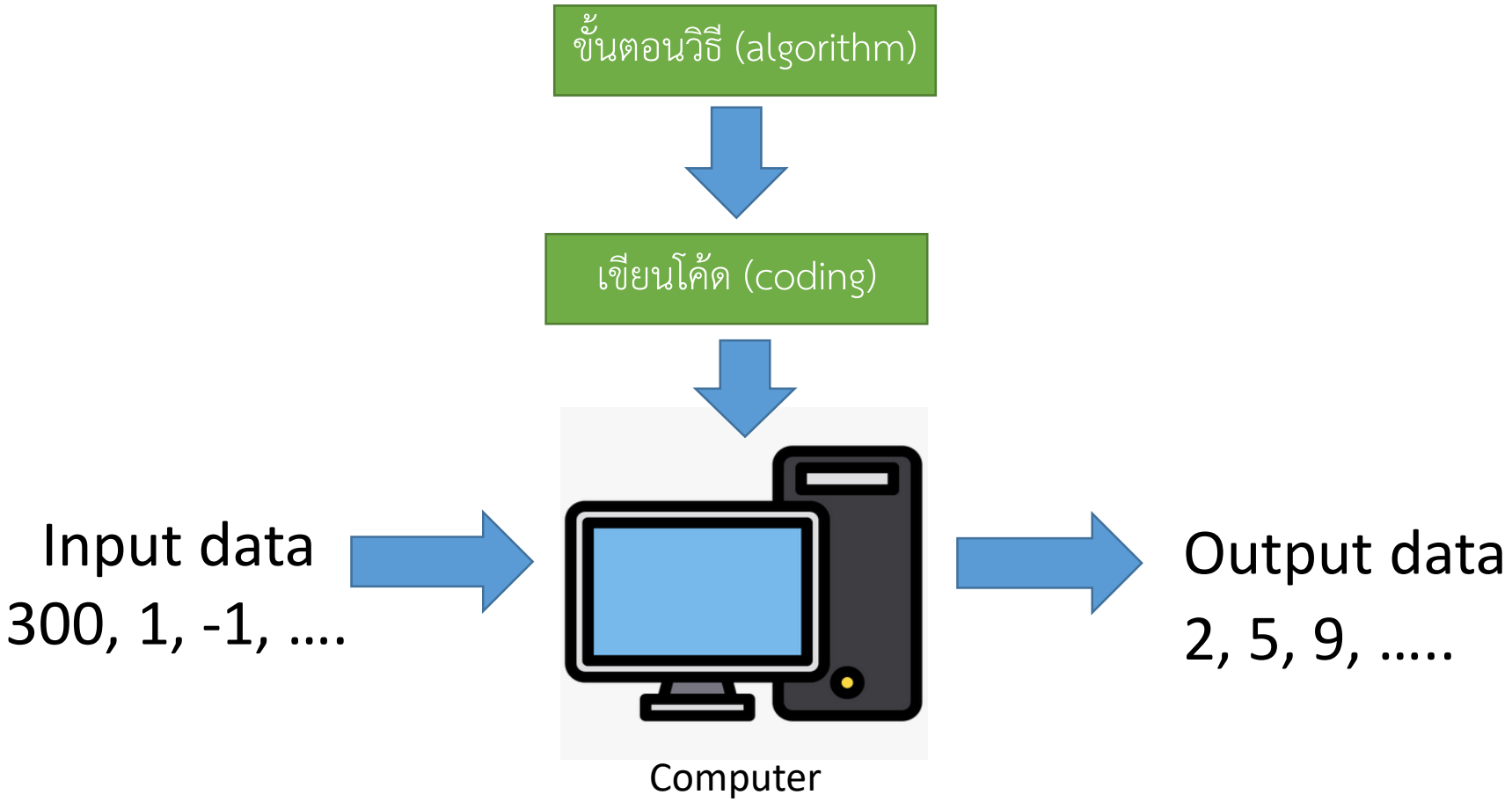
การวิเคราะห์ประสิทธิภาพของอัลกอริทึม (1)

อ. ลือพล พิพานเมฆาภรณ์

ขั้นตอนวิธี (Algorithm)

- เป็นขั้นตอนสำหรับการแก้ปัญหาทางคณิตศาสตร์หรือตรรกะโดยมีลำดับที่ชัดเจน



รหัสเทียม (Pseudo Code)

