การวิเคราะห์ประสิทธิภาพของอัลกอริทึม

Analysis of Algorithm Efficiency

อ.ลือพล พิพานเมฆาภรณ์

luepol.p@sci.kmutnb.ac.th

Content

- **การวิเคราะห์ความซับซ้อนเชิงเวลา (time complexity analysis)**
  - ขนาดของอินพุต (input's size)
- **เทคนิคการวิเคราะห์ความซับซ้อนเชิงเวลา**
  - Elementary operation counting
  - Basic operation counting
  - Worst-case, Best-case & Average-case analysis
- **ลำดับการเติบโต (Order of Growth) และการเปรียบเทียบ**
  - ทฤษฎีกฎ L'Hôpital
- **สัญกรเชิงเส้นกำกับ (Asymptotic Notations) และการพิสูจน์นิยาม**
  - สัญกรบิ๊กโอ (Big-Oh)
  - สัญกรโอเมก้า (Omega)
  - สัญกรเทต้า (Theta)

Time ccomplexity analysis

- อัลกอริทึมที่ดีไม่ใช่แค่ทำงานได้ถูกต้องเท่านั้น แต่ต้องทำงานได้อย่างมีประสิทธิภาพด้วย ดังนั้นการประเมินประสิทธิภาพการทำงานของอัลกอริทึม (algorithm efficiency) ถือเป็นงานที่มีความสำคัญสำหรับโปรแกรมเมอร์

- ในการวิเคราะห์ประสิทธิภาพของอัลกอริทึม มักจะวัดในเชิงเวลาทำงานของอัลกอริทึมตั้งแต่เริ่มต้นจนจบการทำงาน
  - การวิเคราะห์เวลาโดยตรง (Empirical analysis)
  - การวิเคราะห์ทางทฤษฎี (Theoretical analysis)

- อย่างไรก็ตามการวัดเวลาทำงานของอัลกอริทึมโดยตรง อาจไม่เหมาะสมเนื่องจากมักจะขึ้นอยู่กับหลายปัจจัย เช่น
  - ความเร็วการทำงานของซีพียู
  - คุณภาพของตัวแปลภาษา (Complier/ Interpreter)
  - ความยาวของ machine codes

Efficiency as a function of input size

- เมื่อขนาดข้อมูลอินพุตมากขึ้น (input size) อัลกอริทึมส่วนใหญ่จะใช้เวลาทำงานมากขึ้น

- ในบางปัญหาขนาดของอินพุตไม่ได้ขึ้นอยู่กับจำนวนข้อมูลที่ถูกป้อนเข้ามา แต่อาจขึ้นอยู่กับค่าข้อมูลแทน
  - ปัญหาการทดสอบจำนวนเฉพาะ (testing prime number)

- เราสามารถวัดและเปรียบเทียบประสิทธิภาพของอัลกอริทึมได้ หากทราบขนาดของอินพุต โดยปกติจะแทนด้วยตัวแปร **n**

Running time estimation

- เทคนิคที่ใช้ในการวัดประสิทธิภาพด้านเวลาของอัลกอริทึม ได้แก่
  - **การนับโอเปอเรชันทั้งหมด (Elementary operation counting)**
    - นับจำนวนครั้งของทุกบรรทัดคำสั่ง (operation) ของอัลกอริทึม
  - **การนับโอเปอเรชันพื้นฐาน (Basic operation counting)**
    - นับเฉพาะบรรทัดที่ถูก execute มากที่สุด สัมพันธ์ขนาดอินพุต n
  - **การวิเคราะห์แบบ worst-case, best-case, และ average case**
    - นับจำนวนครั้งของโอเปอเรชันพื้นฐาน สอดคล้องกับรูปแบบของข้อมูลอินพุต

Example: Elementary Operation Counting

```
1. ALGORITHM power (x, n)
2.    product <- 1
3.       for i <- 1 to n do
4.          product <- product*x
5.       endfor
6.         return product
7. END ALGORITHM
```

Operation	Time	Repetitions
1	t1	1
2	t2	1
3	t3	n+1
4	t4	n
5	t5	n
6	t6	1

$T(n) = t1+t2+t3(n+1)+t4n+t5n+t6$

$T(n) = (t3+t4+t5)n + t1+t2+t3+t6$

Example: Elementary Operation Counting

```
1. ALGORITHM mystery (x, n)
2. S = 0
3.     for i = 1 to n do
4.         for j = 1 to n do
5.             S = S + 1
6.         endfor
7.     endfor
8. END ALGORITHM
```

Operation	Time	Repetitions
1	t1	1
2	t2	1
3	t3	n+1
4	t4	n(n+1)
5	t5	n*n
6	t6	n*n
7	t7	n

$T(n) = t1 + t2 + t3(n+1) + t4(n^2+n) + t5n^2 + t6n^2 + t7n$

$T(n) = (t4+t5+t6)n^2 + (t3+t4+t7)n + t1+t2$

Exercise: Elementary Operation Counting

```
1. sum(n)
2.    S = 0
3.    i = 1
4.   while i <= n do
5.      S = S + 1
6.      i = i + 1
7.   endwhile
8.  Return S
```

t(n) =

Operation	Time	Repetitions
1	t1	
2	t2	
3	t3	
4	t4	
5	t5	
6	t6	
7	t7	

Exercise: Elementary Operation Counting

```
1. sum(n)
2.    S = 0
3.    i = 1
4.   while i <= n do
5.      S = S + 1
6.      i = i + 1
7.   endwhile
8.  Return S
```

Operation	Time	Repetitions
1	t1	1
2	t2	1
3	t3	1
4	t4	n+1
5	t5	n
6	t6	n
7	t7	n

$t(n) = t1+t2+t3+ t4(n+1) + t5n + t6n + t7n$

$t(n) = (t4+t5+t6+t7)n + t1+t2+t3+t4$

Exercise: Elementary Operation Counting

```
1.  product_matrix(a[1..m,1..n], b[1..n][1..p])
2.  for i=1 to m do
3.    for j =1 to p do
4.       c[i,j] = 0;
5.       for k = 1 to n do
6.          c[i,j] = c[i,j] + a[i,k]*b[k,j]
7.       end for
8.    end for
9.  end for
10. Return c[1..m,1..p]
```

