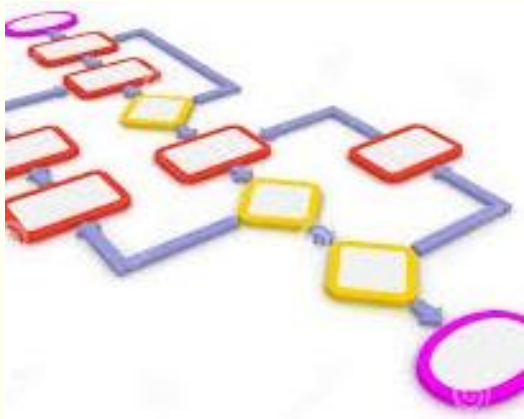


รายวิชา **การออกแบบและวิเคราะห์ขั้นตอนวิธี**

Design and Analysis of Algorithms

รหัสวิชา **ST2022112**

อ.ธิดาวรร คล้ายศรี

บทที่2

การวิเคราะห์ความต้องการเชิงเวลาและเนื้อที่

2.1 ประสิทธิภาพเชิงเวลา (Time Efficiency)

2.2 สัญลักษณ์เชิงเส้นกำกับทางเวลา Big O

2.3 ประมาณความต้องการเชิงเวลา (Time Estimation)

2.4 ประสิทธิภาพเชิงเนื้อที (Space Efficiency)

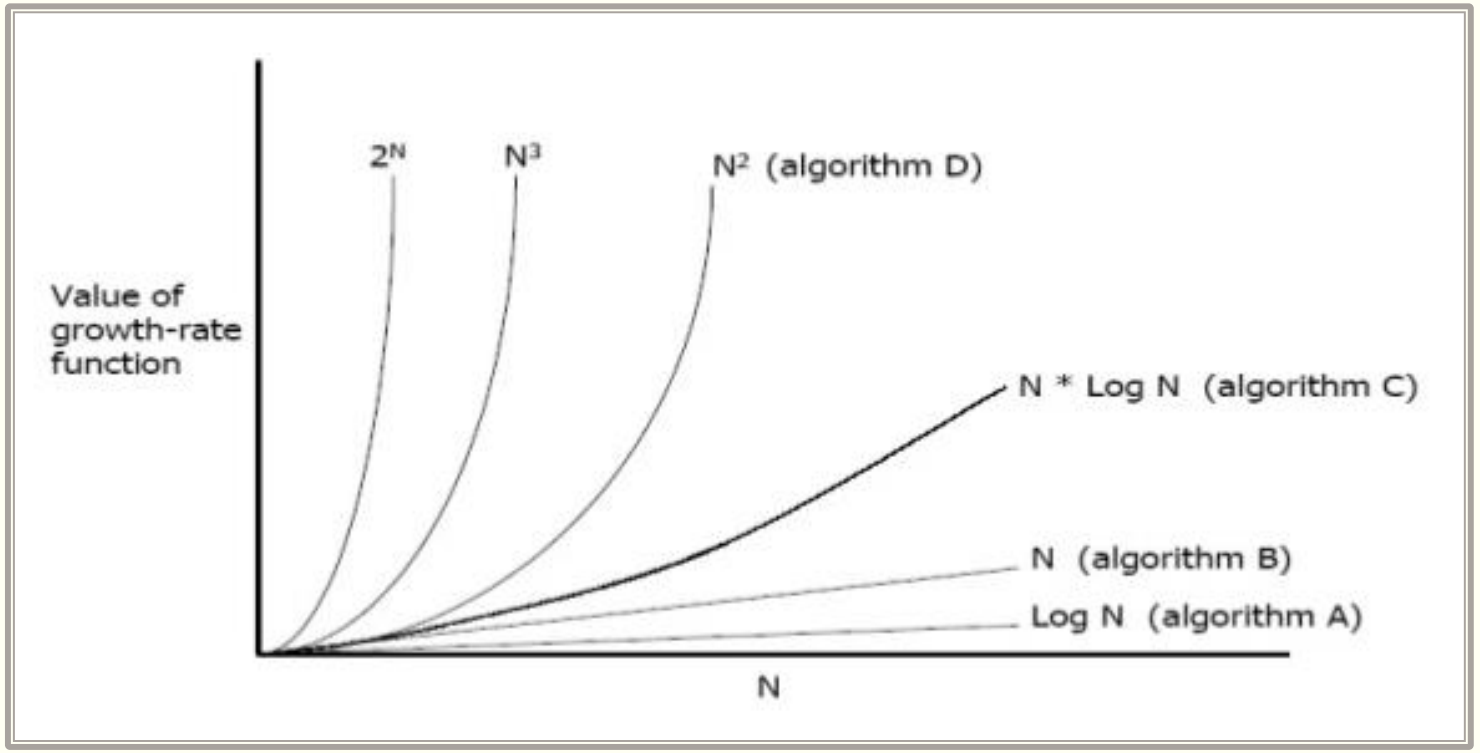
2.5 การวิเคราะห์ชั้นความซับซ้อน

การวัดประสิทธิภาพของโปรแกรม

- จากการใช้เนื้อที่ในหน่วยความจำ (Space/Memory)
- ประสิทธิภาพของเวลาในการทำงาน (Time) แต่การวัดเวลาทำการเปรียบเทียบได้ลำบาก
- พิจารณาอัตราการเติบโตของฟังก์ชัน (Growth Rates)

