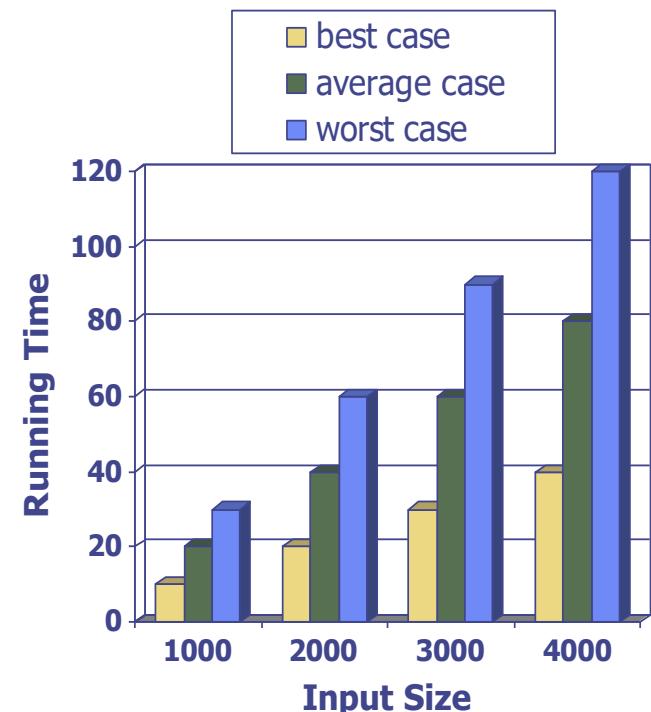


Complexity

Lecturer : Kiatnarong Tongprasert

การวัดเวลาการทำงาน

- ผลลัพธ์จากการทำงานส่วนมากจะเปลี่ยนจำนวนข้อมูล input ไปเป็นจำนวนข้อมูล output
- เวลาที่ใช้ในการทำงานส่วนมากจะเพิ่มขึ้นตามจำนวนข้อมูล input
- ค่าเวลาเฉลี่ยเป็นสิ่งที่ยากในการหาค่า
- ดังนั้นจึงพิจารณาเวลาการทำงานในกรณีที่แย่ที่สุด ซึ่งมีข้อดีคือ
 - ง่ายในการวิเคราะห์
 - มีบทบาทสำคัญในการพัฒนาแอ��พพลิเคชัน เช่น โปรแกรมเกมส์ โปรแกรมด้านการเงิน และ การการควบคุมหุ่นยนต์

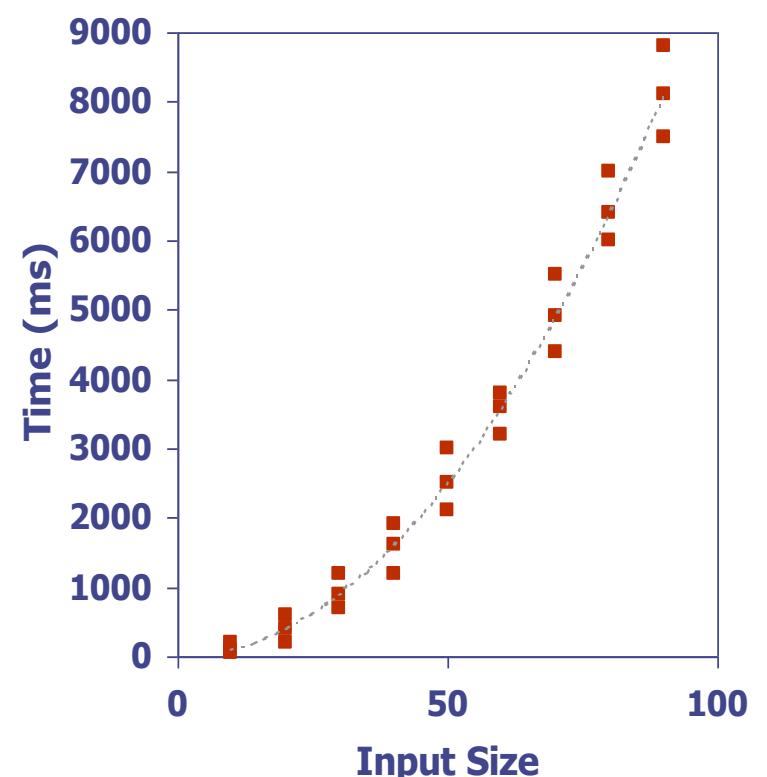


ทดลองวัดเวลาการทำงานของอัลกอริทึม

- เขียนโปรแกรมที่สร้างตามอัลกอริทึมนั้น
- ให้โปรแกรมทำงานโดยป้อนค่า input ที่มีจำนวนและชนิดที่แตกต่างกัน เพื่อดูเวลาที่ใช้

