3 จะพบว่า ลำดับของข้อมูลที่ถูกนำมาเปรียบเทียบเสมือนการแวะผ่านต้นไม้แบบหลังลำดับ นั่นคือมีการเปรียบเทียบข้อมูลตามลำดับดังนี้ (6,7), (8,5), (6,5), (0,2), (3,1), (0,1) และสุดท้าย (5,0) แล้วได้คำตอบคือ 0



รูปที่ 1–1 ต้นไม้แบบทวิภาคจำลองขั้นตอนการหาค่าน้อยสุดในอาเรย์ด้วยวิธีที่ 3

ให้สังเกตว่า ตัวน้อยสุดอันดับสองต้องเป็นข้อมูลตัวใดตัวหนึ่ง ในบรรดาตัวที่เคยเปรียบเทียบกับตัวน้อยสุด แล้วแพ้มาก่อน จากตัวอย่างพบว่า ตัวที่เคยเปรียบเทียบกับ 0 คือ 5, 1, และ 2 ดังนั้น เราเพียงนำรายการของข้อมูลตัวที่เคยเปรียบเทียบกับตัวน้อยสุด มาหาตัวน้อยสุด ก็จะได้ตัวน้อยสุดอันดับที่สอง

แล้ววิธีนี้ดีตรงไหน ? คำตอบอยู่ตรงที่ จำนวนข้อมูลที่เคยเปรียบเทียบกับตัวน้อยสุดนั้นมีอย่างมากเท่ากับความสูงของต้นไม้ เนื่องจากต้นไม้แบบทวิภาคนี้ได้ดุล ดังนั้นจำนวนข้อมูลที่ต้องนำมาพิจารณาเพื่อหาตัวน้อยสุดอันดับที่สองจึงมีอย่างมากเพียง $\lfloor \log_2 n \rfloor$ ตัว ถ้ามีข้อมูล n ตัว การหาตัวน้อยสุดและตัวน้อยสุดอันดับที่สอง ใช้การเปรียบเทียบข้อมูลไม่เกิน $n + \lfloor \log_2 n \rfloor - 2$ ครั้ง (n–1 ครั้งเพื่อหาตัวน้อยสุด ที่เหลือไม่เกิน $\lfloor \log_2 n \rfloor$–1 ครั้งเพื่อหาตัวน้อยสุดอันดับที่สอง)

วิธีที่อธิบายมานี้คงเขียนเป็นโปรแกรมได้ไม่สั้นกระทัดรัดเท่ากับวิธีแรก ในกรณีที่ใช้กับข้อมูลพื้นฐานทั่วไป อาจทำงานช้ากว่าแบบลุยง่าย ๆ ก็ได้ในทางปฏิบัติ แต่หากการเปรียบเทียบข้อมูลมีภาระมาก (เช่น เป็นการเปรียบเทียบข้อความที่ยาว ๆ) วิธีที่ได้นำเสนอมาที่มีจำนวนการเปรียบเทียบที่น้อยกว่า ก็เป็นวิธีที่น่าสนใจ

การหาตัวหมู่มาก

ให้ A เป็นอาเรย์ที่เก็บข้อมูล n ตัว อยากทราบว่า A มีข้อมูลที่เป็นตัวหมู่มาก (majority) หรือไม่ (ตัวหมู่มากคือ ข้อมูลตัวที่ปรากฏซ้ำกันในอาเรย์เป็นจำนวนเกินครึ่งหนึ่งของปริมาณข้อมูลทั้งหมด)

ลุยหาสักครู่ก็ได้คำตอบ (ดังแสดงข้างล่างนี้) เพียงแค่นับว่า ข้อมูลแต่ละตัวมีอยู่ใน A เกินครึ่งหรือไม่ การนับข้อมูลแต่ละตัว ต้องลุยนับทั้งอาเรย์ การนับทั้ง n ตัว จึงต้องใช้เวลารวมทั้งหมดที่เป็นฟังก์ชันที่แปรตาม n^2 แน่นอน

```
01: hasMajority( A[1..n] )
02: {
03:   for (i = 1 to n) {
04:     c = 0
05:     for (j = 1 to n)
06:       if (A[i] == A[j]) c++
07:     if (c > n/2) return true    // พบตัวหมู่มากแล้ว
08:   }
09:   return false
10: }
```

ถ้าต้องการเร็วกว่านี้ จะมีวิธีอื่นหรือไม่ เราควรสงสัยก่อนเลยว่ามี เพราะวิธีแรกนั้นเป็นแบบลุย ที่อ ๆ ใคร ๆ ก็คิดได้ สังเกตได้ว่า การนับจะง่ายขึ้นมาก ถ้าเราเรียงลำดับข้อมูลในอาเรย์เสียก่อน ดังตัวอย่างข้างล่างนี้

2 3 2 3 4 5 3 3 2 2 2 2 1 1 2 2 2 2 3 3 3 2 2 3 2

เรียงลำดับจากน้อยไปมากได้

1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 5

หลังจากเรียงลำดับแล้ว สามารถนับได้ง่ายขึ้นดังนี้

```
01: hasMajority( A[1..n] )
02: {
03:   sort( A )
04:   c = 0; i = 1
05:   for (j = 1 to n)
06:     if (A[i] == A[j]) {
07:       if (++c > n/2) return true
08:     } else {
09:       i = j; c = 1
10:     }
11:   return false
12: }
```

เวลาการทำงานเท่ากับเวลาในการเรียงลำดับข้อมูล บวกกับเวลาในการนับ การนับที่เขียนให้ดูนี้ทำงานเป็นฟังก์ชันแปรตาม n การเรียงลำดับข้อมูลที่เราจะได้ศึกษาในบทที่ 8 ใช้เวลาเป็นฟังก์ชันที่แปรตาม $n \log_2 n$ สรุปว่า เวลาการทำงานรวมถูกกำหนดโดยการเรียง-ลำดับข้อมูล

มีเร็วกว่านี้อีกหรือไม่ ? (ถามอย่างนี้แสดงว่าควรจะมี) จากตัวอย่างข้างบน ให้สังเกตว่าหลังการเรียงลำดับข้อมูล มี 2 เป็นตัวหมู่มาก และ 2 ก็เป็นตัวมัธยฐานของอาเรย์ด้วย ลองคิดดู ตัวที่เป็นตัวหมู่มากย่อมต้องปรากฏอยู่ในอาเรย์เกินครึ่ง เมื่อนำมาเรียงลำดับ ตัวหมู่มากทั้งหลายก็จะเกาะติดกันยาวเกินครึ่ง ไม่ว่าจะเริ่มที่ไหนในอาเรย์ ก็ต้องเกาะกลุ่มกันจนผ่านเส้นแบ่งครึ่งอาเรย์ ทำให้เราได้เงื่อนไขจำเป็นของตัวหมู่มากว่า ต้องเป็นตัวมัธยฐานด้วย (แต่ในทางกลับกันไม่จำเป็นต้องเป็นจริง นั่นคือ ตัวมัธยฐานไม่จำเป็นต้องเป็นตัวหมู่มาก) ได้วิธีที่ 3 ที่หาตัวมัธยฐาน แล้วนับในอาเรย์ว่า ค่าของตัวมัธยฐานมีเกินครึ่งหรือไม่ เวลาการทำงานจึงเท่ากับเวลาการหาตัวมัธยฐานบวกกับเวลาการวิ่งนับในอาเรย์

```
01: hasMajority( A[1..n] )
02: {
03:   c = 0, m = findMedian( A )
04:   for ( i = 1 to n )
05:     if ( m == A[i] ) ++c
06:   return ( c > n/2 ) ? TRUE : FALSE
07: }
```