Analysis of Algorithms (**การวิเคราะห์อัลกอริธึ่ม**)

ในการแก้ปัญหาทางคอมพิวเตอร์ เราอาจออกแบบอัลกอริธึ่มไว้หลายอัลกอริธึ่ม เช่น 4 อัลกอริธึ่ม (A-D) ข้างล่างนี้



[ภาพ: internet 2017]

2.1 ประสิทธิภาพเชิงเวลา (Time Efficiency)

เวลาในการประมวลผลของโปรแกรมได้แก่

- Compile Time คือ เวลาที่ใช้ในการตรวจสอบไวยากรณ์ (syntax) ของ code ว่าเขียนได้ถูกต้องหรือไม่
- Run Time หรือ Execution Time คือ เวลาที่เครื่องทำการรันโปรแกรม
- คอมพิวเตอร์ใช้ในการประมวลผลลัพธ์

การวิเคราะห์อัลกอริธึ่ม

โดยปกติประสิทธิภาพของอัลกอริธึ่มจะพิจารณาเป็น 3 cases ได้แก่

- Worst case -เมื่อข้อมูลที่เข้ามาประมวลผลส่งผลให้อัลกอริธึ่มมีประสิทธิภาพต่ำที่สุดและใช้เวลาในการประมวลผลนานที่สุด-อัตราการเติบโตของฟังก์ชั่นสูง
- Average case –เมื่อข้อมูลที่เข้ามาประมวลผลส่งผลให้อัลกอริธึ่มโดยเฉลี่ยที่สามารถระบุ/ประมาณการได้
- Best case -เมื่อข้อมูลที่เข้ามาประมวลผลส่งผลให้อัลกอริธึ่มมีประสิทธิภาพสูงที่สุดและใช้เวลาในการประมวลผลสั้นที่สุด-อัตราการเติบโตของฟังก์ชั่นต่ำ

2.2 สัญลักษณ์เชิงเส้นกำกับทางเวลา Asymptotic Notations

เป็นสัญลักษณ์ที่ใช้นำเสนอความซับซ้อนด้านเวลาของอัลกอริธึ่ม มีหลายแบบสัญลักษณ์ ได้แก่

- **Big-O** notation **=** Asymptotic upper bound
  ที่บอกขอบเขตบนของฟังก์ชั่น → Worst-case

- **Big-Ω** (Big-Omega) notation **=** Asymptotic lower bound
  ที่บอกขอบเขตล่างของฟังก์ชั่น → best case

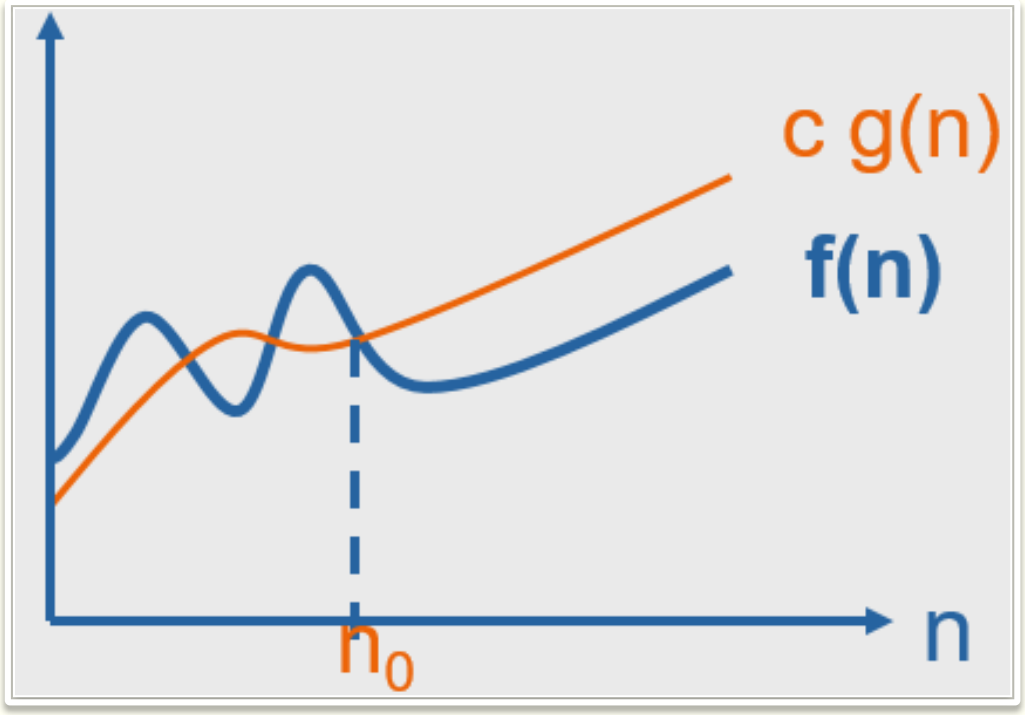
- **Big-θ** (Big-Theta) notation **=** Asymptotic tight bound
  ที่บอกขอบเขตในช่วงกลางของฟังก์ชั่น → average case