Exercise: Elementary Operation Counting

Opr.	Time	Repetitions
1	t1	1
2	t2	m+1
3	t3	m.(p+1)
4	t4	m.p
5	t5	m.p.(n+1)
6	t6	m.p.n
7	t7	m.p.n
8	t8	m.p
9	t9	m

```
1.   product_matrix(a[1..m,1..n], b[1..n][1..p])
2.   for i=1 to m do
3.     for j =1 to p do
4.        c[i,j] = 0;
5.        for k = 1 to n do
6.           c[i,j] = c[i,j] + a[i,k]*b[k,j]
7.        end for
8.     end for
9.   end for
10.  return c[1..m,1..p]
```

t(m, p, n) =

Basic operation counting Technique

- การนับทุกบรรทัดคำสั่ง ทำได้ยากและเสียเวลา
- การประมาณเวลา โดยนับเฉพาะ Basic operation ทำได้ง่าย และนิยมมากกว่า
- Basic operation คือบรรทัดคำสั่งซึ่งถูก execute มากที่สุด สัมพันธ์กับขนาดอินพุต

$$T(n) \approx C_o * C(n)$$

Co = เวลาทำงานจริงของ basic operation

C(n) = คือจำนวนครั้งของการทำงาน basic operation ขึ้นอยู่กับขนาดของอินพุต n

Basic operation counting

สมมติว่าฟังก์ชันเวลาของอัลกอริทึมหนึ่ง คือ

$$T(n) = 60n + 5$$

สำหรับอินพุต n ที่มีขนาดใหญ่มาก T(n) จะขึ้นอยู่กับเทอม 60n เท่านั้น

n	T(n) = 60n + 5	T(n) ~ 60n	Error
10	605	600	0.826
100	6,005	6,000	0.083
1,000	60,005	60,000	0.008
10,000	600,005	600,000	0.001
100,000	6,000,005	6,000,000	0.000

Example: Basic operation counting

```
1. ALGORITHM power (x, n)                    // t1
2.   product <- 1                            // t2
3.     for i <- 1 to n do                    // t3
4.        product <- product *x              // t4
5.     endfor                                // t5
6.          return product                   // t6
7. END ALGORITHM                             // t7
```

Basic Operation

T(n) ~ t4n

Example: Basic operation counting

```
1. ALGORITHM mystery (x, n)    // t1
2. S = 0                       // t2
3.    for i = 1 to n do        // t3
4.        for j=1 to n do      // t4
5.           S = S + 1         // t5
6.       endfor                // t6
7.    endfor                   // t7
8. END ALGORITHM               // t8
```

Basic Operation

$T(n) \sim t5n^2$

Example: Basic operation counting

```
1. sum(n)
2.    S = 0
3.    i = 1
4.   while i <= n do
5.     S = S + 1
6.     i = i + 1
7.   endwhile
8.  Return S
```

Basic operation สามารถจะเป็นบรรทัดที่ 4, 5, หรือ 6 ก็ได้ เนื่องจากมีจำนวนรอบทำงานไม่แตกต่างกันมากนัก

$$T(n) = t5*n$$

Basic operation counting

- ในทางปฏิบัติ ถึงแม้ว่าเราไม่ทราบค่าที่แท้จริงของ Co แต่เราอาจหาเวลาการทำงานที่เพิ่มขึ้นหรือลดลง เมื่อขนาดของอินพุตเปลี่ยนแปลงได้ เช่น
  - สมมติว่าอัลกอริทึมหนึ่ง $C(n) = C_o * n^2$ หากจำนวนอินพุตเพิ่มขึ้นเป็น 2 เท่าเวลาทำงานของอัลกอริทึมนี้จะเป็นเท่าไร

$$\frac{T(2n)}{T(n)} = \frac{C_o * (2n)^2}{C_o * (n)^2} = \frac{4n^2}{n^2} = 4$$

Analysis of Sequential Search

```
1.  Sequential_Search (A[0..n-1], K)
2.       i := 0
3.        while A[i] ≠ k do
4.           i := i + 1
5.        if i < n then return i
6.        else   return -1
```

What is the time complexity of the algorithm?

Best-case, average-case, worst-case Analysis

- นอกจากขนาดอินพุต n แล้ว ในบางอัลกอริทึมเวลาในการทำงานจะขึ้นอยู่ลักษณะของข้อมูลด้วย
- เพื่อให้ง่ายในการวิเคราะห์เวลา เราแบ่งการวิเคราะห์ออกเป็น 3 กรณี ได้แก่
  - กรณีเลวร้ายสุด (Worst Case) W(n) คือเวลาการทำงานที่มากที่สุดที่เป็นไปได้ สำหรับข้อมูลอินพุต n
  - กรณีดีที่สุด (Best Case) B(n) คือเวลาการทำงานที่น้อยที่สุดที่เป็นไปได้ สำหรับข้อมูลอินพุต n
  - กรณีเฉลี่ย (Average Case) A(n) คือเวลาการทำงานเฉลี่ย สำหรับข้อมูลอินพุต n

Best-case, average-case, worst-case

- *Problem:* กำหนดให้มีชุดข้อมูลใน array จำนวน n ชุด ให้หาชุดข้อมูลที่มีค่าเท่ากับ K
- *Algorithm:* ทำการตรวจสอบข้อมูลทีละตัวไปเรื่อยๆ ว่ามีตัวใดมีค่าเท่ากับ K จนกว่าจะพบ (*successful search*) หรือจะหมดข้อมูลที่จะทำการค้นหา (*unsuccessful search*)
- Worst case : $t_w(n) = n$
- Best case : $t_b(n) = 1$
- Average case : $t_a(n) = ?$