เราสามารถหาตัวมัธยฐาน (จะศึกษาในบทที่ 8) ได้โดยใช้เวลาเป็นฟังก์ชันที่แปรตาม n แบบเชิงเส้น ทำให้เราสามารถหาตัวหมู่มากได้ในเวลาเป็นฟังก์ชันเชิงเส้นกับ n ด้วย

แล้วมีวิธีดีกว่านี้หรือไม่ ? หลายคนอาจเกิดคำถามในใจตอนนี้แล้วว่า จะไปรู้ได้อย่างไรว่ามีวิธีดีกว่านี้หรือไม่ ก็คงเป็นการดีถ้าเราจะรู้ว่า ไม่มีวิธีดีกว่านี้แล้ว อันนี้ต้องใช้การพิสูจน์เข้าช่วยจึงจะสามารถบอกได้ว่า ปัญหาที่เผชิญอยู่ ถ้าพบอัลกอริทึมที่มีประสิทธิภาพเท่าใดจึงจะถือว่าดีที่สุด สำหรับปัญหาตัวหมู่มากนี้ เราพบอัลกอริทึม (หาตัวมัธยฐาน+ นับ) ที่ใช้เวลาเป็นฟังก์ชันเชิงเส้นกับ n แล้ว ถามว่า เรามีโอกาสออกแบบอัลกอริทึมอื่นที่ใช้เวลาเป็นฟังก์ชันที่โตช้ากว่า n หรือไม่ อาทิเช่น $\sqrt{n}$ หรือ $\log n$ ได้หรือไม่ คำตอบคือ ไม่มีทาง เพราะการที่จะรู้ว่า อาเรย์ที่มีข้อมูล n ตัวมีตัวหมู่มากหรือไม่นั้น อย่างน้อยเราก็ต้องพิจารณาข้อมูลในอาเรย์เกิน $n/2$ ตัว (เพื่อจะได้สรุปว่า เป็นตัวหมู่มาก) ซึ่งเป็นฟังก์ชันเชิงเส้นกับ n จึงสรุปได้ว่า ไม่มีทางที่จะหาตัวหมู่มากได้โดยพิจารณาข้อมูลน้อยกว่า $n/2$ ตัว นั่นคือ เราพบอัลกอริทึมที่ดีแล้ว (ซึ่งก็คือการหาตัวมัธยฐาน + การนับ)

ขอเตือนว่า อย่าเพิ่งดีใจมากเกินไปที่พบขั้นตอนวิธีที่ดีสุด เพราะไม่ได้หมายความว่า จะไม่มีวิธีอื่นอีกซึ่งใช้เวลาเป็นเชิงเส้นเหมือนกัน แต่ซับซ้อนน้อยกว่า มีความชันของเส้นฟังก์ชันเวลาการทำงานที่ลาดกว่า หมายความว่า ทำงานได้เร็วกว่า

ยังมีอีกวิธีหนึ่งในการหาตัวหมู่มากนี้ที่ออกพิกล ๆ คือ แทนที่จะไปหาตัวมัธยฐานให้เสียเวลาแล้วค่อยมานับ ทำไมเราไม่เสี่ยงสุ่มข้อมูลมาหนึ่งตัวในอาเรย์ แล้วนับเลย ก็คงมีคนแย้งทันทีว่า วิธีนี้มีโอกาสผิด แน่นอนว่า วิธีนี้มีโอกาสผิด แต่ก็สามารถพูดได้อีกนัยหนึ่ง (แบบเข้าข้างตัวเอง) ว่า วิธีนี้ก็มีโอกาสถูก สิ่งที่น่าสนใจก็คือ โอกาสถูกเป็นเท่าใด เราลองมาแยกเป็นสองกรณี

1. กรณีที่อาเรย์ A ไม่มีตัวหมู่มาก : กรณีนี้สบายใจได้เลยว่า จะต้องได้คำตอบที่ถูกต้องแน่นอน เพราะไม่ว่าเราจะสุ่มข้อมูลใดมานับ ย่อมได้ไม่เกินครึ่งของจำนวนข้อมูล (เพราะอาเรย์ A ไม่มีตัวหมู่มาก)
2. กรณีที่อาเรย์ A มีตัวหมู่มาก : เรามีโอกาสเกินครึ่งที่สุ่มข้อมูลได้เป็นตัวหมู่มาก เพราะข้อมูลที่เป็นตัวหมู่มากมีจำนวนเกินครึ่ง

เห็นอย่างนี้แล้วยังกังวลอยู่หรือเปล่า ถ้าเป็นนักพนันก็คงยอมเสี่ยงใช้วิธีนี้ เพราะเขียนก็ง่าย รับรองว่า ทำงานเร็วแน่ ๆ (แค่ สุ่ม+นับ) แถมโอกาสที่จะได้คำตอบถูกมีเกินครึ่ง เนื่องจากเราไม่ใช่นักพนัน เราก็ไม่อยากใช้วิธีนี้ แต่ว่าคนส่วนใหญ่เชื่อเรื่องดวง เชื่อว่าคนหนึ่งเกิดมาคงไม่โชคร้ายตลอดชีวิตหรอก ทำไมเราไม่ "สุ่ม+นับ" แบบนี้สัก 30 ครั้งดู ถ้ามีสักหนึ่ง

ครั้งในคำตอบบอกว่า "มีตัวหมู่มาก" เราก็สบายใจว่า มีแน่ (เพราะโชคดีได้สุ่มพบตัวหมู่มากจริง ๆ จึงนับแล้วเกินครึ่ง) แต่ถ้าทั้ง 30 ครั้งมีแต่คำตอบว่า "ไม่มีตัวหมู่มาก" เราก็น่าจะเชื่อว่า เราคงไม่โชคร้ายขนาดว่า ความจริงมีตัวหมู่มาก แต่สุ่มเท่าไรก็ไม่พบ และยิ่งถ้าเราเป็นนักสถิติ ย่อมเคยเรียนมาว่า โอกาสถูกต่อครั้งมีเกินครึ่ง แสดงว่าโอกาสที่ผิดทั้ง 30 ครั้ง (คือโชคร้ายจริง !!!) มีน้อยกว่า 1 ในพันล้าน (2^{-30}) ทั้งนี้เนื่องจากการสุ่ม+นับแต่ละครั้งไม่ขึ้นต่อกัน ว่าไง คราวนี้น่าใช้วิธี "สุ่ม+นับ k ครั้ง" หรือไม่ ถ้ายังไม่กล้าเสี่ยง ลองให้ k เป็นสัก 50 แล้วคำนวณความโชคร้ายที่จะได้คำตอบผิดดูสิ จะพบว่า มีน้อยกว่าโอกาสที่วงจรอิเล็กทรอนิกส์ของเครื่องคอมพิวเตอร์จะทำงานผิดพลาดเสียอีก

```
01: hasMajority( A[1..n], k )
02: {
03:   for ( j = 1 to k ) {  // สุ่ม + นับ อย่างมาก k ครั้ง
04:     c = 0, m = A[ random(1, n) ]
05:     for ( i = 1 to n )
06:       if ( m == A[i] ) ++c
07:     if ( c > n/2 ) return TRUE
08:   }
09:   return FALSE
10: }
```

