Design and Analysis of Data Structures and Algorithms

Warin Wattanapornprom Ph.D.

การวิเคราะห์อัลกอริทึม

- ทำไมต้องเรียนโครงสร้างข้อมูล
  - เข้าใจ code ต่าง ๆ ได้ง่าย
  - เลือกใช้โครงสร้างข้อมูลได้อย่างถูกต้องเหมาะสม
- เลือกยังไง
  - เขียนง่าย
  - เข้าใจง่าย สื่อสารง่าย
  - เร็ว

การวิเคราะห์อัลกอริทึม

- อาศัยทักษะและความรู้การเขียนโปรแกรมในการพัฒนาโครงสร้างข้อมูลที่เราเลือกหรือออกแบบไว้ให้เห็นจริง
- อาศัยทักษะและความรู้ทางคณิตศาสตร์ การนับ การวิเคราะห์ ตรรกศาสตร์และการจำลอง ในการวิเคราะห์ออกแบบขั้นตอนวิธีในการจัดการข้อมูลให้มีประสิทธิภาพ

การวิเคราะห์อัลกอริทึม

1. นักศึกษาได้รู้จักโครงสร้างข้อมูลที่ใช้ในปัจจุบัน
   - ประโยชน์ในแต่ละแบบ (Benefit)
   - ค่าใช้จ่ายในแต่ละแบบ (Cost)

2. นักศึกษาสามารถเลือกใช้โครงสร้างข้อมูลได้เหมาะสมกับงาน
   - นำมาใช้กับงานได้เลย ไม่ต้องคิดใหม่
   - ดัดแปลงจากของที่มีอยู่ให้เข้ากับงานของตน
   - ออกแบบโครงสร้างข้อมูลใหม่ให้เข้ากับงานของตน

การวิเคราะห์อัลกอริทึม

- ไม่มีโครงสร้างข้อมูลใดที่ใช้ได้ดีกับทุกงาน
- ดังนั้นเราเรียนเพื่อจะได้รู้ว่าโครงสร้างข้อมูลแบบไหนเหมาะกับงานประเภทไหน
  - รู้การเรียกใช้การดำเนินการต่าง ๆ ที่เกี่ยวกับโครงสร้างข้อมูลนี้ได้ เช่น search, insert, update, delete
  - รู้เงื่อนไขที่เกี่ยวข้อง
  - แปลงแนวคิดการจัดเก็บและจัดการข้อมูลออกเป็นโปรแกรมได้

การวิเคราะห์อัลกอริทึม

หัดตั้งคำถามให้ได้

- เราจะใช้เวลาในการประมวลผลเงินเดือนของพนักงานในบริษัทของเราเท่าไหร่ ?
- ตัวละครในเกมมีกี่ตัว แต่ละตัวใช้เวลา และ หน่วยความจำเท่าไหร่?
- เราควรจะซื้อโปรแกรมเราจากบริษัท ABC หรือ ควรซื้อจากบริษัท XYZ?
- โปรแกรมใช้เวลาในการประมวลผลนานมาก เป็นเพราะการออกแบบขั้นตอนวิธีไม่ดีหรือว่าเป็นเพราะปัญหาที่แก้เป็นปัญหาที่ยาก

การวิเคราะห์อัลกอริทึม

- มีวิธีมากมายในการที่เราจะออกแบบและเขียนโปรแกรม เราจะเลือกยังไงดี

- หัวใจของการออกแบบโปรแกรมมีสองวัตถุประสงค์ที่บางครั้งก็โคตรขัดแย้งกันเอง
  1. เขียนโปรแกรมให้เข้าใจง่าย โค๊ดสวย แก้ง่าย
  2. เขียนโปรแกรมให้รีดประสิทธิภาพการทำงานของเครื่องให้ได้มากที่สุด

ต้นทุนต่ำ หรือ กำไรเยอะ ถ้าทำได้ดีทั้งคู่ก็รวย

การวิเคราะห์อัลกอริทึม

- ประสิทธิภาพวัดยังไง

1. รันโปรแกรมตรงๆ (เสร็จชาติไหนไม่รู้)
2. วิเคราะห์จากโครงสร้างอัลกอริทึม

อัลกอริทึมส่วนใหญ่เวลาที่ใช้ขึ้นอยู่กับขนาดของข้อมูลนำเข้าหรืออินพุต

เวลาที่ใช้อยู่ในรูปแบบของฟังก์ชั่นเวลา หรือ $\mathrm{T}(n)$ โดยที่ n *คือขนาดของข้อมูลนำเข้า.*