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time - start_time
```

- แสดงเป็นกราฟระหว่างจำนวน input กับเวลาที่ใช้



ข้อจำกัดของการวัดเวลาด้วยการทดลอง

- ต้องสร้างโปรแกรมขึ้นมาจากการอัลกอริทึมนั้น ๆ ซึ่งอาจยากก็ได้
- ผลลัพธ์อาจไม่แสดงถึงเวลาที่ใช้กับ input อื่นๆ ที่ไม่รวมอยู่ในการทดสอบ
- หากต้องการเปรียบเทียบ 2 อัลกอริทึม ต้องใช้สภาพแวดล้อม ฮาร์ดแวร์และซอฟต์แวร์เดียวกัน



การวิเคราะห์เชิงทฤษฎี

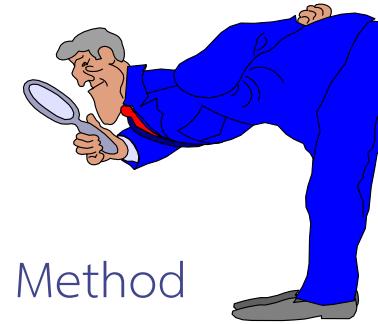


- ใช้การอธิบายระดับสูงแทนการลงมือเขียนโปรแกรม
- โดยแสดงลักษณะของเวลาการทำงานเป็นฟังก์ชันของขนาด input ตามจำนวนข้อมูล n ตัว
- ด้วยการคำนึงถึงโอกาสที่ input จะเป็นไปได้ทั้งหมด
- สามารถประเมินความเร็วของอัลกอริทึมที่ไม่ขึ้นกับสภาพแวดล้อมของฮาร์ดแวร์/ซอฟต์แวร์

Pseudocode

- คำอธิบายระดับสูงของอัลกอริทึม
- มีลักษณะเป็นโครงสร้างมากกว่าประโยคภาษาอังกฤษทั่วไป
- มีรายละเอียดน้อยกว่าการเขียนโปรแกรม
- ชอนปัญหาการออกแบบโปรแกรม

Pseudocode Details



- การควบคุมเส้นทาง
 - **if ... then ... [else ...]**
 - **while ... do ...**
 - **repeat ... until ...**
 - **for ... do ...**
 - ใช้การเยื่องแท่นวงเล็บ
- การประกาศ method

Algorithm **method (arg [, arg...])**

Input ...

Output ...

- การเรียก Method
method (arg [, arg...])
- การส่งค่ากลับ
return expression
- นิพจน์
 - ← การกำหนดค่า
 - = การทดสอบการเท่ากัน
 - n^2 อนุญาตให้ใช้ตัวยกและการจัดรูปแบบทางคณิตศาสตร์อื่นๆ ได้

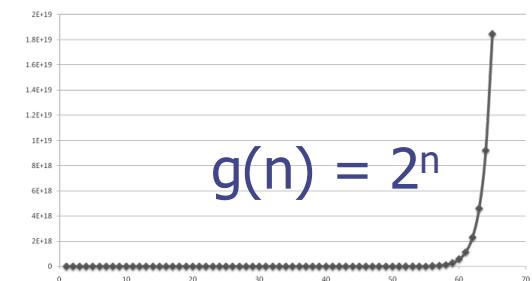
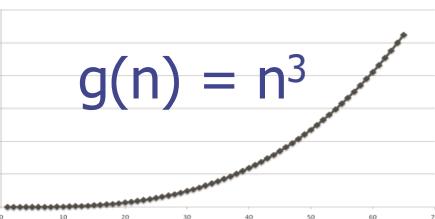
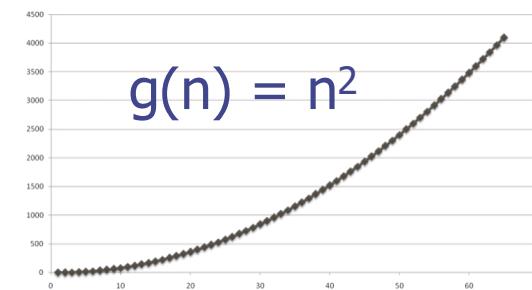
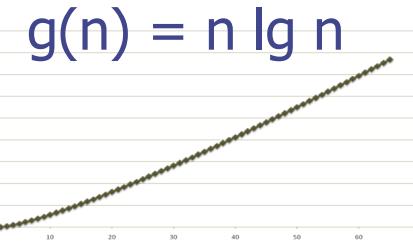
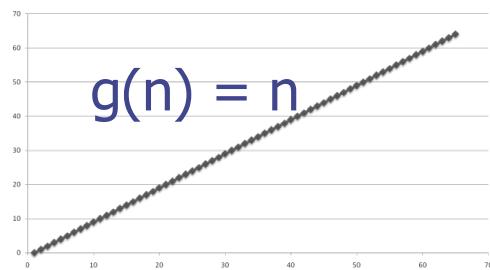
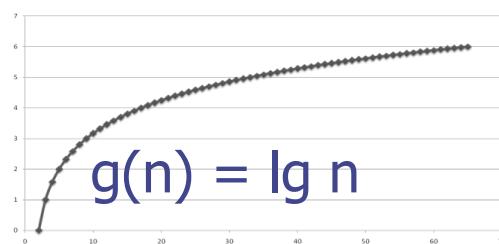
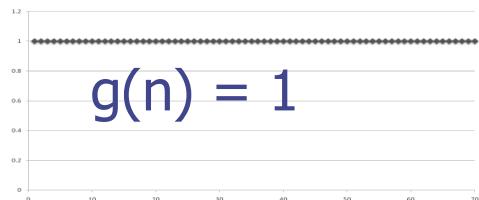
ฟังก์ชันหลักที่พบบ่อยในการวิเคราะห์อัลกอริทึม

□ ฟังก์ชันที่พบบ่อยในการวิเคราะห์อัลกอริทึม:

- Constant ≈ 1
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$

□ จาก log-log chart ความชันของเส้นกราฟแสดง
อัตราการเติบโต (growth rate)

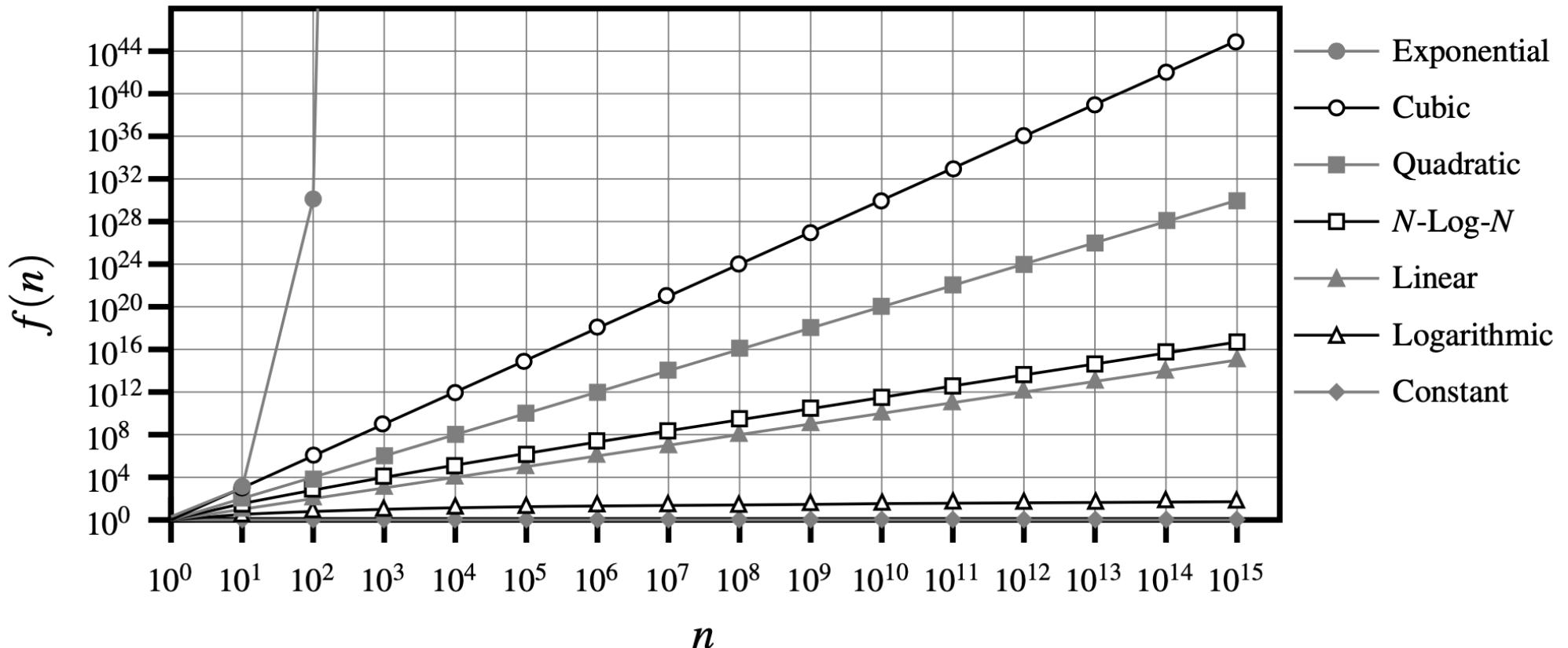
ฟังก์ชันหลักที่พบบ่อยในการวิเคราะห์อัลกอริทึม



Using “Normal” Scale

Slide by Matt Stallmann
included with permission.

พังก์ชันหลักที่พบบ่อยในการวิเคราะห์อัลกอริทึม



Primitive Operations (การทำงานพื้นฐาน)



- อัลกอริทึมเขียนด้วยการคำนวณพื้นฐาน
- เขียนแบบ pseudocode
- เป็นอิสระจากภาษาเขียนโปรแกรม
- ไม่ต้องมีคำจำกัดความที่แน่นอน
- ให้ถือว่าใช้เวลาคงที่ในทุกประเภทเครื่องคำนวณ
- ตัวอย่างเช่น:
 - นิพจน์การคำนวณ
 - การกำหนดค่าให้ตัวแปร
 - การซื้อข้อมูลในอาร์เรย์
 - การเรียกเมธอด
 - การส่งค่ากลับจากเมธอด

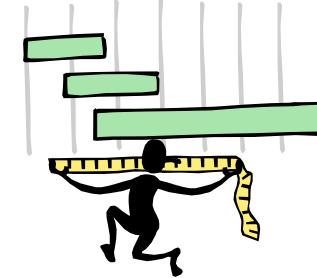
Counting Primitive Operations

- การพิจารณาการทำงานใน pseudocode ทำให้สามารถคำนวณค่ามากสุดในการทำงานตามคำสั่งพื้นฐานเพื่อทำงานของฟังก์ชันตามจำนวนอินพุท

