♡ BigO = ไปดูแบบฝึกในสมุด ♡

บทที่ 11 การวิเคราะห์ประสิทธิภาพของอัลกอริทึม

ในการเขียนโปรแกรม ผู้เขียนโปรแกรมสามารถเขียนแล้วรันออกมาได้ผลลัพธ์เหมือนกัน แต่บางครั้งโปรแกรมเหล่านี้อาจจะใช้ระยะเวลาในการประมวลผลหรือทรัพยากรที่ใช้ในการประมวลผลแตกต่างกัน แน่นนอนว่าโปรแกรมที่ดีต้องเป็นโปรแกรมที่ใช้เวลาน้อยและใช้ทรัพยากรน้อยที่สุด ซึ่งการที่จะวิเคราะห์ได้ว่าโปรแกรมไหนดีหรือไม่ นั้นต้องวิเคราะห์จากขั้นตอนวิธีที่ใช้ในการเขียนโปรแกรมนั้น ดังนั้นขั้นตอนวิธีจึงเป็นวิชาที่สำคัญมาก เป็นพื้นฐานของการศึกษาในสาขาด้านคอมพิวเตอร์ต่อไป

11.1 ความหมายขั้นตอนวิธี

ขั้นตอนวิธีหรืออัลกอริทึม (Algorithms) คือ ลำดับของขั้นตอนการคำนวณที่ใช้แก้ปัญหา โดยการเปลี่ยนข้อมูลนำเข้าของปัญหา (input) ออกมาเป็นผลลัพธ์ (output) ขั้นตอนวิธีดังกล่าวนั้นจะสามารถนำมาเขียนเป็นโปรแกรมในคอมพิวเตอร์ได้ หรือกระบวนการแก้ปัญหาที่สามารถเข้าใจได้ โดยมีลำดับหรือวิธีการในการแก้ไขปัญหาใดปัญหาหนึ่งอย่างเป็นขั้นเป็นตอนและชัดเจน เมื่อนำเข้าอะไร แล้วจะต้องได้ผลลัพธ์เช่นไร โดยทั่วไปขั้นตอนวิธีจะประกอบด้วย วิธีการเป็นขั้นๆ และมีส่วนที่ต้องทำแบบวนซ้ำ (iterate) หรือ เวียนเกิด (recursive) โดยใช้ตรรกะ (logic) และ/หรือ ในการเปรียบเทียบ (comparison) ในขั้นตอนต่างๆ จนกระทั่งเสร็จสิ้นการทำงาน ในการทำงานหรือแก้ปัญหาอย่างเดียวกัน การเลือกขั้นตอนวิธีที่ใช้แก้ปัญหาที่แตกต่างกันนั้น อาจได้ผลลัพธ์ออกมาเหมือนกันหรือไม่ก็ได้ โดยขั้นตอนวิธีที่แตกต่างกันนั้นจะส่งผลให้เวลา (time) และขนาดหน่วยความจำ (space) ที่ต้องการต่างกัน หรือเรียกได้อีกอย่างว่ามีความซับซ้อน (complexity) ต่างกัน

การนำขั้นตอนวิธีไปใช้ ไม่จำกัดเฉพาะการเขียนโปรแกรมคอมพิวเตอร์ แต่สามารถใช้กับปัญหาอื่น ๆ ได้เช่น การออกแบบวงจรไฟฟ้า, การทำงานเครื่องจักรกล, หรือแม้กระทั่งปัญหาในธรรมชาติ เช่น วิธีของสมองมนุษย์ในการคิดเลข หรือวิธีการขนอาหารของแมลง

หนึ่งในขั้นตอนวิธีอย่างง่าย คือ ขั้นตอนวิธีที่ใช้หาจำนวนที่มีค่ามากที่สุดในรายการซึ่งไม่ได้เรียงลำดับไว้ ในการแก้ปัญหานี้ต้องพิจารณาจำนวนทุกจำนวนในรายการ ซึ่งมีขั้นตอนวิธีดังนี้ และรหัสเทียม (Pseudo Code) สำหรับขั้นตอนวิธีนี้

1) พิจารณาแต่ละจำนวนในรายการ ถ้ามันมีค่ามากกว่า จำนวนที่มีค่ามากที่สุดที่เคยพบจดค่านั้นเอาไว้