- **Little-o** notation → Asymptotic bound ของฟังก์ชั่น

2.2 สัญลักษณ์เชิงเส้นกำกับทางเวลา Big-O Notation

- การใช้สัญลักษณ์ Big-O มีวัตถุประสงค์เพื่ออธิบายขอบเขต(บน) ของฟังก์ชันการเติบโตทางด้านเวลา
- ใช้พิจารณาความซับซ้อนด้านเวลาของฟังก์ชั่น เพื่อเลือกอัลกอริธึ่มที่มีประสิทธิภาพสูงสุด (ไม่ได้มีวัตถุประสงค์เพื่อวัดเวลา)
- โดยจะพิจารณาปริมาณอินพุทข้อมูลมากๆ → Worst-case

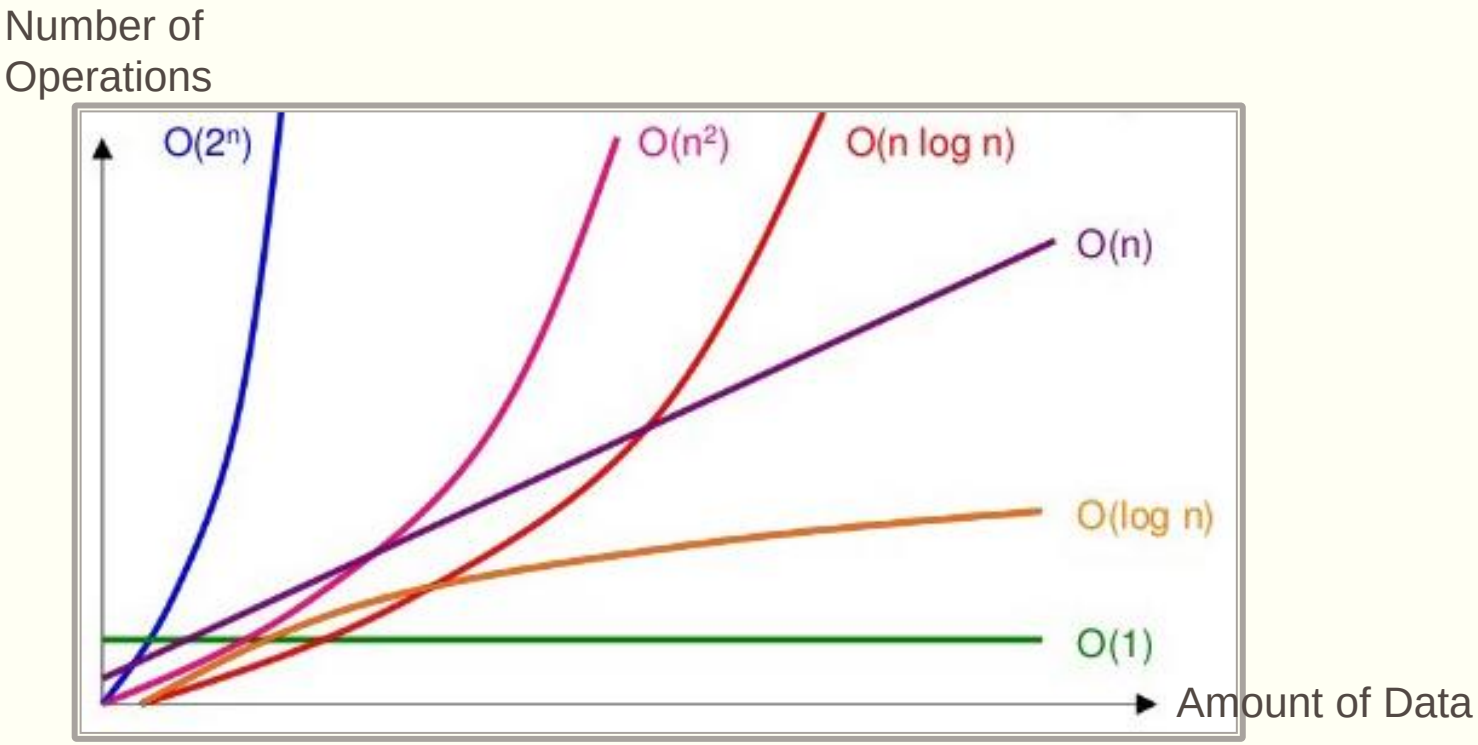
Big-O Notation



f(n) = O(g(n)) ก็ต่อเมื่อ มีค่าคงที่ c และ n_0

ที่ทำให้ f(n) <= c g(n) สำหรับทุกค่า n >= n_0

Big-O Notation



เปรียบเทียบ Big-O

[ภาพ: internet 2017]

2.3 ประมาณความต้องการเชิงเวลา (Time Estimation)

การวิเคราะห์เวลาที่เครื่องคอมพิวเตอร์ต้องใช้ในการประมวลผลอัลกอริธึ่ม ซึ่งวิเคราะห์เพื่อ:

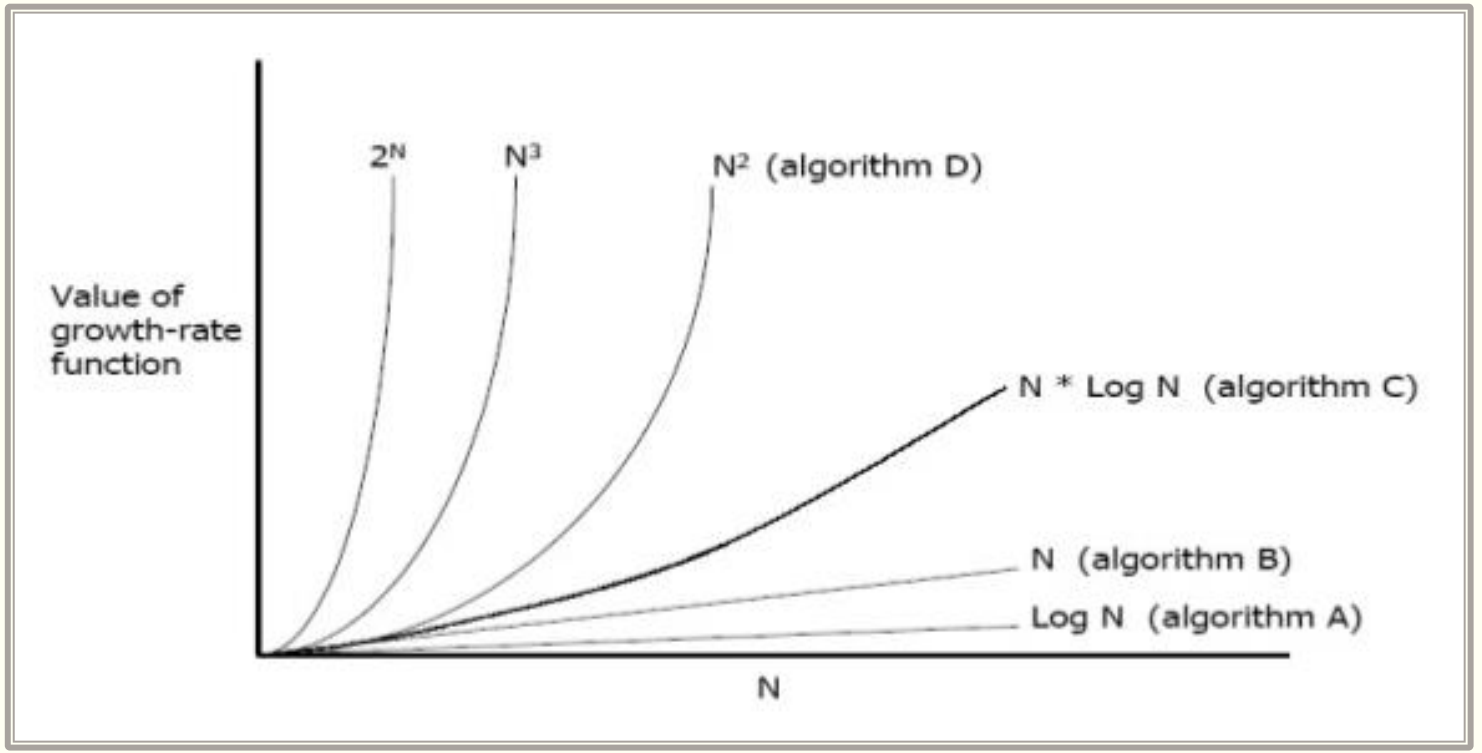
- ประมาณการเวลาทั้งหมดที่ต้องใช้ในโปรแกรมได้
- มุ่งแก้ไขไปที่อัลกอริธึ่มที่ใช้เวลาในการประมวลผลนานๆ ทำให้ไม่ต้องแก้ไขทั้งโปรแกรม
- เลือกคุณลักษณะของคอมพิวเตอร์ที่จะใช้ติดตั้งโปรแกรมที่พัฒนาขึ้นได้อย่างเหมาะสม

Growth of Functions (แนวโน้มการเพิ่มขึ้นของฟังก์ชั่น)

หลังจากที่เราได้ออกแบบอัลกอริธึ่มไว้หลายอัลกอริธึ่ม หรือเมื่อเราต้องเลือกอัลกอริธึ่มที่มีประสิทธิภาพที่สุดจาก 4 อัลกอริธึ่ม (A-D)

คำถาม เราจะพิจารณาจาก?

