Chapter 1
Algorithm Analysis

ผศ.ดร.สิลดา อินทรโสธรฉันท์

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยขอนแก่น

เวลาการทำงาน

- เมธอดหนึ่งอาจเขียนขั้นตอนการทำงานได้หลายรูปแบบ จึงต้องมีการประเมินประสิทธิภาพทางด้านเวลาการทำงานของเมธอด เพื่อเลือกใช้วิธีที่ดีกว่าในการทำงาน
- ฟังก์ชันที่แสดงความสัมพันธ์ของจำนวนครั้งที่คำสั่งพื้นฐานในเมธอดทำงาน กับปริมาณข้อมูล เนื่องจากจำนวนครั้งที่คำสั่งพื้นฐานในเมธอดทำงานจะสัมพันธ์โดยตรงกับเวลาการทำงานจริง

เวลาการทำงาน

- กำหนดให้ n แทนปริมาณข้อมูลที่จัดเก็บด้วยวิธีที่ได้ออกแบบ
- **การค้นข้อมูลวิธีที่ 1** ต้องใช้งานคำสั่งพื้นฐานเป็น**จำนวน n + 9 ครั้ง** จะได้ว่า หากข้อมูล**เพิ่มขึ้น 16 เท่า** เวลาการทำงานของการค้นข้อมูลก็จะ**เพิ่มขึ้น 16 เท่า**เช่นกัน
- **การค้นข้อมูลวิธีที่ 2** ต้องใช้งานคำสั่งพื้นฐานเป็น**จำนวน $2\log_2 n$ ครั้ง** จะได้ว่า ถ้าปริมาณข้อมูล**เพิ่มขึ้น 16 เท่า** การค้นข้อมูลจะใช้เวลา**เพิ่มขึ้น $\log_2 16 = 4$ เท่า**

เวลาการทำงาน

- ถ้าปริมาณข้อมูลมีมาก วิธีการจัดเก็บแบบที่ 2 ย่อมใช้เวลาการค้นข้อมูลที่น้อยกว่าแบบแรก
- การวิเคราะห์ประสิทธิภาพเชิงเวลา ก็คือการหาฟังก์ชันที่แสดงความสัมพันธ์ของจำนวนครั้งที่คำสั่งพื้นฐานถูกใช้งานกับปริมาณข้อมูล

คำสั่งพื้นฐาน

- คำสั่งพื้นฐาน คือ คำสั่งที่ใช้เวลาการทำงานคงตัวไม่ขึ้นกับปริมาณข้อมูลที่จัดเก็บและก็ไม่ขึ้นกับค่าของตัวถูกดำเนินการหรือพารามิเตอร์
- คำสั่งและตัวดำเนินการต่างๆ ของภาษาการเขียนโปรแกรมส่วนใหญ่เป็นคำสั่งพื้นฐาน เช่น = + - * / % == != < > return เป็นต้น

คำสั่งพื้นฐาน

- บางเมธอดใช้เวลาคงตัวไม่ขึ้นกับค่าของพารามิเตอร์ที่ได้รับหรือปริมาณข้อมูลที่จัดเก็บก็ถือได้ว่า เป็นคำสั่งพื้นฐาน
- เช่น Math.max(a,b) เป็นเมธอดที่คืนตัวที่มีค่ามากกว่าระหว่างตัวแปร a และ b ภายใน max ประกอบด้วยคำสั่งพื้นฐานเป็นจำนวนคงตัว เรียกใช้ max เมื่อใดก็ใช้งานคำสั่งพื้นฐานภายในเป็นจำนวนคงตัว

คำสั่งพื้นฐาน

- ถ้าเป็นคำสั่งที่ใช้เวลาไม่คงตัว ขึ้นกับค่าของพารามิเตอร์หรือปริมาณข้อมูล ก็ถือว่าไม่ใช่คำสั่งพื้นฐาน
- เช่น int[ ] x = new int[n] ซึ่งเป็นการสร้างแถวลำดับ โดยจะมีการตั้งค่าแต่ละช่องเป็น 0 แสดงว่าต้องใช้เวลาใส่ค่า 0 จำนวน n ช่อง จึงไม่เป็นคำสั่งพื้นฐาน
- การเรียกเมธอด Arrays.sort(x) มีหน้าที่เรียงลำดับข้อมูลในแถวลำดับ x จะใช้เวลามากหรือน้อยก็ขึ้นกับปริมาณข้อมูลที่ส่งไปเรียงลำดับ จึงไม่ใช้คำสั่งพื้นฐาน