Analysis of average-case of sequential search

- เพื่อที่จะวิเคราะห์อัลกอริทึมในกรณีเฉลี่ย เราจำเป็นจะต้องหาความน่าจะเป็นของอินพุตทุกรูปแบบเป็นไปได้ทั้งหมด จากนั้นทำการหาค่าเฉลี่ยเวลาของการทำงาน

- ตัวอย่างเช่น อัลกอริทึม Sequential Search มีอินพุตทั้งหมด 2 รูปแบบคืออินพุตไม่ปรากฏข้อมูล k และไม่ปรากฏข้อมูล k
  - เราทราบว่าหากไม่พบข้อมูล k แสดงว่าเราต้องเปรียบเทียบคีย์ต้องแต่สมาชิกตัวแรกยันตัวสุดท้าย ซึ่งใช้เวลา n เสมอ
  - หากเราพบข้อมูล k ในอาร์เรย์ ก็ขึ้นอยู่ตำแหน่งที่พบ เช่น ถ้าพบ k ในตำแหน่งแรก เวลาทำงานจะเป็น 1 แต่ถ้าพบในตำแหน่ง n เวลาทำงานจะเป็น n
  - กำหนดให้ p คือความน่าจะเป็นที่ค่า k ปรากฏในข้อมูลอินพุต ดังนั้น (1-p) จะหมายถึงความน่าจะเป็นที่ไม่พบค่า k ในอินพุต
  - ดังนั้น p/n คือความน่าจะเป็นเฉลี่ยที่จะพบ k ในแต่ละตำแหน่ง สำหรับข้อมูลอินพุตขนาด n

Average case analysis of sequential search

ให้ j เป็นลำดับของอินพุตใน sequential search โดยที่ j = 1... m

ที่ j = 1 t1(n) = 1
ที่ j = 2 t2(n) = 2
ที่ j = 3 t3(n) = 3

.....

ที่ j = m tm(n) = m

$$T_{avg}(n) = \frac{1+2+3+4....+m}{m}$$

แต่เนื่องจาก m = n

$$T_{avg}(n) = \frac{\frac{n(n+1)}{2}}{n}$$

$$T_{avg}(n) = \frac{n(n+1)}{2n} = \frac{(n+1)}{2}$$

Exercise

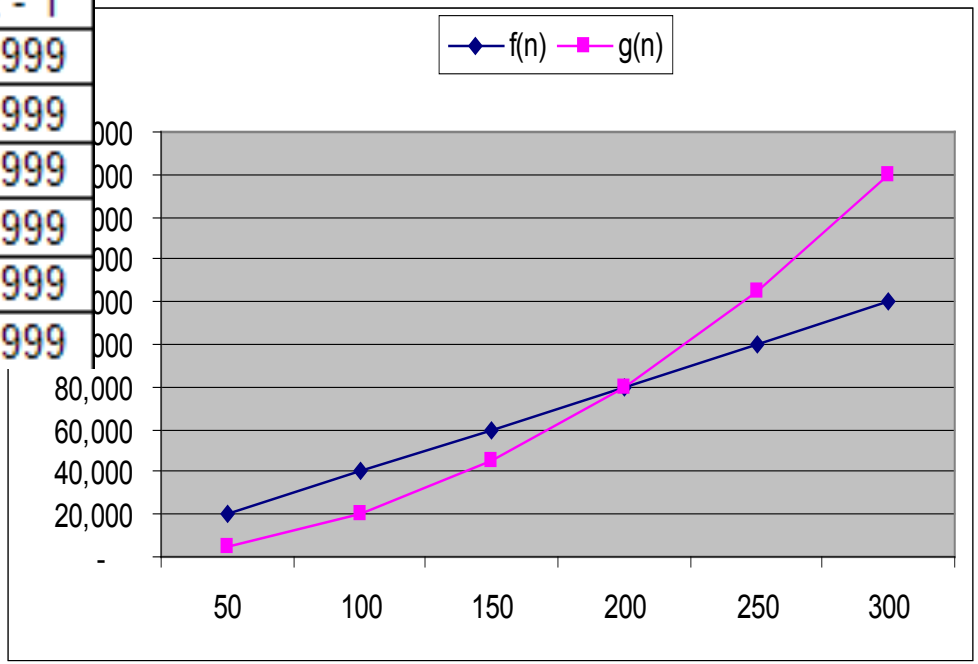
- หา worst-case, best-case, และ average case ของอัลกอริทึมต่อไปนี้

```
1.  minimum (A[1..n])
2.       m := A[1]
3.       for i = 2 to n do
4.           if m > A[i] then
5.               m = A[i]
6.           endif
7.       endfor
8.  Return m
```