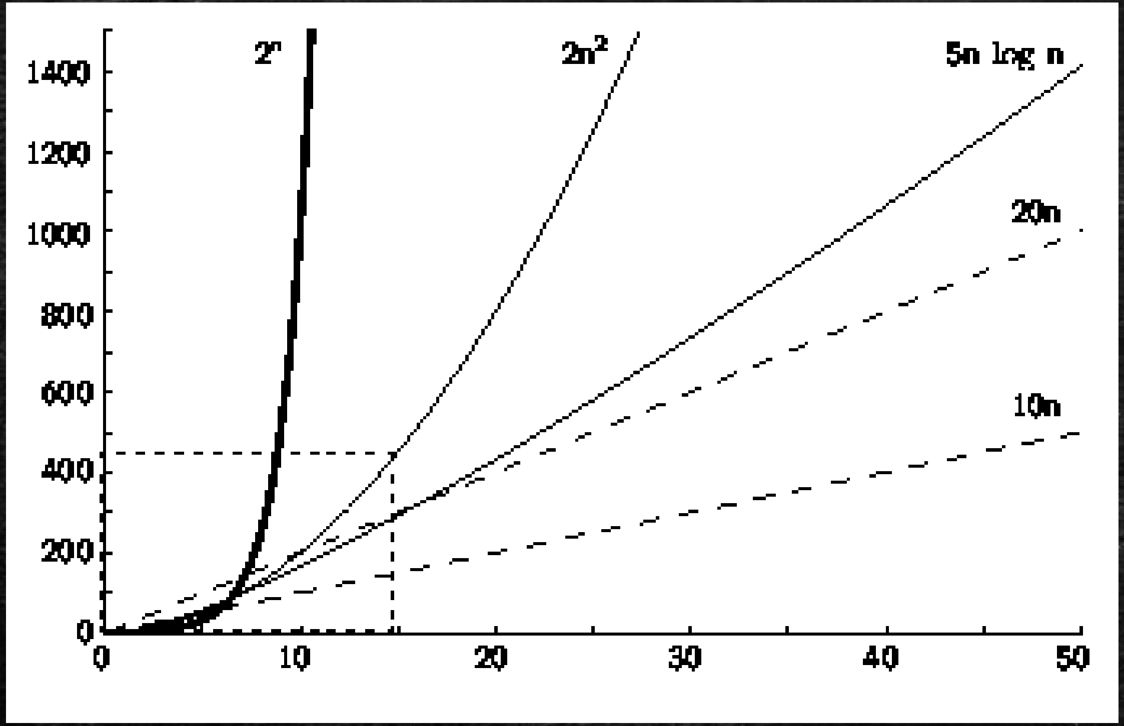
การวิเคราะห์อัลกอริทึม

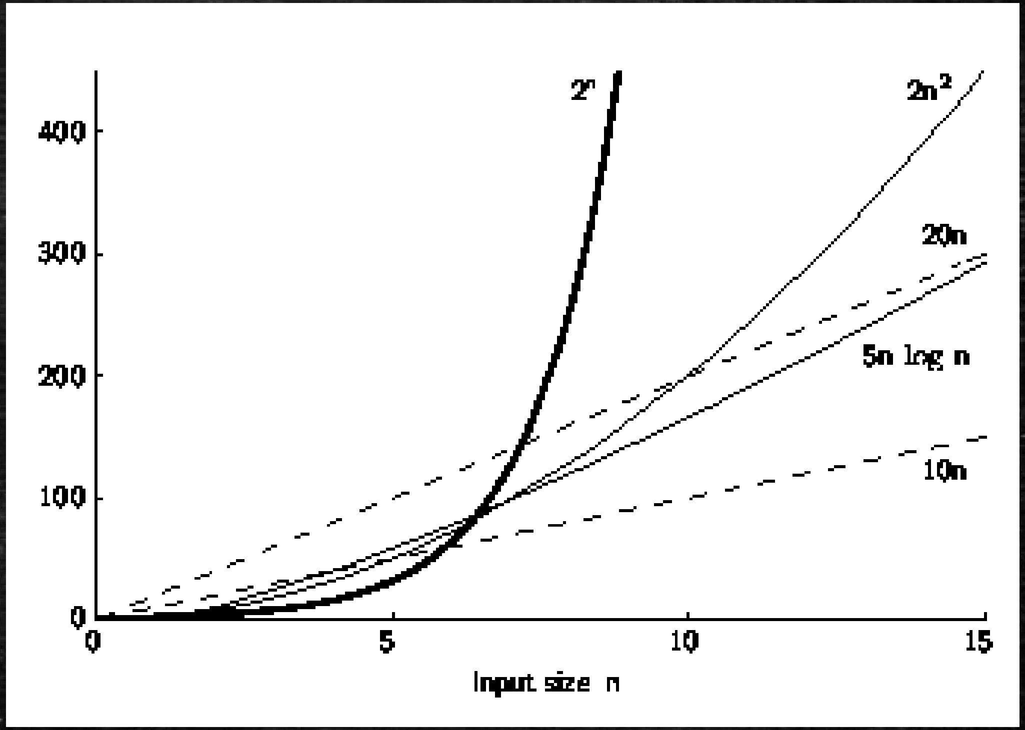
- การวิเคราะห์เชิงเส้นกำกับ (asymptotic algorithm analysis)
  - ช่วยประเมินประสิทธิภาพทางด้านเวลาที่ใช้ในการทำงานของโปรแกรม
  - เป็นการวิเคราะห์ที่ไม่ยาก
  - คำนึงถึงปัจจัยที่ส่งผลต่อเวลาของโปรแกรมมากสุด
  - ใช้ในกรณีที่มีข้อมูลจำนวนมาก
  - ได้เห็นอัตราการเติบโตของฟังก์ชันเวลาการทำงานกับ**จำนวนข้อมูล**

