

**Complexity**

# ความซับซ้อนของอัลกอริธึม Algorithmic Complexity

```
def fac (n): # n>=0
    if n == 0 or n == 1:
        return 1
    else:
        x = fac (n-1)
        return n * x
```

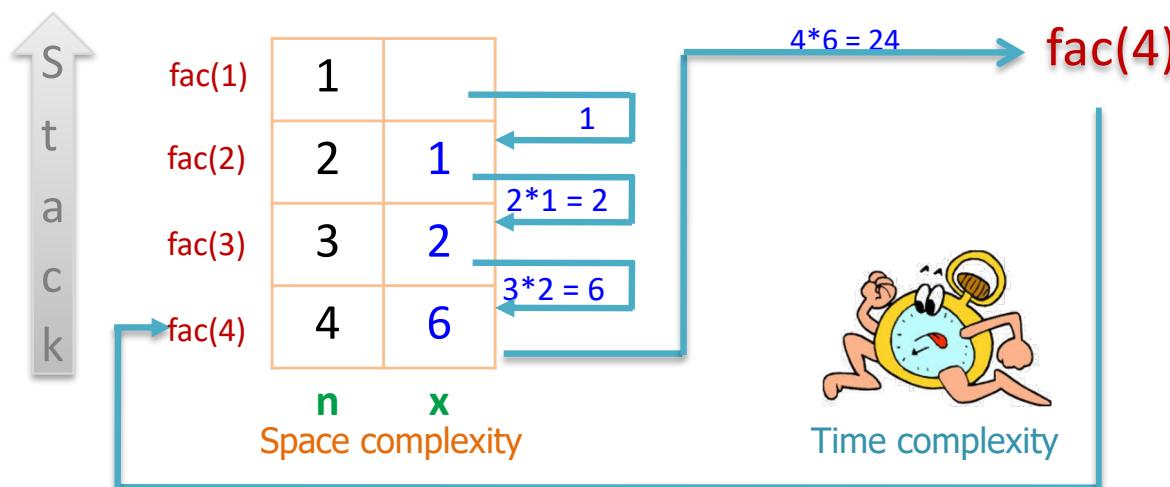
Algorithm : Good ? Bad ?

- Efficiency ประสิทธิภาพ
- Complexity ความซับซ้อน
  - Space complexity จำนวน memory ที่ต้องใช้ให้รันเสร็จ
  - Time complexity จำนวน CPU time ที่ต้องใช้ให้รันเสร็จ
    - Worst case
    - Average case
    - Best Case

✓ ซับซ้อนมาก → รันช้า / ใช้ space มาก → ประสิทธิภาพ ต่ำ

✓ ซับซ้อนน้อย → รันเร็ว / ใช้ space น้อย → ประสิทธิภาพ สูง

- Performance Analysis ประมาณค่า complexity ล่วงหน้า
- Performance Measurement วัด space & time ในการรันครั้งนั้น



# Examples of Time Function T( $n$ )

แผ่นละ 4 นาที  $n$  แผ่น =  $4 \times n$  input size  $n$

$$T(n) = 4n$$



Algorithm 1	Algorithm 2	Algorithm 3
<pre>for i=1 to n do     a = 1; b = 2;     c = 3;</pre>	<pre>for i=1 to n do     a = 1; b = 2;     c = 3; d = 4;</pre>	<pre>for i=1 to n do     for j=1 to n do         a = 1;</pre>
ถ้า แต่ละ assignment ใช้เวลาคงที่ 1 หน่วย		
$T_1(n) = 3n$	$T_2(n) = 4n$	$T_3(n) = n^2$

ถ้า  $T_1(n)$ ,  $T_2(n)$ ,  $T_3(n)$  ทำงานอย่างเดียวกัน จะเลือกใช้ algorithm ใด ?

# Asymptotic Analysis

time เป็น  
มากเสมอ

Algorithm ไหน เร็ว-ช้ากว่า?

บอกไม่ได้ ขึ้นกับค่า n

ถ้าให้เลือก  $4n$  ?  $n^2$  ? เลือก  $4n$  ทำไม ?

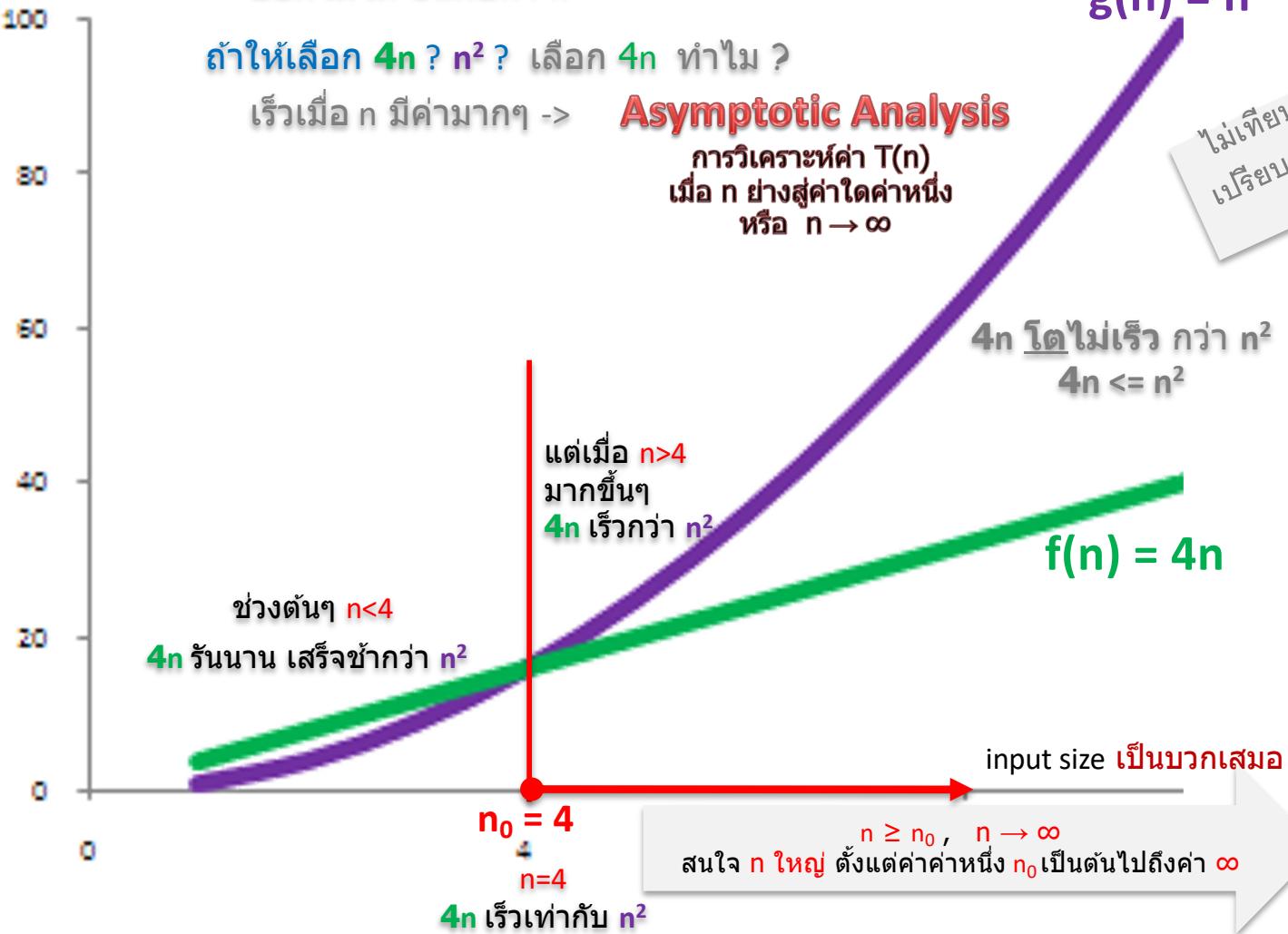
เร็วเมื่อ n มีค่ามากๆ ->

## Asymptotic Analysis

การวิเคราะห์ค่า  $T(n)$   
เมื่อ n ย่างสูงค่าใดค่าหนึ่ง  
หรือ  $n \rightarrow \infty$