Order of growth

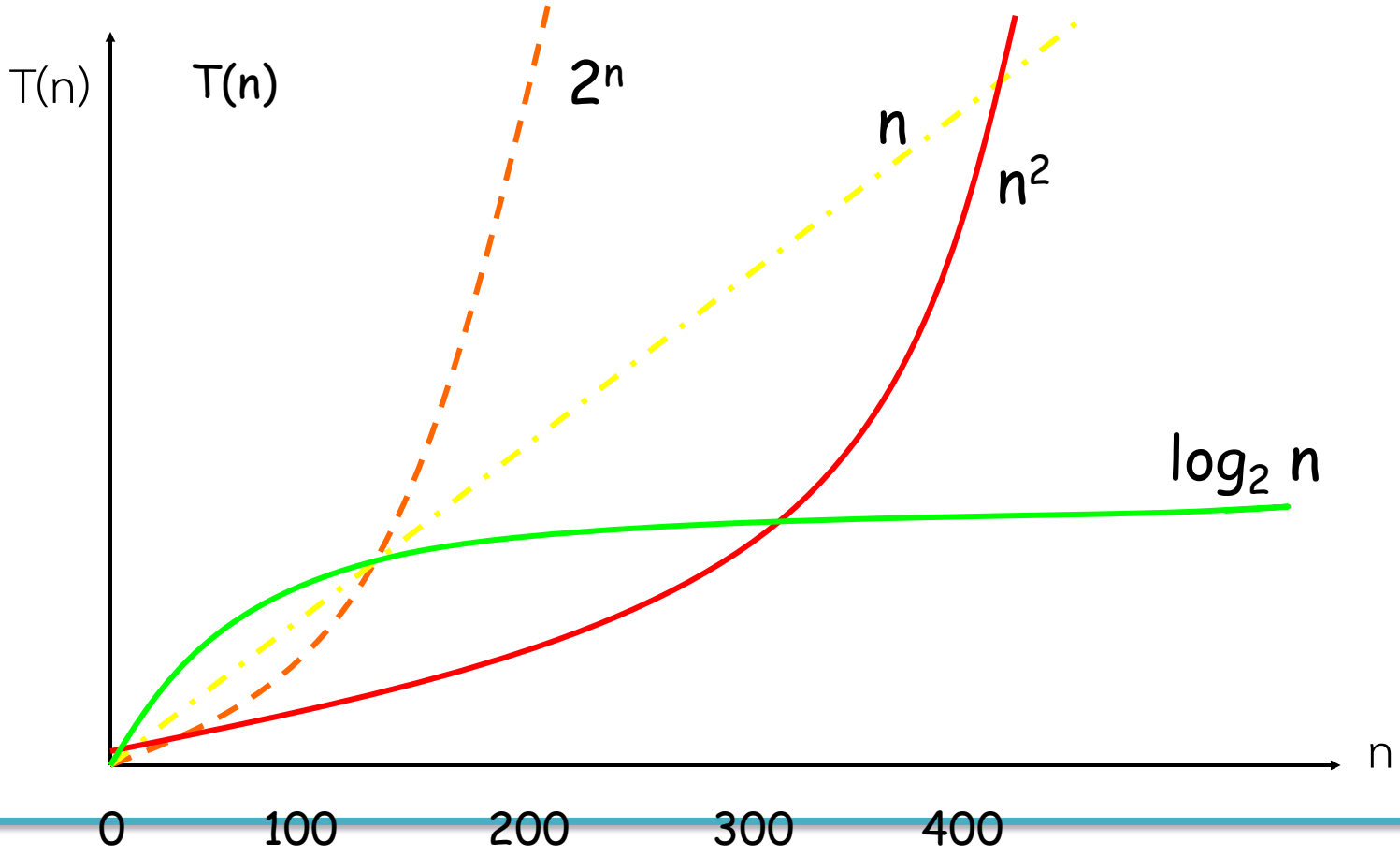
- สมมติว่าอัลกอริทึม A และ B สัมพันธ์กับฟังก์ชันเวลาคือ f(n) = 400n + 23 และ g(n) = $2n^2$-1 ตามลำดับ อัลกอริทึมใดทำงานดีกว่า ?

n	f(n) = 400n + 23	g(n) = 2n^2 - 1
50	20,023	4,999
100	40,023	19,999
150	60,023	44,999
200	80,023	79,999
250	100,023	124,999
300	120,023	179,999



Order of growth

- เรียงลำดับฟังก์ชันเวลาต่อไปนี้ 2^n, n^2, n, $\log_2 n$ จากเติบโตช้าไปเร็วเมื่ออินพุต n มีขนาดใหญ่



การเติบโตของฟังก์ชันเวลา เมื่อ n เพิ่มขึ้น

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Effect of coefficient term

- พิจารณาฟังก์ชันเวลาของอัลกอริทึม ต่อไปนี้

$T1(n) = 100n, \quad T2(n) = 0.01n^2$

$T1(n) = 1{,}000 * \log n, \quad T2(n) = n$

ฟังก์ชันใดทำงานเติบโตได้รวดเร็วและช้ากว่ากัน

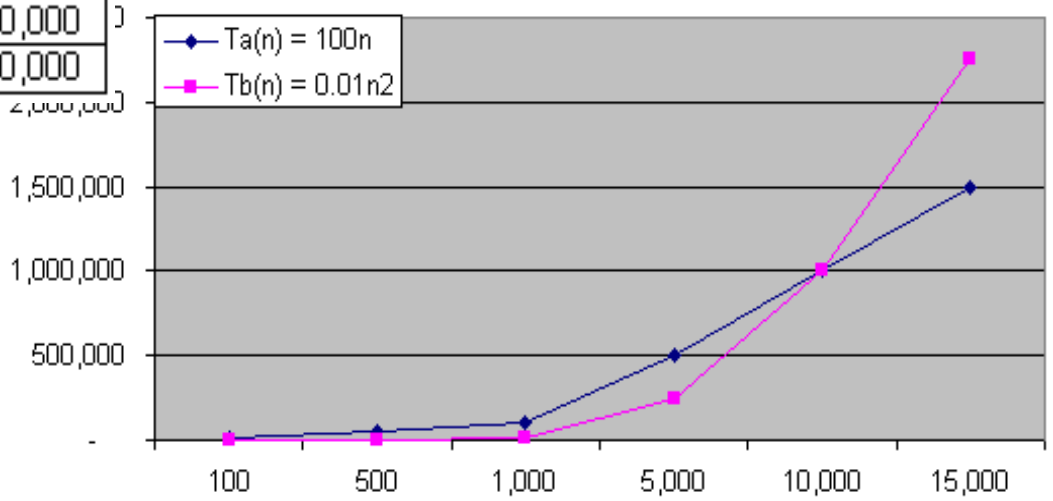
Example 1

Example T1(n) = 100n , T2(n) = $0.01n^2$

Input size	Ta(n) = 100n	Tb(n) = $0.01n^2$
100	10,000	100
500	50,000	2,500
1,000	100,000	10,000
5,000	500,000	250,000
10,000	1,000,000	1,000,000
15,000	1,500,000	2,250,000

$$T_2(n) \succ T_1(n)$$

T2(n) growth faster than T1(n)