2) จำนวนที่จดไว้ตัวสุดท้าย จะเป็นจำนวนที่มีค่ามากที่สุด

ขั้นตอนวิธีการหาค่ามากสุดระหว่าง จำนวน 3 จำนวน a, b, c

บรรทัด

```
1   If (a > b) then
2     If (a > c) then
3       return a
4     else
5       return c
6   else if (b > c) then
7     return b
8   else
9     return c
```

ขั้นตอนวิธี เพื่อหาค่ามากสุดของชุดข้อมูล $a_1, a_2, ..., a_n$
โดย a คือ อาร์เรย์ โดยถ้า a_1 คือตำแหน่งสมาชิกตัวที่ 1 เช่น a มีสามารถ คือ 5, 2, 4, 7, 8 ตามลำดับ ค่าของ a_1 คือ 5 และค่าของ a_5 คือ 8 ในกรณีนี้ n คือขนาดของอาร์เรย์

บรรทัด

```
1   large = a1
2   i = 2
3   if ( ai > large ) then
4     large = ai
5   i = i+1
6   If ( i > n ) then
7     return large
8   else
9     goto line 3
```

11.2 การวิเคราะห์ขั้นตอนวิธี

11.1 ความหมายการวิเคราะห์ประสิทธิภาพของอัลกอริทึม

การวิเคราะห์ประสิทธิภาพของอัลกอริทึม (Analysis of Algorithm Efficiency) โดยปกติประสิทธิภาพของขั้นตอนวิธี หรือ การวิเคราะห์ขั้นตอนวิธีสามารถพิจารณาได้ 2 ส่วนหลัก

ตัวอย่างการเปรียบเทียบ สมมุติ การแก้โจทย์คณิตศาสตร์ ต้องการหาจุดต่ำสุด (y น้อยสุด) ของสมการนี้ $y = \sum_{i=1}^{n} x_i^2$ โดยที่ x มีค่าอยู่ในช่วง 5 ถึง -5

อัลกอริทึม	เวลา (วัน)	พื้นที่(Mb)	results
A	1	100000000	100 TB มากไป
B	7	1000	เหมาะสม พอจะคำนวณได้
C	1000	1	3 ปีกว่า นานไป
D	90	200000	แย่มากใช้เวลามาก แถมพื้นที่มาก
E	60	10000	แย่มากใช้เวลามาก แถมพื้นที่มาก

ในการเปรียบเทียบ Algorithm ต้องพิจารณาถึง เวลาและพื้นที่ ที่ใช้ในการแก้ปัญหา ถ้าหากเปรียบเทียบ จะสามารถ วิเคราะห์ได้ว่า B > E > D สำหรับ C และ A ไม่เหมาะสมที่จะนำมาใช้แก้ปัญหานี้ เพราะใช้ Memory มากไป

1) **หน่วยความจำ (Memory)** เรียกว่าการวิเคราะห์หน่วยความจำ (Space Complexity) เป็นการวิเคราะห์การจองและการจัดการหน่วยความจำของขั้นตอนวิธี โดยองค์ประกอบคือ จำนวนหน่วยความจำที่ใช้ตอนรัน (run) และตอนคอมไพล์ (compile) สำหรับหน่วยความจำมี 2 แบบ คือ

Static คือ ขนาดหน่วยความจำที่ใช้เป็นค่าคงที่ เช่น อาร์เรย์ (Array)

Dynamic คือ ขนาดหน่วยความจำที่ใช้สามารถเปลี่ยนแปลงได้ เช่น วัตถุ (Object)

หมายเหตุ กรณีการเขียนโปรแกรมรูปแบบเวียนเกิดยิ่งลึกมาก ยิ่งใช้หน่วยความจำมาก

2) **เวลา (Time)** เรียกว่าการวิเคราะห์เวลา (Time Complexity) เป็นการวิเคราะห์ประสิทธิภาพความเร็วในการรันขั้นตอนวิธี โดยองค์ประกอบคือ เวลาที่ใช้ตอนรัน โดยส่วนนี้จะขึ้นอยู่กับโปรแกรมที่เขียน และเป็นส่วนที่ใช้พิจารณาขั้นตอนวิธี เพราะตอนคอมไพล์หรือตรวจสอบวากยสัมพันธ์ (syntax) ส่วนนี้ไม่ขึ้นอยู่กับโปรแกรมที่เขียน การวิเคราะห์เวลาสามารถกระทำได้โดยการจับเวลา และทดลองรันกับข้อมูลหลายชุด ซึ่งบางชุดอาจจะใช้เวลาน้อย บางชุดอาจจะใช้เวลามาก แล้วดูว่าคอมพิวเตอร์ใช้เวลาในการรันนานไหม โดยรันหลายๆ ครั้งแล้วหาค่าเฉลี่ย