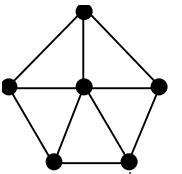
เราจะศึกษาอัลกอริทึมเชิงสุ่มลักษณะเช่นนี้ในบทที่ 12 ซึ่งพบว่า มีใช้กันในทางปฏิบัติจริง

การให้สีปมของกราฟด้วยสีสามสี

ปัญหานี้รับกราฟมาเพื่อหาว่า มีวิธีหรือไม่ในการให้สีกับปมของกราฟด้วยสีเพียงสามสี โดยปมที่ปลายของเส้นเชื่อมเดียวกันในกราฟต้องมีสีไม่เหมือนกัน จะหาคำตอบได้อย่างไร ?

วิธีง่าย ๆ ที่อ ๆ ก็คือ ลุยให้สีทุกรูปแบบ ถ้าลองให้ทุกรูปแบบแล้วไม่มีแบบไหนถูกต้องเลย ก็แสดงว่า ให้สีไม่ได้ แต่ถ้าเจอสักแบบที่ให้สีได้ ก็หยุดการทำงานแล้วก็ตอบว่า ให้สีได้ ก็เท่านั้น คำถามที่ตามมาคือ วิธีนี้ต้องลองให้สีกี่แบบ จำนวนการให้สีขึ้นกับลักษณะของกราฟที่ได้รับ เอาเป็นว่า เราต้องการรู้กรณีช้าสุดก็แล้วกัน (ซึ่งคือกรณีที่ให้สีกราฟไม่ได้ กว่าจะสรุปได้ก็ต้องลองทุกแบบ) ถ้ากราฟมีปม n ปม ในกรณีเลวสุดจะต้องลอง 3^n แบบ (เพราะให้สีปมหนึ่งปมได้สามสี มี n ปมก็ให้สีได้ 3^n แบบ) วิเคราะห์แล้วได้คำตอบแบบนี้ก็คงต้องร้องว่าไม่ไหว มากเกินไป เพราะถ้ากราฟที่ให้มามีสัก 50 ปม ต้องลอง 3^{50} แบบ ลองคิดดูถ้าใช้คอมพิวเตอร์ที่สามารถให้สีและทดสอบได้พันล้านรูปแบบใน 1 วินาที ก็ต้องรอ 3^{50} / 10^9 วินาทีซึ่งมากกว่า 3 ล้านปี !!!

ผู้อ่านบางคนอาจต้องการแย้งว่า ทำไมไปลุยแบบที่อ ๆ ตั้ง 3^n แบบเล่า ก็เพียงแค่ดูซิว่า ในกราฟที่ให้มานี้มีสี่ปมใดที่มีเส้นเชื่อมต่อกันถึงกันหมดหรือไม่ (นั่นคือ มีกราฟย่อยแบบบริบูรณ์ขนาดสี่ปมหรือไม่) ถ้ามีก็แสดงว่า ใช้สามสีไม่พอแน่ ๆ (เพราะกราฟย่อยแบบบริบูรณ์ขนาด k ปมต้องการ k สี) การลองลุยหยิบปมสี่จุดจาก n จุดมาทดสอบว่า เป็นกราฟย่อยแบบบริบูรณ์หรือไม่นั้น ทดลองเพียง $C(n,4) = n(n-1)(n-2)(n-3) / 24$ แบบ ซึ่งน้อยกว่า 3^n มากมายมหาศาล (เช่น ถ้า $n = 50$ จะทดสอบเพียง 2 แสนกว่ากรณีเท่านั้น แค่ดีดนิ้วก็เสร็จแล้ว) ข่าวร้ายก็คือ ถ้าทดลองหยิบ 4 ปมทุกรูปแบบแล้ว ไม่พบกราฟย่อยแบบบริบูรณ์ขนาดสี่จุดเลย ก็ยังสรุปไม่ได้ว่า ใช้สามสีได้ ดูกราฟวงล้อในรูปที่ 1–2 เป็นตัวอย่าง



รูปที่ 1–2 กราฟวงล้อห้าซี่ต้องใช้สี่สีถึงพอ

วิธีหนึ่งที่ได้ผลดีมากในการหาคำตอบของปัญหานี้คือ การใช้กลวิธีการย้อนรอย (ซึ่งจะอธิบายอย่างละเอียดในบทที่ 11) วิธีนี้จะให้คำตอบรวดเร็วมากสำหรับกราฟทั่วไป แต่ก็จะมีกราฟบางแบบที่หาคำตอบได้ในเวลาที่ช้ามากจนรอไม่ไหวเหมือนกัน

ใครจะลองคิดออกแบบวิธีแก้ปัญหานี้ตอนนี้ก็ได้ แต่ขอบอกไว้ว่า ยังไม่มีใครพบวิธีที่ประกันว่า จะได้คำตอบในเวลาอันรวดเร็วกับกราฟทุกกรณี จัดเป็นหนึ่งในกลุ่มปัญหาที่ “สงสัยว่ายาก” ที่เขียนว่า สงสัย ก็เพราะว่า ยังไม่มีใครพิสูจน์ได้ว่ามันยาก และก็ยังไม่มีใครหาวิธีแก้ปัญหาที่ทำงานได้ในเวลาอันรวดเร็ว ปัญหาประเภทนี้พบได้ในทางปฏิบัติ และพบในหลายวงการ จัดเป็นกลุ่มปัญหาที่น่าสนใจ ซึ่งเราจะได้อธิบายกันในบทที่ 13

การออกแบบอัลกอริทึม

จากตัวอย่างที่ได้แสดงในหัวข้อที่แล้ว พอจะสรุปขั้นตอนการออกแบบและวิเคราะห์อัลกอริทึมได้ดังนี้