การนับจำนวนคำสั่งที่ถูกใช้งาน

```
public void remove(Object e) {
    int i = indexOf(e);
    if ( i != -1) {
        elementData[i] = elementData[- - size];
        elementData[size] = null;
    }
}
```

การนับจำนวนคำสั่งที่ถูกใช้งาน

- เมธอด remove
  - บรรทัดที่ 24 เรียก indexOf แล้วตามด้วย = **หนึ่งครั้ง**
  - บรรทัดที่ 25 ทำ i != 1 **หนึ่งครั้ง**
  - ถ้าเป็นจริง ก็ทำบรรทัดที่ 26 - - size **หนึ่งครั้ง** และมี = **หนึ่งครั้ง**
  - บรรทัดที่ 27 มี = **หนึ่งครั้ง**
  - รวมเป็น 5 ครั้ง + เรียก indexOf

การนับจำนวนคำสั่งที่ถูกใช้งาน

```
private int indexOf(Object e){
    for (int i = 0; i < size; i++)
        if(elementData[i].equals(e)) return i ;
    return -1;
}
...
```

การนับจำนวนคำสั่งที่ถูกใช้งาน

- เมธอด indexOf
  - บรรทัดที่ 31 คำสั่ง for ส่วนกำหนดค่า = **หนึ่งครั้ง** ส่วนของเงื่อนไขตรวจสอบ **size + 1 ครั้ง** ส่วนของการ update ค่า ทำงาน **size ครั้ง**
  - บรรทัดที่ 32 เงื่อนไขตรวจสอบ **size ครั้ง**
  - บรรทัดที่ 33 คำสั่ง return **หนึ่งครั้ง**
  - รวม 1 + (size + 1) + 2(size) + 1

การนับจำนวนคำสั่งที่ถูกใช้งาน

- ดังนั้น คำสั่ง remove ทำงานทั้งหมด

  = เมธอด remove + เรียกเมธอด indexOf

  = 5 + 1 + (size + 1) + 2(size) + 1

  = <u>8 + 3(size)</u>

การนับจำนวนคำสั่งที่ถูกใช้งาน

- ถ้า n คือ จำนวนข้อมูล t(n) คือ เวลาการทำงานจริง c(n) คือ จำนวนครั้งที่คำสั่งพื้นฐานทำงาน จะได้ว่า

$$t(n) \leq k\, c(n)$$

k เป็นค่าคงที่ค่าหนึ่งที่แทนเวลาการทำงานของคำสั่งพื้นฐานที่ทำงานนานสุด

การนับจำนวนคำสั่งที่ถูกใช้งาน

```
public void dummy (int n){
        int c = 0;
        for ( int i = 1; i < n ; i++) {
                for (int j = 0; j < i ; j++)
                        c += i + j ;
}
```

การนับจำนวนคำสั่งที่ถูกใช้งาน

```
04          for (int j = 0; j < i ; j++)
05              c += i + j ;
```

- บรรทัดที่ 4 ถึง 5 พบว่า ทำบรรทัดที่ 5 ทั้งสิ้น $\sum_{i=0}^{i-1} 1$
- บรรทัดที่ 5 ทำงานภายใน for ตั้งแต่บรรทัดที่ 3 ถึง 5 จะเป็นจำนวน

$\sum_{i=1}^{n-1}\left(\sum_{i=0}^{i-1} 1\right) = \sum_{i=1}^{n-1}(i) = n(n-1)/2$ ครั้ง

การนับจำนวนคำสั่งที่ถูกใช้งาน

```
03      for ( int i = 1; i < n ; i++) {
04          for (int j = 0; j < i ; j++)
05              c += i + j ;
06  }
```