- เวลากระทำการน้อย (น้อยสุด คือเวลาดีที่สุด best- case time)
- เวลากระทำการมาก (มากสุด คือเวลาแย่ที่สุด worst-case time)
- เวลากระทำโดยเฉลี่ย (เวลาโดยเฉลี่ย average-case time)
-

?ดูที่จำนวนการกระทำ

การพิจารณาตามขั้นตอนวิธี คือการพิจารณาเฉพาะขั้นตอนที่สำคัญของปัญหาที่ต้องการแก้ เช่น จำนวนการกระทำหรือเวลา ตัวนี้คือเอาโค้ด (code) มาวิเคราะห์ว่าต้องใช้ทรัพยากรหรือเวลาในการรันเท่าไรโดยประมาณ

ตัวอย่างที่ 1 จงพิจารณาจำนวนหรือเวลา กระทำคำสั่งเปรียบเทียบ a_i > large และ a คือ อาร์เรย์ โดยถ้า a_1 คือตำแหน่งสมาชิกตัวที่ 1 เช่น a มีค่า 5, 2, 4, 7, 8 ตามลำดับ ค่าของ a_1 คือ 5 และค่าของ a_5 คือ 8 ในกรณีนี้ n คือขนาดของอาร์เรย์

บรรทัด		
1	Large = a_1	ให้ t(n) เป็นจำนวนหรือเวลา สำหรับโค้ดชุดนี้
2	i = 2	สมมุติ n = 5 หมุน 5 รอบ
3	if (a_i > Large) then	รอบแรก i = 3 → 3 > 5
4	Large = a_i	รอบสอง i = 4 → 4 > 5
5	i = i+1	รอบสาม i = 5 → 5 > 5
6	if (i > n) then	รอบสี่ i = 6 → 6 > 5 หยุด → ทำ 4 ครั้งจาก n = 5 ดังนั้น n-1
7	return Large	
8	else	
9	goto line 3	

ตัวอย่างที่ 2 จงพิจารณาจำนวนหรือเวลา // loop ลดลง

บรรทัด		
1	j=n	ให้ t(n) เป็นจำนวนหรือเวลา สำหรับโค้ดชุดนี้
2	while (j ≥ 1)	สมมุติ n = 5
3	{	รอบแรก j = 5 → for i to j ทำอีก 5 รอบ
4	for i = 1 to j	รอบสอง j = 4 → for i to j ทำอีก 4 รอบ
5	x = x+1	รอบสาม j = 3 → for i to j ทำอีก 3 รอบ
6	j = j-1	รอบสี่ j = 2 → for i to j ทำอีก 2 รอบ
7	}	รอบห้า j = 1 → for i to j ทำอีก 1 รอบ

$5 + 4 + 3 + 2 + 1 = 15 = \frac{5(5+1)}{2}$ คือ $\frac{n(n+1)}{2}$ → bigO = n^2

ตัวอย่างที่ 3 จงพิจารณาจำนวนหรือเวลา

บรรทัด		
1	j=n	ให้ t(n) เป็นจำนวนหรือเวลา สำหรับโค้ดชุดนี้
2	while (j > 1)	สมมุติ n = 8
3	{	รอบแรก j = 8 → 4 = 8/2
4	j = j/2	รอบสอง j = 4 → 2 = 4/2
5	}	รอบสาม j = 2 → 1 = 2/2