- วิธีการในการอธิบายขั้นตอนการแก้ปัญหาที่ไม่ขึ้นกับภาษาใดภาษาหนึ่ง

เริ่มต้นทำงาน

1. ให้ i มีค่าเป็น 1 และ Sum มีค่าเป็น 0
2. ถ้า i มากกว่า n ไปขั้นตอน 7
3 ถ้า ai เป็นเลขคี่ ไปขั้นตอน 6
5. นำ ai ไปบวกกับ Sum
6. เพิ่มค่า i ขึ้นไป 1 กลับไปยังขั้นตอน 2
7. แสดงผลลัพธ์ Sum

จบการทำงาน

```
1.  Sum = 0, i = 1 ;
2.  if i <= n then:
3.      if a[i] is even then:
5.          Sum = Sum + a[i];
6.      end if;
7.      i = i + 1;
8.      go to step 2;
9.  end if;
10. Display Sum
```

ทำไมต้องวิเคราะห์ขั้นตอนวิธี

- อัลกอริทึมที่ดีไม่ใช่แค่ทำงานถูกต้อง แต่ต้องประมวลผลไม่ซับซ้อน (complexity) เกินไป
- การประเมินความซับซ้อนของขั้นตอนวิธี สามารถทำได้ 2 วิธี
  1. การประเมินเวลาโดยตรง (Empirical analysis)
  2. การประเมินเวลาโดยนับรอบทำงาน (Counting analysis)
- การวิเคราะห์เวลาโดยตรง อาจไม่ใช่วิธีที่เหมาะสมเพราะขึ้นอยู่กับหลายปัจจัย
  - ความเร็วของเครื่องคอมพิวเตอร์ (Speed)
  - ภาษาคอมพิวเตอร์ที่ใช้
  - คุณภาพของตัวแปลภาษา

การประเมินเวลาโดยนับรอบของบรรทัดคำสั่ง

- แก้ปัญหาการประเมินเวลาที่ขึ้นอยู่กับปัจจัยภายนอก จึงเปลี่ยนมาใช้วิธีการประเมินเวลาโดยการนับรอบของบรรทัดคำสั่ง (counting operations) ขึ้นอยู่กับขนาดอินพุตแทน และเขียนในรูปของฟังก์ชั่นทางคณิตศาสตร์ T(n)

- เนื่องจากอัลกอริทึมทั่วไปจะใช้เวลาในการประมวลผลมากขึ้น เมื่อขนาดอินพุตมากขึ้น
  - Sorting ข้อมูล 10 จำนวน เทียบกับเวลาที่ใช้เรียงข้อมูล 1,000 จำนวน
  - Searching ข้อมูล 10 จำนวน เทียบกับข้อมูล 1,000 จำนวน

- ขณะที่บางปัญหาขนาดอินพุตก็ไม่ได้ขึ้นอยู่กับจำนวนข้อมูลแต่อาจเป็นค่าข้อมูล (value) เช่น การคำนวณค่าแฟคทรอเรียล (n!) หรือการคำนวณเลขยกกำลัง x^n

การประเมินเวลาโดยนับรอบของบรรทัดคำสั่ง

```
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

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

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

