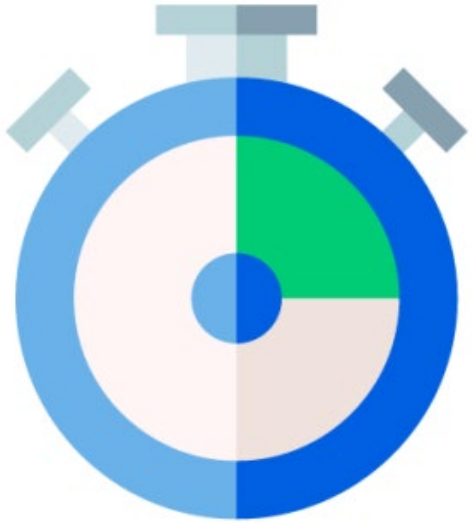
TOI Camp 2 @silpakorn

ความซับซ้อนของอัลกอริทึม

Algorithm Complexity



อ.ดร.ปัญญนัท อ้นพงษ์

ภาควิชาคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร

aonpong_p@su.ac.th

Outline

- **Optimization คืออะไร**
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- การวัดจากพื้นฐานการทำงานจริง
- การวัดความซับซ้อนของอัลกอริทึม
- แนวคิดและการหาค่า Big-O
- Big-O ประเภทต่าง ๆ

Optimization คืออะไร

- ในโปรแกรมที่มีจุดมุ่งหมายเดียวกัน เราจะพยายามทำให้โปรแกรมสร้างคำตอบที่ดีที่สุดและใช้ทรัพยากรคุ้มค่ามากที่สุด
  - อาจทำให้การประมวลผลสามารถทำได้เร็วขึ้น
  - อาจจัดสรรทรัพยากรให้ใช้งานได้อย่างเต็มที่
    - ทรัพยากรในที่นี้รวมไปถึงอุปกรณ์ที่มี กำลังคน และเวลา
- การพยายามทำให้โปรแกรมทำงานได้ผลลัพธ์เหมือน (หรือใกล้เคียง) ผลลัพธ์เดิม แต่ใช้ทรัพยากรคุ้มค่าขึ้น เรียกว่า Optimization

Optimization คืออะไร

- ก็เหมือนกับการที่โปรแกรมบางอย่างทำงานได้ช้าในช่วงแรก แต่เมื่อเวลาผ่านไป ผู้พัฒนามีการอัพเดทแล้วสามารถทำงานได้เร็วขึ้น (ผู้พัฒนาพยายาม Optimize ให้ใช้ทรัพยากรได้คุ้มค่าขึ้น)

Outline

- Optimization คืออะไร
- **ความสำคัญของการวิเคราะห์ความซับซ้อน**
- การวัดจากพื้นฐานการทำงานจริง
- การวัดความซับซ้อนของอัลกอริทึม
- แนวคิดและการหาค่า Big-O
- Big-O ประเภทต่าง ๆ

ความสำคัญของการวิเคราะห์ความซับซ้อน

- กลับมาที่การเขียนโปรแกรมในระดับของเรา
- ปัญหาทางคอมพิวเตอร์เดียวกันมักมีแนวทางการแก้ปัญหาได้หลายวิธี ซึ่งเรารู้จักกันในชื่อ “อัลกอริทึม” (ชุดคำสั่งที่มีลำดับการทำงานชัดเจน)
- ลองจินตนาการว่ามีโจทย์ข้อหนึ่ง และเราแก้ปัญหาออกมาได้ 3 วิธี
  - วิธีที่ 1 ใช้หน่วยความจำน้อยทำงานช้า
  - วิธีที่ 2 ใช้หน่วยความจำเยอะทำงานไวกว่าวิธีแรก
  - วิธีที่ 3 ใช้หน่วยความจำน้อย ทางานไว ความแม่นยำต่ำ
  - เราจะเลือกใช้วิธีไหนดี

ความสำคัญของการวิเคราะห์ความซับซ้อน

- ควรตรวจสอบสิ่งต่อไปนี้
  - ความถูกต้องของอัลกอริทึม (ให้น้ำหนักเป็นอันดับ1)
  - เวลาที่ใช้ในการทำงาน
  - หน่วยความจำที่ใช้ไปจากการแก้ปัญหาด้วยอัลกอริทึมนี้

- ความถูกต้องของการทำงานเป็นสิ่งสำคัญที่สุด ถ้าอัลกอริทึมให้ผลลัพธ์ที่ผิดพลาด อาจส่งผลร้ายแรงต่อระบบโดยรวม ซึ่งจะทำลายความน่าเชื่อถือของโปรแกรมและบริษัทผู้พัฒนา
- ตัวอย่าง การค้นหาสิ่งที่ต้องการ ถ้าอัลกอริทึมทำงานผิดพลาด อย่างการหาไม่พบหรือหาแต่ระบุตำแหน่งผิดพลาดแบบนี้อาจสร้างปัญหาทำให้ไม่มีคนใช้ระบบนี้อีกต่อไป