จำนวนรอบทั้งหมด คือ 3 รอบ ซึ่งมีค่าเท่ากับ $\log_2 8 = 3 = \log_2 n$

ตัวอย่างที่ 4 จงพิจารณาจำนวนหรือเวลา for ซ้อน for = n^2

บรรทัด		
1	j=n	ให้ t(n) เป็นจำนวนหรือเวลา สำหรับโค้ดชุดนี้
2	while (j >= 1)	สมมุติ n = 5
3	{	รอบแรก j = 5 → for i to n ทำอีก 5 รอบ
4	i=n	รอบสอง j = 4 → for i to n ทำอีก 5 รอบ
5	while (i >= 1)	รอบสาม j = 3 → for i to n ทำอีก 5 รอบ
6	{	รอบสี่ j = 2 → for i to n ทำอีก 5 รอบ
7	i = i -1	รอบห้า j = 1 → for i to n ทำอีก 5 รอบ
8	}	จำนวนรอบทั้งหมด คือ 25 รอบ ซึ่งมีค่าเท่ากับ nxn หรือ n^2
9	j = j-1	
10	}	

การวิเคราะห์แบบ worst case, best case, average case นับจำนวนครั้งของ operation พื้นฐาน สอดคล้องกับรูปแบบข้อมูล input นอกจาก input n แล้ว algorithm ขึ้นอยู่กับลักษณะของข้อมูลด้วย แบ่งเป็น 3 กรณี

worst case $t_w(n)$ คือ เวลาการทำงานที่มากที่สุด ที่เป็นไปได้สำหรับ input n

best case $t_b(n)$ คือ เวลาการทำงานที่น้อยที่สุด ที่เป็นไปได้สำหรับ input n

average case $t_a(n)$ คือ เวลาการทำงานเฉลี่ย ที่เป็นไปได้สำหรับ input n กรณีนี้ เราต้องหา input ทุกรูปแบบที่เป็นไปได้ทั้งหมด จากนั้นทำการหาค่าเฉลี่ยเวลาของการทำงาน

ขั้นตอนวิธีต้องทำงานให้ถูกต้อง แต่ต้องทำงานอย่างมีประสิทธิภาพ ดังนั้น การประเมินประสิทธิภาพของขั้นตอนวิธี (algorithm effeciency) ถือเป็นงานที่มีความสำคัญสำหรับ programmer ในการวิเคราะห์ ประสิทธิภาพของขั้นตอนวิธี วัดในเชิงเวลาตั้งแต่เริ่มต้นจนจบการทำงาน

* ไม่นิยม *

1) การวิเคราะห์เวลาโดยตรง (Empirical analysis) การวัดเวลาโดยตรง (Real time measurement) ไม่เหมาะสมเพราะมีหลายปัจจัย เช่น CPU speed, ตัว complier, OS, ความยาวของ code เป็นวิธีการจับเวลารันโปรแกรมตามจริง ซึ่งในความจริงมีปัจจัยหลายอย่าง เช่น คอมพิวเตอร์เครื่องใหม่เปรียบเทียบกับคอมพิวเตอร์เก่าถึงโปรแกรมคอมเครื่องเก่าจะดีกว่าคอมเครื่องใหม่ แต่เนื่องจากคอมเครื่องใหม่เร็วกว่าจึงประมวลผลเสร็จก่อน

2) การวิเคราะห์ทางทฤษฎี (Theoretical analysis) ใช้วิธีนับจำนวนรอบของการทำงานของโปรแกรม ซึ่งการวิเคราะห์รูปแบบนี้ต้องพิจารณาผลกระทบจากจำนวนอินพุทข้อมูล (Efficiency as a function of input size) คือ อินพุทข้อมูลขนาดใหญ่ ส่วนมากแล้ว ขั้นตอนจะใช้เวลามากขึ้น เช่น sort ตัวเลข 100 ตัว กับ sort ตัวเลข 10000000 ตัว แต่บางครั้งขนาดอินพุทข้อมูลไม่ได้ขึ้นอยู่กับจำนวนข้อมูล แต่อยู่กับค่าอินพุท เช่นทดสอบจำนวนเฉพาะ ถ้าใส่ 23 กับ 13223596 การตรวจสอบว่าเป็นจำนวนเฉพาะไหม ขนาดอินพุทข้อมูลโดยปกติจะแทนด้วยตัวอักษร n สำหรับอัลกอริทึมส่วนมากใช้เวลามากขึ้น เมื่อข้อมูลมีปริมาณมากขึ้น ดังนั้นประสิทธิภาพอัลกอริทึมจึงเป็นฟังก์ชันที่ขึ้นอยู่กับขนาดของพารามิเตอร์ n ที่ระบุขนาดอินพุท การวัดขนาดอินพุทมักเกี่ยวข้องกบคุณสมบัติของตัวเลข เช่น การตรวจสอบว่า เป็นจำนวนเฉพาะหรือไม่ เป็นต้น