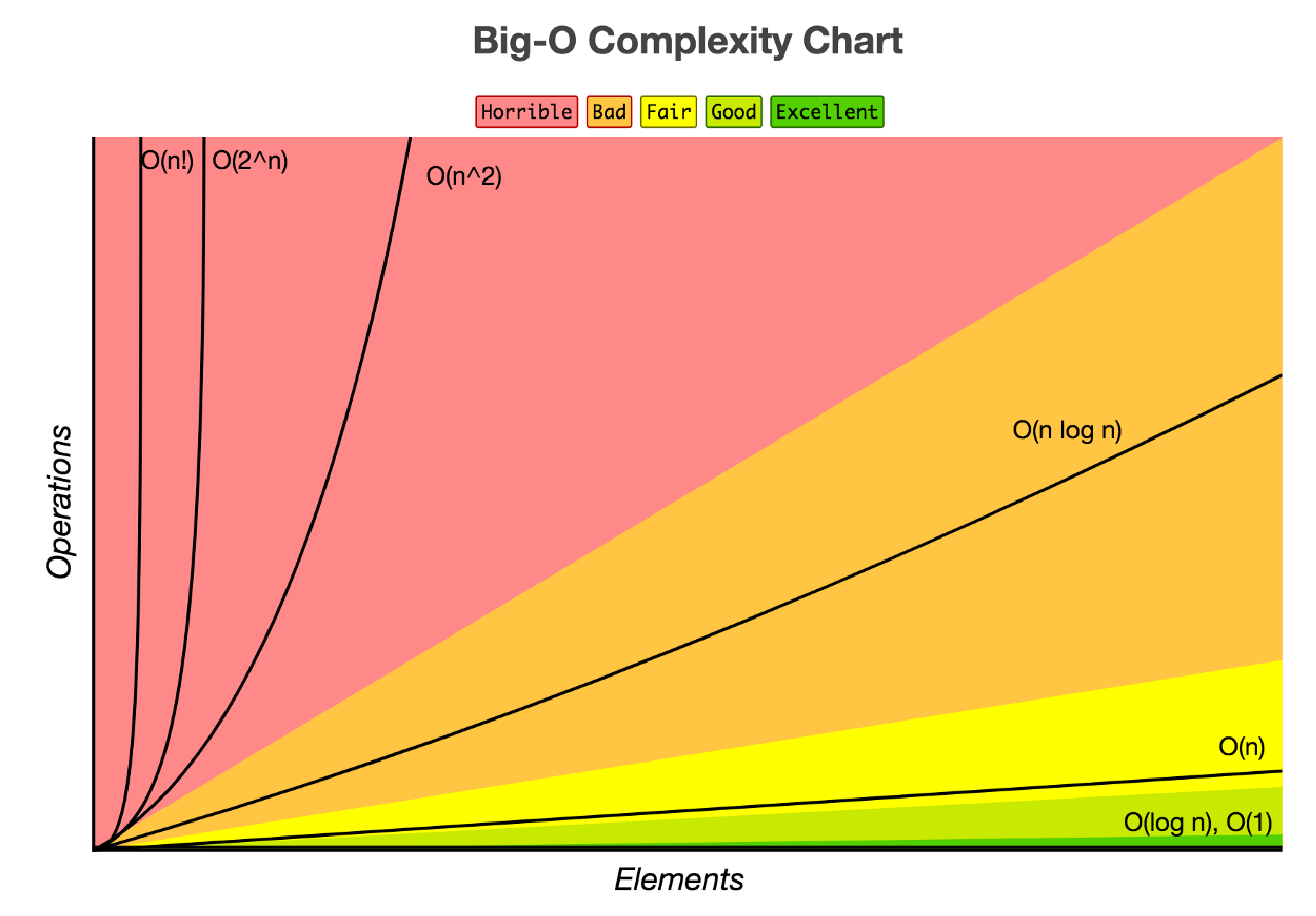
การวิเคราะห์อัลกอริทึม

- growth rate หรืออัตราการโตคืออัตราของต้นทุนเวลาของอัลกอริทึมที่โตขึ้นตามขนาดของข้อมูลนำเข้า

การวิเคราะห์อัลกอริทึม






Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

การวิเคราะห์อัลกอริทึม

```
for (i=1; i<=1000; i=i+1)
   <<application code>>;
```