```
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
```

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

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

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

การประเมินเวลาโดยการนับรอบของโอเปอเรชันพื้นฐาน (basic operation counting)

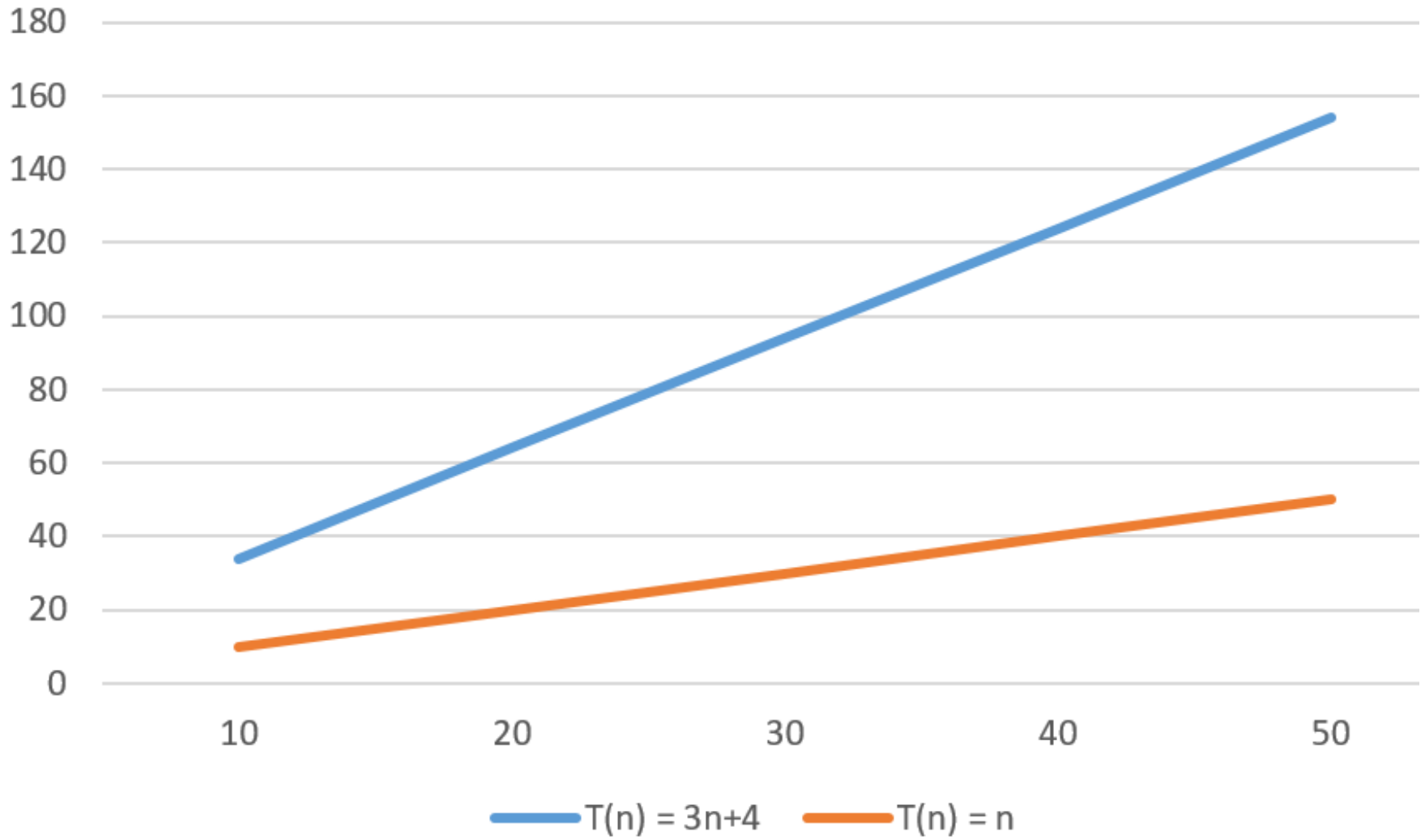
- **โอเปอเรชั่นพื้นฐาน** (Basic operation) คือบรรทัดคำสั่งที่ถูกประมวลผลมากที่สุดในอัลกอริทึม ซึ่งใช้เป็นตัวแทนของเวลาประมวลผลในอัลกอริทึม (มักเป็นบรรทัดที่อยู่ในลูปในสุด)

```
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
```

$T(n) \cong t4*n$

การประเมินเวลาโดยการนับรอบของโอเปอเรชันพื้นฐาน

กำหนดให้บรรทัดคำสั่ง ti มีเวลาคงที่เท่ากับ 1



การนับรอบทำงานของโอเปอเรชันพื้นฐาน (basic 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
```

$T(n) = t5*n^2$

ตัวอย่าง

```
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 สามารถเป็นบรรทัดที่ 5 หรือ 6 ก็ได้ เนื่องจากมีรอบทำงานไม่แตกต่างกัน

$T(n) = ??$

ตัวอย่าง

```
1.  result = 1,  j=0;
2.  for(i= 2; i <= n-1; i++)
3.   {    result = result * n;
4.        j = j + 1;
5.   }
6.  print result;
```

T(n) =

ตัวอย่าง

```
1. result = 1;
2. for( i=1 ; i <= n ; i++ )
3.  {  for( j=2; j <= i; j++)
4.        {  result = result * n;
5.        }
6.  }
7. print result;
```

T(n) =

สูตรช่วยนับรอบ for

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$$

```
for( i = 1; i <= n; i++)
    s = s + 1
```

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

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = n * (n + 1)/2$$

ตัวอย่าง