สำหรับนับจำนวนรอบของการทำงานของโปรแกรมมีวิธีคิด 2 แบบ คือ 1 การนับโอเปอเรชั่นทั้งหมด (Elementary operation counting) และ 2 การนับโอเปอเรชั่นพื้นฐาน (basic operation counting) ดังตัวอย่างต่อไปนี้

2.1) การนับโอเปอเรชั่นทั้งหมด (Elementary operation counting) นับจำนวนครั้งของทุกบรรทัดคำสั่งของ algorithm ตัวอย่างเช่น

โค้ดโปรแกรม	จำนวนรอบ
int power(int x, int n)	1
{	
p = 1	1
for (int i=0 ; i < n ; i++)	n+1 loop
{	
p = p * x;	n
}	
return p;	1
}	รวมแล้วเป็น T(n) = 1 + 1 + (n+1) + n + 1
int mystery (x,n)	1
{	
S = 0	1
for(int i = 1 ; i < n ; i++)	n+1 loop
{	
for(int j=1 ; j < n ; j++)	$(n^2 + n)$ loop ซ้อน loop
{	
S = S + 1	n^2 * อยู่ใน loop ซ้อน loop *
}	
}	
return S;	1
}	รวมแล้วเป็น $T(n) = 1 + 1 + (n+1) + (n^2 + n) + n^2 +1$
int sum(n)	1
{	
S = 0	1
i = 1	1
while(i <= n)	n+1
{	
S = S + 1;	n * ถ้าอยู่ใน loop เดียว => n *
i = i +1;	n
}	
return S;	1
}	รวมแล้วเป็น T(n) = 1 + 1 +1 + (n+1) + n + n + 1
int product_matrix(a[m] [n], b[n] [p])	1
{	
for(int i = 1 ; i < m ; i++)	m+1
{	99
for(int j = 1 ; j < p ; j++)	m*(p+1)
{	
c[i, j] = 0;	m*p
for(int k = 1 ; k < n ; k++)	m*p*(n+1)

`{`	
`c[i][j] = c[i][j] + a[i][k] * b[k] [j];`	m*p*n
`}`	
`}`	
`}`	
`return c[m][p];`	1
`}`	รวมแล้วเป็น T(m,p,n) = 1 + (m+1) + [m*(p+1)] + [m*p] + [m*p*(n+1)] + [m*p*n] + 1

2.2) การนับโอเปอเรชั่นพื้นฐาน (basic operation counting) นับเฉพาะบรรทัดที่ถูก execute มากที่สุด สัมพันธ์ขนาด input n เพราะ การนับทุกบรรทัด ทำได้ยากและเสียเวลา, การประมาณเวลา โดยนับเฉพาะ basic operation ทำได้ง่ายและนิยมมากกว่า, basic operation คือบรรทัดคำสั่งซึ่งถูก execute มากที่สุด สัมพันธ์กับขนาด input เวลาในการประมวลผลโปรแกรมที่พัฒนาจากอัลกอริทึม T(n) สามารถหาได้จากสูตรนี้คือ $T(n) \approx C_0 \times C(n)$ โดยที่ C_0 คือ เวลาที่ใช้ในการประมวลผลการทำงานพื้นฐาน (basic operation) บนคอมพิวเตอร์เครื่องหนึ่ง และ C(n) = จำนวนครั้งของการทำงานของ basic operation ในอัลกอริทึม ขึ้นอยู่กับขนาดของอินพุต n สมมติว่าฟังก์ชันเวลาของขั้นตอนวิธี คือ T(n) = 60n + 5 สำหรับอินพุต n มีขนาดมากสุดขึ้นอยู่กับค่าของ 60n ตัวนี้เท่านั้น

n	T(n) = 60n + 5	T(n) ~ 60n	Error
10	605	600	0.826
100	6,005	6,000	0.083
1,000	60,005	60,000	0.008
10,000	600,005	600,000	0.001
100,000	6,000,005	6,000,000	0.000