```
1 def find_max(data):
2     """Return the maximum element from a nonempty Python list."""
3     biggest = data[0]                      # The initial value to beat
4     for val in data:                       # For each value:
5         if val > biggest:                 # if it is greater than the best so far,
6             biggest = val                  # we have found a new best (so far)
7     return biggest                         # When loop ends, biggest is the max
```

- Step 1: 1 ops 3: 1 ops 4: n ops
 5: n ops 6: 0 to n ops 7: 1 op

Estimating Running Time



□ อัลกอริทึม `find_max` ใช้เวลาสำหรับกรณีแย่ที่สุด $3n + 3$

และใช้เวลากรณีดีที่สุด $2n + 3$ กำหนดให้:

a = เวลาที่ดีที่สุด

b = เวลาที่แย่ที่สุด

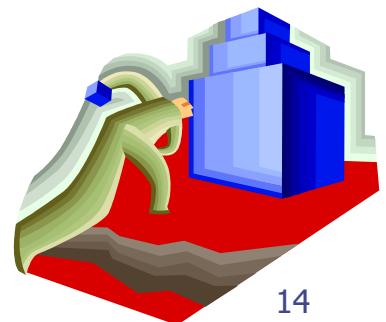
□ ให้ $T(n)$ เป็นเวลาที่แย่ที่สุดของ `find_max`.

$$a(2n + 3) \leq T(n) \leq b(3n + 3)$$

□ ดังนั้น เวลาที่ใช้ในการทำงาน $T(n)$ อยู่ในขอบเขตของฟังก์ชันแบบเส้นตรง

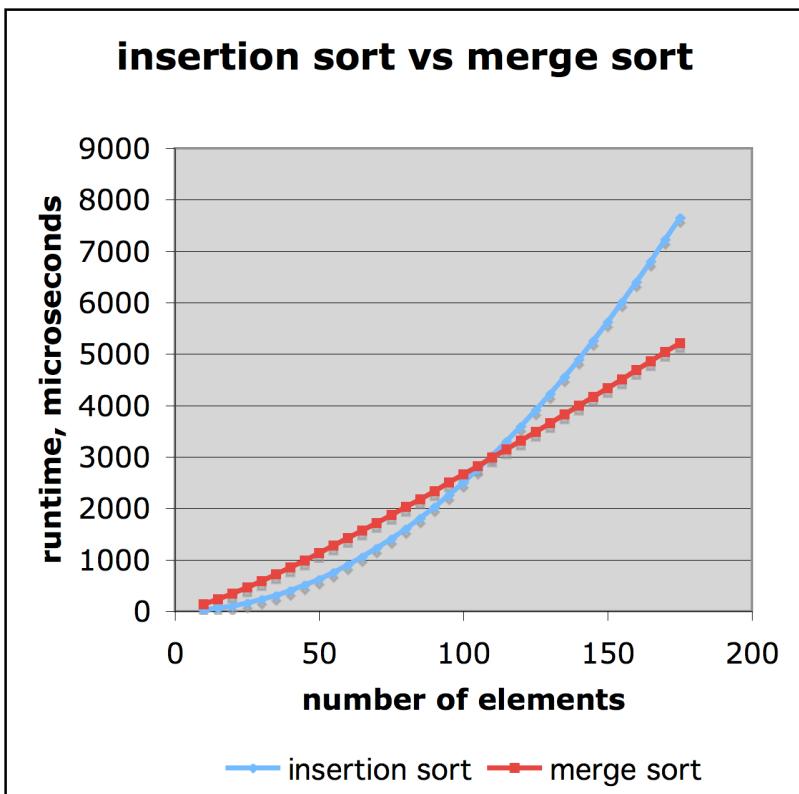
Growth Rate of Running Time

- การเปลี่ยนฐานรากแวร์หรือซอฟต์แวร์
 - มีผลต่อ $T(n)$ ในลักษณะค่าคงที่
 - ไม่เปลี่ยนแปลงอัตราการเติบโตของ $T(n)$
- อัลกอริทึม `find_max` มีอัตราการเติบโตของเวลาในการทำงานแบบเชิงเส้น



Slide by Matt Stallmann
included with permission.

Comparison of Two Algorithms



insertion sort is

$$n^2 / 4$$

merge sort is

$$2n \log n$$

sort a million items?

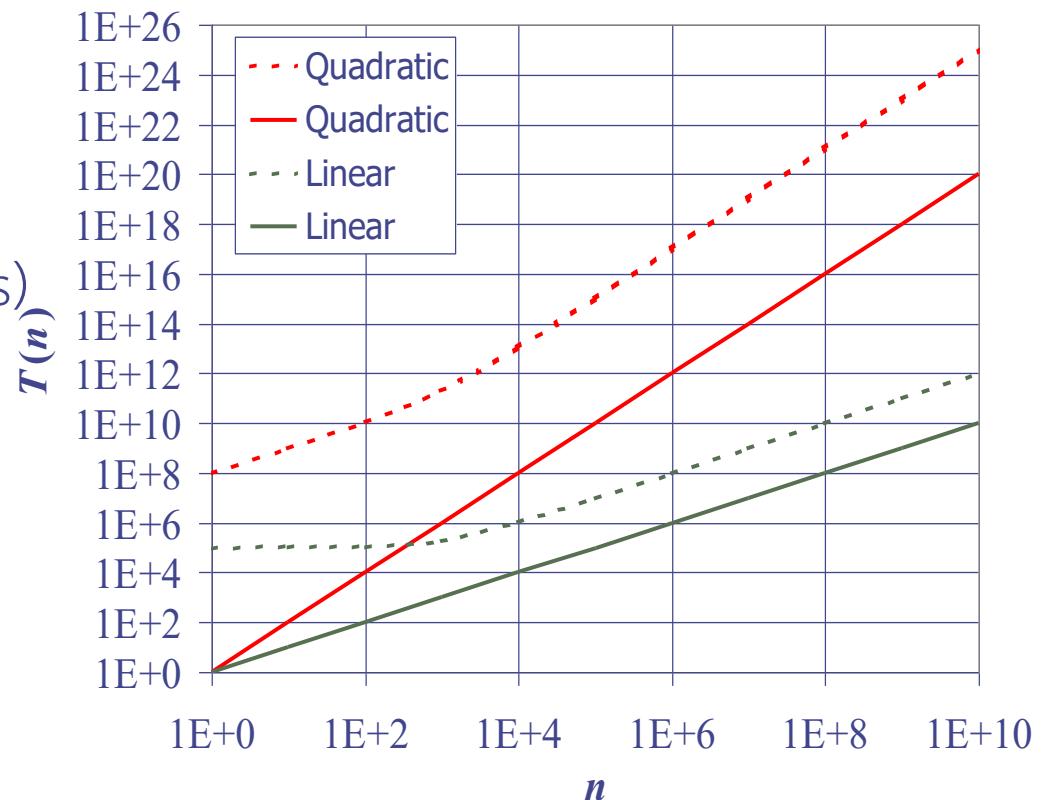
insertion sort ใช้เวลาประมาณ 70 ชม.

merge sort ใช้เวลาประมาณ 40 วินาที

สำหรับเครื่องที่เร็วขึ้น เวลาที่ใช้คือ 40 นาที