ความสำคัญของการวิเคราะห์ความซับซ้อน

ตัวอย่างความพังพินาศของชื่อเสียง

ความสำคัญของการวิเคราะห์ความซับซ้อน

- ในการสอบโอลิมปิกระดับประเทศ นอกจากความถูกต้องแล้วยังเน้นเรื่องเวลาการประมวลผลด้วย
- เกรดเดอร์ที่ใช้ในงาน TOI จะมีแนวคิดแตกต่างจากเกรดเดอร์ของเรา (ของเขาเป็นรุ่นใหม่ อ้างจากปี 2021)
  - เกรดเดอร์จะแบ่ง Test case ออกเป็นชุดย่อย ๆ เรียกว่า Subtask
  - แต่ละ Subtask จะทำงานได้ในลิมิตที่แตกต่างกัน
  - ถ้ามี เคสใดใน Subtask ที่ไม่ถูกต้อง ก็จะไม่ได้คะแนนทั้ง Subtask

ความสำคัญของการวิเคราะห์ความซับซ้อน

#	Outcome	Details	Execution time	Memory used
1	Correct	Output is correct	0.001 sec	236 KiB
2	Correct	Output is correct	0.001 sec	236 KiB
3	Not correct	Output isn't correct	0.001 sec	236 KiB
4	Correct	Output is correct	0.001 sec	384 KiB

ความสำคัญของการวิเคราะห์ความซับซ้อน

- จากหน้าต่างที่แสดงในหน้าที่แล้วจะพบว่า นอกจากความถูกต้องของผลลัพธ์แล้ว การแข่งขันยังเน้นเรื่องเวลา และหน่วยความจำที่ใช้อีกด้วย
- ดังนั้น นอกจากจะทำโปรแกรมของเราให้ถูกต้องแม่นยำแล้ว เราจะเริ่มให้ความสำคัญกับเวลาและหน่วยความจำมากขึ้น (เน้นไปทางเวลา)
  - เวลาเกิน -> Time limit exceeds
  - หน่วยความจำเกิน -> Memory limit exceeds

Outline

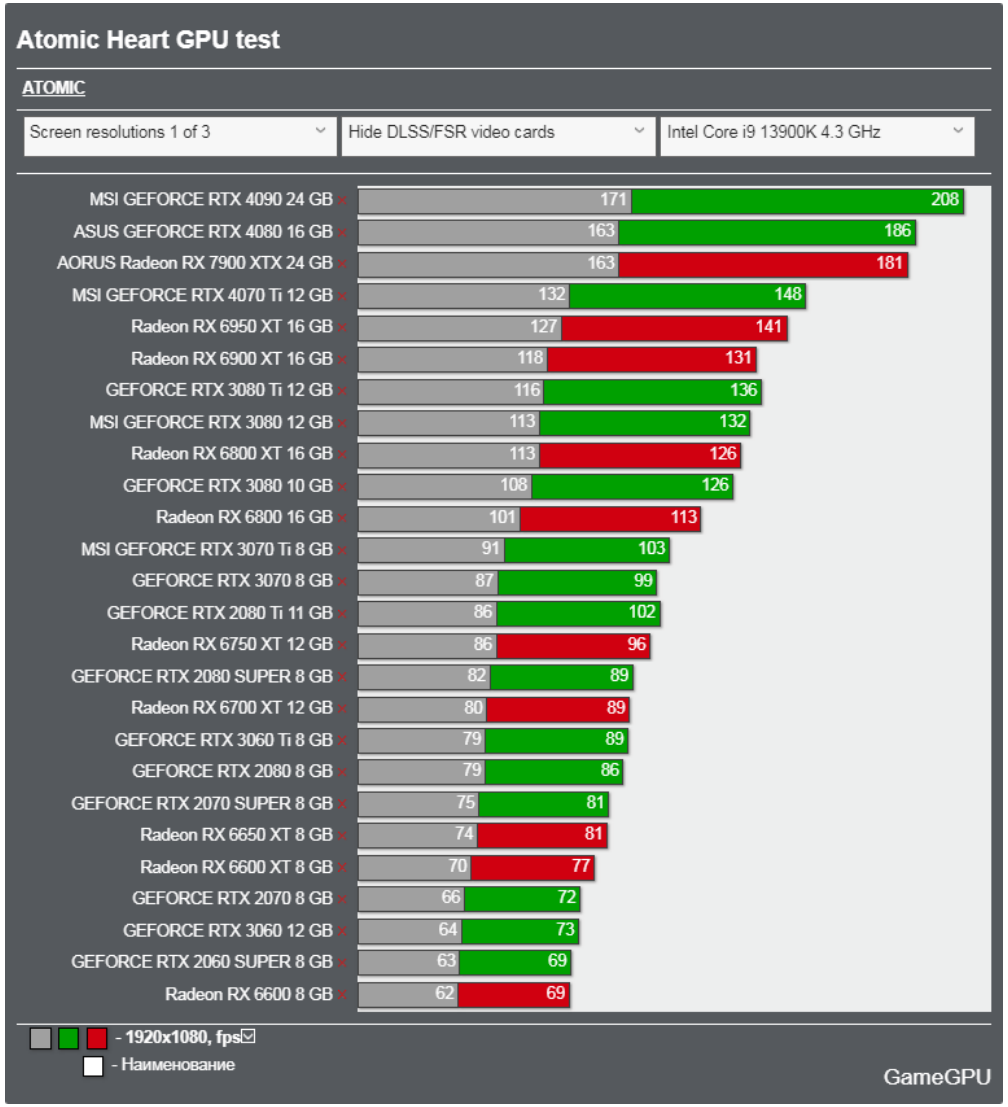
- Optimization คืออะไร
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- **การวัดจากพื้นฐานการทำงานจริง**
- การวัดความซับซ้อนของอัลกอริทึม
- แนวคิดและการหาค่า Big-O
- Big-O ประเภทต่าง ๆ