- ต้องเข้าใจตัวปัญหาอย่างถ่องแท้ อะไรคือข้อมูลขาเข้า มีขนาดใหญ่ได้เท่าไร (10, 1000, 10^6,…) อะไรคือผลลัพธ์ที่ต้องการ ต้องการคำตอบเดียว หรือทุกคำตอบ ต้องการคำตอบที่ดีที่สุด หรือแค่คำตอบดี ๆ ก็พอ ต้องการคำตอบที่ถูกต้องแน่ หรือยอมเสี่ยงผิดได้แต่โอกาสน้อย ต้องการคำตอบเร็วขนาดไหน (ภายใน 1 วินาที ภายใน 1 นาที ภายใน 1 เดือน…) ปัญหานี้ต้องแก้ไขหาคำตอบถี่ขนาดใด (ครั้งสองครั้งในชีวิต หรือทุกนาที) ปัจจัยเหล่านี้จะเป็นตัวช่วยกำหนดแนวทางการออกแบบ
- ปัญหาที่ต้องการแก้ไขนี้เหมือนหรือสามารถแปลงไปเป็นปัญหาที่พบได้ใน “ท้องตลาด” หรือไม่ (เช่น จากหนังสือ วารสาร หรืออินเตอร์เน็ต) ถ้าเหมือน จะได้ไม่ต้องเสียเวลาคิด (แล้วเอาเวลาไปคิดแก้ปัญหาที่ไม่มีใครแก้ไขมาก่อนจะดีกว่า) ถ้าไม่เหมือน ปัญหาของเราเป็นกรณีพิเศษ หรือกรณีทั่วไปของปัญหาใน “ท้องตลาด” หรือไม่ ถ้าใช่ อย่างน้อยก็มีแนวทางการแก้ปัญหาอยู่บ้าง อย่างไรก็ตามขอเน้นว่า ปัญหาส่วนใหญ่ที่จะพบในหนังสือเล่มนี้เป็นปัญหา “ท้องตลาด” โดยมีจุดประสงค์ให้ผู้อ่านสามารถหาแนวทางแก้ไขปัญหาได้ โดยไม่ต้องไปดูใน “ท้องตลาด”
- การออกแบบอัลกอริทึมต้องทำหลายรอบ หลังการออกแบบในแต่ละรอบต้องวิเคราะห์ว่าได้ผลตามที่ต้องการหรือไม่ โดยทั่วไปการวิเคราะห์ทำให้เราเข้าใจพฤติกรรม จุดเด่นจุดด้อย ที่มักนำไปสู่การปรับปรุงตัวอัลกอริทึมให้ดีขึ้น จนได้ประสิทธิภาพตามข้อกำหนด
- โครงสร้างข้อมูลมีผลต่อประสิทธิภาพของอัลกอริทึมด้วย เพราะอัลกอริทึมเป็นกระบวนการที่รับข้อมูลขาเข้ามาประมวลผลให้ได้ผลลัพธ์ที่ต้องการ จึงต้องจัดเก็บและจัดการข้อมูลขาเข้าและข้อมูลเสริมต่าง ๆ ระหว่างการแก้ไขปัญหา โครงสร้างข้อมูลที่เลือกเพื่อจัดเก็บและจัดการข้อมูลจึงมีผลโดยตรงต่อประสิทธิภาพทั้งเชิงเวลา และเนื้อที่หน่วยจำที่ใช้
- ความยากง่ายของปัญหามีผลกับกลวิธีในการออกแบบอัลกอริทึม
- ถ้าง่าย มักใช้กลวิธีการแบ่งแยกและเอาชนะ (divide and conquer) กำหนดการพลวัต (dynamic programming) หรืออัลกอริทึมเชิงละโมบ (greedy algorithm) แต่ถ้าต้องการง่าย ๆ เร็ว ๆ ก็อาจใช้อัลกอริทึมเชิงสุ่ม (randomized algorithms)

- ถ้ายาก มักใช้การค้นปริภูมิสถานะ (state space search) เช่น การย้อนรอย (backtracking) หรือการขยายและจำกัดเขต (branch and bound) ถ้าลดความทะเยอทะยานของคำตอบลง เช่น ต้องการคำตอบที่ดี ๆ ก็พอไม่จำเป็นต้องดีที่สุด ก็อาจใช้การค้นเฉพาะที่ (local search) หรืออัลกอริทึมเชิงประมาณ (approximation algorithm) เข้าช่วย
- แต่ไม่ว่าปัญหาจะง่ายหรือยาก ถ้าเป็นปัญหาที่มีขนาดเล็กจริง ๆ บางทีลุยเอา (แบบที่เขาเรียกกันว่า brute force) ก็เพียงพอ

เนื้อหาที่จะนำเสนอ

กล่าวโดยสรุป เราจะศึกษาเรื่องสามเรื่องด้วยกันในหนังสือเล่มนี้คือ การวิเคราะห์อัลกอริทึม การออกแบบอัลกอริทึม และการศึกษาความซับซ้อนของปัญหา

การวิเคราะห์อัลกอริทึมนั้นจะเน้นที่ประสิทธิภาพเชิงเวลาของอัลกอริทึม (จะเน้นเรื่องเนื้อที่หน่วยความจำบ้างเป็นครั้งคราว) การวิเคราะห์ประสิทธิภาพเชิงเวลาคือ การนับจำนวนครั้งที่คำสั่งทำงานระหว่างการแก้ไขปัญหา โดยจะนับอย่างคร่าว ๆ (จะเห็นต่อไปว่าคร่าว ๆ อย่างไร) เพียงพอให้เราสามารถเปรียบเทียบอัลกอริทึมในแง่การเติบโตของเวลากับขนาดของปัญหา

จากนั้นเราจึงมาศึกษากลวิธีมาตรฐานทั่วไป ในการออกแบบอัลกอริทึม อันประกอบด้วยการแบ่งแยกและเอาชนะ กำหนดการพลวัต อัลกอริทึมเชิงละโมบ การค้นคำตอบในปริภูมิสถานะ การย้อนรอย การขยายและจำกัดเขต อัลกอริทึมเชิงสุ่ม และอัลกอริทึมเชิงประมาณ แต่ละกลวิธีจะเหมาะกับปัญหาในแต่ละลักษณะ การนำเสนอในหัวข้อเหล่านี้จะเป็นการนำเสนอด้วยตัวอย่าง โดยนำเสนอปัญหา “ของเล่น” ฟังง่าย ๆ แต่วิธีแก้ไขปัญหาและการวิเคราะห์บางทีอาจยุ่งยาก ปัญหา “ของเล่น” เหล่านี้ฟังดูแล้วรู้สึกเป็นปัญหายกเมฆขึ้นมา แต่ในโลกแห่งความเป็นจริงแล้ว มีให้พบให้เห็น และต้องการวิธีแก้ไขกันจริง ๆ และเราจะนำเสนอโครงสร้างข้อมูลใหม่ ๆ (นอกเหนือจากที่ได้ศึกษากันมาในวิชาโครงสร้างข้อมูลเบื้องต้น) เสริมในกรณีที่ประสิทธิภาพของอัลกอริทึมที่นำเสนอนั้นขึ้นกับโครงสร้างข้อมูลที่เหมาะสม