```
1. k = 0;
2. for( i=1; i <= n; i++)
3.  {    k = k + i ;
4.  }
5. for(j = n;  j > 0 ; j--)
6.  {   k = k  + j;
7.  }
8. print k;
```

T(n) =

ตัวอย่าง

```
1. result = 1,  j = n;
2. while ( j > 1)
3. {     result = j + n;
4.          j = j / 2;
5. }
6. print result;
```

T(n) =

ตัวอย่าง

1. result = 1, j = n;
2. **while (j > 1)**
3. {   result = j + n;
4.   j = j / 2;
5. }
6. print result;

หาความสัมพันธ์ระหว่างรอบทำงาน และขนาดของอินพุตที่เปลี่ยนแปลง

รอบที่ 1 n = n / 2 = n / 2^1

รอบที่ 2 n = n / 4 = n / 2^2

รอบที่ 3 n = n / 8 = n / 2^3

.....

เมื่อได้ความสัมพันธ์แล้วจะหารอบสุดท้าย โดยสมมติ n = 2^k

รอบที่ k 1 = n / 2^k

แก้สมการหาค่า k ในรูปของ n

ดังนั้น $k = \log_2 n$

$T(n) = \log_2 n$

การวิเคราะห์เวลากรณี best-case / worst-case / average-case