$$g(n) = n^2$$

ไม่เที่ยบที่ n ค่าใดค่าหนึ่ง  
เปรียบเทียบ growth rate  
อัตราการเติบโต



n	$4n$	$n^2$
1	4	1
2	8	4
3	12	9
4	16	16
5	20	25
6	24	36
7	28	49

# Classes of Algorithms

อย่างประเมิน algorithms โดยแบ่งกลุ่มตาม พังก์ชันมาตรฐานง่ายๆ เช่น

constant      กลุ่ม  $c, 1$        $2 \quad 1 \quad 10$

Logarithmic      กลุ่ม  $\log n$        $2 \log n$

Linear      กลุ่ม  $n$        $4n \quad 2n + 1 \quad 100n$

Linearithmic      กลุ่ม  $n \log n$        $4n \log n + 5$

Quadratic      กลุ่ม  $n^2$        $10n^2 + 5 \quad 3n^2 - 75 \quad 20n^2 + 87$

เมื่อ  $n$  มากๆ (asymptotic) เทอมที่กำลังน้อยกว่า (lower-order terms) แทนไม่มีความหมาย

สัมประสิทธิ์ (constant factors) อาจแตกต่างกัน แต่มองเหมือนค่าคงที่ ดังนั้นมองแต่ term หลัก

ไม่ได้แปลว่าห้ามคูณสำคัญในการวิเคราะห์ algorithm แต่เมื่อมองภาพใหญ่ เช่นอย่างเปรียบเทียบ algorithms asymptotic analysis สามารถนำทางได้ว่าอันไหนดีกว่าอันไหน โดยเฉพาะอย่างยิ่งกับ input ขนาดใหญ่

# Asymptotic Analysis & Big Oh

- constant factor  $c$  (Harry = 2, Tim = 30)

ต่างกันได้ ไม่ถือเป็นสำคัญ วัดตัว algorithm ที่ทำ

$$T(n) = cn + d$$

แผ่นละ  $c$  นาที  $n$  แผ่น =  $c \times n$  **input size  $n$**

- lower-order terms เช่น มีช่วงพัก

เมื่อ  $n$  มีค่ามาก (Asymptotic)

ส่วนนี้มีค่าน้อยเมื่อเทียบกับ term หลัก



$$T(n) = O(n)$$

Running time is **big oh** of  $n$

อ่าน  $T$  of  $n$  is big oh of  $n$

Algorithm เดียวกัน เร็ว-ช้า, ใช้ space มาก-น้อย ต่างกัน ได้จาก ปัจจัยอื่น

- Architecture - CPU speed, Instruction set, Disk speed
- Language
- Compiler - dependent details

## ANALYSIS OF ALGORITHMS BIG-OH NOTATION

ต้องการ วัด algorithm แบบ

- หยาบ พอที่จะ ไม่นำสิ่งเหล่านี้มาเป็นประเด็น
- ละเอียด พอที่จะ ทำนาย เปรียบเทียบ ระหว่าง algorithm ที่แตกต่างกัน โดยเฉพาะอย่างยิ่งบน large inputs



# Example : One Loop

หาว่า  $t$  อยู่ใน array  $A$  ที่มี  $\text{length} = n$  หรือไม่ ?

Algorithm :

```
for i = 1 to n do
    if A[i] == t
        return TRUE
return FALSE
```

Running time = ? (Depend on input  $n$ )

- ก)  $O(1)$
- ข)  $O(\log n)$
- ✓ ค)  $O(n)$
- ง)  $O(n^2)$

ขึ้นกับ เจอที่ไหนใน array : worst case  $\rightarrow$  ทำ loop  $n$  ครั้ง

constant factor :  $c n$

• ในการ access array  $\rightarrow c$

lower order term :  $c^2$

ในการ return boolean

$$c n + c^2 \rightarrow O(n)$$

## Example : Two Loops

A and B เป็น array มี length n  
หาว่า t อยู่ใน A หรือ B ?

Algorithm :

```
for i = 1 to n do
    if A[i] == t
        return TRUE

for i = 1 to n do
    if B[i] == t
        return TRUE

return FALSE
```

Running time = ? (Depend on input n)

- ก) O(1)
- ข) O(log n)
- ✓ ค) O(n)
- ง) O( $n^2$ )

worst case running time เป็น 2 เท่าของ algorithm ที่แล้ว  
constant factor :  $2n$

## Example : Two Nested Loops

A and B เป็น array มี length n

หาว่า A และ B มี data ร่วมกันหรือไม่ ?

Algorithm :

```
for i = 1 to n do
    for j = 1 to n do
        if A [i] == B [j]
            return TRUE

return FALSE
```

Running time = ? (Depend on input n)

ก)  $O(1)$

ข)  $O(\log n)$

ค)  $O(n)$

✓ ง)  $O(n^2)$

running time = quadratic  
คือ double n แล้วได้  
 $= \text{runtime} \text{เดิม} \times 4$

แต่ละครั้งของ outer for ทำ n ครั้งของ inner for  
outer for ทำ n ครั้ง

## Example : $\log_2 n$

กำหนด  $n$  = จำนวนผู้เข้าแข่งขัน

คัดออกทีละครึ่งจนเหลือผู้ชนะ 1 คน ต้องแข่งกี่รอบ ?

นิยามของ  $\log$  :

$$\log_2 n = x \rightarrow 2^x = n \rightarrow 2 \text{ คูณกันกี่ครั้งได้ } n \rightarrow n \text{ ถูกหารด้วย } 2 \text{ กี่ครั้งจึงเหลือ } 1$$

Algorithm :

```
i = n  
c = 0  
while i >= 1 do  
    i = i / 2  
    c += 1  
return c
```

Running time = ? (Depend on input  $n$ )

ก)  $O(1)$

✓ ข)  $O(\log n)$

ค)  $O(n)$

ง)  $O(n^2)$

$$\log_x n = \log_x y \times \log_y n$$

( $\log_x y$  เป็น constant)