1000 times

```
for (i=1; i<=1000; i=i+2)
   <<application code>>;
```

500 times

```
for (i=1; i<1000; i=i*2)
   <<application code>>;
```

10 times

```
for (i=1000; i>=1; i=i/2)
   <<application code>>;
```

10 times

การวิเคราะห์อัลกอริทึม

```
for (j=1; j<=10; j=j+1)      10 times
   for (i=1; i<=10; i=i+1)   10 iterations
      <<application code>>;
                                          100 iterations

for (j=1; j<=10; j=j+1)     10 iterations
   for (i=1; i<=10; i=i*2)  log₂10 iterations
      <<application code>>;
                                      10*log₂10 iterations

for (j=1; j<=10; j=j+1)   10 times
   for (i=1; i<=j; i=i+1)  (10+1)/2 times
      <<application code>>;
                                          55 iterations
```

การวิเคราะห์อัลกอริทึม

```
for (i=1; i<=n; i=i+1)
    <<application code>>;
```

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

```
for (i=1; i<=n; i=i+2)
   <<application code>>;
```

$$T(n) = n/2 = O(n)$$

```
for (i=1; i<n; i=i*2)
   <<application code>>;
```

$$T(n) = \lceil \log_2 n \rceil = O(\log n)$$

```
for (i=n; i>=1; i=i/2)
   <<application code>>;
```

$$T(n) = \lceil \log_2 n \rceil = O(\log n)$$

การวิเคราะห์อัลกอริทึม

```
for (j=1; j<=n; j=j+1)
   for (i=1; i<=n; i=i+1)
         <<application code>>;
```

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

```
for (j=1; j<=n; j=j+1)
   for (i=1; i<=n; i=i*2)
        <<application code>>;
```

$$T(n) = \lceil n \log_2 n \rceil = O(n \log n)$$

```
for (j=1; j<=n; j=j+1)
   for (i=1; i<=j; i=i+1)
        <<application code>>;
```

$$T(n) = n\left(\frac{n+1}{2}\right) = O(n^2)$$

การวิเคราะห์อัลกอริทึม

Big O Notation

- หรือ อันดับขนาด (Order of Magnitude)
  - หมายถึงปริมาณที่เครื่องคอมพิวเตอร์ทำไม่ขึ้นกับขนาดของโปรแกรมหรือจำนวนบรรทัดของโปรแกรม
  - เป็นฟังก์ชันที่ได้จากการประมาณค่าทางคณิตศาสตร์ซึ่งเป็นฟังก์ชันที่สัมพันธ์กับขนาดของปัญหา
- ให้ข้อมูลในการเปรียบเทียบอัลกอริทึม
  -
    ```
    for(int i =1;i<n;i++){
        dosomething();
    }
    ```
  - $O(n)$

O-notation

- **นิยาม** : ความหมายของ $\mathrm{O}(n)$ คือ

  ฟังก์ชั่นนั้น ๆ ใช้เวลาทำงานช้าที่สุด $\leq n$

- $\mathrm{O}(g(n)) = \{ f(n)$: there exist positive constants
  c and n_0 such that
  $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0 \}$

ตัวอย่าง เช่น อัลกอริทึม $a1$ มีประสิทธิภาพเป็น $\mathrm{O}(n^2)$

– ถ้า $n = 10$ แล้ว $a1$ จะใช้เวลาทำงานช้าที่สุด 100 หน่วยเวลา
(รับประกันว่าไม่ช้าไปกว่านี้ - แต่อาจจะเร็วกว่านี้ได้)

Ω-notation

- **นิยาม :** ความหมายของ $\Omega\ (n)$ คือ

  ฟังก์ชั่นนั้น ๆ ใช้เวลาทำงานเร็วที่สุด $\geq n$

- $\Omega(g(n)) \;=\; \{ f(n)$: there exist positive constants
  c and n_0 such that
  $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$ $\}$

- <u>ตัวอย่าง</u> เช่น อัลกอริทึม $a1$ มีประสิทธิภาพเป็น $\Omega(n)$
  – ถ้า $n = 10$ แล้ว $a1$ จะใช้เวลาทำงานเร็วที่สุด 10 หน่วยเวลา (รับประกันว่าไม่เร็วไปกว่านี้ - แต่อาจจะช้ากว่านี้ได้)

Θ-notation

- นิยาม : $f(n) = \Theta(g(n))$ ก็ต่อเมื่อ $f(n) = O(g(n))$
  และ $f(n) = \Omega(g(n))$
- $\Theta(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

การวิเคราะห์อัลกอริทึม