กับ น้อยกว่า 0.5 วินาที

Constant Factors

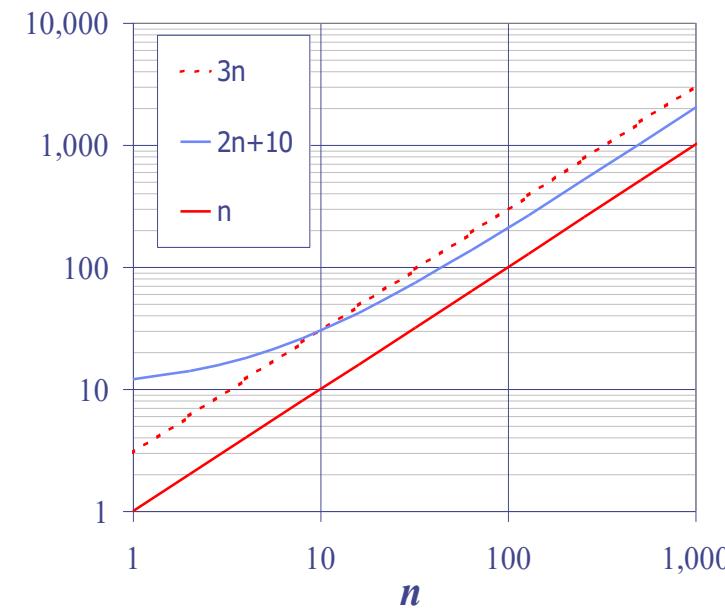
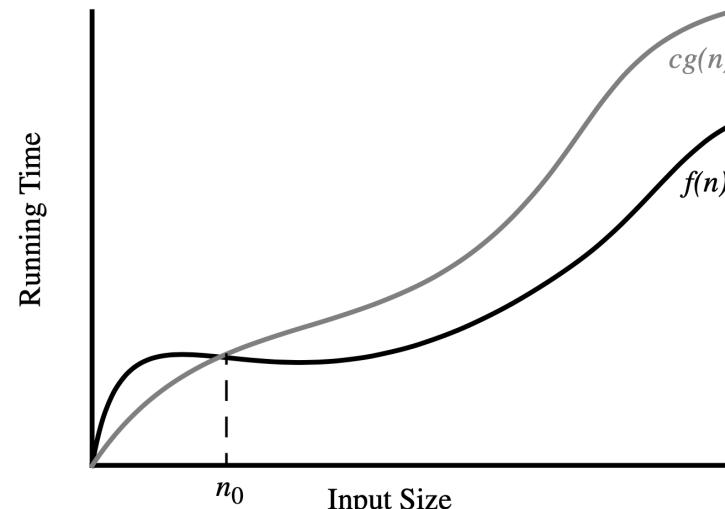
- อัตราการเติบโต ไม่ขึ้นกับปัจจัยต่อไปนี้
 - ค่าคงที่
 - เทอมที่มีดีกรีน้อยกว่า (lower-order terms)
- ตัวอย่าง
 - $10^2n + 10^5$ เป็นฟังก์ชันเชิงเส้น
 - $10^5n^2 + 10^8n$ เป็นฟังก์ชัน quadratic



Big-Oh Notation

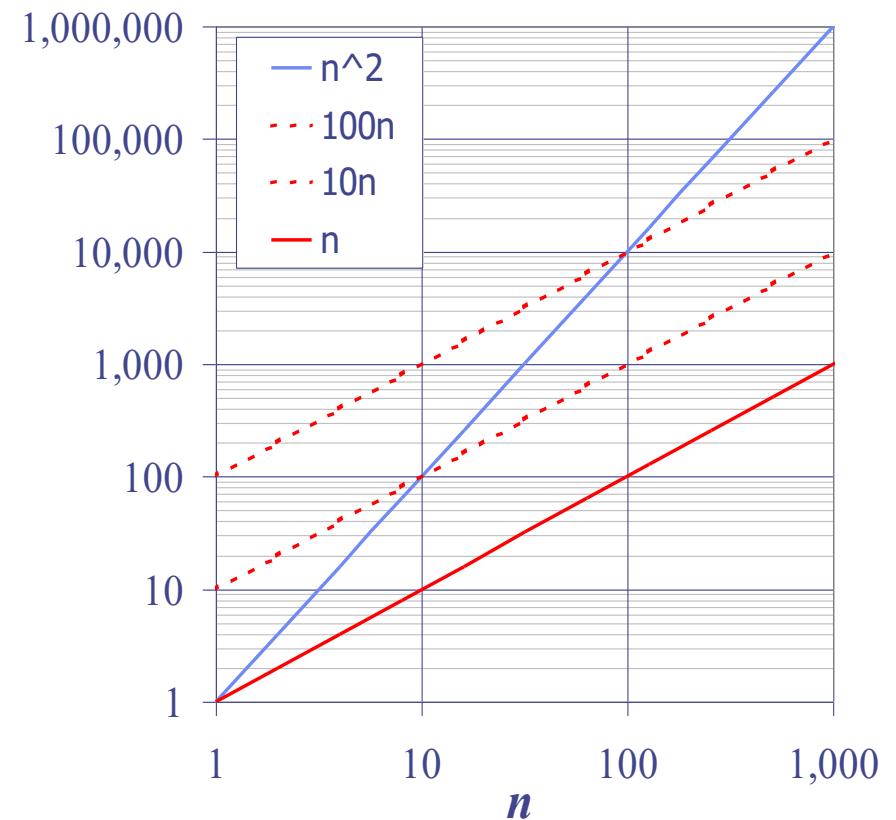
- ให้ฟังก์ชัน $f(n)$ และ $g(n)$, จะเรียก
$$f(n) = O(g(n)) \text{ ถ้ามีค่าคงที่} c \text{ และ } n_0 \text{ ที่ทำให้}$$
$$f(n) \leq cg(n) \text{ สำหรับ } n \geq n_0$$

- ตัวอย่าง: $2n + 10 = O(n)$
 - $2n + 10 \leq cn$
 - $(c - 2)n \geq 10$
 - $n \geq 10/(c - 2)$
 - เลือก $c = 3$ และ $n_0 = 10$



Big-Oh Example

- ตัวอย่าง: ฟังก์ชัน n^2 ไม่เท่ากับ $O(n)$
 - $n^2 \leq cn$
 - $n \leq c$
 - ไม่สามารถแก้สมการด้านบนได้เนื่องจาก หากค่าคงที่ c ที่มากกว่า n ไม่ได้



More Big-Oh Examples

◆ $7n^2$

$$7n^2 = O(n)$$

ต้องการ $c > 0$ และ $n_0 \geq 1$ ที่ทำให้ $7n^2 \leq c \cdot n$ สำหรับ $n \geq n_0$ เป็นจริงเมื่อ $c = 7$ และ $n_0 = 1$



◆ $3n^3 + 20n^2 + 5$

$$3n^3 + 20n^2 + 5 = O(n^3)$$

ต้องการ $c > 0$ และ $n_0 \geq 1$ ที่ทำให้ $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ สำหรับ $n \geq n_0$ เป็นจริง
เมื่อ $c = 4$ และ $n_0 = 21$

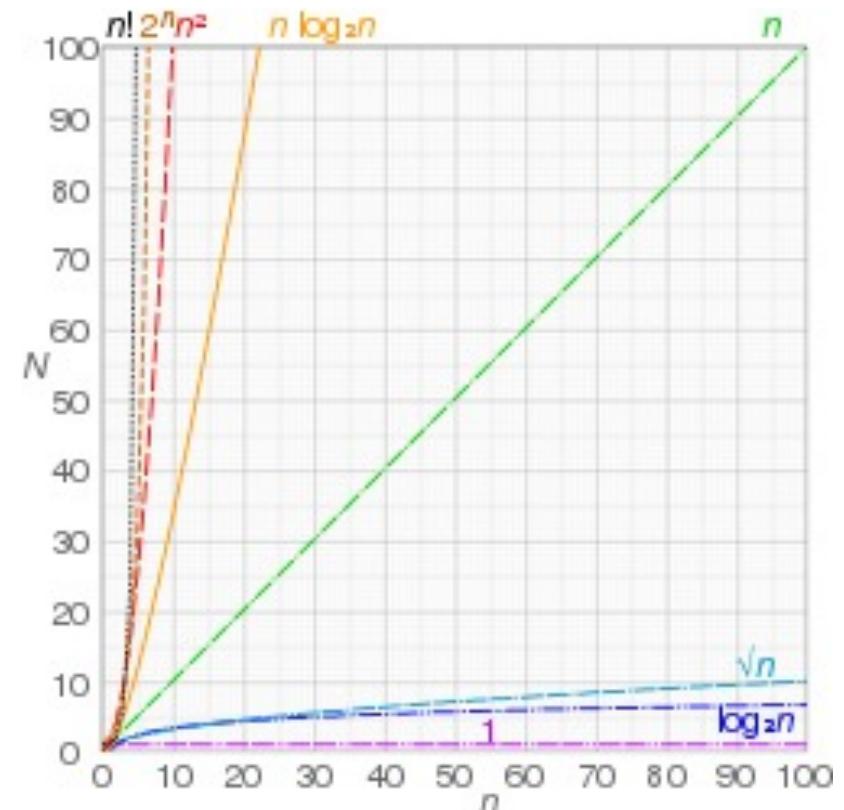
◆ $3 \log n + 5$

$$3 \log n + 5 = O(\log n)$$

ต้องการ $c > 0$ และ $n_0 \geq 1$ ที่ทำให้ $3 \log n + 5 \leq c \cdot \log n$ สำหรับ $n \geq n_0$ เป็นจริง
เมื่อ $c = 8$ และ $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation แสดงค่าขอบเขตบน upper bound ของอัตราการเติบโตของฟังก์ชัน