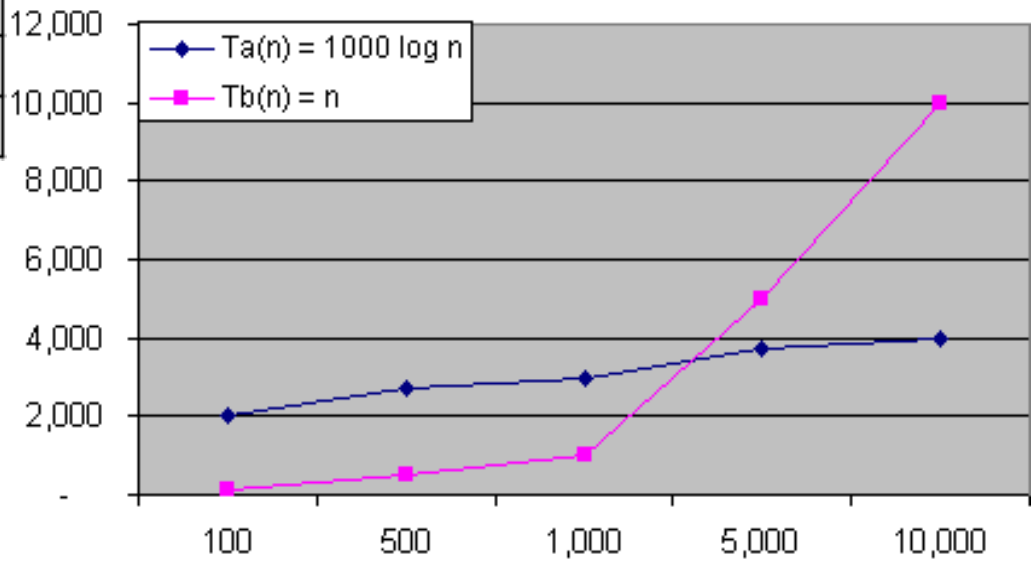
Example 2

Example T1(n) = 1000*log n , T2(n) = n

Input size	Ta(n) = 1000 log n	Tb(n) = n
50	1,699	50
100	2,000	100
500	2,699	500
1,000	3,000	1,000
5,000	3,699	5,000
10,000	4,000	10,000

$$T_1(n) \prec T_2(n)$$

T1(n) growth lower than T2(n)



ทฤษฎี L'Hôpital's rule

- วิธีการที่สะดวกและรวดเร็วกว่าในการเปรียบเทียบลำดับการเติบโตของฟังก์ชันเวลา คือ ทฤษฎีลิมิตของโลปิตา

- กำหนดให้ f(n) และ g(n) เป็นฟังก์ชันเวลาของอัลกอริทึม

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0 & f(n)\prec g(n)\\ c & f(n)\equiv g(n)\\ \infty & f(n)\succ g(n)\end{cases}$$

- จะเห็นได้ว่า เราสนใจเปรียเทียบเวลา เมื่ออินพุตมีขนาดเข้าใกล้ infinity

การหา Derivative ของฟังก์ชั่น

สมมติว่า $\lim_{n\to\infty} f(n) = \infty$ และ $\lim_{n\to\infty} g(n) = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

by $f'(n)$ and $g'(n)$ are derivative of $f(n)$ and $g(n)$

ตัวอย่างการเปรียบเทียบเวลา โดย L'Hôpital's rule

Example T1(n) = 100n , T2(n) = 0.01n²

$$\lim_{n\to\infty}\frac{100n}{0.01n^2}=10{,}000\lim_{n\to\infty}\frac{n}{n^2}=10{,}000\lim_{n\to\infty}\frac{1}{n}=10{,}000\lim_{n\to\infty}\frac{1}{\infty}=10{,}000\lim_{n\to\infty}0=0$$

Therefore $T1(n)\prec T2(n)$

Example T1(n) = 1,000 log n , T2(n) = n

$$\lim_{n\to\infty}\frac{1000.\log n}{n}=1{,}000\lim_{n\to\infty}\frac{\log n}{n}=1{,}000\lim_{n\to\infty}\frac{\log\infty}{\infty}=1{,}000\lim_{n\to\infty}\frac{\infty}{\infty}=\frac{\infty}{\infty}$$

ตัวอย่างการเปรียบเทียบเวลา โดย L'Hôpital's rule

$$= 1{,}000 \lim_{n\to\infty} \frac{\log_{10} e^{\frac{1}{n}}}{1} = 1{,}000 \lim_{n\to\infty} \frac{1}{n} = 1{,}000 \lim_{n\to\infty} \frac{1}{\infty} = 1{,}000 \lim_{n\to\infty} 0 = 0$$

Therefore $T1(n) \prec T2(n)$

Example T1(n) = 1/2*n*(n-1) , T2(n) = n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1 - \frac{1}{n}) = \frac{1}{2}$$

Therefore $T1(n) \equiv T2(n)$

Asymptotic Notation and Basic efficiency class

ในบางครั้งประสิทธิภาพของอัลกอริทึมอาจถูกอธิบายอยู่ในรูปแบบที่เข้าใจได้ง่าย ผ่านการใช้สัญกรเชิงเส้นเพื่อประมาณเวลาที่ใกล้เคียงในการทำงานของอัลกอริทึม

- สัญกรเชิงเส้นกำกับ (Asymptotic notation) โดยทั่วไปมีสามประเภทหลัก

- **O(*g*(*n*))**: class of functions *t*(*n*) that grow *no faster* than *g*(*n*)
- **Θ** (*g*(*n*)): class of functions *t*(*n*) that grow *at same rate* as *g*(*n*)
- **Ω**(*g*(*n*)): class of functions *t*(*n*) that grow *at least as fast* as *g*(*n*)