จึงไม่นิยมเขียนฐาน  $\log$

# Big Oh - Simple Standard Functions

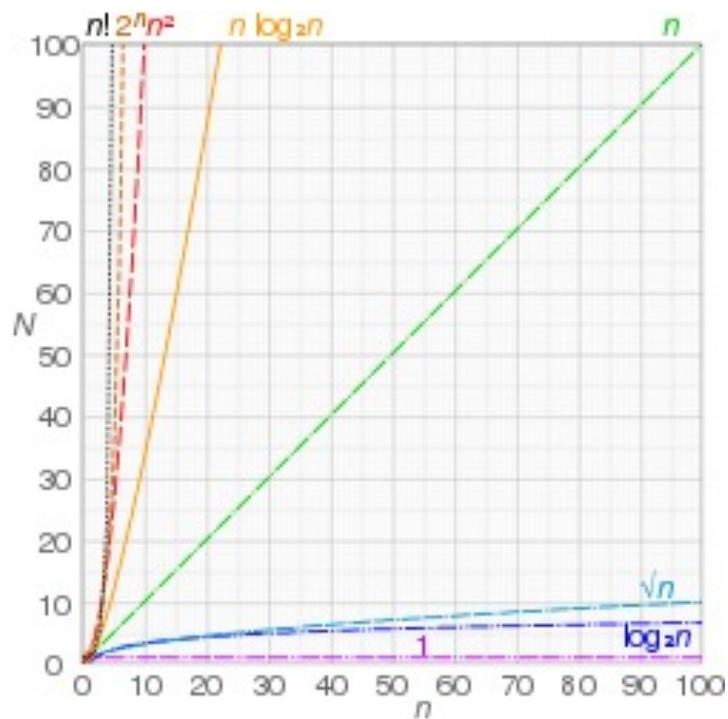
สำหรับ Big Oh

constant factor & lower order terms

ถูกมองข้าม ไม่คิดเป็นสำคัญ

จึงนิยมแสดงในรูปของ

ฟังก์ชันมาตรฐานแบบง่ายๆ

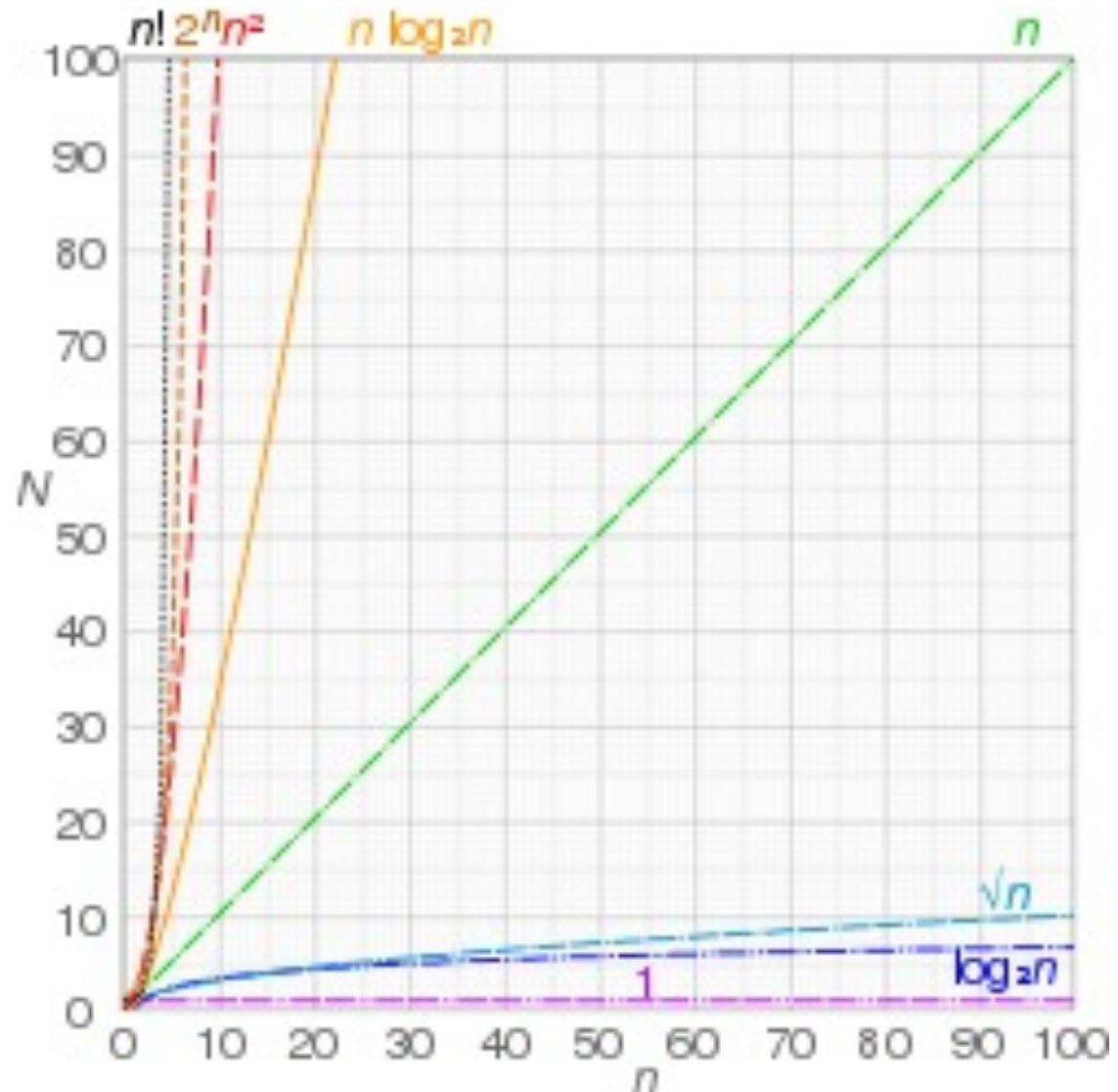


สัญลักษณ์	ชื่อ
$O(1)$ , $O(c)$	Constant
$O(\log \log n)$	double logarithmic
$O(\log n)$	logarithmic
$O((\log n)^c)$	polylogarithmic
$O(n)$	linear
$O(n \log n)$	linearithmic/loglinear/quasilinear
$O(n^2)$	quadratic
$O(n^c)$	polynomial หรือ algebraic
$O(c^n)$	exponential
$O(n!)$	factorial

# Estimating Algorithms

การวัด algorithm :

- จัดว่า algorithm อยู่ในกลุ่ม class ไหน (class : พังก์ชั้นมาตรฐานง่ายๆ)
- ประมาณค่า algorithm โดยเปรียบเทียบ กับ algorithm อื่น นิยมเปรียบเทียบกับ พังก์ชั้นง่ายๆ
  - upper bound → Big Oh** บอกว่า running time โดยอย่างมาก ที่สุดเท่านี้ ไม่มากกว่านี้ at most (แต่มันอาจดีกว่านี้)
  - lower bound → Big Omega**
  - ...