- ข้อความที่กล่าวว่า “ $f(n) = O(g(n))$ ” หมายความว่า อัตราการเติบโตของ $f(n)$ ไม่มากกว่าอัตราการเติบโตของ $g(n)$
- สามารถใช้ the big-Oh notation เพื่อจัดลำดับฟังก์ชันตามอัตราการเติบโต



Big-Oh Rules

□ ถ้า $f(n)$ เป็นพหุนามดีกรี d และ $f(n) = O(n^d)$

1. ตัดเทอมที่ดีกรีน้อยกว่าออก
2. ตัดค่าคงที่ออก

□ ใช้ตัวแทนที่ค่าน้อยที่สุดในกลุ่มของฟังก์ชัน

- เช่น “ $2n = O(n)$ ” แทนที่ “ $2n = O(n^2)$ ”

□ ใช้นิพจน์ในรูปแบบง่าย

- เช่น “ $3n + 5 = O(n)$ ” แทนที่ “ $3n + 5 = O(3n)$ ”

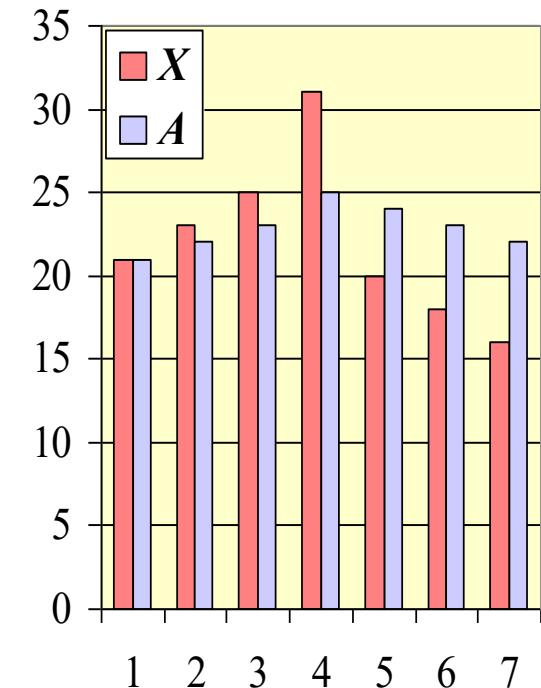


Asymptotic Algorithm Analysis

- การวิเคราะห์เชิงซีมโถติกของอัลกอริทึมกำหนดเวลาทำงานในรูปแบบ big-Oh
- การวิเคราะห์เชิงซีมโถติก
 - คนหาจำนวนคำสั่งพื้นฐานในการทำงานกรณีที่แย่ที่สุดของอัลกอริทึมในรูปแบบของฟังก์ชันที่สัมพันธ์กับจำนวน input
 - แสดงฟังก์ชันด้วยสัญกรณ์ big-Oh
- ตัวอย่าง:
 - จากอัลกอริทึมสามารถแสดงได้ว่า find_max “ใช้เวลาการทำงาน $O(n)$ ”
- เนื่องจากค่าคงที่ และ เทอมในลำดับรองลงมาจะมีความสำคัญลดลง จึงไม่นำคำนึงถึงในการนับจำนวนคำสั่งพื้นฐานในการทำงานของอัลกอริทึมนั้น

Computing Prefix Averages

- ขอแสดงการวิเคราะห์เชิงซีมโทติกกับอัลกอริทึมสำหรับคำ prefix averages
- โดยที่ i -th เป็น prefix average ของลิสต์ X คือค่าเฉลี่ยของข้อมูล $(i + 1)$ ตัวแรกของ X :
$$A[i] = (X[0] + X[1] + \dots + X[i])/(i+1)$$
- การคำนวณคำ prefix averages ของลิสต์ X ใช้ในแอพพลิเคชันเกี่ยวกับการวิเคราะห์ด้านการเงิน



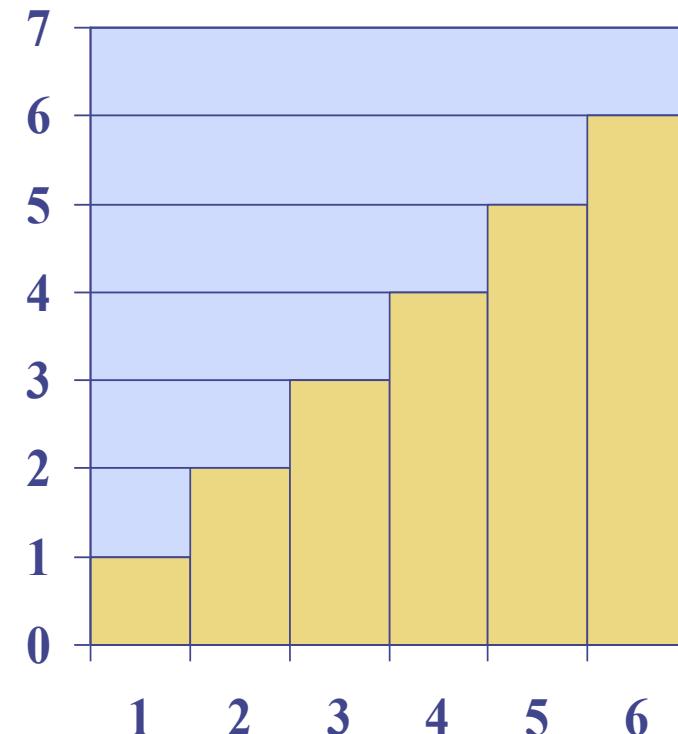
Prefix Averages (Quadratic)

- ◆ อัลกอริทึมต่อไปนี้แสดงการแสดงการหาค่า prefix averages โดยใช้เวลาแบบ quadratic