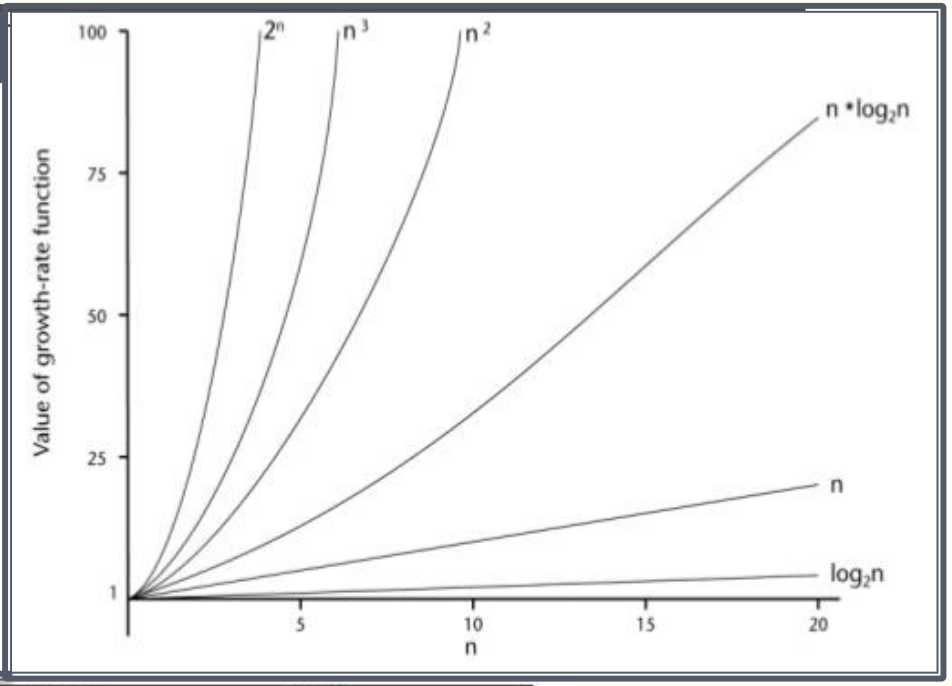
คำตอบ แนวโน้มการเพิ่มขึ้นของฟังก์ชั่น



[ภาพโดย อ. จีรพร วีระพันธุ์]

Functions of Growth Rate (อัตราการเติบโตของฟังก์ชั่น) เรียงจากน้อยไปมาก

Function	Name	Algorithms
c	Constant	Array index lookup
$\log N$	Logarithmic	Binary search
$\log^2 N$	Log-squared	
N	Linear	Graph traversal
$N \log N$	N log N	Merge, Quick sort
N^2	Quadratic	Shortest path
N^3	Cubic	Dynamic programming
2^N	Exponential	Traveling salesman



g(n) / n	log n	n	n log$_2$ n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

[ภาพ: internet 2017]

ประสิทธิภาพของอัลกอริธึ่ม

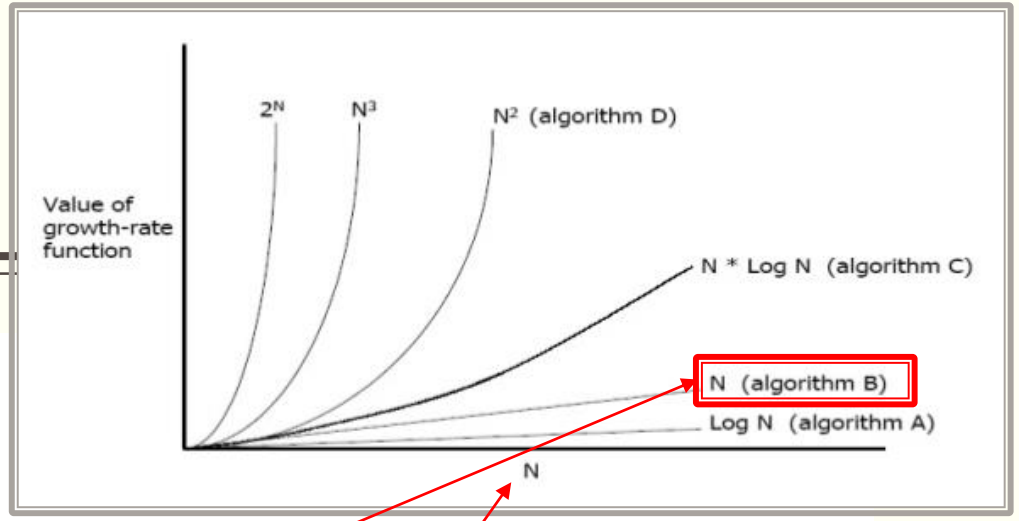
พิจารณาจำนวนอิลิเมนต์ (Elements) ที่จะประมวลผลหรือจากจำนวนรอบการทำงานของตัวดำเนินการนั้นๆ

- f(n) = efficiency

→ อัตราการเติบโตของฟังก์ชั่น (growth rates) ที่บอกความสัมพันธ์ระหว่างจำนวนข้อมูลนำเข้ากับความเร็วในการประมวลผล

- พิจารณาจากส่วนการทำงานของวนรอบประมวลผล (loop) เป็นสำคัญ

ลูปแบบเชิงเส้น (Linear Loops)



= มีการเพิ่ม หรือลดค่าภายในลูปแบบคงที่

```
for ( i = 0; i < 1000; i++ )
        application code
```