```
// Find Max array value
int FindMax(int array[], int n) {
int MaxSoFar = 0;
  for (int i=1; i<n; i++)
    if (array[currlarge] < array[i])
      MaxSoFar = i;
  return MaxSoFar;
}
```

การวิเคราะห์อัลกอริทึม – Time Complexity

- Best Case Time Complexity
  - เวลาที่ดีสุด (minimum time) ที่ algorithm ใช้ในการประมวลผลสำหรับข้อมูลนำเข้าขนาด n
- Worst Case Time Complexity
  - เวลาที่มากสุด(maximum time) ที่algorithm ใช้ในการประมวลผลสำหรับข้อมูลนำเข้าขนาด n
  - **รับประกันเวลาได้ว่า algorithm ที่ใช้จะไม่ใช้เวลาเกินนี้**
- Average Case Time Complexity
  - เวลาที่เฉลี่ย(average time) ที่ algorithm ใช้ในการประมวลผลสำหรับข้อมูลนำเข้าขนาด n
  - จำเป็นต้องกำหนดเงื่อนไขขนาดของ n

การวิเคราะห์อัลกอริทึม – Time Complexity

ค้นหาค่า k จาก array ขนาดเท่ากับ n:

- เริ่มที่ตำแหน่งแรกสุดไล่ไปจนกว่าจะเจอค่า k หรือจนกว่าจะจบอาร์เรย์

Best case: 1

Worst case: n

Average case: $n/2$

การวิเคราะห์อัลกอริทึม – Time Complexity

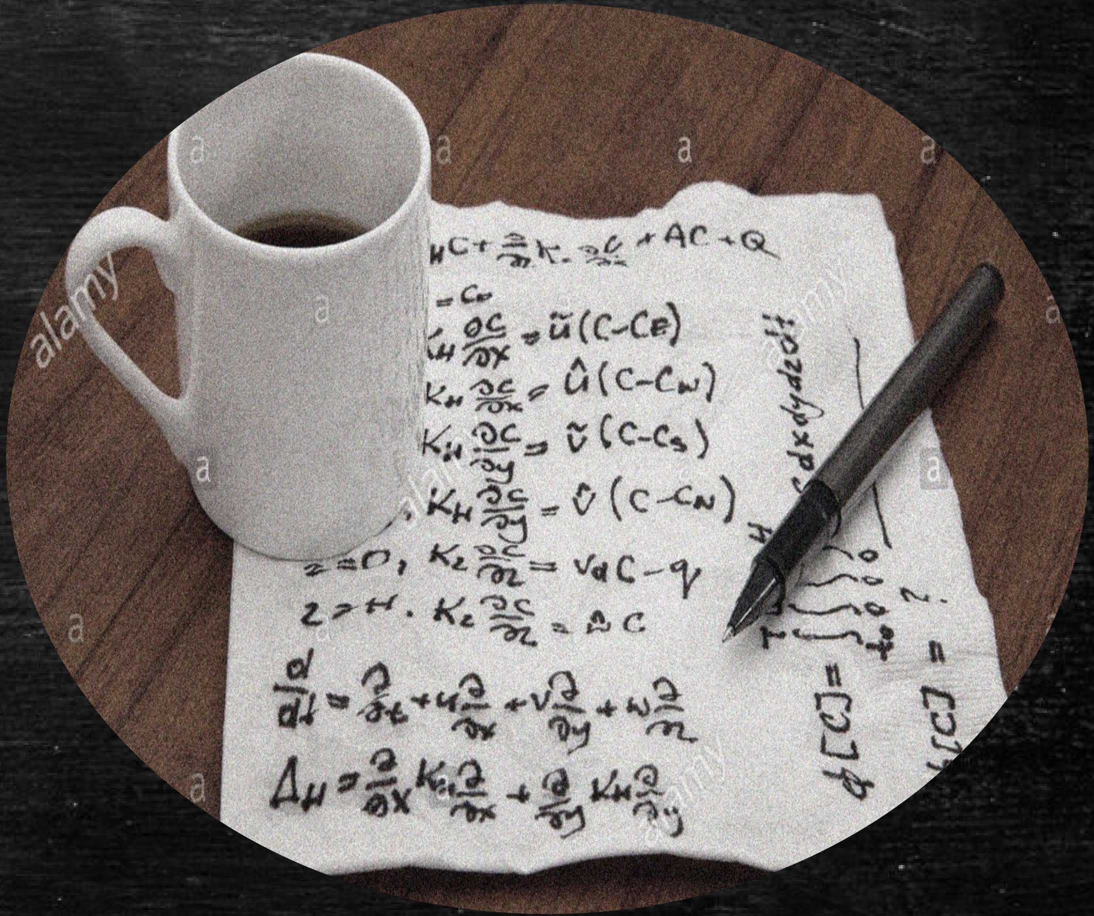
- Big O

$$T(N) = O(f(n)) \text{ ถ้า } T(N) <= c(f(N))$$

- โดยมี c กับ $N0$ เป็นค่าคงที่และ $N >= N0$
- นี่คือการบอกว่า $T(N)$ เติบโตอย่างไร

Asymptotic Notations

- Deal with the behaviour of functions in the limit (for sufficiently large value of its parameters)
- Permit substantial simplification (napkin mathematic, rough order of magnitude)
- Classify functions by their growth rates



การหา BIG O จาก loop ต่างๆ (ต่อ)

- นิยามของการคิด Nested loop เป็นดังนี้
  - ถ้า $T_1(N) = O(f(N))$ และ $T_{2(N)} = O\left(g(N)\right)$ แล้ว

$$T_1(N) * T_2(N) = O(f(N) * g(N))$$

  - จากตัวอย่างนั้น $f(n) = g(n) = n$
  - จึงตอบเป็น $O(n^2)$

Loop ติดกัน

- Statement ที่เรียงต่อกันเป็นบรรทัด