`int power(int x, int n)`	
`{`	
`p = 1`	
`for (int i=0 ; i < n ; i++)`	
`{`	
`p = p * x;`	t4 = Basic Operation
`}`	
`return p;`	
`}`	รวมแล้วเป็น T(n) = t4 = n
`int mystery (x,n)`	
`{`	
`S = 0`	
`for(int i = 1 ; i < n ; i++ )`	
`{`	n^2
`for(int j=1 ; j < n ; j++)`	
`{`	
`S = S + 1`	t5 = Basic Operation
`}`	
`}`	
`return S;`	
`}`	รวมแล้วเป็น T(n) = t5 = n^2 ✓✓
`int sum(n)`	
`{`	
`S = 0`	
`i = 1`	
`while( i <= n )`	t4
`{`	
`S = S + 1;`	t5
`i = i +1;`	t6
`}`	
`return S;`	
`}`	รวมแล้วเป็น T(n) = t4 หรือ t5 หรือ t6 = n

t4, t5, t6 เป็น basic operation ได้ เนื่องจากมีจำนวนรอบทำงานไม่แตกต่างกันมาก

int sequential_search (A[n], K)
{
i = 0;
while (i < n)
{
if (A[i] == K) //ไม่แน่นอน
{
return true;
}
i = i + 1;
}
return false;
}
	จาก code นี้ไม่สามารถสรุปเวลาที่ชัดเจนได้
	ถ้า { 1 2 3 4 5 } ถ้าหาเลข 1 เวลาแค่ 1 รอบ best case
	ถ้าหาเลข 5 ใช้เวลา 5 รอบ worst case

int minimun(A[n])
{
int m = A[0];
for (int i=2 ; i < n ; i++) n
{
if (m > A[i])
{
m = A[i];
}
}
return m;
}
	จำนวน Loop คงที่ คือ n ดังนั้น $t_w(n) = t_b(n) = t_a(n)$ คือ n

ในทางปฏิบัติ เราไม่ทราบ C_0 ที่แท้จริงแต่สามารถทราบเวลาที่เพิ่มขึ้นลดลงได้ เมื่อ input เปลี่ยน เช่น $C(n) = C_0 \times n^2$ หากจำนวน input เพิ่มขึ้น 2 เท่า เวลาทำงาน เพิ่มขึ้น 4 เท่า

ตัวอย่างปัญหา sequential search

$t_w(n) = n$

$t_b(n) = 1$

$t_a(n) = ?$ สำหรับกรณีนี้ มี input ทั้งหมด 2 รูปแบบ คือ input ไม่พบคำตอบ กับ พบคำตอบ กรณี ไม่พบคำตอบหา n ตัว กรณีพบคำตอบตัวแรกคือ 1 กรณีพบตัวสุดท้าย คือ n กำหนดให้ p คือ ความน่าจะเป็นที่ค่า k ปรากฏในข้อมูล ดังนั้น (1-p) ความน่าจะเป็นที่ไม่พบค่า k ดังนั้น p / n คือ ความน่าจะเป็นเฉลี่ยที่จะพบ k ในแต่ละตำแหน่ง สำหรับข้อมูล input ขนาด n

$t_a(n)$ = (1 {หาพบตัวแรก} + 2 {หาพบตัวสอง} + 3 + ... n {หาพบตัวสุดท้าย}) / n

$t_a(n) = [n(n+1) / 2] / n = (n+1) / 2$

ปัญหากำหนดให้มีชุดข้อมูลใน array จำนวน n ชุด ให้หาชุดข้อมูลทีมีค่า เท่ากับ K สำหรับขั้นตอนวิธีทำการตรวจสอบข้อมูลทีละตัวไปเรื่อยๆ วามีตัวใดมีค่าเท่ากับ K จนกว่าจะพบ (successful search) หรือจะหมดข้อมูลที่จะทำการค้นหา (unsuccessful search) โดย กรณี Worst case : n , กรณี Best case : 1, กรณี Average case : n/2