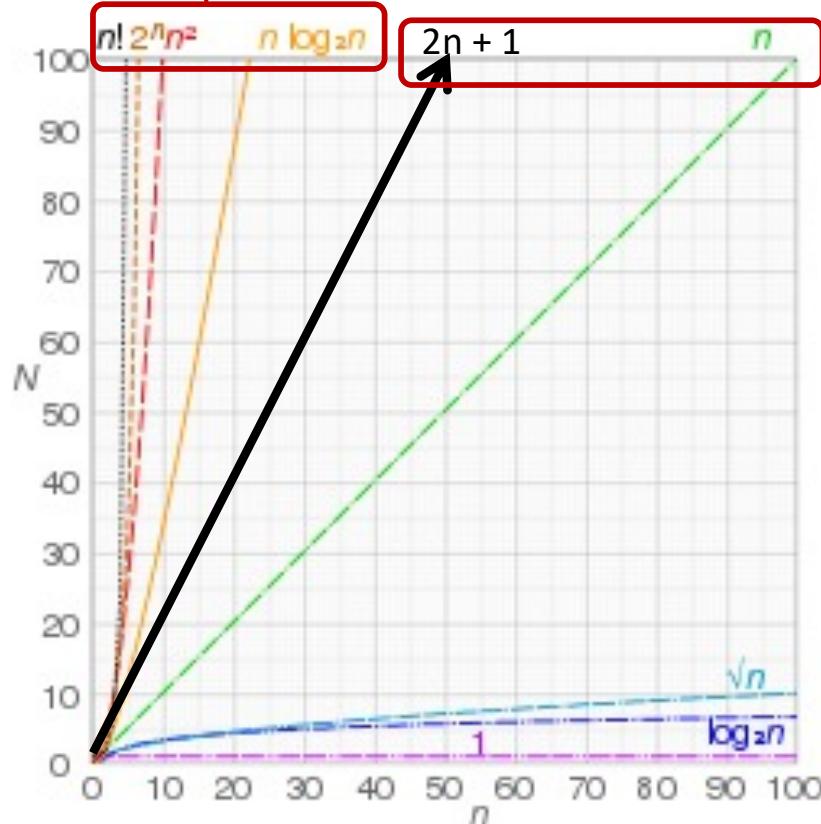


# Asymptotic Upper Bound

ฟังก์ชันอะไรบ้างเป็น Asymptotic Upper Bound ของ  $2n + 1$  ?

Asymptotic : when  $n \rightarrow \infty$

จากราฟ ชัดเจนว่า  $n^n$   $n!$   $2^n$   $n^2$   $n \log_2 n$   
ต่างเป็น Asymptotic Upper Bound ของ  $2n + 1$



เป็น Asymptotic Upper Bound ของ  $2n + 1$  ?

Asymptotic : when  $n \rightarrow \infty$   
 $n$  &  $2n + 1$  อยู่ในกลุ่มเดียวกัน  
ส.p.ส & lower-order term ไม่ใช่สิ่งสำคัญ

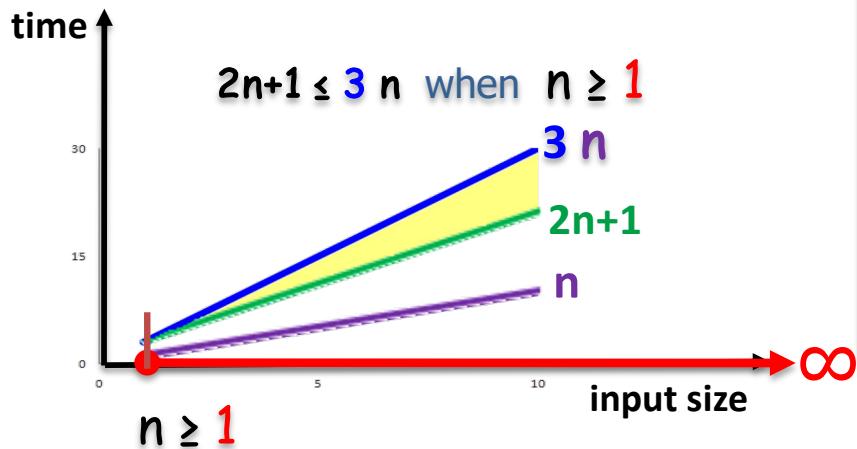
$n$  เป็น Asymptotic Upper Bound ของ  $2n + 1$

# Asymptotic Upper Bound & Big Oh

สำหรับ Asymptotic : when  $n \rightarrow \infty$  :  $n$  &  $2n + 1$  อยู่ในกลุ่มเดียวกัน ดังนั้น อย่างไรทั้ง

1.  $2n + 1$  Asymptotic upper bounds  $n$  ซึ่งเป็นจริงอยู่แล้ว :  $n \leq 2n + 1$ ,  $n \geq 0$

2.  $n$  Asymptotic upper bounds  $2n + 1$  จะเป็นไปได้เมื่อ  $2n + 1 \leq c n$  เมื่อ  $c$  คือ some positive  $c$ ,  $n_0$



$n$  เป็น Asymptotic Upper Bound ของ  $2n + 1$

$$2n + 1 = O(n)$$

เพราะ  $2n + 1 \leq 3n$  :  $n \geq 1$  ( $n_0$ )

Asymptotic Upper Bound :

เมื่อมีค่าคงที่  $c$ ,  $n_0$  ที่ทำให้  
ตั้งแต่  $n \geq n_0$  เป็นต้นไป  $T(n) \leq c g(n)$  เมื่อ  
เรียกว่า  $g(n)$  bound อยู่ข้างบน  $T(n)$  แบบ Asymptotic  
(Asymptotic Upper Bound)

$$T(n) = O(f(n))$$
 ค่านว่า  $T$  of  $n$  is big oh of  $f$  of  $n$

if and only if there exist constants  $c$ ,  $n_0 > 0$  such that

$$T(n) \leq c f(n) : \text{for all } n \geq n_0$$

$c$ ,  $n_0$  cannot depend on  $n$

$$\text{แต่ } n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n! \leq n^n$$

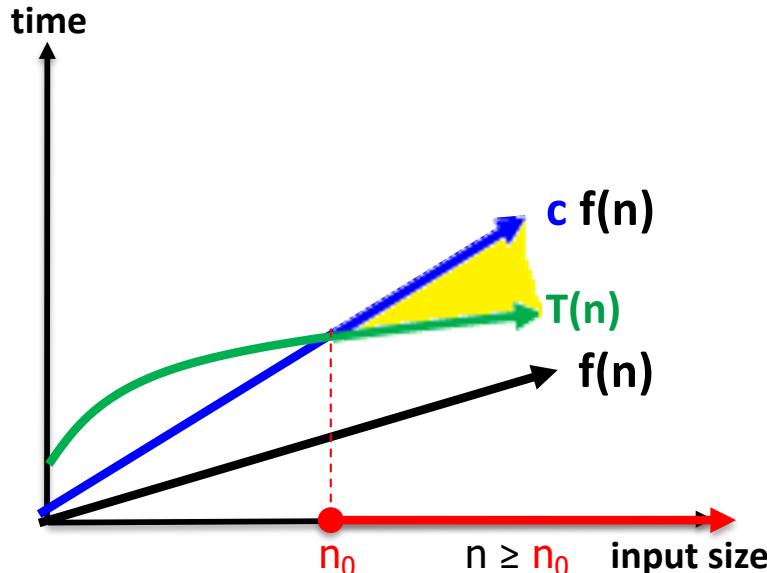
ดังนั้น ทุกตัวเป็น big oh ของ  $2n + 1$

$$2n + 1 = O(n \log n), 2n + 1 = O(n^2), 2n + 1 = O(n^3), 2n + 1 = O(2^n), 2n + 1 = O(n!), 2n + 1 = O(n^n)$$

คำถาม : running time มี order ลำดับ ?