การวัดจากพื้นฐานการทำงานจริง

วัดเวลาที่ใช้ในการทำงาน

- เป็นวิธีปกติที่ใช้กัน เฟรมเวิร์กไหนมาแรง ภาษาใหม่น่าใช้ไหม เราจะไม่ใช้แค่ความรู้สึก แต่จะจับสิ่งเหล่านี้มาทำงานแก้ปัญหาแล้ววัดเวลาที่ใช้ในการทำงาน โดยการจับเวลาในการทำงาน เราเรียกว่า benchmarking หรือ profiling

การวัดจากพื้นฐานการทำงานจริง



บางคนอาจเป็นนัก oc หรือเป็น gamer หรือแม้กระทั่งคนที่ติดตามเทคโนโลยี น่าจะคุ้นเคยกับคำว่า Benchmarking

แต่ส่วนใหญ่พวกนี้จะวัดประสิทธิภาพของอุปกรณ์โดยใช้ซอฟแวร์เป็นมาตรฐาน ต่างจากงานของเราที่ต้องการวัดประสิทธิภาพของซอฟแวร์บนคอมพิวเตอร์เครื่องเดียวกัน

การวัดจากพื้นฐานการทำงานจริง

Python

```python
import time

# ขนาดข้อมูล
problemSize = 10000000
print("%12s%16s" % ("Problem Size", "Seconds"))

# ทำการวนลูป 5 ครั้งโดยแต่ละครั้งจะเพิ่มขนาด problemSize เป็นสองเท่า
for loopCount in range(5):
    startTime = time.time()
    sum = 0
    for i in range(problemSize):
        sum += 1

    # หาว่าใช้เวลาทำงานนานเท่าไร
    elapsedTime = time.time() - startTime

    print("%12d%16.3f" % (problemSize, elapsedTime))
    problemSize *= 2
```

Code

```
Problem Size         Seconds
    10000000           1.799
    20000000           3.800
    40000000           7.288
    80000000          15.141
   160000000          31.174
```

ด้วยคอมพิวเตอร์เครื่องเดียวกัน สังเกตว่าเมื่อขนาดตัวเลข (Problem size) เพิ่ม เวลาในการประมวลผลก็เพิ่มขึ้นด้วย ทำให้เราประมาณเวลาในการประมวลผลของโปรแกรมนี้ได้

การวัดจากพื้นฐานการทำงานจริง

Code

Problem Size	Seconds
10000000	1.799
20000000	3.800
40000000	7.288
80000000	15.141
160000000	31.174

มีทางที่จะระบุประสิทธิภาพของโปรแกรมโดยไม่ขึ้นกับสภาพแวดล้อมของฮาร์ดแวร์หรือไม่?

- แต่ตัวเลขนี้อธิบายได้เพียงการทำงานบนเครื่องเดียวกันเท่านั้น
- ถึงแม้วิธีการวัดความเร็วของการประมวลผลโดยการจับเวลา ทำให้เห็นภาพรวมของประสิทธิภาพ แต่ก็มีข้อโต้แย้งที่สำคัญคือ ถ้ารันผลด้วยฮาร์ดแวร์ที่ต่างกัน จะได้ความเร็วที่ต่างกัน ดังนั้นประสิทธิภาพที่เราเห็นจากเวลาการทำงานจึงเป็นประสิทธิภาพของโปรแกรมที่รันบนฮาร์ดแวร์และสิ่งแวดล้อมที่จำเพาะ ทั้งระบบปฏิบัติการ และภาษาโปรแกรมที่ใช้

การวัดจากพื้นฐานการทำงานจริง

มีทางที่จะระบุประสิทธิภาพของโปรแกรมโดยไม่ขึ้นกับสภาพแวดล้อมของฮาร์ดแวร์หรือไม่?