```
1 def prefix_average1(S):
2     """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n
5     for j in range(n):
6         total = 0
7         for i in range(j + 1):
8             total += S[i]
9         A[j] = total / (j+1)
10    return A
```

Arithmetic Progression

- เวลาที่ใช้สำหรับ *prefixAverage1* คือ $O(1 + 2 + \dots + n)$
- การบวกเลขจำนวนเต็ม n ตัวแรก คำนวณได้จากสูตร $n(n + 1) / 2$
- ดังนั้นน้อยกว่าอัลกอริทึม *prefixAverage1* ใช้เวลา $O(n^2)$



Prefix Averages 2 (Looks Better)

◆ อัลกอริทึมต่อไปนี้เขียนด้วยภาษาไพทอนทำให้เข้าใจง่ายขึ้น

```
1 def prefix_average2(S):
2     """ Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n                      # create new list of n zeros
5     for j in range(n):
6         A[j] = sum(S[0:j+1]) / (j+1)  # record the average
7     return A
```

◆ อัลกอริทึม *prefixAverage2* ยังคงใช้เวลา $O(n^2)$!

Prefix Averages 3 (Linear Time)

◆ อัลกอริทึมต่อไปนี้คำนวณค่า prefix averages ที่ใช้เวลาแบบเชิงเส้น