11.3 การเปรียบเทียบขั้นตอนวิธี

ขั้นตอนวิธีที่ได้ บางครั้งไม่สามารถบอกจำนวนหรือเวลาที่แท้จริงได้ ถ้าต้องการเปรียบเทียบขั้นตอนวิธีเหล่านี้ที่แก้ปัญหาเดียวกัน จะต้องปรับขั้นตอนวิธีให้เข้าสู่ตัวอ้างอิงตัวเดียวกัน โดยการตัวอ้างอิงที่นิยมใช้ คือ บิ๊กโอ (Big O) โดย *O* คือสัญลักษณ์ของ Big O ซึ่งเป็นการปรับให้มีค่าโดยประมาณที่มากที่สุด โดยหลักการคือการยุบเศษเข้ากับตัวมากที่สุดเพื่อให้ได้ค่าที่มากที่สุดค่าของขั้นตอนวิธี โดยเรียงลำดับประสิทธิภาพจากมากไปหาน้อยดังนี้ $c > \log N > N > N^2 > 2^n > N!$ แย่สุด ดีสุด

บิ๊กโอ คือ กรณีที่ใช้เวลากระทำการมาก (worst case) ของขั้นตอนวิธี ซึ่งหมายความว่า ขั้นตอนวิธีนี้จะทำงานไม่แย่ไปกว่า Big O ของมันแล้ว ซึ่งก็เหมือนกับเป็นตัวบอกถึง ประสิทธิภาพของขั้นตอนวิธี เพื่อใช้ในการเปรียบเทียบขั้นตอนวิธีหลายวิธีเข้าด้วยกัน

ขั้นตอนวิธีที่อยู่ใน O(n) ก็หมายความว่า ถ้าใช้ขั้นตอนวิธีนี้ประมวลข้อมูล n อย่าง มีขั้นตอนในการประมวลผลประมาณ n ครั้ง และไม่ช้าไปกว่านี้ ส่วนขั้นตอนวิธีที่อยู่ใน $O(n^3)$ ก็หมายความว่าถ้าใช้ขั้นตอนวิธีนี้ประมวลข้อมูล n อย่าง มีขั้นตอนในการประมวลผลประมาณ n^3 ครั้ง และไม่ช้าไปกว่านี้ ด้วยวิธีนี้ Big O ของขั้นตอนวิธีใดๆ จะสามารถเปรียบเทียบความสามารถในการแก้ปัญหาของขั้นตอนวิธีนั้นๆ เช่นตัวอย่างนี้ O(n) นั้นเร็วกว่า $O(n^3)$ ถ้าประมวลข้อมูลเดี่ยวกัน

ในแง่ของการเปรียบเทียบ Big O สามารถนำมาวัดระดับความสามารถของขั้นตอน เช่นกัน O(log n) อันนี้เร็วมาก ส่วน $O(2^n)$ อันนี้ช้ามาก ถ้าให้เรียงลำดับ Big O มาตรฐานที่พบบ่อย มีเปรียบเทียบกับความเร็วจากเร็วมากไปหาเร็วน้อยนั้นมีรายละเอียดดังนี้ O(1) > O(log n) > O(n) > O(n log n) > $O(n^2)$ > $O(2^n)$ > O(n!)

โดย O(1) คือ Constant Time Algorithm เป็นขั้นตอนวิธีที่เร็วที่สุด คือ ทำงานทุกครั้งใช้เวลาเท่าเดิมเสมอ ไม่ว่า ข้อมูลนำเข้าจะมีค่ามากแค่ไหน ขั้นตอนวิธีประเภทนี้ไม่ค่อยมีให้เห็นนัก และแก้ปัญหาได้ไม่กี่อย่าง เช่น การแทรกค่าใหม่เข้าไปที่จุดท้ายสุดของรายการโยง (linked list) หรือโปรแกรมที่ไม่มีลูปเลย หรือลูปที่กำหนดตายตัว

โดย O(log n) คือ Logarithmic Algorithm เป็นขั้นตอนที่เริ่มมีจริง เรียกได้ว่าเป็นขั้นตอนวิธีที่ที่เร็วที่สุดในความเป็นจริงแล้ว ตัวอย่างเช่น สมมุติค่าเป็นลอการิทึมฐาน 2 ใส่ค่าไป 8 แต่วนซ้ำแค่ 3 ครั้ง ก็สามารถได้ผลลัพธ์ เป็นต้น

โดย O(n) คือ Linear Algorithm เป็นขั้นตอนที่ทำงานเท่ากับตัวข้อมูลนำเข้า ตัวอย่างเช่น สมมุติใส่ค่าไป 10 แต่วนซ้ำแค่ 10 ครั้ง ก็สามารถได้ผลลัพธ์ เป็นต้น