```
1. Sequential_Search (A[],k,n)
2.  {    i = 0;
3.       while(i <= n)
4.        {    if (A[i]==k)
5.                return i;
6.             i++;
7.        }
8.       return -1;
9.  }
```

การวิเคราะห์เวลากรณี best-case / worst-case / average-case

- 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

การวิเคราะห์เวลากรณี best-case / worst-case / average-case (Sequential Search)

- *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 : $t_a(n) = ?$

การวิเคราะห์เวลากรณี best-case / worst-case / average-case (Sequential Search)

- การวิเคราะห์เวลากรณีเฉลี่ยของอัลกอริทึม Sequential Search จะต้องหาเวลาของอินพุตที่เป็นไปได้ทุกกรณีแล้วนำมาเฉลี่ยรวมกัน
- ถึงแม้ว่าจะมีความเป็นไปได้ของอินพุตจำนวนมาก แต่เราสามารถจำแนกรูปแบบของอินพุต ได้ดังนี้
  - รูปแบบ 1 ค่าเป้าหมายที่ตำแหน่งแรกของอาร์เรย์ ใช้เวลาเท่ากับ 1
  - รูปแบบ 2 ค่าเป้าหมายที่ตำแหน่งสองของอาร์เรย์ ใช้เวลาเท่ากับ 2

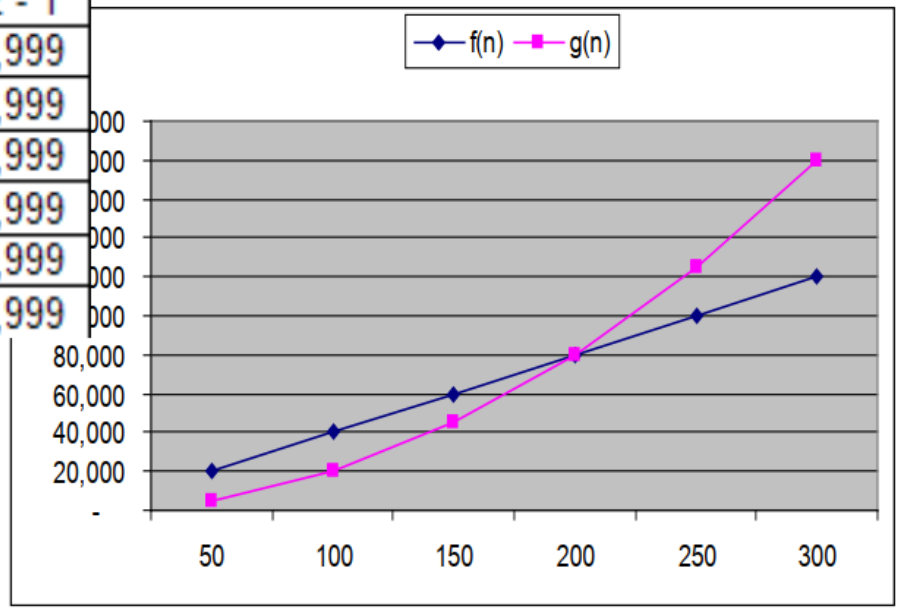
    .....
  - รูปแบบ n ค่าเป้าหมายที่ตำแหน่งสุดท้ายหรือไม่เจอ ใช้เวลาเท่ากับ n

ดังนั้น Tavg(n) = (1 + 2 + 3+ ... + n) / n =

ลำดับเติบโต (Order of Growth)

- ในการเปรียบเทียบเวลาทำงานของขั้นตอนวิธีมักไม่สามารถทำได้โดยตรง เนื่องจากแต่ละขั้นตอนวิธีมีรายละเอียดที่แตกต่างกัน
  - 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



การเปรียบเทียบขั้นตอนวิธี

- วิธีการที่สะดวกและรวดเร็วกว่าในการเปรียบเทียบลำดับการเติบโตของฟังก์ชันเวลาคือใช้ ทฤษฎีลิมิตของโลปิตา (L’ Hopital Rule)

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