ปิดท้ายเนื้อหาที่จะนำเสนอด้วยการจำแนกปัญหาตามความซับซ้อน (หรือจะพูดว่าตามความยากง่าย) ของปัญหา โดยเน้นกลุ่มปัญหาหนึ่งที่เรียกว่า เอ็นพีบริบูรณ์ (NP-

complete) ซึ่งเป็นกลุ่มปัญหาที่น่าสนใจเพราะว่า เป็นกลุ่มปัญหาที่พบมากในทางปฏิบัติ อีกทั้งเป็นกลุ่มปัญหาที่ทุกคนสงสัยว่ายาก แต่ยังพิสูจน์ไม่ได้ อีกทั้งทุกปัญหาในกลุ่มนี้มีความซับซ้อนเท่าเทียมกันทั้งหมด ถ้าใครพบอัลกอริทึมที่มีประสิทธิภาพที่แก้ปัญหาหนึ่งในกลุ่มนี้ ก็หมายความว่า ได้พบวิธีแก้ทุกปัญหาในกลุ่มด้วย (คนนั้นดังแน่ ๆ)

สิ่งที่จะไม่สนใจ

เนื่องจากประสิทธิภาพการทำงานของโปรแกรมขึ้นกับปัจจัยหลายอย่างมาก สิ่งที่เราจะสนใจคือ อัลกอริทึมของโปรแกรม ซึ่งเป็นแนวคิด หรือ “กึ๋น” การทำงานของโปรแกรม เราจะไม่มาสนใจหรือโต้เถียงกันในประเด็นต่อไปนี้ ถึงแม้ว่าจะมีผลโดยตรงกับเวลาและปริมาณหน่วยความจำที่ใช้ก็ตาม

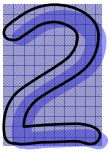
- *ใช้ภาษาคอมพิวเตอร์นี้ดีกว่า* จะไม่มาสนใจหรอกว่าใช้ machine code เร็วสุด น่าใช้คำสั่ง MMX หรือคำสั่งใหม่ ๆ หรือว่า ใช้ Java แล้วช้า เวลาการทำงานก็ไม่แน่นอน ใช้ C ดีกว่า อะไรทำนองนี้
- ใช้ตัวแปลของบริษัท A ดีกว่าของบริษัท B อันนี้เราจะไม่มาโฆษณาผลิตภัณฑ์
- *ใช้คำสั่งที่เหมาะสม* เรื่องจุกจิกเกี่ยวกับคำสั่งในภาษาอาทิเช่น ++k จะเร็วกว่า k = k+1 นิดหนึ่ง เป็นต้น
- *ใช้ความรู้เรื่องสถาปัตยกรรมคอมพิวเตอร์* เราจะสนใจเฉพาะอัลกอริทึมซึ่งใช้หน่วยประมวลผลเพียงตัวเดียว อีกทั้งจะไม่สนใจเรื่องที่ไปขึ้นกับสถาปัตยกรรมคอมพิวเตอร์ อาทิเช่น การอ้างอิงหน่วยความจำที่ตำแหน่งที่หาร 4 ลงตัว จะเร็วกว่าที่หารไม่ลงตัว หรือการอ้างอิงหน่วยความจำในอาเรย์สองมิติแบบใช้วงวน for สองวงซ้อนกันแล้วเปลี่ยน แถวแนวนอนก่อนแนวตั้ง จะช้ากว่าเปลี่ยนแถวแนวตั้งก่อนแนวนอนเพราะเรื่องของ data cache เป็นต้น
- *ใช้ลูกเล่นการเขียนโปรแกรม* โปรแกรมที่ทำงานตามแนวคิดของอัลกอริทึมเดียวกัน อาจมีได้หลากหลายรูปแบบ นักเขียนโปรแกรมสามารถใช้ลูกเล่นมากมายเพื่อให้โปรแกรมของตัวเองทำงานได้เร็วขึ้น อาทิเช่น การใช้ตัวเฝ้ายาม (sentinel) การเปลี่ยนการเรียกซ้ำ (recursive) เป็นแบบวงวนธรรมดา เป็นต้น

ขอย้ำอีกครั้งว่า ปัจจัยต่าง ๆ ข้างต้นล้วนมีสำคัญอย่างยิ่งต่อประสิทธิภาพของโปรแกรม แต่สิ่งที่เราต้องการเน้นคือ อัลกอริทึมหรือแนวคิดการทำงาน ตัวอัลกอริทึมมีผลโดยตรงอย่างมหาศาลมากกว่าเรื่องที่พูดถึงเสียอีก ใช้อัลกอริทึมที่ดีแต่เขียนโปรแกรมไม่มีลูกเล่น จะได้ผลที่ดีกว่าใช้อัลกอริทึมที่ไม่เหมาะสมแต่เขียนโปรแกรมสุดยอด (ความจริงแล้ว เราน่าจะมีความสามารถทั้งออกแบบอัลกอริทึมที่ดีและเขียนโปรแกรมได้สุดยอด)

พื้นฐานที่ต้องมี

ขอเน้นตรงนี้ว่า ผู้ที่จะศึกษาศาสตร์ทางอัลกอริทึมให้ได้ผลดี ต้องหมั่นคิดและปฏิบัติ พื้นฐานที่ผู้อ่านต้องมีคือ ต้องผ่านวิชาการเขียนโปรแกรมคอมพิวเตอร์ โครงสร้างข้อมูล และคณิตศาสตร์ดีสครีต แทบว่าจะขาดอย่างใดอย่างหนึ่งไม่ได้เอาเสียเลย จะออกแบบอัลกอริทึมที่ดีได้ก็ต้องวิเคราะห์ด้วยคณิตศาสตร์เป็น ต้องเลือกใช้โครงสร้างข้อมูลที่เหมาะสม เพราะมีผลโดยตรงต่อประสิทธิภาพการทำงาน และแน่นอนว่า ต้องเปลี่ยนอัลกอริทึมที่เป็นแนวคิดให้เป็นรูปธรรมโดยการเขียนเป็นโปรแกรมสมบูรณ์ได้

ปัญหาและอัลกอริทึม



ก่อนจะเข้าเรื่องการวิเคราะห์และออกแบบอัลกอริทึม บทนี้ขอสร้างความเข้าใจและสร้างความคุ้นเคยเกี่ยวกับลักษณะของปัญหาที่เราจะพบกันในหนังสือเล่มนี้ว่า มีลักษณะใดกันบ้าง จะอธิบายความหมายของคำว่าตัวอย่างปัญหา ขนาดของตัวอย่างปัญหา และคำว่าอัลกอริทึมในบทนี้ด้วย