```
for (i = 0; i <= n; i++)
        statement1;
for (j = 0; j <= n; j++)
    for (k = 0; k <= n; k++)
         statement2;
```

$O(n)$

$O(n^2)$

เอาตัวมากสุดมาตอบ นั่นคือ $O(n^2)$

Loop ติดกัน

- นิยามของการหา running time จาก Statement ที่เรียงต่อกัน
  - ถ้า $T_1(N) = O(f(N))$ และ $T_2(N) = O(g(N))$ แล้ว

$$T_1(N) + T_2(N) = \max(O(f(N), O(g(N)))$$

จากตัวอย่าง $f(n) = O(n), g(n) = O(n^2)$

จึงตอบเป็น $O(n^2)$

การหา BIG O จาก loop ต่างๆ

- ประโยคแบบมีเงื่อนไข

```
1:    if (condition)
2:            Statement1
3:    else
4:            Statement2
```

Statement1 $\rightarrow O(f(n))$

Statement2 $\rightarrow O(g(n))$

เอาตัวมากสุดมาตอบ นั่นคือ

$$\max(O(f(n), O(g(n)))$$

Recursive Programming

- คือ การเขียนโปรแกรมที่ฟังก์ชันในโปรแกรมนั้นมีการเรียกใช้งาน ตัวมันเอง (Call itself)
- เรียกฟังก์ชันที่มีการเรียกใช้งานตัวมันเองว่า “Recursive Function”
- บางปัญหาสามารถที่จะเขียนในรูปแบบของ recursion จะง่ายกว่า

<u>หลักการแก้ปัญหาในรูปแบบ recursion</u>

1. นิยามปัญหาในรูปการเรียกตัวเอง
2. มีเงื่อนไขสำหรับจบการทำงาน

Recursive Function

- Function หาค่า Factorial n!

```
double factorial(double n) {
  double result = 1;
for(double i = 2; i<=n; ++i) {
     result = result*i;
}return result;
}
```

$n!=1*2*3*\ldots.*n$

- เปลี่ยนเป็น Recursive Function ได้ดังนี้

```
double factorial(double n) {
  double result;
     if(n <= 1) return 1;
     result = n * factorial(n - 1);
     return result;
}
```

$n!=n*(n-1)!$

Recursive Function

- ลองรันโปรแกรมดู

```
#include <iostream.h>

double factorial(double);

void main(void) {
	double number;
	cout << "Please enter a positive integer: ";	cin >> 
number;
	cout << number << " factorial is: " << factorial(number) 
<< endl;
}
```

การหา Big O จาก recursion

```
mymethod (int n) {
    if (n == 1) {
        return 1;
    }else{
        return 2*mymethod(n - 1) + 1;
    }
}
```

n ครั้ง, big O $= O(n)$

Recursive Function

- Function หาค่า Fibonacci n (0,1,1,2,3,5,8,13,21,…)