คำตอบ : ต้องเลือก least asymptotic upper bound : เพราะเป็นตัวชี้ภาพของ  $f^n$   $2n + 1 = O(n)$  Linear Order

# Big Oh Definition



$$T(n) = O(f(n))$$

อ่านว่า

T of n is big oh of f of n

if and only if there exist constants  $c, n_0 > 0$

such that

$$T(n) \leq c f(n) : \text{for all } n \geq n_0$$

$c, n_0$  cannot depend on n

- ความหมายของ  $T(n) = O(f(n))$  คือ ในที่สุด (เมื่อ n มีค่ามากพอ)  $T(n)$  จะถูก bound ข้างบนโดย constant คูณ คุณ  $f(n)$  (multiple of  $f(n)$ )
- Big Oh ให้ asymptotic upper bound หลายตัว ไม่ได้ทำให้เห็น asymptotic tight bound ซึ่งเป็นตัวชี้ภาพของ  $T(n)$  ดังนั้นเวลาใช้จะต้องใช้ least upper bound
- Big Oh บอกว่า running time โตอย่างมากที่สุดเท่านี้ (at most) ไม่มากกว่านี้ (แต่อาจดีกว่านี้ เพราะ เราใช้ worst case ของ  $T(n)$ )  
บางที่ใช้คำว่า โตไม่เร็วกว่า ( $\leq$ ) (มากกว่าหรือเท่ากับ) เช่น  $2n + 1$  โตไม่เร็วกว่า  $n$   
จะเห็นว่า เป็นความจริงที่  $2n + 1$  โต (แบบ asymptotic) ไม่มากกว่า  $n, n^2, n^3, \dots$   
ใช้เมื่อต้องการพูดว่า guarantee ว่าไม่ช้ากว่านี้ (ตัวที่เป็น tight, least asymptotic upper bound ไม่ใช่ทุก big oh)

# Asymptotically bound, $c g(n)$ , $n \geq n_0$

## Big O Notation

ใช้เปรียบเทียบกับฟังก์ชัน  $T(n)$  ของอัลกอริธึม กับ ฟังก์ชัน  $g(n)$  ที่เราเลือกมาใช้เปรียบเทียบ

เมื่อหาค่าคงที่  $c$  มาตรฐาน  $g(n)$  ของ Big O และ

$T(n) \leq c g(n)$  เช่นเดียวกันแต่  $n \geq n_0$

เรียก  $g(n)$  bound ออยู่ข้างบน  $T(n)$  แบบ asymptotic

เลือก  $g(n)$  ใกล้ (tight) พอดียืนให้เห็น limit ของ  $T(n)$

นิยมเลือก  $g(n)$  เป็นรูปมาตรฐานง่าย ๆ

$1, c$  Constant

$\log n$  Logarithmic

$\log^2 n$  Log-squared

$n$  Linear

$n \log n$  Log-linear

$n^{1.5}$

$n^2$  Quadratic

$n^3$  Cubic

$n^k$  Polynomial

$x^n$  Exponential

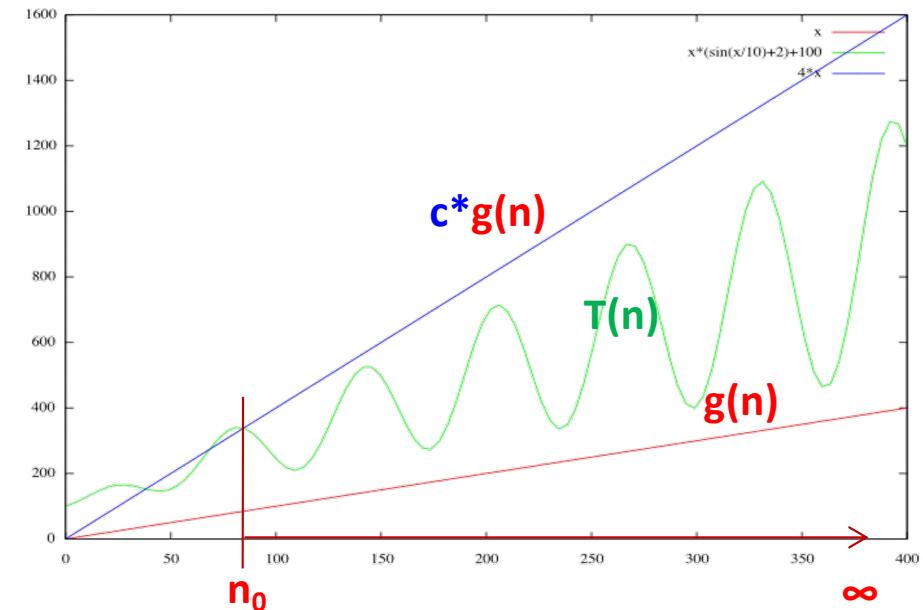
Big แปลว่า capital  
O แปลว่า order  
Big O = order of complexity  
= ลำดับของความซับซ้อน