ตอบ **มี** คือการวิเคราะห์ความซับซ้อนของอัลกอริทึมที่ใช้นั่นเอง

ในความเป็นจริงที่ว่า ถ้าอัลกอริทึมมีความซับซ้อนสูง ไม่ว่าย้ายไปในเครื่องคอมพิวเตอร์ใดๆก็ตาม มันก็จะทำงานช้ากว่าอัลกอริทึมที่มีความซับซ้อนต่ำกว่านั่นเอง

นั่นคือ เราควรพยายามเขียนโปรแกรมให้มีความซับซ้อนน้อยที่สุดเท่าที่จะสามารถแก้ปัญหาที่ได้มาได้

การวัดจากพื้นฐานการทำงานจริง

เป็นสิ่งที่น่าสนใจอีกอย่างคือความเร็วกับอัตราการใช้หน่วยความจำมักสวนทางกัน คือ อัลกอริทึมที่ทำงานไวมักใช้หน่อยความจำเยอะ การใช้หน่วยความจำน้อยจึงไม่สามารถบอกว่าอัลกอริทึมมีประสิทธิภาพในแง่ของเวลา (แต่ขอแค่หน่วยความจำไม่เกินที่กำหนดก็พอ)

Outline

- Optimization คืออะไร
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- การวัดจากพื้นฐานการทำงานจริง
- **การวัดความซับซ้อนของอัลกอริทึม**
- แนวคิดและการหาค่า Big-O
- Big-O ประเภทต่าง ๆ

การวัดความซับซ้อนของอัลกอริทึม

การวิเคราะห์ความซับซ้อนของอัลกอริทึม เป็นขั้นตอนนำไปสู่การตัดสินใจเลือกอัลกอริทึม ว่าอัลกอริทึมใดมีประสิทธิภาพระดับใดเมื่อเทียบกับจำนวนข้อมูลนำเข้า โดยตัดเรื่องอื่นที่ไม่เกี่ยวข้องออกไป ทั้งการทำงานของแพลตฟอร์ม ภาษาโปรแกรมที่ใช้และระบบปฏิบัติการ

ลองพิจารณาโปรแกรมต่อไปนี้

```
p = 0;
for( int i=2, i<=n, i++){
  p = (p+i)*i;
}
```

- ถ้า n เป็นจำนวนเต็มบวกใดๆ ในแต่ละรอบ มีการบวกและการคูณอย่างละครั้ง รวมเป็น 2 ครั้ง ทำการวนลูปทั้งหมด n-1 รอบ ดังนั้นโปรแกรมนี้มีการทำงานทั้งสิ้น 2*(n-1) ครั้ง
- หรือกล่าวได้ว่า T(n) = 2*(n-1) หรือ T(n) = 2n-2

การวัดความซับซ้อนของอัลกอริทึม

```
p = 0;
for( int i=2, i<=n, i++){
  p = (p+i)*i;
}
```

n	2n-2
1000000	1999998
2000000	3999998
3000000	5999998

- ตัวอย่างข้างต้นอัลกอริทึมทำงาน 2n-2 ครั้ง
- ขนาดของ n เป็นปัจจัยหลักที่ทาให้จำนวนรอบเพิ่มขึ้น ดังนั้น n จึงเป็น dominant term หรือส่วนโดดเด่นที่สุดที่ต้องนำมาพิจารณาในการวิเคราะห์อัลกอริทึม
- การวิเคราะห์อัลกอริทึมจากค่า n มากๆ แบบนี้เรียกว่า asymptotic analysis (การวิเคราะห์เชิงกำกับ)

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

- การวิเคราะห์อัลกอริทึมจากค่า n มากๆ แบบนี้เรียกว่า asymptotic analysis (การวิเคราะห์เชิงกำกับ
- เวลาในการทำงานของ Algorithm จะแตกต่างกันไปตามขนาด Input ที่ต่างกัน
- โดยทั่วไปมักพิจารณาจากความซับซ้อนของระยะเวลาที่แย่ที่สุด (worst case complexity) ซึ่งเป็นระยะเวลาสูงสุดที่จำเป็นสำหรับ Input ที่มีขนาดที่กำหนด
- เรามักใช้ Asymptotic notation ในการแสดง worst case complexity

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

- กรณีของ $T(n) = 2n - 2$ สามารถบอกว่าอัลกอริทึมของเราเป็น $O(n)$
- ถ้าอัลกอริทึมของเราทำงาน $T(n) = 9n^2 + 4$ รอบจะได้ $O(n^2)$
- แน่นอนว่า $O(n)$ ย่อมดีกว่า $O(n^2)$ ลองพิจารณาตารางด้านล่าง

n	n^2
100	10,000
10,000	100,000,000
100,000,000	10,000,000,000,000,000

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

- Asymptotic notation เป็นเครื่องหมายทางคณิตศาสตร์อธิบายถึงลิมิตของพฤติกรรมของ Function ซึ่งถูกคิดค้นขึ้นโดย Paul Bachmann, Edmund Landau Landau และคณะ
- โดยพวกเขาเรียกมันว่า Bachmann Landau notation หรือ asymptotic notation
- Asymptotic notation แบ่งออกได้เป็น Asymptotic function อีก 3 function คือ Big Θ (theta) และ Big Ω (omega) และ Big O notation

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

- Asymptotic notation เป็นเครื่องหมายทางคณิตศาสตร์อธิบายถึงลิมิตของพฤติกรรมของ Function ซึ่งถูกคิดค้นขึ้นโดย Paul Bachmann, Edmund Landau Landau และคณะ
- โดยพวกเขาเรียกมันว่า Bachmann Landau notation หรือ asymptotic notation
- Asymptotic notation แบ่งออกได้เป็น Asymptotic function อีก 3 function คือ Big **Θ** (theta) และ Big **Ω** (omega) และ **Big O notation**