- ```
  int fib (int N) {
    int k1, k2, k3;
    k1 = k2 = k3 = 1;
    for (int j = 3; j <= N; j++) {
        k3 = k2 + k1;
        k1 = k2;
        k2 = k3;
    }
    return k3;
  }
  ```

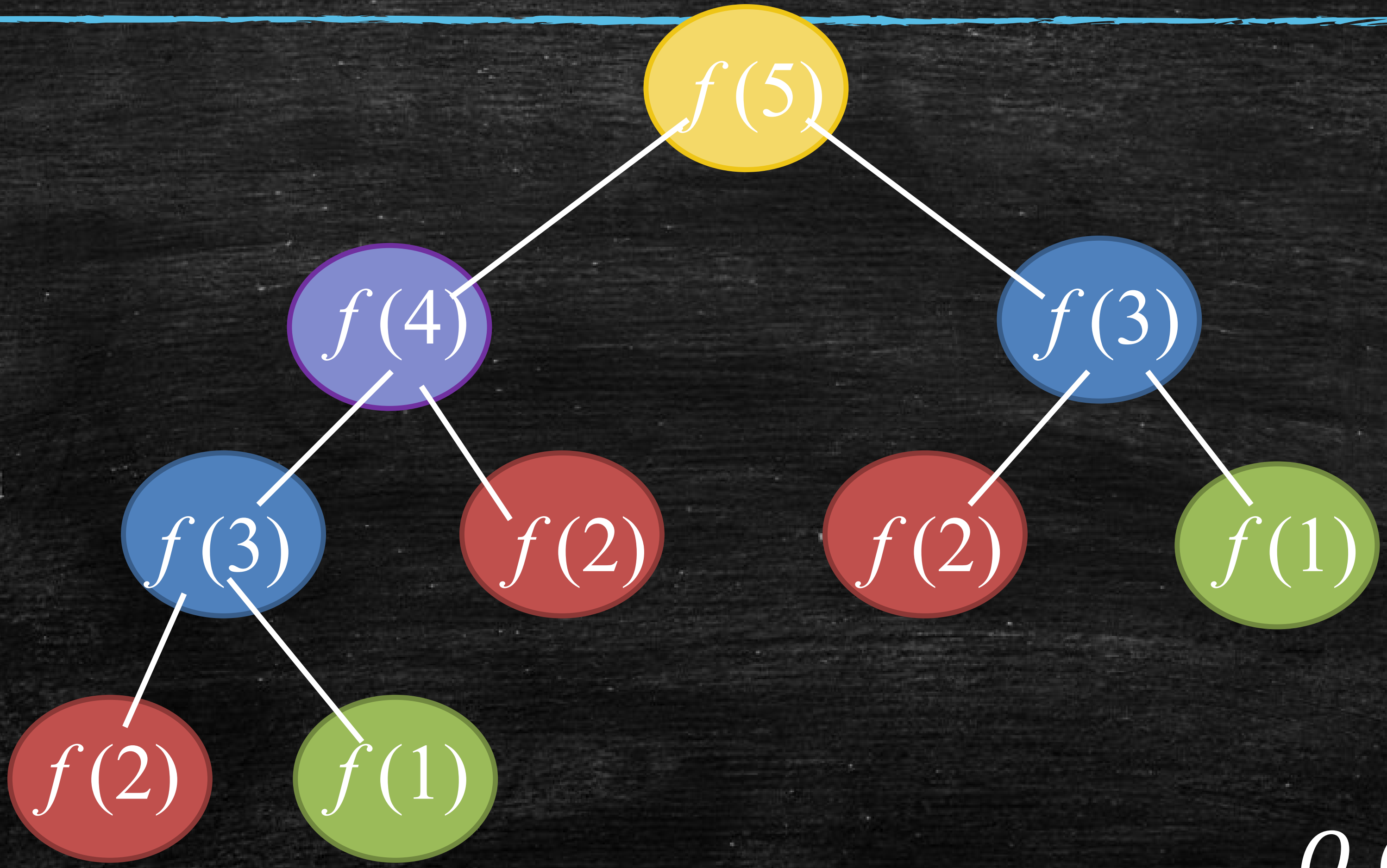
  $O(n)$

  เปลี่ยนเป็น Recursive Function ได้ดังนี้

- ```
  int fib (int N) {
    if ((N == 1) || (N == 2)) return 1;
    else return (fib (N-1) + fib (N-2));
  }
  ```

  $O(\varphi^n)$

Fibonacci Number



$O(\varphi^n)$

Recursive Function

- Function หาค่า Great Common Devisor

```
int findGCD(int n1,int n2){
   int  gcd;
   for(int i=1;i<=n1&&i<=n2;i++){
       if(n1%i==0 && n2%i == 0 ){
       gcd=i;}}
return gcd;
}
```

$$O\ (\max(n1,n2))$$

เปลี่ยนเป็น recursive

```
int findGCD(int n1, int n2)
{
    if (n2!=0)
       return findGCD (n2, n1%n2);
    else
       return n1;
}
```

$$O\ (?????)$$
$$<\ O(\log_? n)$$

การยกกำลังเลข x^n ด้วยวิธี divide and conquer