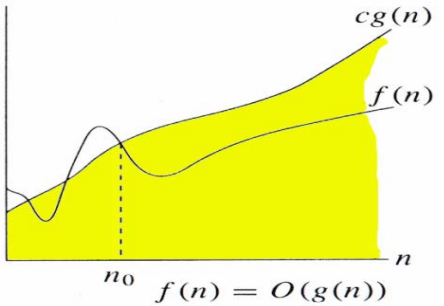
โดยที่ t(n) คือฟังก์ชั่นเวลาของอัลกอริทึม และ g(n) คือ basic efficiency class

Basic asymptotic efficiency class g(n)

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Big-Oh

$$t(n) \in \mathrm{O}(g(n))$$



- O(g(n)) is the set of all functions with a smaller or same order of growth as g(n)

$$t(n) \in O(g(n)) \qquad \text{iff.} \qquad t(n) \prec O(g(n))$$

By f(n) is a algorithm 's basic operation, g(n) is a basic efficiency class

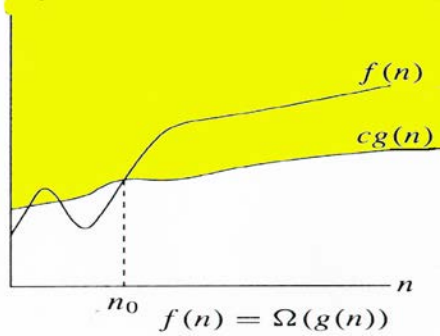
$$n \in O(n^2) \qquad 100n + 5 \in O(n^2) \qquad \frac{1}{2}n(n-1) \in O(n^2)$$

but

$$n^3 \notin O(n^2) \qquad 0.00001n^3 \notin O(n^2) \qquad n^3 + n + 1 \notin O(n^2)$$

Omega

$$t(n) \in \Omega(g(n))$$



- Ω(g(n)) is the set of all functions with a larger or same order of growth as g(n)

$$t(n) \in \Omega(g(n)) \qquad \text{iff.} \quad t(n) \succ \Omega(g(n))$$

Ex.

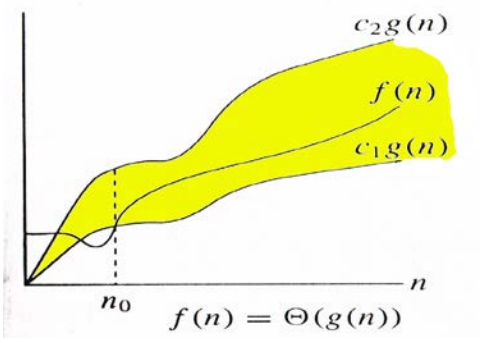
$$n^3 \in \Omega(n^2) \qquad \frac{1}{2}n(n-1) \in \Omega(n^2) \qquad 0.00001n^4 \notin \Omega(n^3)$$

but

$$1{,}000n + 5 \notin \Omega(n^2) \qquad n^2 + 50\log n \notin \Omega(n^3)$$

Theta

$$t(n) \in \Theta(g(n))$$



- θ(g(n)) is the set of all functions that have same order of growth as g(n)

$$t(n) \in \Theta(g(n)) \qquad \text{iff.} \qquad t(n) \approx \Theta(g(n))$$

Ex.

$$\frac{1}{2}n(n-1) \in \Theta(n^2) \qquad n^2 + \sin n \in \Theta(n^2) \qquad n^2 + \log n \in \Theta(n^2)$$

but

$$n^2 + \sin n \notin \Theta(n) \qquad 1{,}500n^2 \notin \Theta(n^3)$$

ตัวอย่างการพิสูจน์

Example prove that $100n + 5 \in O(n^2)$

$100n + 5 \leq c.n^2$ for all $n \geq n_0$ (1)

Divide by n^2

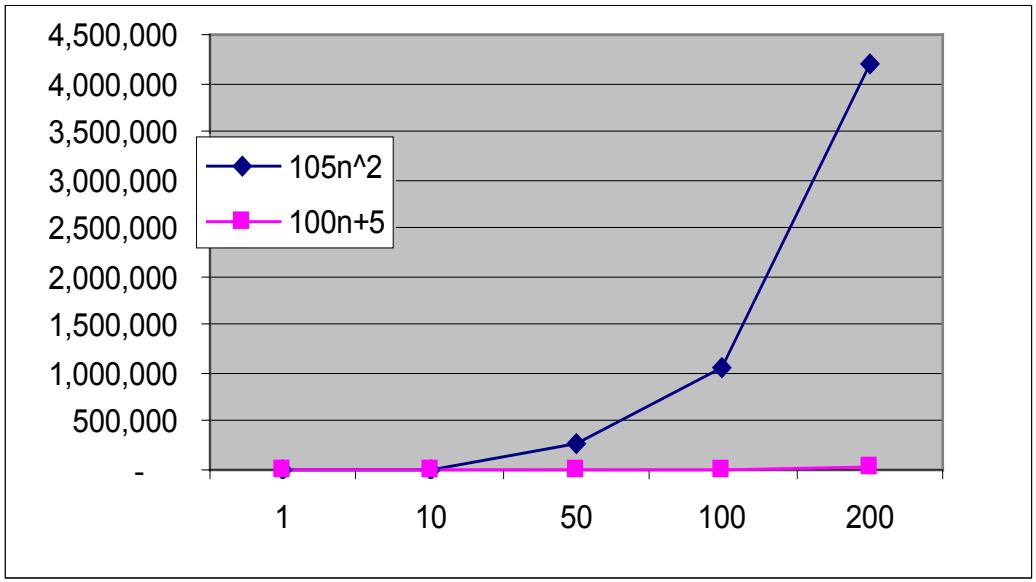
$$\frac{100}{n} + \frac{5}{n^2} \leq c$$