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

การหาอนุพันธ์ของฟังก์ชั่น

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 $g(n)$

ตัวอย่าง

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

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

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

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

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

ตัวอย่าง

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

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

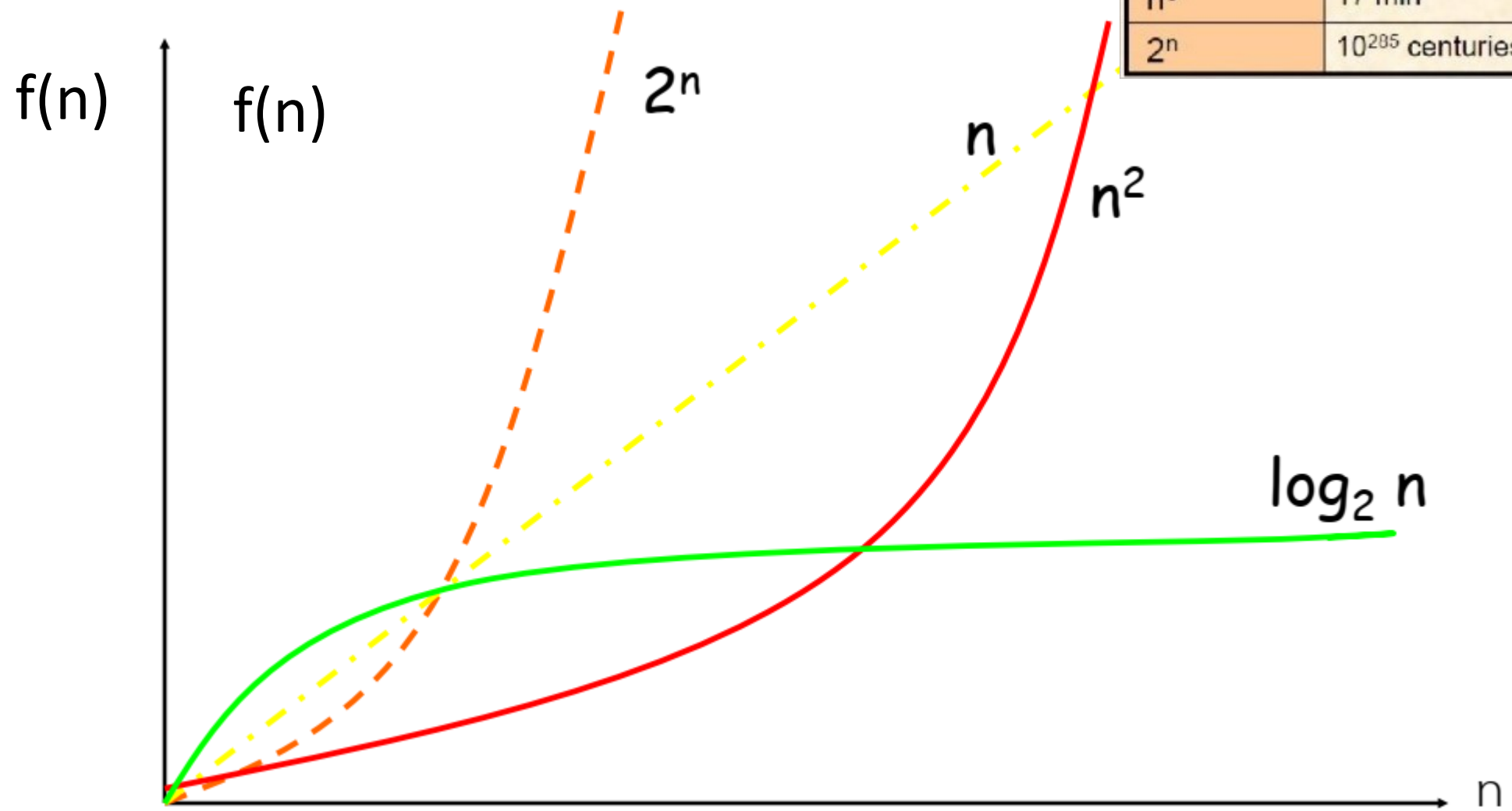
สัญกรเชิงเส้นกำกับ (Asymptotic notation)

- เพื่อให้การเปรียบเทียบเวลาทำงานของอัลกอริทึมทำได้ง่ายขึ้นจึงต้องมีการปรับฟังก์ชั่นเวลาให้เข้าสู่ฟังก์ชั่นเวลาอ้างอิง (reference function) เดียวกันก่อน โดยจะใช้สัญกรเชิงเส้นกำกับ (asymptotic notation) อธิบายฟังก์ชั่นเวลาของอัลกอริทึมด้วยฟังก์ชั่นอ้างอิงที่ใกล้เคียงที่สุด

- สัญกรเชิงเส้นกำกับ สามารถแบ่งออกเป็น 3 ประเภทหลักๆ ได้แก่
  - O(g(n)) อ่านว่า บิ๊กโอ (Big-oh) ใช้อธิบาย T(n) ว่าเติบโตไม่เร็วไปกว่า g(n)
  - Ω(g(n)) อ่านว่า บิ๊กโอเมก้า (Big-omega) ใช้อธิบาย T(n) ว่าเติบโตไม่ช้าไปกว่า g(n)
  - Θ(g(n)) อ่านว่า บิ๊กเทต้า (Big-theta) ใช้อธิบาย T(n) ว่าเติบโตเทียบเท่า g(n)

ฟังก์ชั่นอ้างอิงและอัตราเปลี่ยนแปลงเวลา

f(n)	$n=10^3$	$n=10^5$	$n=10^6$
$\log_2(n)$	10^{-5} sec	$1.7 * 10^{-5}$ sec	$2 * 10^{-5}$ sec
n	10^{-3} sec	0.1 sec	1 sec
$n*\log_2(n)$	0.01 sec	1.7 sec	20 sec
n^2	1 sec	3 hr	12 days
n^3	17 min	32 yr	317 centuries
2^n	10^{285} centuries	10^{10000} years	10^{100000} years