```
▪ long power (long x, int n) {
              if (n==0)
                     return 1;
              if (isEven (n))
                     return power (x*x, n/2);
              else
                     return power (x*x, n/2)*x;
  }
▪ long power (long x, int n) {
              if (n==1) return 1;
                     return x*power(x, n-1);
  }
▪ long power (long x, int n) {
       long result;
              for(int i=1;i<n;i++)
              result=result*x;
       return result;
  }
```

Big O คือ O ($\log_2 n$)

ปัญหาถูกแบ่งเป็น 2 ครึ่ง (ประมาณ) เท่ากันในแต่ละการเรียกใช้ method

$$x^n = (x * x)^{n/2}$$

$$x^n = x * x^{n-1}$$

$$x^n = x * x * x * \cdots * x$$

การวิเคราะห์อัลกอริทึม

Constant	$O(1)$	การเข้าถึงสมาชิกตัวที่ i ในแถวลำดับ
Logarithmic	$O(\log n)$	การค้นหาแบบ Binary Search
Linear	$O(n)$	การค้นหาแบบ Sequential
Linear logarithmic	$O(n(\log n))$	การจัดเรียงแบบ Merge Sort
Quadratic	$O(n^2)$	การจัดเรียงแบบธรรมดา

Comparison of Functions

$$f \leftrightarrow g \;\approx\; a \leftrightarrow b$$

$$f(n) = O(g(n)) \;\approx\; a \leq b$$

$$f(n) = \Omega(g(n)) \;\approx\; a \geq b$$

$$f(n) = \Theta(g(n)) \;\approx\; a = b$$

$$f(n) = o(g(n)) \;\approx\; a < b$$

$$f(n) = \omega(g(n)) \;\approx\; a > b$$

Asymptotic Notation Properties

- Transitivity
- Reflexivity
- Symmetry
- Transpose symmetry

Transitivity

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

Reflexivity

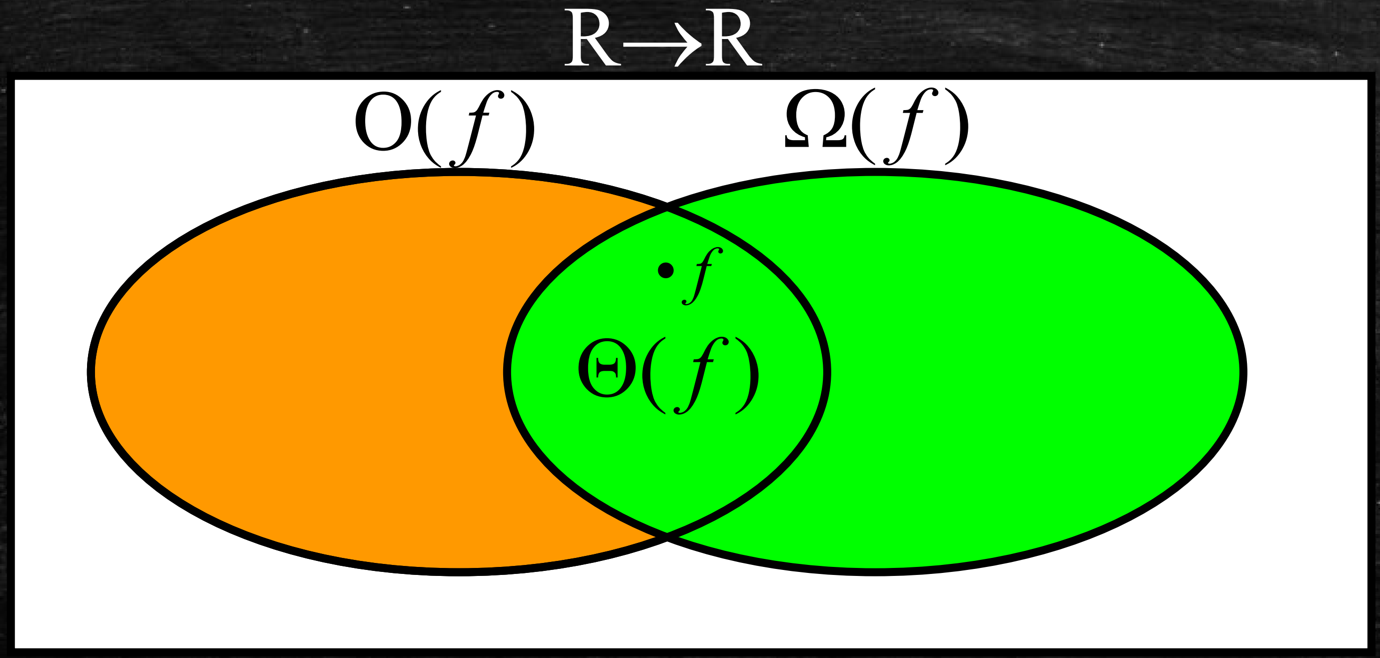
- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Symmetry and Transpose Symmetry

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

Θ, Ω, O

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



Manipulating Asymptotic Notations

- $c\,O(f(n)) = O(f(n))$
- $O(O\,f(n)) = O\,(f(n))$
- $O\,(f(n))O(g(n)) = O(f(n)\,g(n))$
- $O\,(f(n)\,g(n)) = f(n)O(g(n))$
- $O\,(f(n) + g(n)) = O(\max(f(n), g(n))\,)$

Examples

- $2n^3 = O(n^3)$:
- $n^2 = O(n^2)$:
- $1000n^2+1000n = O(n^2)$:

More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \le cn$, $\forall n > n_0$.

Let $c=31$, $n_0=8$. Assume $n>n_0=8$.

Then

$$
\begin{aligned}
cn &= 31n \\
&= 30n + n > 30n+8,
\end{aligned}
$$

so

$$30n+8 < cn.$$