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

- Time Complexity คือความซับซ้อนของการคำนวณที่จะอธิบายระยะเวลาที่ใช้เรียกใช้ Algorithm (เราได้เห็นฟังก์ชัน $T(n)$ กันไปในตอนต้นแล้ว)
- ส่วน Asymptotic notation (ส่วนของ Big O notation) เป็นการหาความซับซ้อนของระยะเวลา (time execution) ที่แย่ที่สุด (worst case complexity)

การวัดความซับซ้อนของอัลกอริทึม: asymptotic analysis

ตัวอย่าง

- สมมติว่ามี Array 26 ช่อง เก็บตัวอักษร A-Z เอาไว้แบบเรียงกันหมด เราต้องการหาตัว F ซึ่งอยู่เป็นอักษรตัวที่ 5 เราก็เขียนโปรแกรมเรียก Array ช่องที่ 5 ออกมาเลย เนื่องจากเรารู้อยู่แล้วว่า A-Z ตัว F คือตัวอักษรตัวที่ 6 แบบนี้เราจะได้ O(1) ซึ่งเป็นวิธีที่ไวมาก
- ใน Array เก็บตัวอักษร A-Z แบบไม่เรียงกัน เราก็จะไม่ทราบว่าตัวอักษรที่ต้องการหานั้นอยู่ที่ช่องไหนของ Array เราจำเป็นต้องใช้ Loop เพื่อหาตัวอักษรที่ต้องการ ถ้าโชคไม่ดีอาจจะพบที่ช่องสุดท้าย เราจะได้ O(n)

Outline

- Optimization คืออะไร
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- การวัดจากพื้นฐานการทำงานจริง
- การวัดความซับซ้อนของอัลกอริทึม
- **แนวคิดและการหาค่า Big-O**
- Big-O ประเภทต่าง ๆ

แนวคิดและการหาค่า Big-O

แนวคิด

- จำให้ฝังใจว่า Assume the worse
  - หลายครั้งที่มีความเป็นไปได้ที่โปรแกรมจะรันแล้วจบในรอบเดียว
  - แต่ในการคำนวณหาค่า Big O เราจะคิดถึงกรณีที่แย่ที่สุดเสมอ
- ตัวอย่างเช่นหากต้องการวนซ้ำใน Array และค้นหาข้อมูลบางอย่าง เราอาจจะเจอตั้งแต่รอบแรก หรือรอบสุดท้าย
- กรณีนี้ให้ถือว่าเราต้องคิด Big O เป็นรอบที่แย่ที่สุด จะได้ O(n) นั่นเอง

แนวคิดและการหาค่า Big-O

ขั้นตอนแบบตามทฤษฎี

1. แยกอัลกอริทึมออกเป็นขั้นตอนย่อย ๆ
2. หา Time complexity ของแต่ละ Operation แล้วนำมารวมกัน (จำนวนงานที่ต้องทำ)
3. นำ T(n) มาหา O(n) โดย
    1. ตัดค่าคงที่ (ตัวเลข) ที่ทำหน้าที่ +, -, *, / ออก เหลือไว้แค่เลขชี้กำลัง
    2. ถ้าเหลือหลายพจน์ เช่น $\mathrm{T(n)} = n^2 + n$ ก็ให้เก็บตัวที่มีเลขชี้กำลังสูงสุดไว้ ในที่นี้คือ n^2
    3. ดังนั้น Big-O notation คือ $O(n^2)$

แนวคิดและการหาค่า Big-O

ตัวอย่าง ขั้นตอนแบบตามทฤษฎี

1. แยกอัลกอริทึมออกเป็นขั้นตอนย่อย ๆ
2. หา Time complexity ของแต่ละ Operation แล้วนำมารวมกัน (จำนวนงานที่ต้องทำ)

```
p = 0;
for( int i=2, i<=n, i++){
  p = (p+i)*i;
}
```

แนวคิดและการหาค่า Big-O

ตัวอย่าง ขั้นตอนแบบตามทฤษฎี

1. แยกอัลกอริทึมออกเป็นขั้นตอนย่อย ๆ
2. หา Time complexity ของแต่ละ Operation แล้วนำมารวมกัน (จำนวนงานที่ต้องทำ) เช่นจากตัวอย่างจะได้ว่า $T(n) = 2 * (n - 1)$
3. นำ T(n) มาหา O(n) โดย
   1. ตัดค่าคงที่ (ตัวเลข) ที่ทำหน้าที่ +, -, *, / ออก เหลือไว้แค่เลขชี้กำลัง ดังนั้นข้อนี้ จะเหลือแค่ n ตัวเดียว
   2. ไม่ต้องคิดต่อเพราะเหลือพจน์เดียวแล้ว ดังนั้น Big-O notation ของข้อนี้คือ $O(n)$

```
p = 0;
for( int i=2, i<=n, i++){
  p = (p+i)*i;
}
```