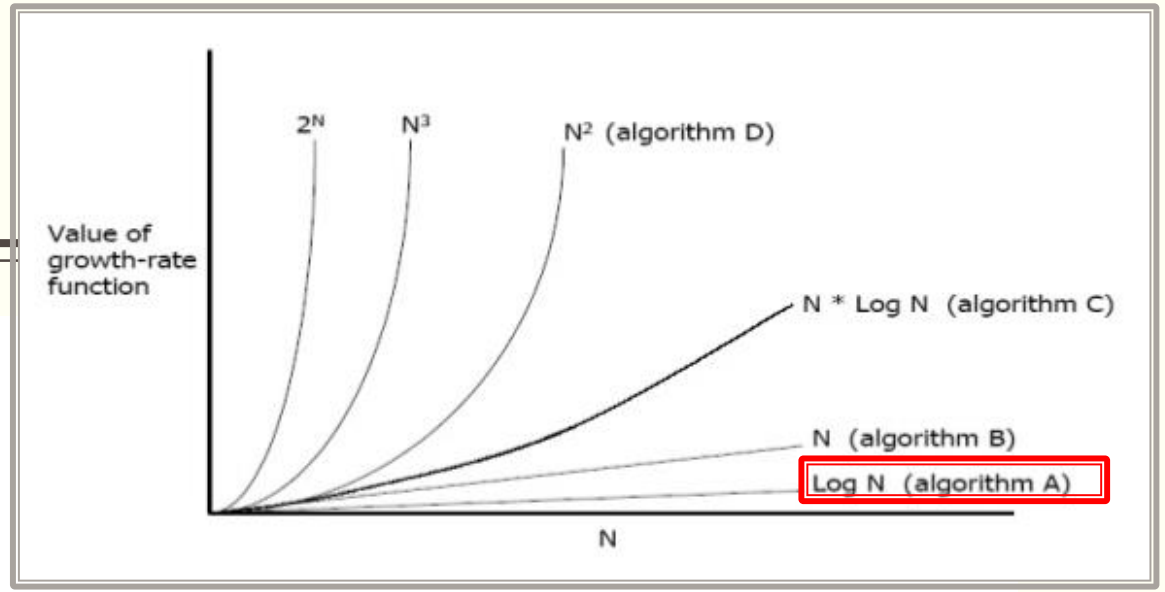
ประสิทธิภาพของอัลกอริธึ่มคือ f(n) = n

```
for ( i = 0; i < 1000; i+= 2 )
        application code
```

ประสิทธิภาพของอัลกอริธึ่ม f(n) = n / 2

→เมื่อพล๊อตจุดลงกราฟจะมีลักษณะเป็นเส้นตรงเหมือนกัน

ลูปแบบลอการิธมิค (Logarithmic Loops)



= เพิ่มหรือลดค่าภายในลูปสองเท่า

- Multiply Loops

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

- Divide Loops

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

ประสิทธิภาพของอัลกอริธึ่ม $f(n) = [\log_n]$ หรือ $[\log_2 n]$

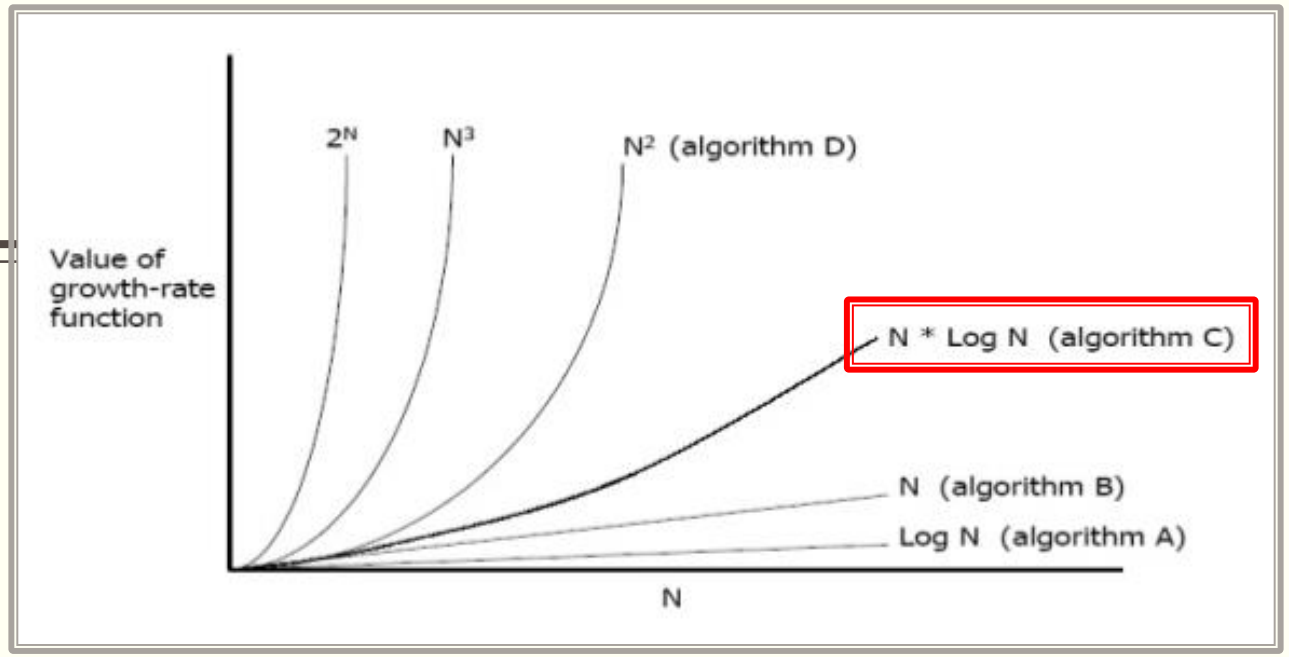
ลูปแบบซ้อน (Nested Loops)