```
1 def prefix_average3(S):
2     """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n                      # create new list of n zeros
5     total = 0                         # compute prefix sum as S[0] + S[1] + ...
6     for j in range(n):
7         total += S[j]                 # update prefix sum to include S[j]
8         A[j] = total / (j+1)          # compute average based on current sum
9     return A
```

◆ อัลกอริทึม *prefixAverage3* ใช้เวลา $O(n)$

Math you need to Review



- ◆ Summations
 - ◆ Logarithms and Exponents
 - ◆ Proof techniques
 - ◆ Basic probability
- **properties of logarithms:**
 $\log_b(xy) = \log_bx + \log_by$
 $\log_b(x/y) = \log_bx - \log_by$
 $\log_bxa = a\log_bx$
 $\log_ba = \log_xa/\log_xb$
 - **properties of exponentials:**
 $a^{(b+c)} = a^b a^c$
 $a^{bc} = (a^b)^c$
 $a^b / a^c = a^{(b-c)}$
 $b = a^{\log_a b}$
 $b^c = a^{c * \log_a b}$

Relatives of Big-Oh



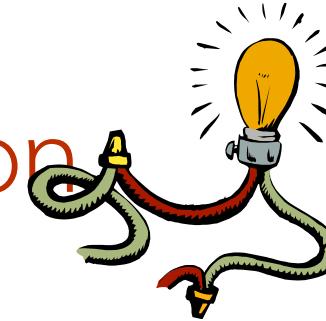
◆ big-Omega

- $f(n) = \Omega(g(n))$ ถ้ามีค่าคงที่ $c > 0$ และค่าจำนวนเต็มคงที่ $n_0 \geq 1$ ที่ทำให้
 $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

◆ big-Theta

- $f(n) = \Theta(g(n))$ ถ้ามีค่าคงที่ $c' > 0$ และ $c'' > 0$ และค่าจำนวนคงที่ $n_0 \geq 1$ ที่ทำให้ $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

สรุป Asymptotic Notation



Big-Oh

- $f(n) = O(g(n))$ ถ้า $f(n)$ น้อยกว่าหรือเท่ากับ (ไม่มากไปกว่า) $g(n)$

big-Omega

- $f(n) = \Omega(g(n))$ ถ้า $f(n)$ มากกว่าหรือเท่ากับ (ไม่น้อยไปกว่า) $g(n)$

big-Theta

- $f(n) = \Theta(g(n))$ ถ้า $f(n)$ เท่ากับ $g(n)$

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
```