แนวคิดและการหาค่า Big-O

ขั้นตอนแบบตามทฤษฎี

1. แยกอัลกอริทึมออกเป็นขั้นตอนย่อย ๆ
2. หา Big-O ของแต่ละ Operation เช่นจากตัวอย่าง จะได้ว่า $T(n) = 1 + n(n+1)$
3. ลองทำต่อเอง...

```
void function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            cout << "*";

        }
      cout << endl;
    }
}
```

แนวคิดและการหาค่า Big-O

ขั้นตอนแบบในทางปฏิบัติ

1. ไม่ต้องคิดละเอียด สุดท้ายก็ตัดเลขคงที่ออกเกือบหมดอยู่ดี ดังนั้นเราจะมุ่งเป้าไปยังเลขชี้กำลังเป็นหลัก
2. **ถ้ามี Nested Loop (ซ้อนกัน) ให้นำ Big-O ของแต่ละ Loop มาคูณกัน (มองด้วยตาเลยว่าทั้ง loop นั้นทำงานกี่รอบ ไม่จำเป็นต้องใส่ใจงานพื้นฐานใน loop)**
3. **ถ้า Loop อยู่ระดับเดียวกัน ให้นำ Big-O มาบวกกัน**
4. ตัวไหนเป็นตัวเลขที่ +, -, *, / ตัดออก (ไม่ตัดตัวชี้กำลัง)
5. พอทำการคูณและบวกเสร็จแล้วจะพบว่ามีหลายพจน์ เราก็จะเลือกตัวที่เลขชี้กำลังแย่ที่สุดก็พอ

*ถ้ามีการเรียกฟังก์ชัน ต้องคำนวณ Big-O ของฟังก์ชันด้วย

แนวคิดและการหาค่า Big-O

ขั้นตอนแบบในทางปฏิบัติ

1. ถ้ามี Nested Loop (ซ้อนกัน) ให้นำ Big-O ของแต่ละ Loop มาคูณกัน (มองด้วยตาเลยว่าทั้ง loop นั้นทำงานกี่รอบ) ข้อนี้ทำงาน n-1 รอบ
2. ตัดตัวเลขคงที่ เหลือ n
3. ตอบ O(n)

```
p = 0;
for( int i=2, i<=n, i++){
  p = (p+i)*i;
}
```

แนวคิดและการหาค่า Big-O

ขั้นตอนแบบในทางปฏิบัติ

1. ถ้ามี Nested Loop (ซ้อนกัน) ให้นำ Big-O ของแต่ละ Loop มาคูณกัน (มองด้วยตาเลยว่าทั้ง loop นั้นทำงานกี่รอบ) ข้อนี้ทำงาน n รอบ ซ้อนกับ n รอบ จึงได้ว่า $n * n = n^2$
2. ตัดค่าคงตัว ซึ่งไม่มี
3. ตอบได้เลยว่า $O(n^2)$

```
void function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            cout << "*";

        }
      cout << endl;
    }
}
```

แนวคิดและการหาค่า Big-O

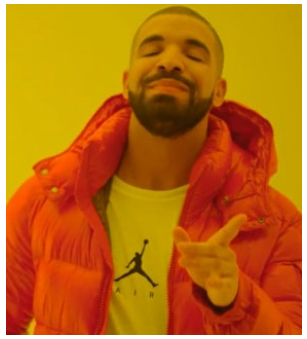
ลองทำดู

```
void function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            cout << "*";
            break;
        }
      cout << endl;
    }
}
```

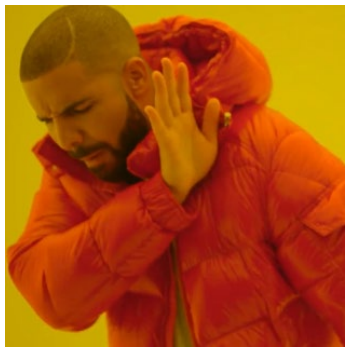
Outline

- Optimization คืออะไร
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- การวัดจากพื้นฐานการทำงานจริง
- การวัดความซับซ้อนของอัลกอริทึม
- แนวคิดและการหาค่า Big-O
- **Big-O ประเภทต่าง ๆ**