Assume n=1

$$100 + 5 \leq c$$
$$105 \leq c$$

Replace c=105 in (1) $100n + 5 \leq 105n^2$ for all $n \geq 1$

ตัวอย่างการพิสูจน์



$$c = 105 \qquad n \geq 1$$

- จะเห็นได้ว่าเมื่อคูณ 105 เข้ากับ n^2 เมื่อ n >= 1 ฟังก์ชัน 100n+5 จะโตช้ากว่า n^2 เสมอ

ตัวอย่างการพิสูจน์

Example prove that $0.1n^3 \in \Omega(n^2)$

$0.1n^3 \le c.n^2$ for all $n \ge n_0$ (1)

Divide by n^2 $0.1n \le c$

Assume n=1 $0.1(1) \le c$

$0.1 \le c$

Replace c=0.1 in (1) $0.1n^3 \le 0.1n^2$ for all $n \ge 1$

ตัวอย่างการพิสูจน์

Example prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Upper bound prove $\frac{1}{2}n(n-1) \le c1.n^2$ for all $n \ge n_0$

assign c1 = 1/2 $\frac{n^2}{2} - \frac{n}{2} \le \frac{1}{2}n^2$ for all $n \ge 0$

Lower bound prove $\frac{1}{2}n(n-1) \ge c2.n^2$

$$\frac{n^2}{2} - \frac{n}{2} \ge c2.n^2$$

ตัวอย่างการพิสูจน์

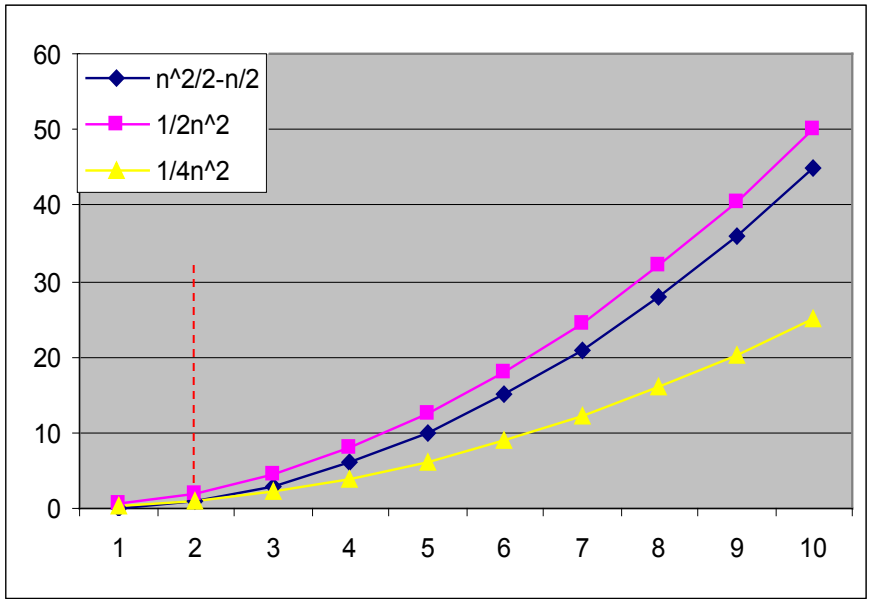
$$\frac{1}{2}-\frac{1}{2n}\geq c2$$

Assume n=2

$$\frac{1}{2}-\frac{1}{4}\geq c2$$

$$\frac{1}{4}\geq c2$$

$$c1=\frac{1}{2}, c2=\frac{1}{4}$$ at n=0 and n=2



Workshop: Asymptotic Notation

- ตรวจสอบฟังก์ชันเวลาต่อไปนี้ว่าเป็นจริงหรือไม่

$$1{,}000n \in O(n^2)$$

$$\sqrt{n} \in O(n)$$

$$\frac{n}{2}.\log(\frac{n}{2}) \in \Omega(n.\log n)$$

$$2n^2 + 500n + 1{,}000\log n \in \Theta(n^2)$$

Mathematical analysis of non-recursive algorithms

- ระบุขนาดของข้อมูล (*input size)* และ basic operation ของอัลกอริทึม
  - Basic operation มักที่อยู่ในลูปชั้นในสุด
- พิจารณาว่าอัลกอริทึมมี *worst*, *average*, and *best* case หรือไม่
  - จำนวนครั้งการทำงานของ basic operation แตกต่างกันเมื่อข้อมูลต่างกัน แม้ขนาดข้อมูลเท่ากัน
- ตั้งสมการ summation ของ basic operation
- แก้สมการเพื่อหา Order of growth

Useful Summation Formulas and Rules

- $\sum_{i=1}^{n} 1 = 1 + 1 + \cdots + 1 = n \in \Theta(n)$
- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$
- $\sum_{i=1}^{n} i^k = 1 + 2^k + \cdots + n^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$
- $\sum_{i=1}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \in \Theta(a^n)$
- $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i \qquad \sum_{i=1}^{n} c\, a_i = c \sum_{i=1}^{n} a_i$

Find Maximum

```
MaxElement (A[1...n])
 Maxval:= A[1]
 for i:= 2 to n do
    if A[i] > maxval then
          maxval:= A[i]
return maxval
```

$$T(n) = \sum_{i=2}^{n} 1$$

$$T(n) = \sum_{i=2}^{n} 1 = n - 2 + 1 = n - 1 \in \Theta(n)$$

$$T(n) \in \Theta(n)$$

Matrix multiplication

```
MatrixMultiplication (A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
for i ⟵ 0 to n-1 do
    for j ⟵ 0 to n-1 do
        C[ i, j ] ⟵ 0.0
        for k ⟵ 0 to n-1 do
            C[ i, j ] ⟵ C[ i, j] + A[ i, k] * B[ k, j]
return C
```