Big O ที่ใกล้พอดีจะช่วยแบ่งประเภท  $T(n)$  ได้

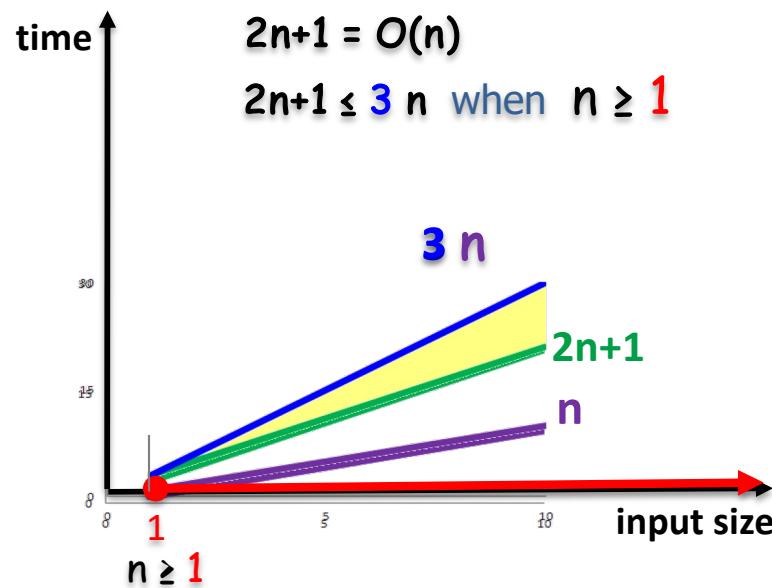
กลุ่มเดียวกันแน่นด้วย Big O เดียวกัน

ช่วยจัดอันดับ(order) ความซับซ้อนของ  $T(n)$

Big O นำมาใช้ประเมินการใช้งานของ resources ต่าง ๆ ของอัลกอริธึม



# Proof: $2n+1 = O(n)$



Pf ว่า  $2n+1$  มีบิกโอดีน  $n$   $2n+1 = O(n)$

ต้องหา constants บวก  $C$  และ  $n_0$  ซึ่งทำให้

$$2n+1 \leq c n \quad \text{เมื่อ } n \geq n_0^1$$

ต้องได้ว่า  $2n+1 \leq 3n$  เมื่อ  $n \geq 1$

ต้องได้ว่า  $2n+1 \leq 2n+n$  เมื่อ  $n \geq 1$  ซึ่งเป็นจริง

$\therefore 2n+1 \leq 3n, n \geq 1$  เป็นจริง

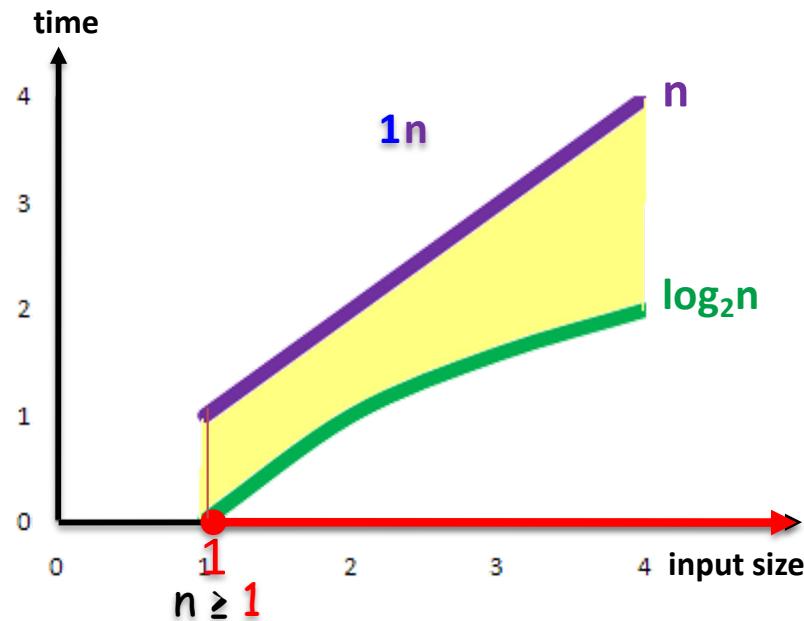
$2n+1 = O(n) \because \text{มี } c = 3, n_0 = 1 \text{ ทำให้}$

$2n+1 \leq 3n \quad \text{when } n \geq 1$

# Proof: $\log_2 n = O(n)$

$$\log_2 n = O(n)$$

$$\log_2 n \leq 1n \text{ when } n \geq 1$$



Pf ว่า  $\log_2 n$  มีบิกโอเป็น  $n$

$$\log_2 n = O(n)$$

ต้องหา constants บวก  $C$  และ  $n_0$  ซึ่งทำให้

$$\log_2 n \leq cn \quad \text{เมื่อ } n \geq n_0$$

ได้  $c = 1$  และ  $n_0 = 1$  ซึ่งทำให้

$$\log_2 n \leq n \quad , n \geq 1$$

ดังนั้น  $\log_2 n = O(n)$

# Proof: $3n^2+10 = O(n^2)$

Pf

$$3n^2+10 = O(n^2)$$

$$3n^2+10 = O(n^2)$$

$$3n^2+10 \leq 4n^2 \text{ when } n \geq 10$$

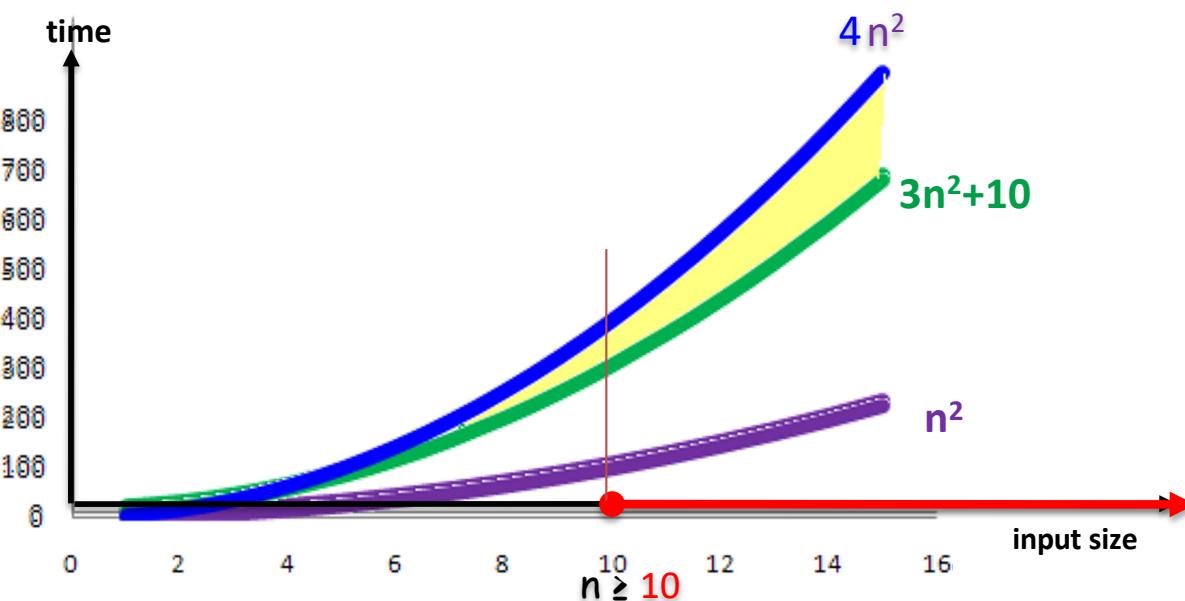
ต้องหา constants บวก  $C$  และ  $n_0$  ซึ่งทำให้

$$3n^2+10 \leq 4n^2 \text{ เมื่อ } n \geq n_0$$

$$3n^2+10 \leq 3n^2 + n^2 \text{ เมื่อ } n \geq 10$$

$$\therefore 3n^2+10 \leq 4n^2, n \geq 10$$

ดังนั้น  $3n^2+10 = O(n^2)$



Proof:  $a_k n^k + \dots + a_1 n + a_0 = O(n^k)$

$$a_k n^k + \dots + a_1 n + a_0 = O(n^k)$$

Proof : เลือก  $n_0 = 1$  และ  $c = |a_k| + \dots + |a_1| + |a_0|$

ต้องแสดงว่า

$$T(n) \leq cn^k \text{ เมื่อ } n \geq 1$$

เมื่อ  $n \geq 1$

$$T(n) \leq |a_k| n^k + \dots + |a_1| n + |a_0|$$

$$\begin{aligned} T(n) &\leq |a_k| n^k + \dots + |a_1| n^k + |a_0| n^k \\ &= c n^k \end{aligned}$$

ดังนั้น  $a_k n^k + \dots + a_1 n + a_0 = O(n^k)$

## Proof : $n^k \neq O(n^{k-1})$

ในการพิสูจน์ Big Oh โดยหาค่า constants บวก  $c$  และ  $n_0$  มีได้หลายคู่ เลือกคู่ใดก็ได้

แต่ หาก Big Oh  $g(n)$  ไม่เป็น asymptotical upper bound ของ  $f(n)$

จะหา constants บวก  $c$  และ  $n_0$  ไม่ได้เลยที่ทำให้  $f(n) \leq c g(n)$  เมื่อ ( $n \geq n_0$ )

เช่น  $n^k \neq O(n^{k-1})$

เพราะ ไม่มี constants บวก  $c$  และ  $n_0$  ซึ่งทำให้  $n^k \leq cn^{k-1}$  ( เมื่อ  $n \geq n_0$  )

Proof : โดย contradiction สมมติว่า  $n^k = O(n^{k-1})$

ดังนั้น ต้องมี constants บวก  $c$  และ  $n_0$  ซึ่งทำให้

$$n^k \leq cn^{k-1} \text{ เมื่อ } n \geq n_0$$

แต่ เมื่อตัด  $n^{k-1}$  ออกทั้งสองข้างจะได้

$$n \leq c \text{ เมื่อ } n \geq n_0$$

ซึ่ง เป็นไปไม่ได้

ดังนั้นสมมติฐานที่ให้ไว้ไม่จริง

ดังนั้น  $n^k \neq O(n^{k-1})$

# Big O Notation

คิดโดย Edmund Landau และ Paul Bachmann เพื่อใช้ในคณิตศาสตร์บริสุทธิ์

เริ่มพบโดยนักจำนวนชาวเยอรมัน แพยแพร ปี 1894

**Paul Gustav Heinrich Bachmann**

(22 June 1837 – 31 March 1920)