ปัญหา

คำว่า *ปัญหา* (problem) ที่เราจะสนใจกันเป็นปัญหาที่คำนวณหาคำตอบได้ (computational problem) หรืออีกนัยหนึ่งคือ ปัญหาที่ให้คอมพิวเตอร์แก้ไขหาคำตอบให้ได้ อาทิเช่น ปัญหาการหาตัวน้อยสุด การทดสอบจำนวนเฉพาะ การให้สีปมของกราฟ การเรียงลำดับข้อมูล การหาตัวหมู่มาก และอื่น ๆ อีกสารพัด การบรรยายลักษณะของปัญหานั้นจะต้องระบุลักษณะของข้อมูลขาเข้า และผลที่ต้องการให้ชัดเจน เช่น

ปัญหาการหาค่าน้อยสุด :

ข้อมูลขาเข้า : เซตของจำนวนจริง $S = \{ a_1 , a_2 , \ldots, a_n \}$
ผลที่ต้องการ : a_k โดยที่ $a_k \in S$ และ $a_k \le a_j$ สำหรับค่า $j = 1, 2, \ldots, n$

ปัญหาการทดสอบความเป็นจำนวนเฉพาะ :

ข้อมูลขาเข้า : จำนวนเต็มบวก p, $p > 1$
ผลที่ต้องการ : จริง ถ้า p เป็นจำนวนเฉพาะ, เท็จ ถ้า p เป็นจำนวนประกอบ

ปัญหาการให้สีปมของกราฟ :

ข้อมูลขาเข้า : กราฟ $G = (V, E)$ และจำนวนเต็มบวก k
ผลที่ต้องการ : จริง ถ้าเราสามารถให้สีปมใน G ด้วยสีอย่างมาก k สี
เท็จ ถ้าไม่มีวิธีให้สีปมใน G ด้วยสีอย่างมาก k สี

ปัญหาที่คำนวณหาคำตอบได้พอแบ่งเป็นหมวดหมู่ได้ดังนี้

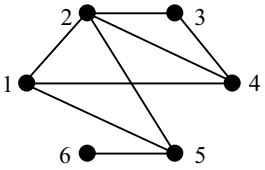
1. ปัญหาทางเซต ลำดับ และสตริง (set sequence and string problems) เช่น การเรียงลำดับ (sorting) การค้น (searching) การเลือก (selection) การแบ่งส่วน (partitioning) เป็นต้น
2. ปัญหาเชิงจำนวน (numerical problems) เช่น การทดสอบความเป็นจำนวนเฉพาะ (primality testing) การแยกตัวประกอบ (factorization) การยกกำลัง (exponentiation) การคูณเมทริกซ์ (matrix multiplication) การสร้างจำนวนสุ่ม (random number generation) เป็นต้น
3. ปัญหาทางกราฟ (graph problems) เช่น สภาพเชื่อมโยง (connectivity) ต้นไม้แบบทอดข้ามเล็กสุด (minimum spanning tree) วิถีสั้นสุด (shortest path) การไหลในข่ายงาน (network flow) การเดินทางของพนักงานขาย (traveling salesperson) การให้สีกราฟ (graph coloring) เป็นต้น
4. ปัญหาทางเรขาคณิตเชิงคำนวณ (computational geometry problems) เช่น เปลือกนูน (convex hull) จุดใกล้กันที่สุด (closest point) การสามเหลี่ยม (triangulation) เป็นต้น

ตัวอย่างปัญหา

ให้อัลกอริทึม A ถูกออกแบบมาไว้แก้ปัญหา P สิ่งที่ A รับเป็นข้อมูลขาเข้าคือ ข้อมูลขาเข้าที่เป็นไปได้ของ P เราเรียกข้อมูลขาเข้าหนึ่งของ P ว่า *ตัวอย่างข้อมูลขาเข้า* (input instance) หรือบางทีเรียกว่า *ตัวอย่างปัญหา* (problem instance) หรือบางครั้งเรียกสั้น ๆ ว่า ตัวอย่าง (instance) เฉย ๆ เช่น

- {3, 4, 5, 7, 30, 0, 2, 3, 5, 6} เป็นตัวอย่างหนึ่งของปัญหาการหาค่าน้อยสุด
- 13746687628251 เป็นตัวอย่างหนึ่งของปัญหาการทดสอบความเป็นจำนวนเฉพาะ

- กราฟ $G = (V, E)$ โดยที่ $V = \{1,2,3,4,5,6\}$, $E = \{ (1,2), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (5,6) \}$ ซึ่งแทนกราฟในรูปที่ 2–1 และ $k = 3$ ก็เป็นตัวอย่างหนึ่งของปัญหาการให้สีปมของกราฟ



รูปที่ 2–1 กราฟในตัวอย่างของปัญหาการให้สีกราฟ

ขนาดของตัวอย่างปัญหา

ให้ $f_A(n)$ เป็นฟังก์ชันที่แทนประสิทธิภาพของอัลกอริทึม A ตัว n คือ ขนาดของตัวอย่างปัญหาที่อัลกอริทึม A รับไปหาคำตอบ ขอเน้นว่า n คือ ขนาดของตัวอย่างปัญหา ไม่ใช่ค่าของตัวอย่างปัญหา หน่วยของขนาดที่ว่านี้ จะเป็นบิต ไบต์ เวิร์ด ตัว ก้อน ปม เส้น อะไรก็แล้วแต่สะดวก เช่น รายการ {3, 4, 5, 7, 0, 2, 3, 5, 6} มีขนาด 9 ตัว หรือจะกล่าวว่า ใช้ที่เก็บ 9 เวิร์ดก็ได้ เลข 13746286234876298251 มีขนาด 20 หลักฐานสิบ หรือจะพูดว่า 64 บิตฐานสอง (เพราะต้องใช้ที่เก็บ $1 + \lfloor \log_2 13746286234876298251 \rfloor = 64$ บิต) ก็ได้ หรือกราฟในรูปที่ 2–1 มีขนาด 6 ปม 8 เส้น จากสามตัวอย่างข้างบนนี้ พอสรุปขนาดของตัวอย่างปัญหาได้ดังนี้

ปัญหาการหาค่าน้อยสุด :

ข้อมูลขาเข้า : เซตของจำนวนจริง $S = \{ a_1 , a_2 , \ldots, a_n \}$

ขนาดของตัวอย่างปัญหา : n

ปัญหาการทดสอบจำนวนเฉพาะ :

ข้อมูลขาเข้า : จำนวนเต็มบวก p, $p > 1$

ขนาดของตัวอย่างปัญหา : $1 + \lfloor \log_2 p \rfloor$

ปัญหาการให้สีปมของกราฟ :