- บรรทัดที่ 5 ทำงานภายใน for ตั้งแต่บรรทัดที่ 3 ถึง 5 จะเป็นจำนวน $\sum_{i=1}^{n-1}\left(\sum_{i=0}^{i-1} 1\right) = \sum_{i=1}^{n-1}(i) = n(n-1)/2$ ครั้ง

Asymptotic notations

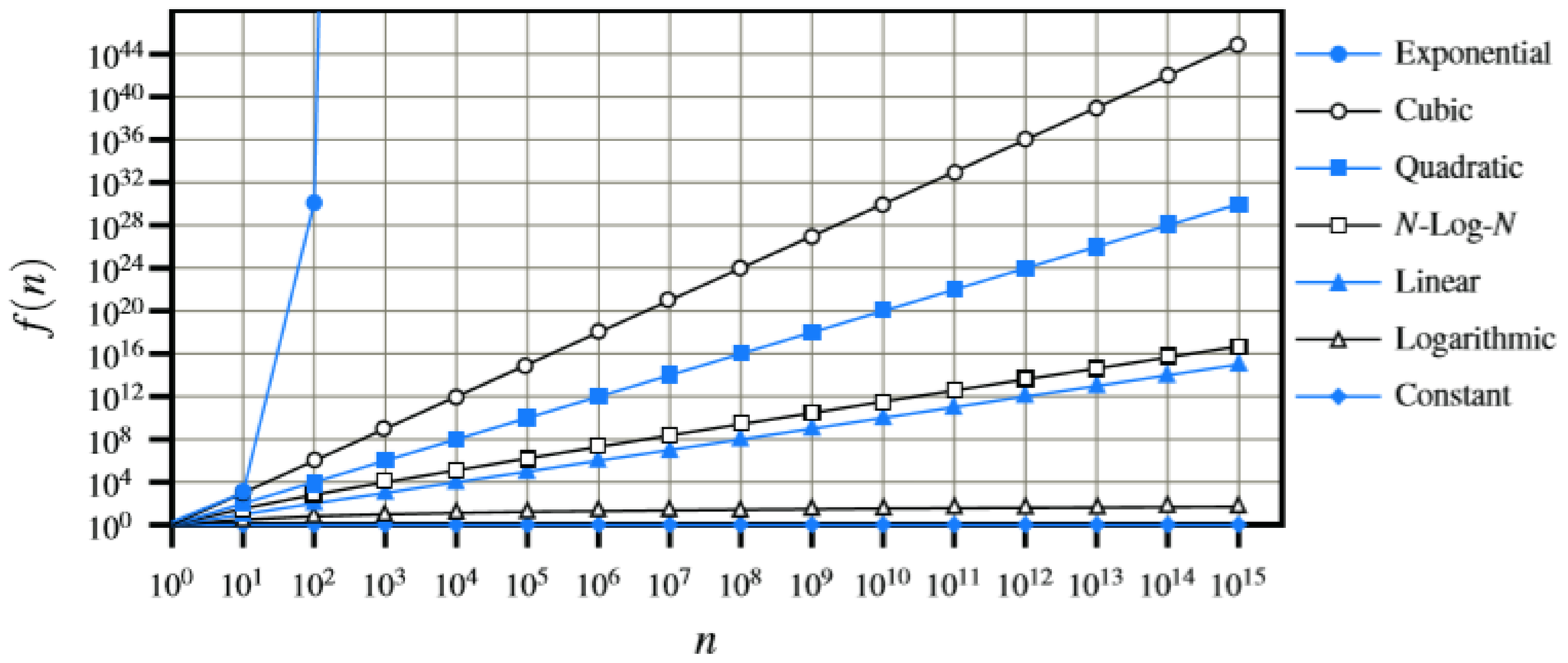
- จากฟังก์ชันของเวลาการทำงานที่หามาได้ของแต่ละวิธีมาเปรียบเทียบกัน สิ่งที่สนใจเปรียบเทียบ คือ อัตราการเติบโตของฟังก์ชัน
- ฟังก์ชันใดโตเร็วกว่ากัน เมื่อข้อมูลมีปริมาณมาก หมายความว่า เมื่อเพิ่มจำนวนข้อมูลในปริมาณเท่ากัน จะส่งผลให้ใช้เวลาการทำงานมากกว่า
- **ฟังก์ชันที่โตช้ากว่าจะมีประสิทธิภาพดีกว่า**