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

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
= runtime เดิม $\times 4$

Example : $\log_2 n$

กำหนด n = จำนวนผู้เข้าแข่งขัน

คัดออกทีละครึ่งจนเหลือผู้ชนะ 1 คน ต้องแข่งกี่รอบ ?

นิยามของ \log :

$\log_2 n = x \rightarrow 2^x = n \rightarrow$ 2 คูณกันกี่ครั้งได้ $n \rightarrow n$ ถูกหารด้วย 2 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)

O(1) Constant Order

Constant time runtime คงที่ ไม่ขึ้นกับ input size n

Examples:

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

```
def sum(n):  
    sum = 0  
    for i in range(n)  
        sum += i  
    return sum
```

เฉพาะที่ **high light** สีฟ้าเป็น **Constant time** runtime คงที่ ไม่ขึ้นกับ input size n

O(n) Linear Order (Linear Search)

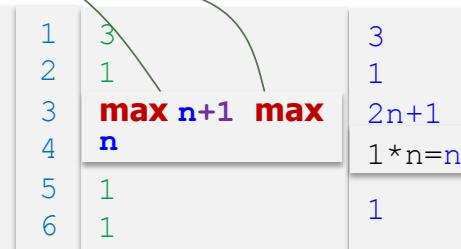
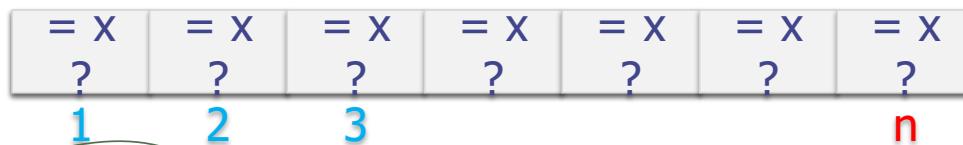
Linear time runtime โตเป็น linear กับ n ที่เพิ่ม

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

กี่ probes ?

probes = n times (worst case)

```
search(x, a, n) :  
    i = 0  
    while((i<n) and (a[i]!=x)):  
        i += 1  
    if i!=n :  
        return i  
    return -1
```



Loop :
 x จำนวนครั้งของลูป

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

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

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

$O(n^3)$ Cubic Order

```
def maxSubSum1(a, N):
    maxSum = 0
    for i in range(N):
        for j in range(i,N):
            thisSum = 0;
            for k in range(i,j+1):
                thisSum += a[k]
            if thisSum > maxSum :
                maxSum = thisSum
    return maxSum
```

อะไรทำมากที่สุด ?

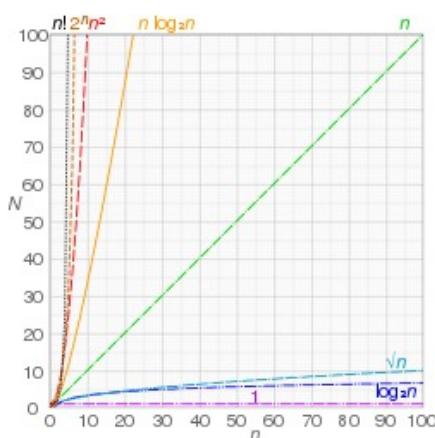
1
2
3
4
5

loop size N
loop size $N-i$ (worst case N)
loop size $j-i+1$ (worst case N)
line 5 $x \ n \ x \ n \ x \ n$ $= O(N^3)$

O(log n) Logarithmic Order

Logarithmic time runtime โตเป็น log กับ n ที่เพิ่ม

```
def log_2(n): //n>=1
    times = 0
    i = n
    while i >= 2 :
        i /= 2
        time += 1
    return times
```



O(log n)

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

อะไรทำมากที่สุด ?

loop

ทำ loop กี่ครั้ง ?

เริ่ม ?
จบ ?

i = n
i = 1

ทุกครั้ง sizeลดลง $\frac{1}{2}$

$$\begin{aligned} 1 &= n / 2^d \\ 2^d &= n \\ d &= \log_2 n \end{aligned}$$

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

$O(\log n)$

```
def gcd(m, n):  
    while n != 0:  
        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 $M = 10, N = 6 \therefore 10 \% 6 < 10-6 < M/2$

O(n) Simple Recursive

Recursive Function : หลักการนี้

- recursion ทำแค่ for loop วิเคราะห์เหมือน for loop → O(n)

O(n)

```
def fac(n):  
    if n<=1:  
        return 1  
    else  
        return n * fact(n-1)
```

```
def fib(n):  
    if n<=1:  
        return n  
    else  
        return fib(n-1) + fib(n-2)
```

เป็นอีกกรณีหนึ่ง →

$O(c^n)$ Exponential Order

```
def fib(n) :
    if n<=1
        return n
    return fib(n-1) + fib(n-2)
```

1
2
3

ให้ run time เมื่อเรียก $\text{fib}(n) = T(n)$ และค่า $\text{fib } n$ แทนด้วย $\text{fib}[n]$ จะ Pf. ว่า $T(n) \geq \text{fib}[n]$ นั่นคือ $T(n) = O(x^n)$ โตเป็น exponential

- A. $T(0) = T(1) = 1$ (จาก code)
- B. $T(n) = T(n-1) + T(n-2) + c$ when $n \geq 2$ (จาก code)

pf. base case: ว่า $T(n) \geq \text{fib}[n]$ เมื่อ $n=0, n=1$

Pf. $1 = T(0) \geq \text{fib}[0] = 0, 1 = T(1) \geq \text{fib}[1] = 1$ (จาก A)

pf. induction case: ถ้าให้ $T(n) \geq \text{fib}[n], T(n-1) \geq \text{fib}[n-1]$ เป็นจริง จะต้องพิสูจน์ว่า $T(n+1) \geq \text{fib}[n+1]$ เป็นจริง

$$\begin{aligned} 1. \quad T(n+1) &= T(n) + T(n-1) + c \quad (\text{จาก B}) \\ &\geq \quad \geq \quad \geq \end{aligned}$$

$$2. \quad \text{fib}[n+1] = \text{fib}[n] + \text{fib}[n-1] \quad (\text{จากกฎของ fib})$$

$T(n+1) \geq \text{fib}[n+1]$ จริง เพราะข้างบนของสมการบันมากกว่าข้างขวาสมการล่าง \rightarrow ซ้ายบน \geq ซ้ายล่าง : induction case เป็นจริง

- ก) $T(n) \geq \text{fib}[n]$ for $n \geq 0$ (จากการพิสูจน์แบบ induction)
- ข) $\text{fib}[n] > (3/2)^n$ for $n > 10$ (สามารถพิสูจน์แบบ induction ได้)
- ค) $T(n) \geq \text{fib}[n] \geq (3/2)^n$ for $n > 10$ (จาก ก) & ข))

$T(n) = O(c^n)$
โตเป็น exponential

การพิสูจน์โดย Mathametical Induction ต้องพิสูจน์ 2 กรณี

1. base case พิสูจน์ว่ากรณี n ตั้งแต่น้อยๆ เป็นจริง เช่น $n = 0, 1$
2. induction case พิสูจน์ว่า ถ้าที่ $n \leq x$ ใดๆ เป็นจริงแล้ว หากพิสูจน์ได้ว่าที่ $n = x + 1$ เป็นจริง
ดังนั้นสิ่งที่ต้องการพิสูจน์จะเป็นจริงทุก n ใดๆ