ได้รับความนิยมและขยายผล โดยนักจำนวนชาวเยอรมัน

**Edmund Georg Hermann Landau**

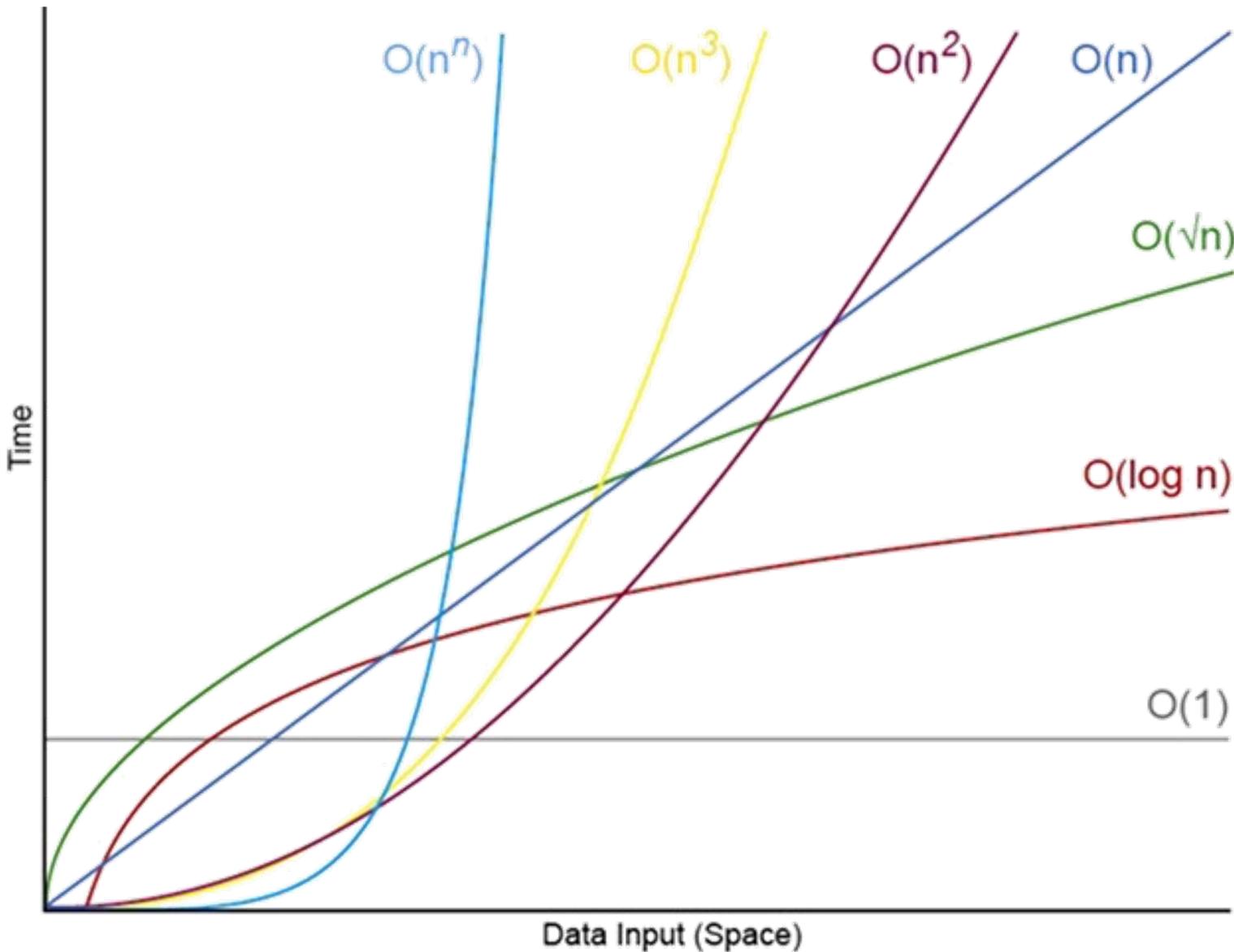
(14 February 1877 – 19 February 1938)



Big Oh notation จึงเรียกหลายแบบ

- Big-O notation
- Big Oh notation
- Landau notation
- Bachmann-Landau notation
- **Asymptotic notation**
- Big **Omicron** (ตัว **ο** ภาษากรีก) notation

# Classification of Algorithms



# Big-Oh & Related Notations

		<b>if there are positive constants c and <math>n_0</math> such that</b>	
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c g(n)$	
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c g(n)$	<b>when <math>n \geq n_0</math></b>
little Oh	$f(n) = o(g(n))$	$f(n) < c g(n)$ or $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$	
little Omega	$f(n) = \omega(g(n))$	$f(n) > c g(n)$ or $f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$	

Big Theta	$f(n) = \Theta(g(n))$	if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
-----------	-----------------------	--

# Big Oh O - Big Omega $\Omega$ - Big Theta $\theta$

		ถ้ามี constants บวก $c$ และ $n_0$ ซึ่ง เมื่อ $n \geq n_0$ และ	ใช้เมื่อ ต้องการพูดถึง running time
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c g(n)$ เสมอ	พูดถึง upper bound อย่างมากที่สุดเท่านี้ ไม่ซักกว่านี้ at most worst case
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c g(n)$ เสมอ	พูดถึง lower bound อย่างดี(เร็ว)ที่สุดเท่านี้ ไม่เร็วกว่านี้ at least best case
Big Theta	$f(n) = \Theta(g(n))$ ก็ต่อเมื่อ	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$	พูดถึง exact bound เป็นสัดสวนกับ $g(n)$

ตย. Binary Search มีลำดับเป็น

- $O(\log_2 n)$  เนื่องจาก worst case เป็น unsuccessful search ดังนั้นอย่างมากที่สุดมันจึง run ไม่นานกว่า  $\log_2 n$
- $\Omega(1)$  เนื่องจาก best case เป็น การหาพบในครั้งแรก
- กรณี worst case เป็น  $\Theta(\log_2 n)$

```
min = A[0]
i = 1
while i < n do
    if A[i] < min
        min = A[i]
return min
```

ตย. search array size  $n$  หา minimum element มี running time เป็น  $\Theta(n)$

# Limit Comparing

อีกวิธีที่ใช้เปรียบเทียบ  $f(n)$  และ  $g(n)$

เอา  $f(n) \div g(n)$  และ take lim เมื่อ  $n \rightarrow \infty$

1. 0  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

$f(n)$  โตช้ากว่า  $g(n)$ ,

$$f(n) = o(g(n)), g(n) = \omega(f(n))$$

ถ้าค่า lim แกร่ง ใช้วิธีนี้ไม่ได้

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

ดูผลที่ได้ 3 กรณี

$$f(n) = 3n^2 - 100n - 25 \quad g(n) = n^3$$

$$\lim_{n \rightarrow \infty} \frac{3n^2 - 100n - 25}{n^3} = \lim_{n \rightarrow \infty} \left( \frac{3}{n} - \frac{100}{n^2} - \frac{25}{n^3} \right) = 0$$