Asymptotic notations

- **Ex1** จงเปรียบเทียบฟังก์ชัน 0.5^n , 1 , log n , n และ 10^n
  - $0.5^n \prec 1 \prec \log n$ เพราะว่า 0.5^n เป็นฟังก์ชันที่มีค่าลดลงเรื่อยๆ เมื่อ n เพิ่มขึ้น ส่วน 1 เป็นค่าคงที่ ในขณะที่ log n เป็นฟังก์ชันเพิ่มเมื่อ n เพิ่ม
  - เทียบ log n กับ n จะได้ว่า $\log n \prec n$
  - และเมื่อเทียบ n กับ 10^n จะได้ว่า $n \prec 10^n$

ดังนั้น $0.5^n \prec 1 \prec \log n \prec n \prec 10^n$

Comparing Growth Rates



constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

การวิเคราะห์เชิงเส้นกำกับ

- วิธีหนึ่งที่ใช้ในการวิเคราะห์ประสิทธิภาพอย่างง่าย คือ วิธีการวิเคราะห์เชิงเส้นกำกับ วิธีการนี้จะช่วยให้เห็นภาพรวมของการเติบโตของฟังก์ชันเวลาการทำงานกับปริมาณข้อมูล
- การวิเคราะห์เวลาการทำงานเชิงเส้นกำกับของเมธอด คือ การหาฟังก์ชันแสดงจำนวนครั้งที่คำสั่งพื้นฐานในเมธอดทำงานในรูปแบบของสัญกรณ์เชิงกำกับ

Big-Oh Notation

- Big-Oh เป็นฟังก์ชันที่แสดงขอบเขตด้านบนของการเติบโตของ f(n)
- ถ้า $f(n) \in O(g(n))$ แสดงว่า การเติบโตของ f(n) ถูกกำหนดขอบเขตด้านบนด้วยลักษณะการเติบโตของ g(n) นั่นคือ มีค่าคงตัวบวก c ที่ $f(n) \le c\, g(n)$
- การอธิบายเวลาในการทำงานด้วยค่า Big-Oh สามารถละในส่วนของ constant factor และ lower-order term ได้ โดยจะอธิบายโดยใช้เพียงส่วนของ main component

Big-Oh Notation

- **ตัวอย่าง จงแสดงให้เห็นว่า** Big-Oh ของฟังก์ชัน $8n + 5$ เป็น $O(n)$
- **วิธีทำ** จากทฤษฎี จะต้องหาค่า constant factor (c) และ lower-order term (n_0) ที่ทำให้ $8n + 5 \le c\,n$ ในทุกๆ $n \ge n_0$
  - เมื่อลองแทนค่า $c = 9$ และ $n_0 = 5$

    จะได้ $8(5) + 5 \le (9)(5)$ อสมการเป็นจริง
  - จึงสรุปได้ว่า Big-Oh ของฟังก์ชัน $8n + 5$ เป็น $O(n)$

Big-Oh Notation

- **ตัวอย่าง จงหาค่า** Big-Oh ของฟังก์ชัน

$$5n^4 + 3n^3 + 2n^2 + 4n + 1$$

- **วิธีทำ**

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)\, n^4 = cn^4$$

สำหรับค่า $c = 15$ เมื่อ $n_0 = 1$

จึงสรุปได้ว่า Big-Oh ของฟังก์ชัน $5n^4 + 3n^3 + 2n^2 + 4n + 1$ เป็น $O(n^4)$

Big-Oh

- **<u>ข้อสังเกต</u>** ถ้า f(n) เป็นฟังก์ชันพหุนาม ที่มีดีกรีสูงสุดเป็น d

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d \; ; \;\; a_d > 0$$

- จะได้ว่า f(n) มีค่า Big-Oh เป็น $O(n^d)$

การวิเคราะห์เชิงเส้นกำกับ

- **จากตัวอย่าง** การวิเคราะห์เมธอด dummy
  - จากตัวอย่างก่อนหน้า คำสั่งพื้นฐานที่ทำงานเป็นจำนวน n(n-1)/2 ครั้ง
  - จะได้ว่า เวลาทำงานจะเป็น $O(n^2)$
- **จากตัวอย่าง** เมธอด remove สั่งงานคำสั่งพื้นฐานเป็นจำนวนไม่เกิน 8 + 3n ครั้ง โดยที่ n คือ จำนวนข้อมูลในคอลเล็กชัน ซึ่งสามารถเขียนแบบเชิงเส้นกำกับได้ว่า เมธอด remove ใช้เวลาการทำงานเป็น $O(n)$