= ภายในลูป จะมีลูปซ้อนอีกลูปหนึ่ง

- ✓ ยอดรวมที่ได้คือ ผลคูณของจำนวนลูปชั้นใน (Inner Loop) กับจำนวนของลูปชั้นนอก (Outer Loop)
- ✓ ลูปแบบซ้อนชนิดลอการิธมิคเชิงเส้น (Linear Logarithmic)
  - ลูปแบบซ้อนชนิดกำลังสอง (Quadratic)
  - ลูปแบบซ้อนกำลังสองชนิดขึ้นต่อกัน (Dependent Quadratic)

ลูปแบบซ้อนชนิดลอการิธมิคเชิงเส้น

```
for (i = 0; i < 10; i++)
      for (j = 1; j <= 10; j*= 2)
            application code
```

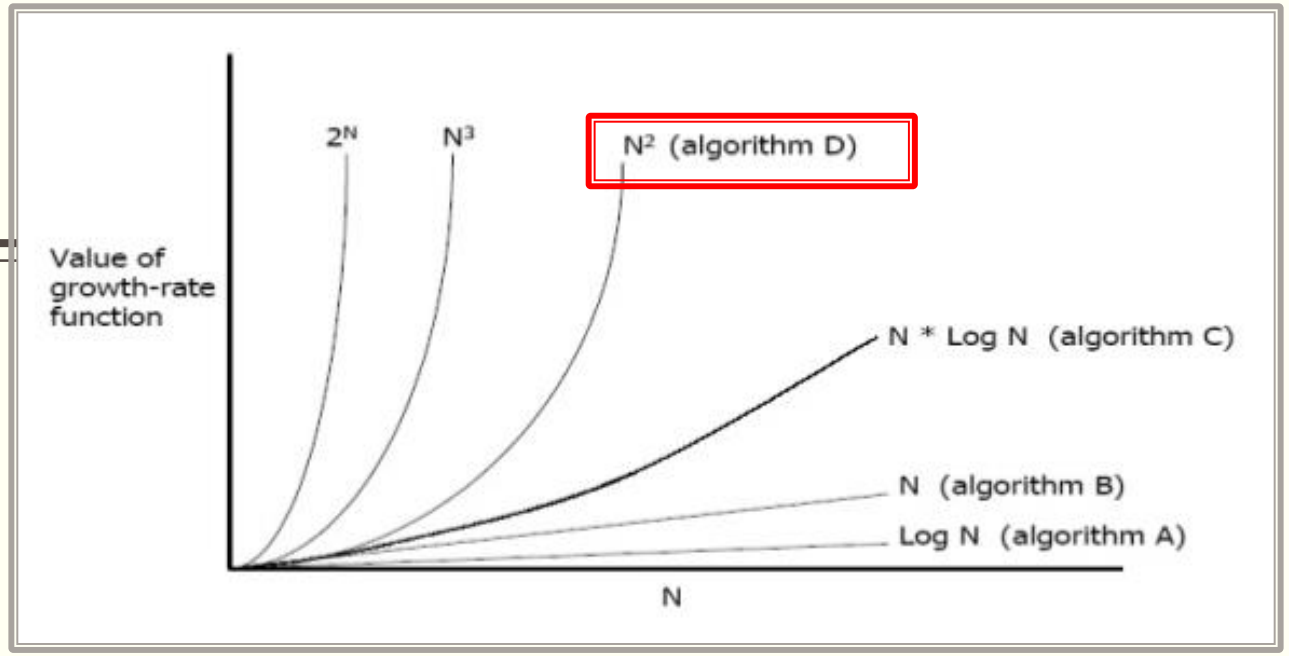


จำนวนรอบคือ 10 $\log_{10}$

f(n) = n log n

ลูปแบบซ้อนชนิดกำลังสอง (Quadratic)

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        application code
```

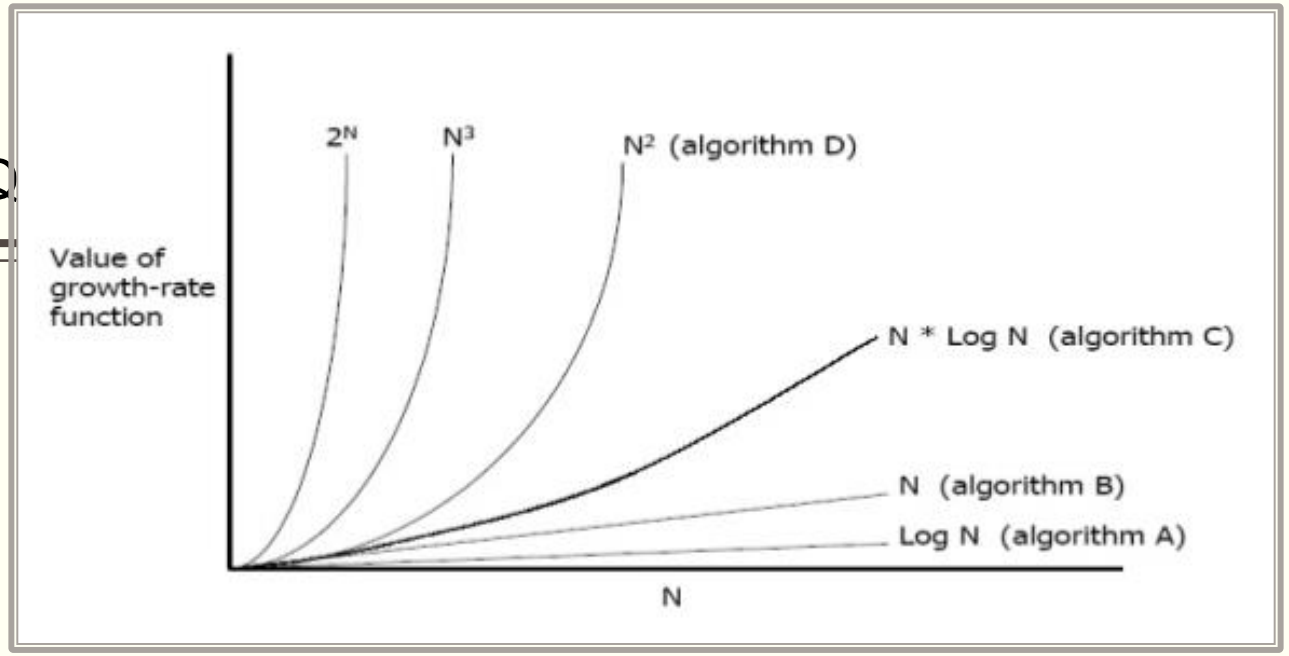


จำนวนรอบคือ 10 * 10

$f(n) = n^2$

ลูปแบบซ้อนชนิดขึ้นต่อกัน (Dependent Q