Selection Sort algorithm

```
Algorithm SelectionSort(A[0..n-1])
  for i←0 to n-2 do
      min←i
      for j←i+1 to n-1 do
          if A[j] < A[min]
              min ← j
      swap A[i] and A[min]
```

Example:

Insertion Sort Algorithm

```
Algorithm InsertionSort( A[0 .. N-1] )
  for i←1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

Example:

31 | 25 12 22 11

25 31 | 12 22 11

12 25 31 | 22 11

12 22 25 31 | 11

BitCount algorithm

algorithm $BitCount(m)$
    // Input: A positive integer m
    // Output: The number of bits to encode m
    $count \leftarrow 1$
    while $m > 1$ **do**
        $count \leftarrow count + 1$
        $m \leftarrow \lfloor m/2 \rfloor$
    return $count$

Mathematical Analysis of recursive algorithms

- **Plan for analysing recursive algorithms**
  - Decide on parameter n indicating input size.
  - Identify algorithm's basic operation(s).
  - Determine worst, average, and best cases for input of size n.
  - Set up a recurrence relation expressing the basic operation count.
  - Solve the recurrence (at least determine it's order of growth).

Recurrence for Factorial Function

algorithm $Factorial(n)$
// Computes $n!$ recursively
// Input: A nonnegative integer n
// Output: The value of $n!$
if $n = 0$ **then return** 1
else return $Factorial(n-1) * n$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: multiplication

Let $M(n)$ = multiplication count to compute $Factorial(n)$.

$M(0) = 0$ because no multiplications are performed to compute $Factorial(0)$.

If $n > 0$, then $Factorial(n)$ performs recursive call plus one multiplication.

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } Factorial(n-1)} + \underbrace{1}_{\text{to multiply } Factorial(n-1) \text{ by } n}$$

Solving the Factorial Recurrence

- **Forward substitution** : $M(1) = M(0) + 1 = 1$
  - $M(2) = M(1) + 1 = 2$
  - $M(3) = M(2) + 1 = 3$
- **Backward substitution** : $M(n) = M(n-1) + 1$
  - $[\, M(n-2) + 1 \,] + 1 = M(n-2) + 2$
  - $[\, M(n-3) + 1 \,] + 2 = M(n-3) + 3$
- Solve the general recurrence:
  - Assume K is the iteration. At K = n
    - $M(n-k) + k = M(n-n) + n = M(0) + n = n \in O(n)$

Recurrence of BitCount

algorithm $BitCount(n)$
// Input: A positive integer n
// Output: The number of bits to encode n
if $m = 1$ **then return** 1
else return $BitCount(\lfloor n/2 \rfloor) + 1$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: division by 2

- Let $D(n)$ = division count to compute $BitCount(n)$.
- $D(1) = 0$ because no divisions are performed to compute $BitCount(1)$.
- If $n > 1$, then $BitCount(n)$ performs recursive call on $\lfloor n/2 \rfloor$ plus one division.

$$D(n) = \underbrace{D(\lfloor n/2 \rfloor)}_{\text{to compute } BitCount(\lfloor n/2 \rfloor)} + \underbrace{1}_{\text{to compute } \lfloor n/2 \rfloor}$$

Recurrence of find minimum

```
algorithm findmin (A[], i, j)
    if i=j then
        return A[i];
    mid = (i+j)/2;
    m1  = findmin(A , i , mid);
    m2  = findmin(A , mid+1 , j);
return m1<m2? m1:m2;
```

Recurrence of MoveDisk algorithm

```
ALGORITHM MoveDisk (N, fromPeg, toPeg)
  if N = 1 then
    PRINT fromPeg + "=>" + toPeg;
 else
    help = 6-fromPeg-toPeg;          // fromPeg + help + toPeg = 6
    MoveDisk(N-1, fromPeg, help);    // Move n-1 disks to helpPeg
    PRINT fromPeg + "=>" + toPeg;    // Move the last disk to toPeg
    MoveDisk(N-1, help, toPeg);      //  Move n-1 disks to toPeg
```

call MoveDisk(4, 1, 3);

$$T(n) = T(n-1) + 1 + T(n-1) \qquad n > 1$$

$$T(1) = 1 \qquad n = 1$$

Recurrence of MoveDisk algorithm

$$T(n) = 2T(n-1) + 1$$

$$T(n) = 2.[2T(n-2) + 1] + 1 = 2^2 T(n-2) + 2^1 + 1$$

$$T(n) = 2^2[2T(n-3) + 1] + 2^1 + 1$$

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

$$T(n) = 2^{n-1}.T(n-(n-1)) + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0$$

$$T(n) = 2^{n-1}T(1) + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0$$

$$T(n) = \sum_{i=0}^{n-1} 2^i = \sum_{i=0}^{n} 2^i - 2^n = \frac{2^{n+1}-1}{2-1} - 2^n$$

$$T(n) = 2^n[2-1] - 1 = 2^n - 1 \in \Theta(2^n)$$ Exponential !!

Exercise

(1) $x(n) = x(n-1) + 5 \text{ for } n > 1, \ x(1) = 0$

(2) $x(n) = 3x(n-1) \text{ for } n > 1, \ x(1) = 4$

(3) $x(n) = x(n/2) + n \text{ for } n > 1, \ x(1) = 1 \ (\text{solve for } n = 2^k)$

Homework

Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + ... + n^3$.

Algorithm $S(n)$
//Input: A positive integer n
//Output: The sum of the first n cubes
if $n = 1$ **return** 1
else return $S(n-1) + n * n * n$

```
Algorithm Parallel-Product(A[1..n]);
    if n = 1 then return;
    for i := 1 to n/2 do
        A[i] := A[i]*A[i+n/2];
    call Parallel-Product(A[1..n/2]);
```

Homework

Algorithm $Q(n)$
//Input: A positive integer n
if $n = 1$ **return** 1
else return $Q(n-1) + 2 * n - 1$

b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.