ฟังก์ชั่นอ้างอิง

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

สัญกรเชิงเส้นกำกับ (Asymptotic notation)

- เพื่อให้การเปรียบเทียบเวลาทำงานของอัลกอริทึมทำได้ง่ายขึ้นจึงต้องมีการปรับฟังก์ชั่นเวลาให้เข้าสู่ฟังก์ชั่นเวลาอ้างอิง (reference function) เดียวกันก่อน โดยจะใช้สัญกรเชิงเส้นกำกับ (asymptotic notation) อธิบายฟังก์ชั่นเวลาของอัลกอริทึมด้วยฟังก์ชั่นอ้างอิงที่ใกล้เคียงที่สุด

- สัญกรเชิงเส้นกำกับ สามารถแบ่งออกเป็น 3 ประเภทหลักๆ ได้แก่
  - O(g(n)) อ่านว่า บิ๊กโอ (Big-oh) ใช้อธิบาย T(n) ว่าเติบโตไม่เร็วไปกว่า g(n)
  - Ω(g(n)) อ่านว่า บิ๊กโอเมก้า (Big-omega) ใช้อธิบาย T(n) ว่าเติบโตไม่ช้าไปกว่า g(n)
  - Θ(g(n)) อ่านว่า บิ๊กเทต้า (Big-theta) ใช้อธิบาย T(n) ว่าเติบโตเทียบเท่า g(n)

บิ๊กโอ / บิ๊กโอเมก้า / บิ๊กเทต้า

- T(n) ∈ O(g(n)) ก็ต่อเมื่อมีค่าคงที่ c คูณกับ g(n) แล้วรับประกันว่า

  T(n) <= c*g(n) เมื่อ n มีค่ามากๆ

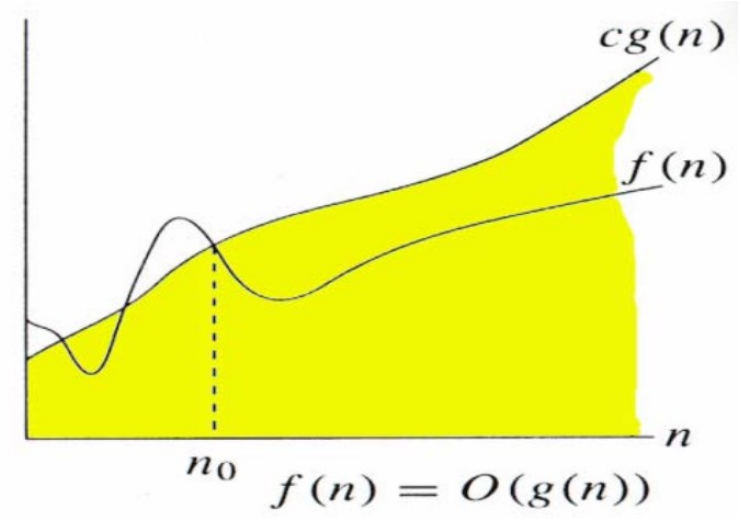
- T(n) ∈ Ω(g(n)) ก็ต่อเมื่อมีค่าคงที่ c คูณกับ g(n) แล้วรับประกันว่า

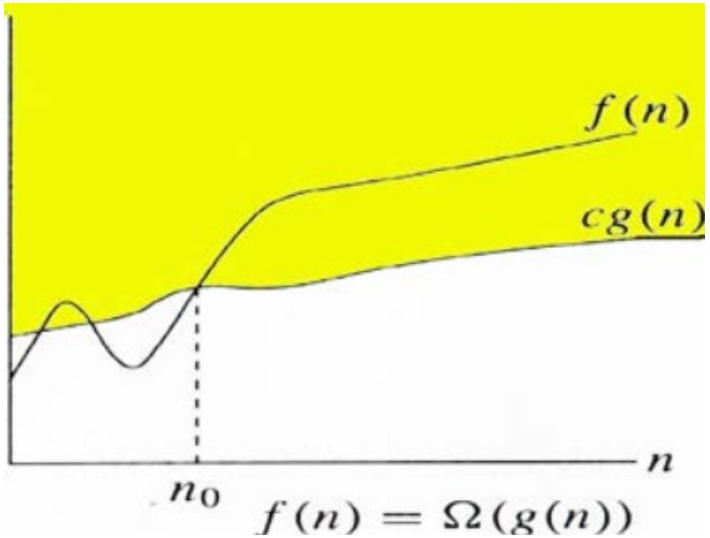
  T(n) >= c*g(n) เมื่อ n มีค่ามากๆ

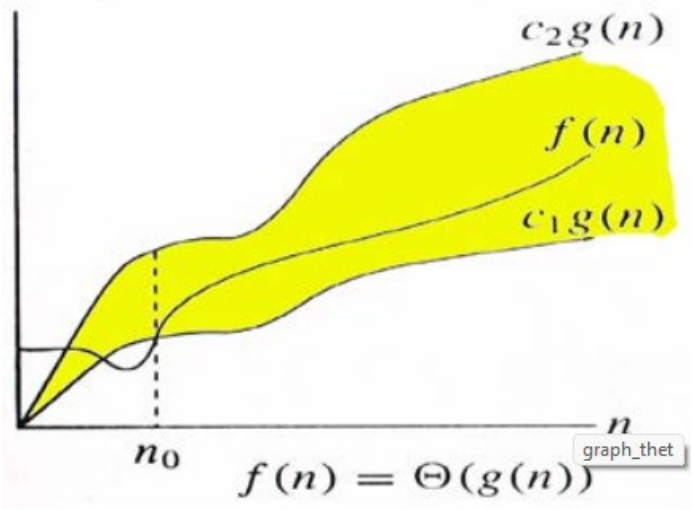
- T(n) ∈ Θ(g(n)) ก็ต่อเมื่อมีค่าคงที่ c1 และ c2 คูณกับ g(n) แล้วรับประกันว่า

  c1*g(n) <= T(n) <= c2*g(n) เมื่อ n มีค่ามากๆ

บิ๊กโอ / บิ๊กโอเมก้า / บิ๊กเทต้า







ตัวอย่าง

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

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

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

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