```
for (i = 0; i < 10; i++)
    for (j = 0; j <= i ; j++)
        application code
```



จำนวนรอบการทำงานลูปในคือ 1+2+3 ... + 10 = 55

f จำนวนรอบเฉลี่ยคือ 55/10 = 5.5

เทียบได้เท่ากับ **(n + 1)/2**

f(n) = n * (n+1)/2

2.4 ประสิทธิภาพเชิงเนื้อที่ (Space Efficiency)

ในการวิเคราะห์ประสิทธิภาพของอัลกอริธึ่มทางคอมพิวเตอร์ที่ได้ออกแบบไว้นั้น
มักพิจารณาจากความซับซ้อนของอัลกอริธึ่มใน 2 ด้านคือ

- **Time Complexity** (ความซับซ้อนด้านเวลา)
  วัดจากจำนวน n รอบที่ใช้ในการคำนวณ (+ - * /) เปรียบเทียบทางตรรกศาสตร์ (> = <) ใน CPU → loops

- **Space Complexity** (ความซับซ้อนด้านเนื้อที่)

  วัดจากเนื้อที่ทั้งหมดที่ใช้ในการเก็บข้อมูล ตัวแปรต่างๆ ไว้ใน main memory แต่จะมีความสำคัญน้อยกว่าความซับซ้อนด้านเวลา

2.5 การวิเคราะห์ Space Complexity

✓ Maximum memory=? ที่รัน algorithm แล้วมีจริงๆ เท่าไหร่

- วิเคราะห์ว่าต้องใช้หน่วยความจำทั้งหมดเท่าไหร่ในการประมวลผลอัลกอริธึ่มนั้น
  รองรับจำนวนข้อมูลที่ส่งเข้ามาประมวลผล ได้มากที่สุดเท่าใด
  เพื่อให้อัลกอริธึ่มนั้นสามารถประมวลผลได้อยู่
- ทราบขนาดของหน่วยความจำที่จะต้องใช้ประมวลผลอัลกอริธึ่มโดยไม่กระทบการประมวลผลอื่นๆ
- เพื่อเลือกคุณลักษณะของคอมพิวเตอร์ที่จะใช้ติดตั้งโปรแกรมที่พัฒนาขึ้นได้อย่างเหมาะสม

องค์ประกอบของ Space Complexity

✓ **Instruction Space**

-จำนวนของหน่วยความจำที่คอมไพเลอร์จำเป็นต้องใช้ขณะทำการคอมไพล์โปรแกรม

✓ **Data Space**

-จำนวนหน่วยความจำที่ต้องใช้สำหรับเก็บค่าคงที่ และตัวแปรทั้งหมดที่ต้องใช้ในการประมวลผลโปรแกรม

- **Static memory allocation**

จำนวนของหน่วยความจำที่ต้องใช้อย่างแน่นอน ไม่มีการเปลี่ยนแปลง ประกอบด้วยหน่วยความจำที่ใช้เก็บค่าคงที่และตัวแปรประเภท array เช่น การประกาศตัวแปร

```
int    a, b;
char   s[10],   c;
```

องค์ประกอบของ Space Complexity

- **Dynamic memory allocation**

จำนวนของหน่วยความจำที่ใช้ในการประมวลผลสามารถเปลี่ยนแปลงได้ และจะทราบจำนวนหน่วยความจำที่จะใช้ก็ต่อเมื่อโปรแกรมกำลังทำงานอยู่ เช่น การใช้ pointer และมีการจองเนื้อที่ให้หน่วยความจำด้วยคำสั่ง malloc();

```
int  *p;
p = malloc(sizeof(int)*2);
```

✓ **Environment Stack Space**

-จำนวนหน่วยความจำที่ต้องใช้ในการเก็บผลลัพธ์ของข้อมูลเอาไว้ เพื่อรอเวลาที่จะนำผลลัพธ์นั้นกลับไปประมวลผลอีกครั้ง (พบใน recursive function)