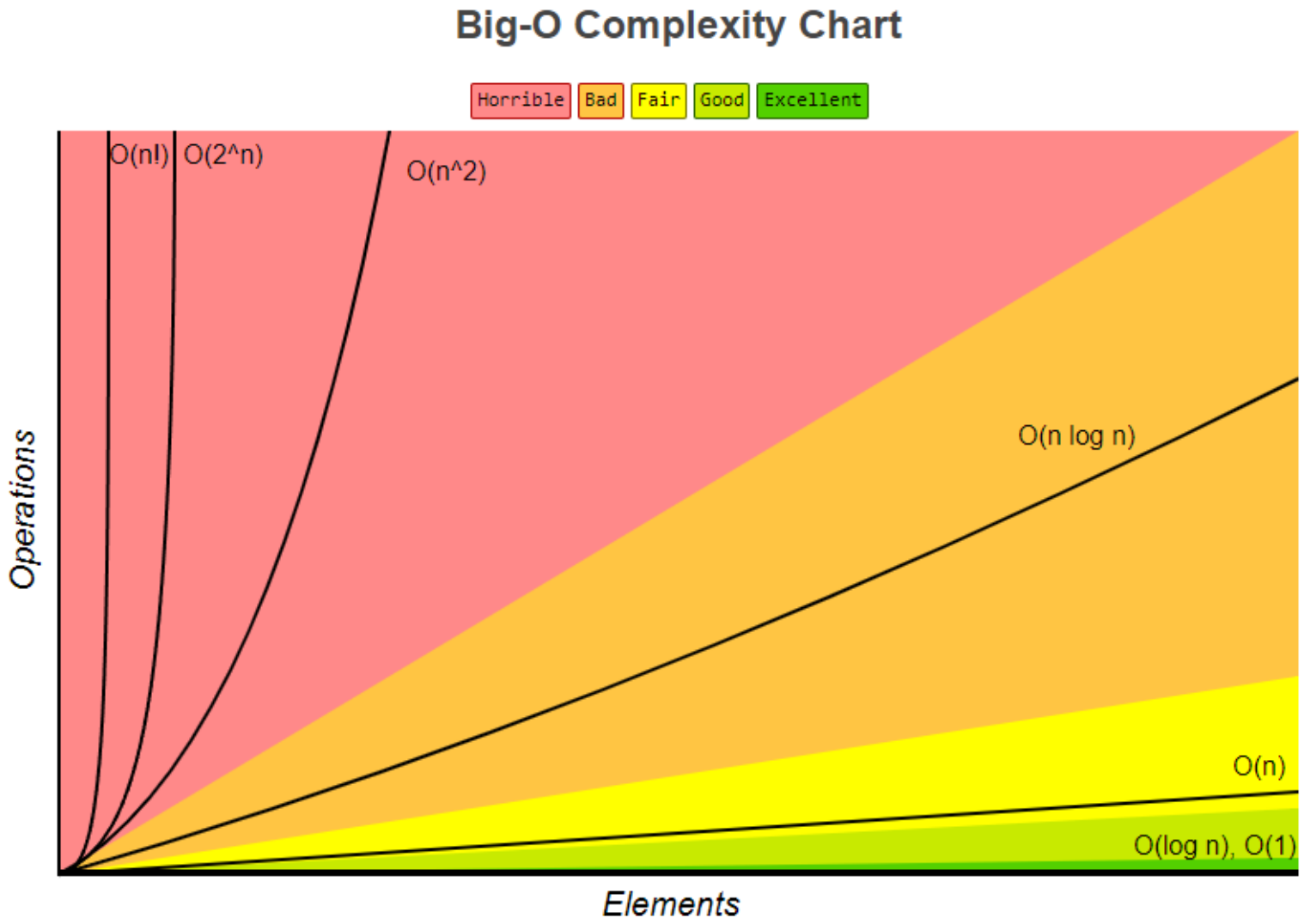
Big-O ประเภทต่าง ๆ



n	n^2	N = 16	N = 1024
O(1)	Constant	1	1
O(lg n)	Lg	4	10
O(n)	Linear	16	1024
O(n lg n)	N lg n	64	10,240
$O(n^2)$	Quadratic	256	1,048,576
$O(n^3)$	Cubic	4096	1,073,741
$O(n^m)$	Polynomial	>>>	
$O(2^n)$	exponential	65536	2^1024
O(n!)	Factorial	16!	1024!



Big-O ประเภทต่าง ๆ



Big-O ประเภทต่าง ๆ

1. Constant: $O(1)$

O(1) เป็นสุดยอดความปรารถนา เพราะระยะเวลาในการประมวลผลไม่ขึ้นกับปริมาณข้อมูล ไม่ว่าข้อมูลจะใหญ่หรือเล็ก ระยะเวลาประมวลผลจะคงที่

```
NewType array[];
cin >> input;
print(array[hash(input)]
```

จากคำสั่งซ้ายมือ มีการรับค่าข้อมูลเข้าแล้วเอาไปเข้า hash function ได้ค่าออกมา สามารถนำไปค้นในอาเรย์ได้เลย แบบนี้ถือว่าขนาดข้อมูลใน array ไม่มีผลกับการค้นหา

Big-O ประเภทต่าง ๆ

1. Constant: $O(1)$

O(1) เป็นสุดยอดความปรารถนา เพราะระยะเวลาในการประมวลผลไม่ขึ้นกับปริมาณข้อมูล ไม่ว่าข้อมูลจะใหญ่หรือเล็ก ระยะเวลาประมวลผลจะคงที่

```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

จากฟังก์ชันซ้ายมือ เป็นการรับค่าอาเรย์เข้าไปในฟังก์ชันและพิมพ์ค่าแรกออกมา แม้ว่าอาเรย์จะใหญ่เป็นพันล้านตัว แต่งานที่มันทำก็มีแค่การพิมพ์ค่าแรกออกมาทุกครั้ง ดังนั้นเวลาประมวลผลจึงคงที่

Big-O ประเภทต่าง ๆ

2. Logarithmic: $O(\log n)$

ตัวอย่างเช่น Binary Search ซึ่งจะเป็นการค้นหาแบบค่อย ๆ แบ่งครึ่งไปเรื่อย ๆ ข้อดีของมันคือไวกว่า O(n) ในหัวข้อถัดไปมาก แต่ข้อเสียคือ ข้อมูลที่จะทำ Binary Search ต้องถูก Sort ก่อน (และการ Sort ก็มีความซับซ้อนพอตัว)

จากอาเรย์ซ้ายมือ ถ้าอยากรู้ว่ามีค่า 32 ในอาเรย์หรือไม่ โดยใช้ Binary search จะต้อง...

Big-O ประเภทต่าง ๆ

3. Linear: $O(n)$

Algorithm ที่ใช้ระยะเวลาในการค้นหาตามปริมาณข้อมูลที่มี ที่แย่ที่สุดจะไม่เกินปริมาณของตัวเอง เช่นข้อมูลมี 26 ตัว การค้นหาก็จะแย่ที่สุดคือ 26 ครั้ง