ข้อมูลขาเข้า : กราฟ $G = (V, E)$ และจำนวนเต็มบวก k

ขนาดของตัวอย่างปัญหา : $|V| + |E|$

บางคนอาจสงสัยว่า ทำไมต้องมาจู้จี้กับจำนวนหลัก จำนวนบิตในปัญหาการทดสอบจำนวนเฉพาะด้วย ในขณะที่ไม่เห็นสนใจว่า จะใช้กี่บิตเลยสำหรับจำนวนในเซต S ของปัญหาการหาค่าน้อยสุด การระบุลักษณะของข้อมูลขาเข้าของปัญหานั้น ถ้าจะให้ละเอียดต้องบอกขนาดของข้อมูลด้วย อาทิเช่น จำนวนเต็มแต่ละจำนวนที่ให้มานั้นจะมีขนาดไม่จำกัดหรือไม่ (หมายถึงกี่หลักก็ได้หรือไม่) สำหรับบางปัญหาโดยทั่วไปถือว่า เป็นที่รู้กัน ไม่ต้องระบุอย่างเด่นชัด เช่น ปัญหาการหาค่าน้อยสุดนั้น แต่ละจำนวนในเซต S มักมีขนาดไม่ใหญ่มาก คำว่า "ไม่ใหญ่มาก" ในที่นี้มักตีความว่า ไม่มากเกินที่เก็บได้ในหนึ่งเวิร์ดของเครื่องคอมพิวเตอร์ นั่นคือ "เล็ก" พอที่จะทำให้การประมวลผล เช่น บวก ลบ คูณ หาร เปรียบเทียบ... จำนวนดังกล่าวใช้เวลาคงตัวไม่เปลี่ยนแปลงตามค่าของจำนวนนั้น (หนึ่งเวิร์ดของคอมพิวเตอร์ในปัจจุบันก็สี่ไบต์ ดังนั้นคำว่า "เล็ก" ในที่นี้ก็เก็บจำนวนเต็มได้เป็นหลายพันล้านทีเดียว)

แต่ถ้าเป็นปัญหาการทดสอบจำนวนเฉพาะ ส่วนใหญ่ถือว่า เป็นจำนวนที่มีค่ามาก จำนวนหลักของ p จึงเป็นตัวกำหนดขนาดของข้อมูล โดยทั่วไปถ้าเป็นปัญหาเชิงจำนวน ซึ่งเป็นปัญหาประเภทที่ยุ่งเกี่ยวกับการประมวลผลค่าของจำนวนโดยตรง จะไม่จำกัดจำนวนหลักของข้อมูลขาเข้าของปัญหา แต่ถ้าเป็นปัญหาที่ยุ่งเกี่ยวกับเซตหรือรายการของจำนวน เช่น การเรียงลำดับข้อมูล การหาผลรวมของข้อมูล การผสานรายการข้อมูล การหาค่าน้อยสุด การเลือกตัวมัธยฐานของกลุ่ม เป็นต้น จะถือว่า แต่ละจำนวนในเซตหรือรายการเหล่านั้นมีขนาด "เล็ก"

ที่ต้องมาเน้นในประเด็นนี้เพราะ หากคิดขนาดของตัวอย่างปัญหาผิดไป จะส่งผลโดยตรงต่อการตีความผลลัพธ์ของการวิเคราะห์อัลกอริทึม มาดูตัวอย่างง่ายต่อไปนี้ สมมติว่า เราออกแบบวิธีการทดสอบจำนวนเฉพาะโดยใช้การลองหารดื้อ ๆ ทื่อ ๆ คือ ถ้าต้องการทดสอบว่า p เป็นจำนวนเฉพาะหรือไม่ ก็ลุยทดลองหาร p ด้วย 2, 3, ... ไปเรื่อย ๆ จนถึง p–1 ถ้าพบตัวที่หาร p ลงตัวก็แสดงว่า p เป็นจำนวนประกอบ แต่ถ้าทดลองหารทั้ง p–2 ตัวแล้วไม่มีตัวใดหาร p ลงตัวเลย ก็แสดงว่า p เป็นจำนวนเฉพาะ แน่นอนว่า การทำงานจะเป็นวงวน หมุนลองหารอย่างมาก p–2 รอบ ถ้ากำหนดให้การทดสอบว่าหารลงตัวหรือไม่นั้นใช้เวลาคงตัว แสดงว่าประสิทธิภาพการทำงานของอัลกอริทึมนี้ใช้เวลาในกรณีช้าสุดเป็นฟังก์ชันเชิงเส้นของ p การพบว่าอัลกอริทึมใช้เวลาเป็นเชิงเส้นนั้น จะรู้สึกว่าเร็ว แต่อย่าลืมมันเป็นเชิงเส้นกับค่าของจำนวนที่มาทดสอบจำนวนเฉพาะ ไม่ใช่ขนาดของจำนวนที่มาทดสอบ สิ่งที่เราอยากรู้ก็คือ ถ้าเพิ่มขนาดของ p ไปอีก 1 หลัก (ไม่ใช่เพิ่มค่าของ p ไปอีก 1) แล้วจะเสียเวลาเพิ่มขึ้นเท่าใด นั่นหมายความว่า เราต้องการฟังก์ชันที่แปรตามขนาดของ

p เนื่องจากอัลกอริทึมนี้เป็นฟังก์ชันเชิงเส้นของค่า p แสดงว่า เป็นฟังก์ชันเลขชี้กำลังของขนาดของ p (เพราะ $p = 2^{\log_2 p}$) ก็จะให้ความรู้สึกทันทีว่า การเพิ่มขนาดของ p ไปอีก 1 บิตจะเสียเวลาเพิ่มขึ้นอีก 1 เท่าตัว แสดงว่าเป็นวิธีที่ช้าเกินไป (ลองคิดดูว่าจะเสียเวลาเท่าไรในการทดสอบเลข 100 บิต ที่ถือว่าเล็กมากสำหรับปัญหาการทดสอบความเป็นจำนวนเฉพาะที่ทำกันทุกวันนี้)

อัลกอริทึม

อัลกอริทึมเป็นคำสำคัญของหนังสือเล่มนี้ จึงขอนิยามคำว่าอัลกอริทึมกันเสียก่อน *อัลกอริทึม* (Algorithm - ศัพท์ราชบัณฑิตใช้คำว่า "ขั้นตอนวิธี" แต่หนังสือเล่มนี้ขอใช้ทับศัพท์ เพราะเป็นคำที่มาจากชื่อคน) หมายถึง ลำดับของขั้นตอนเชิงคำนวณซึ่งแปลงตัวอย่างข้อมูลขาเข้าของปัญหา ไปเป็นผลลัพธ์ที่ต้องการ ขอเน้นว่า ขั้นตอนต่าง ๆ ในอัลกอริทึมต้องเป็นขั้นตอนที่ใช้หลักการคำนวณ หรือพูดง่าย ๆ คือ ขั้นตอนต่าง ๆ สามารถแปลงไปเป็นคำสั่งที่ทำงานด้วยเครื่องคอมพิวเตอร์ได้ (ดังนั้นอะไรที่ใช้อารมณ์ก็ไม่น่าจัดเป็นอัลกอริทึม) อัลกอริทึมของปัญหา P ต้องสามารถหาคำตอบที่ถูกต้องให้กับข้อมูลขาเข้าทุกแบบของปัญหา P ได้ในเวลาอันจำกัด ดังนั้นจุดประสงค์หลักของการออกแบบอัลกอริทึมสำหรับแก้ไขปัญหาหนึ่งคือ การทำงานที่ถูกต้องและมีประสิทธิภาพ [1]