2.  $\infty$

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

$f(n)$  โตเร็วกว่า  $g(n)$ ,

$$f(n) = \omega(g(n)), g(n) = o(f(n))$$

$$f(n) = 3n^2 - 100n - 25 \quad g(n) = n$$

$$\lim_{n \rightarrow \infty} \frac{3n^2 - 100n - 25}{n} = \lim_{n \rightarrow \infty} \left( 3n - 100 - \frac{25}{n} \right) = \infty$$

3.  $c, c \neq 0, c \neq \infty$   $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty$

โตเร็วพอกัน,  $f(n) = \Theta(g(n))$

$$f(n) = 3n^2 - 100n - 25 \quad g(n) = n^2$$

$$\lim_{n \rightarrow \infty} \frac{3n^2 - 100n - 25}{n^2} = \lim_{n \rightarrow \infty} \left( 3 - \frac{100}{n} - \frac{25}{n^2} \right) = 3$$

## กฎของโลปิตอล (L'Hôpital 's rule)

ถ้า  $\lim_{n \rightarrow \infty} f(n) = \infty$  และ  $\lim_{n \rightarrow \infty} g(n) = \infty$  แล้ว

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

โดย  $f'(n)$  และ  $g'(n)$  คือ  
derivatives ของ  $f(n)$  และ  $g(n)$  ตามลำดับ

# $O(1)$ Constant Order

Constant time runtime คงที่ ไม่ขึ้นกับ input size n

Examples:

a[i]  
push(), pop() fix size stack

```
int sum(int n) {  
    sum = 0;  
    for( int i=1 ; i<=n; i++)  
        sum = sum + i;  
    return sum;  
}
```

เฉพาะที่ highlight สีฟ้าเป็น Constant time  
runtime คงที่ ไม่ขึ้นกับ input size n

# $O(n)$ Linear Order (Linear Search)

Linear time    runtime トイเป็น linear กับ  $n$  ที่เพิ่ม

Linear Search    Search หา  $x$  จาก array หรือ linear linked list size  $n$

กี่ probes ?

=  $x$  ?    =  $x$  ?    =  $x$  ?    =  $x$  ?    =  $x$  ?    =  $x$  ?    =  $x$  ?

probes =  $n$  times (worst case)

1                  2                  3                   $\dots$                    $n$

```
int search(int x, int a[], int n)
{   int i = 0;
    while((i<n) && (a[i] !=x)) {
        i++;
    if(i!=n) return i;
        return -1;
}
```

int i = search(5, a, n);

1	3
2	1
3	max $n+1$ max $n$
4	1
5	1
6	1

Loop :

$x$  จำนวนครั้งของลูป

$$= 3n + 6 = O(n)$$

Consecutive statements  
คิดที่ทำมากที่สุด  $n > \text{constant } 6$

# $O(n^3)$ Cubic Order

```
int maxSubSum1(int a[], int N) {  
    int maxSum = 0;  
    for(int i=0; i<N; i++)  
        for(int j=i; i<N; j++) {  
            int thisSum = 0;  
            for(int k=i; k<=j; k++)  
                thisSum += a[k];  
            if(thisSum > maxSum)  
                maxSum = thisSum;  
        }  
    return maxSum;  
}
```

อะไรทำมากที่สุด ?

1 loop size  $N$   
2 loop size  $N-i$  (worst case  $N$ )  
3  
4 loop size  $j-i+1$  (worst case  $N$ )  
5 line 5  $\times n \times n \times n$

$= O(N^3)$

# $O(\log n)$ Logarithmic Order

Logarithmic time runtime โตเป็น log กับ n ที่เพิ่ม

```

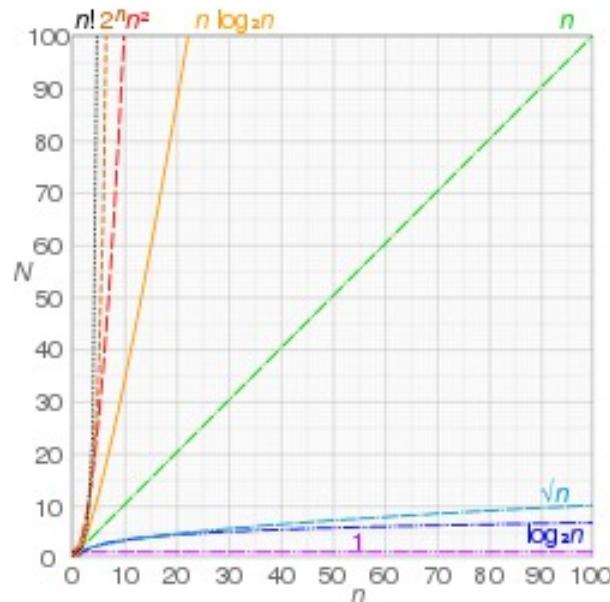
int log_2(int n) { //n>=1
    int times = 0;
    for (int i = n; i >= 2; i = i / 2) {
        times++;
    }
    return times;
}
  
```

อะไรทำมากที่สุด ?

loop      ทำ loop กี่ครั้ง ?

เริ่ม ?       $i = n$   
จบ ?       $i = 1$

ทุกครั้ง sizeลดลง  $\frac{1}{2}$



$O(\log n)$

$\log_x n = \log_y * \log_y n$   
( $\log_y$  เป็น constant)  
จึงไม่นิยมเขียนฐาน log

$$1 = n / 2^d$$
$$2^d = n$$
$$d = \log_2 n$$

ครั้งที่	เหลือ size	
1	$n/2$	$n/2^1$
2	$n/4$	$n/2^2$
3	$n/8$	$n/2^3$
...	...	...
d	1	$n/2^d$

# $O(\log n)$ Euclid's Algorithm

```
// O(log n)
long gcd( long m, long n ) {
    while( n != 0 ) {
        long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

**gcd(1989, 1590)**

$M_1 = 1,989$	$N_1 = 1,590$	$M_1 \% N_1 = 399$
$M_2 = 1,590$	$N_2 = 399$	$M_2 \% N_2 = 393$
$M_3 = 399$	$N_3 = 393$	$M_3 \% N_3 = 6$
$M_4 = 393$	$N_4 = 6$	$M_4 \% N_4 = 3$
$M_5 = 6$	$N_5 = 3$	$M_5 \% N_5 = 0$
$M_6 = 3 = \text{gcd}$	$N_6 = 0$	

แม้จะไม่ได้ใช้เวลาคงที่  $O(1)$  ในการลดขนาดของปัญหา  
แต่สามารถพิสูจน์ได้ว่า จำนวนรอบมากที่สุด =  $O(\log N)$

If  $M > N$  then  $(M \bmod N) < M/2$

Case 1: if  $N \leq M/2 \rightarrow (M \bmod N) < N \leq M/2$  เช่น  $M = 10, N = 4 \therefore 10 \% 4 < N < M/2$

Case 2: if  $N > M/2 \rightarrow (M \bmod N) < M-N < M/2$  เช่น  $M = 10, N = 6 \therefore 10 \% 6 < 10-6 < M/2$

# Simple Recursive

Recursive Function : หลายนรรณี

- recursion ทำแค่ for loop วิเคราะห์เหมือน for loop  $\rightarrow O(n)$

//  $O(n)$

```
long  fact(int n) {  
    if (n<=1)  
        return 1;  
    else  
        return n * fact(n-1);  
}
```

//  $O(2^n)$

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

Q&A

Q&A