```
function findCharacterH(){
  String[] name = {"Helen","Adam","John"};
  for(int i = 0; i < name.length; i++){
    if(name[i] == "John"){
      print(name[i]);
      print(i);
    }
  }
}
```

จากฟังก์ชันซ้ายมือ มีจำนวนการทำงานวนซ้ำมากที่สุดที่จะเป็นไปได้ เท่ากับความยาวชื่อทั้งหมดที่รับเข้าไปในระบบ

Big-O ประเภทต่าง ๆ

4. Linearithmic: $O(n \log n)$

พบใน Algorithm ที่จะมีการใช้ซ้อน Loop โดยปกติถ้าซ้อน Loop ธรรมดา ค่าที่ได้มักจะเป็น $O(n^2)$ เนื่องจากจะเป็นการวนให้ครบให้หมด (worst case complexity) แต่ในเมื่อเป็น $O(n \log n)$ ภายใน Loop ที่สองที่ซ้อนอยู่นั้น มีจำนวนครั้งการทำงานเป็น log n จึงทำให้วิธีนี้ไม่ใช้พลังงานอย่าง $O(n^2)$ เราจะพบ $O(n \log n)$ ได้จาก Merge Sort, Heap Sort หรือ Quick Sort

Big-O ประเภทต่าง ๆ

5. Quadratic: $O(n^2)$

Algorithm ที่ถือเป็น 2-nested loop ก็คือการซ้อน Loop แบบทำงานเต็มจำนวนนั่นเอง เราจะพบ $O(n^2)$ ได้จาก Bubble Sort, Insertion Sort, Selection Sort

```
for (int i = 1; i <= n; i++) {
  for(int j = 1; j <= n; j++) {
    println("I: " + i + " J: " + j);
  }
}
```

Big-O ประเภทต่าง ๆ

6. Qubic: $O(n^3)$

จะเป็น Algorithm ที่ใช้ 3-nested loop คือการ Loop ซ้อน Loop แล้วก็ซ้อน Loop อีกที โดย loop แต่ละตัวทำงานเต็มจำนวน ตัวนี้มีความช้ามาก มากกว่า $O(n^2)$ ยิ่งมีข้อมูลเยอะ เวลาที่ใช้ประมวลผลยิ่งทวีคูณ (พยายามหลีกเลี่ยงถ้าอัลกอริทึมของนักเรียนเดินทางมาถึงจุดนี้)

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
    for (int k = 1; k <= n; k++) {
      println("I: " + i + " J: " + j + " K: " + k);
    }
  }
}
```

Big-O ประเภทต่าง ๆ

7. Exponential: $O(2^n)$

เป็น Algorithm ที่ควรหลีกเลี่ยงเป็นอย่างมาก แค่ n = 4 ก็ต้องใช้เวลาถึง 16 รอบแล้ว ยิ่งถ้าข้อมูลสูงขึ้นเป็น n = 16 ก็จำเป็นต้องทำงานถึง 256 รอบ ไม่ค่อยมีโอกาสได้ใช้หรือพบเห็นในชีวิตจริงเท่าไหร่นัก

```
for (int i = 1; i <= Math.pow(2, n); i++){
    print("round: " + i);
}
```

```
function fib (n) {
  if (n <= 1) {
    return n
  }
  else {
    return fib(n - 2) + fib(n - 1)
  }
}
```

Big-O ประเภทต่าง ๆ

8. Factorial: $O(n!)$

Classic case ที่เป็นตัวอย่างคือปัญหา Traveling Salsman Problem แค่ n มีค่าเป็น 8 ก็ใช้ไป 40,320 รอบแล้ว ตัวอย่างนอกจากนั้น ถ้าไม่พยายามเขียนขึ้นโดยเฉพาะ ก็หายากมาก ๆ

```
void nFacRuntimeFunc(int n) {
  for(int i=0; i<n; i++) {
    nFacRuntimeFunc(n-1);
  }
}
```

Conclusion

- Optimization คืออะไร
- ความสำคัญของการวิเคราะห์ความซับซ้อน
- การวัดจากพื้นฐานการทำงานจริง
- การวัดความซับซ้อนของอัลกอริทึม
- แนวคิดและการหาค่า Big-O
- Big-O ประเภทต่าง ๆ