เราสามารถบรรยายอัลกอริทึมได้ด้วยภาษาเขียน บรรยายไปเป็นย่อหน้าสั้น ๆ ได้ใจความและเข้าใจถึงแนวคิดการทำงาน อาทิเช่น การเรียงลำดับข้อมูลแบบผสานมีขั้นตอนการทำงานดังนี้

การเรียงลำดับข้อมูลแบบผสานนั้นอาศัยการแบ่งชุดข้อมูลที่ต้องการเรียงลำดับออกเป็นสองชุดย่อยขนาดเท่า ๆ กัน จากนั้นนำข้อมูลทั้งสองชุดย่อยไปเรียงลำดับข้อมูล (ซึ่งก็ใช้วิธีแบบผสานเหมือนกัน) เมื่อได้ข้อมูลสองชุดย่อยที่เรียงลำดับแล้ว จึงนำมาผสานกันเพื่อได้ข้อมูลชุดใหญ่ที่เรียงลำดับทั้งหมด

เราอาจบรรยายอัลกอริทึมด้วยรหัสเทียม (pseudo code) ซึ่งเขียนคล้ายโปรแกรมภาษา-คอมพิวเตอร์ แต่ละเลยกฎเกณฑ์จุกจิกเพื่อความกะทัดรัด เช่น ไม่ต้องประกาศประเภทตัว

---

[1] ในบางสถานการณ์ เราอาจลดข้อกำหนดของอัลกอริทึมลงบ้างคือ ยอมให้อัลกอริทึมหาคำตอบที่ผิดหรือไม่ก็ใช้เวลานานมากจนอาจไม่สิ้นสุด แต่ภายใต้ความน่าจะเป็นที่ต่ำมาก ๆ ซึ่งจะได้ศึกษากันในบทที่ 12

แปร หรืออาจปนด้วยคำบรรยายก็ได้สำหรับคำสั่งง่าย ๆ ที่ไม่ต้องการลงรายละเอียด อาทิ เช่น อาจบรรยายการเรียงลำดับข้อมูลแบบผสานดังนี้

```
01: MergeSort( A[], left, right )
02: {
03:   if ( left < right ) {
04:     m = (left + right) / 2
05:     MergeSort( A, left, m    )
06:     MergeSort( A, m+1 , right )
07:     Merge( A, left, m, right )
08:   }
09: }
```

หรืออาจเขียนแบบนี้

```
01: MergeSort( A[left..right] )
02: {
03:   if ( left < right ) {
04:     m ← (left + right) / 2
05:     MergeSort( A[left.. m]   )
06:     MergeSort( A[m+1..right] )
07:     A[left..right]←Merge(A[left..m],A[m+1..right])
08:   }
09: }
```

คำสั่งทั้งหลายเป็นคำสั่งพื้น ๆ พบได้ในภาษาคอมพิวเตอร์ทั่วไป สำหรับในหนังสือเล่มนี้ จะใช้คำสั่งและตัวปฏิบัติการต่าง ๆ ที่คล้ายคลึงกับที่ใช้ในภาษา C, C++ หรือ Java ซึ่งหวังว่าคงเป็นที่คุ้นเคยของผู้อ่านอยู่แล้ว โดยการส่งผ่านข้อมูลต่าง ๆ ระหว่างฟังก์ชันนั้นกระทำในลักษณะที่มีประสิทธิภาพ ตัวอย่างเช่น รหัสเทียมของ `MergeSort` ที่แสดงให้เห็นข้างบนนี้ การส่งอาเรย์ `A` เมื่อเรียกฟังก์ชันนั้น จะส่งตำแหน่งไป ไม่ได้ต้องเสียเวลามานั่งทำสำเนาทั้งอาเรย์ก่อนส่ง เป็นสิ่งที่เป็นไปได้ในทางปฏิบัติที่เราต้องเข้าใจด้วย ระหว่างการวิเคราะห์อัลกอริทึม

คำว่า “algorithm” มาจากชื่อของนักคณิตศาสตร์ชาวเปอร์เซีย *Abu Ja'far Muhammad ibn Musa al-Khwarizmi* (ดูรูปที่ 2–2 ประกอบ) ผู้เขียนหนังสือเกี่ยวกับเรื่องของจำนวนของชาวฮินดูและอาหรับ “*Algoritmi de numero Indorum*” (ภาษาลาติน) ซึ่งแปลว่า “*Al-Khwarizmi on the Hindu Art of Reckoning*” (ภาษาอังกฤษ) เขาเป็นผู้เริ่มใช้เลขศูนย์ในระบบทศนิยม นอกจากนี้คำว่า “*algebra*” ก็มาจากคำว่า “*al-jabr*” ซึ่งเป็นคำในชื่อหนังสือทางพีชคณิตของเขาอีกเล่มหนึ่งเช่นกัน

รูปที่ 2–2 *Al-Khwarizmi* (780-850)

แบบฝึกหัด

1. จงบรรยายปัญหาเชิงคำนวณต่อไปนี้ ด้วยลักษณะของข้อมูลขาเข้า และผลลัพธ์ที่ต้องการ (ปัญหาใดที่ไม่เคยรู้จักมาก่อน ก็ค้นคว้าตามแหล่งข้อมูลอื่นประกอบ)
    ก) การเดินทางของพนักงานขาย (traveling salesperson)
    ข) ต้นไม้แบบทอดข้ามเล็กสุด (minimum spanning tree)
    ค) เปลือกนูน (convex hull)
    ง) การสามเหลี่ยม (triangulation)
    จ) การจับคู่สตริง (string matching)
    ฉ) เซตอิสระ (independent set)
    ช) ผลรวมของเซตย่อย (sum of subset)
    ซ) การแบ่งส่วนเชิงเส้น (linear partition)
2. จงบอกตัวกำหนดขนาดของข้อมูลขาเข้า ของปัญหาต่าง ๆ ในข้อที่ 1
3. จงบรรยายอัลกอริทึมการเรียงลำดับข้อมูลแบบฟอง (bubble sort) ด้วยรหัสเทียม
4. จงบรรยายอัลกอริทึมการเรียงลำดับข้อมูลแบบฟอง ด้วยข้อความไม่เกิน 3 บรรทัด
5. จงบรรยายขั้นตอนการค้นแบบทวิภาค (binary search) ด้วยข้อความไม่เกิน 3 บรรทัด
6. ค้นคว้าจากแหล่งข้อมูลอื่นเพื่อบรรยายลักษณะของปัญหาดังต่อไปนี้
    ก) Satisfiability
    ข) Maximum 2-Satisfiability
    ค) Bin packing
    ง) Integer programming
7. ค้นคว้าจากแหล่งข้อมูลอื่นเพื่อบรรยายลักษณะของปัญหาดังต่อไปนี้
    ก) Halting problem
    ข) $3x+1$ problem
    ค) Word correspondence problem
    ง) Hilbert's tenth problem