การวิเคราะห์เชิงเส้นกำกับ

- O(1) หรือ Constant

  เป็นอัลกอริทึมที่ดีที่สุด ไม่ว่า input จะมีขนาด 1, 1000, 1M+ หรือเท่าไหร่ก็ตาม ระยะเวลาที่ใช้ในการประมวลผลก็ยังไม่เปลี่ยนแปลง คือ เท่ากับ 1 เสมอ

- O(log n) หรือ logarithmic

  เป็นอัลกอริทึมที่ในการวนลูปแต่ละรอบ จะตัดจำนวนข้อมูลที่ไม่มีโอกาสเกิดขึ้นออกไปทีละครึ่งนึง ที่เราใช้กันจริงๆ บ่อยๆ ก็คือ binary search

การวิเคราะห์เชิงเส้นกำกับ

- O(n) หรือ linear

 ประสิทธิภาพดี โดยระยะเวลาขึ้นอยู่กับจำนวน input ที่ใส่เข้ามา ซึ่งถ้าจำนวน input เพิ่มขึ้นสองเท่า ระยะเวลาที่ใช้ในการประมวลผลก็จะเพิ่มขึ้นสองเท่าด้วย ยกตัวอย่าง เช่น การค้นหา item ใน array ที่ยังไม่ได้ sort ข้อมูล

การวิเคราะห์เชิงเส้นกำกับ

- O(n log n) หรือ linearithmic

ความเร็วปานกลาง อธิบายง่ายๆ จะเป็นการวนลูปสองรอบ ลูปชั้นนอกวนแบบปกติ (n รอบ) แต่ลูปชั้นในวนแบบตัดข้อมูลที่ไม่เกี่ยวข้องออกไปทีละครึ่งด้วย (log n) จึงกลายเป็น O(n log n) ตัวอย่างที่ใช้กันจริง ก็เช่น merge sort

การวิเคราะห์เชิงเส้นกำกับ

- $O(n^2)$ หรือ quadratic

ประสิทธิภาพช้า การเพิ่มขนาด input เข้ามาสองเท่าจะทำให้ระยะเวลาเพิ่มขึ้น **สี่เท่า** เลยทีเดียว ตัวอย่างง่ายๆ เลยก็คือ 2-nested loop เช่น อัลกอริทึมค้นหาว่ามี item ซ้ำกันใน array หรือไม่

การวิเคราะห์เชิงเส้นกำกับ

- $O(n^3)$ หรือ cubic

ช้ามาก การเพิ่มขนาด input เข้ามาสองเท่า จะทำให้ระยะเวลาเพิ่มขึ้น **แปดเท่า**เลยทีเดียว ตัวอย่างง่ายๆ เลยก็คือ 3-nested loop ถ้าไม่จำเป็นจริงๆ อย่าพยายามให้เกิดขึ้นจะดีกว่า ตัวอย่างของ $O(n^3)$ เช่น matrix multiplication หรือ operation บางอย่างใน array 3 มิติ เป็นต้น

Big-Oh

- **แบบฝึกหัด** จงหาค่า Big-Oh ของฟังก์ชันต่อไปนี้
  - $5n^2 + 3n\log n + 2n + 5$
  - $20n^3 + 10n\log n + 5$
  - $3\log n + 2$
  - 2^{n+2}
  - $2n + 100\log n$

เอกสารอ้างอิง

- **Data Structures and Algorithms ,** Michael T. Goodrich.
- **โครงสร้างข้อมูล ฉบับวาจาจาวา,** สมชาย ประสิทธิ์จูตระกูล