โดย $O(n^2)$ คือ Quadratic Algorithm เป็นขั้นตอนที่ทำงานยกกำลังเท่ากับตัวข้อมูลนำเข้า ตัวอย่างเช่น สมมุติใส่ค่าไป 10 แต่วนซ้ำ 100 ครั้ง ก็สามารถได้ผลลัพธ์ เป็นต้น พวกนี้มักเป็นลูปซ้อนลูปอีกที

โดย $O(2^n)$ คือ Polynomial Algorithm เป็นขั้นตอนที่พบในกรณีเวียนเกิด (recursive) เช่น ปริศนาหอคอยฮานอยจำนวนครั้งในการทำงานคือ $a_n = 2^n-1$ เป็นต้น

โดย O(n!) คือ Factorial Algorithm เป็นขั้นตอนที่พบในกรณีเวียนเกิด เช่น การสลับเปลี่ยนค่า สำหรับ {1,2} สามารถสลับได้ 2 ครั้ง คือ {1,2} กับ {2,1} หรือ 2! สำหรับ {1,2,3} สามารถสลับได้ {1,2,3} {1,3,2} {2,1,3} {2,3,1} {3,2,1} {3,1,2} จำนวน 6 ครั้ง หรือ 3! เป็นต้น

ตัวอย่างที่ 1 f(n) = 2n-1 จงหา Big O

f(n) = 2n-1 คือ 2n มีค่าเยอะมาก n คืออนันต์ แล้วเศษคือ -1 เอาเศษยุบรวมเข้าไปที่ n

= 2n + n

$\boxed{O(f(n)) = \cancel{3}n}$ โดยที่ $f(n) \leq 3n$ แต่การเขียน Big O ไม่คิดตัวเลขหน้า n ดังนั้น คำตอบคือ $\boxed{n}$

หมายเหตุ สำหรับกรณี Big O ไม่ว่าพจน์ที่ความเร็วมากกว่าพจน์ที่ช้าที่สุด ในกรณีนี้คือ 1 ซึ่งไม่ว่าจะเป็นบวกหรือลบ ให้บวกเพิ่มไปอีก 1 สำหรับพจน์ที่มีค่ามากที่สุดเสมอ ในกรณีนี้คือ 2n กลายเป็น 3n แต่การเขียน Big O สามารถไม่คิดตัวเลขหน้า n ที่ช้าที่สุด เพราะจะทำให้เปรียบเทียบกันได้ง่ายกว่า ดังนั้นจึงนิยมไม่เขียนตัวเลขหน้า n ที่ช้าที่สุด ทำให้คำตอบที่ได้ n

ตัวอย่างที่ 2 $f(n) = n^2+n+3$ จงหา Big O

$f(n) = n^2+n+3$ คือ n^2 มีค่าเยอะมาก เมื่อเปรียบเทียบกับ n เอาเศษยุบรวมเข้าไปที่ n^2

$\boxed{O(f(n)) = \cancel{2}n^2}$ โดยที่ $f(n) \leq 2n^2$ แต่การเขียน Big O ไม่คิดตัวเลขหน้า n ดังนั้น คำตอบคือ $\boxed{n^2}$

ตัวอย่างที่ 3 $f(n) = 2n^2+7n$ จงหา Big O

$f(n) = 2n^2+7n$ คือ n^2 มีค่าเยอะมาก เมื่อเปรียบเทียบกับ n เอาเศษยุบรวมเข้าไปที่ n^2

$\boxed{O(f(n)) = \cancel{3}n^2}$ โดยที่ $f(n) \leq 3n^2$ แต่การเขียน Big O ไม่คิดตัวเลขหน้า n ดังนั้น คำตอบคือ $\boxed{n^2}$

11.4 การพิสูจน์อุปนัยเชิงคณิตศาสตร์

การให้เหตุผลแบบนิรนัย (Deductive Reasoning) คือ การนำความรู้พื้นฐาน หรือนิยามซึ่งเป็นสิ่งที่รู้มาก่อน และยอมรับเป็นจริงนำไปสู่ข้อสรุป เช่น จำนวนคู่ คือจำนวนที่หาร 2 ลงตัว ความรู้คือ 10 หาร 2 ลงตัว ดังนั้น 10 คือ จำนวนคู่ เป็นต้น การให้เหตุผลแบบนิรนัยเป็นการนำความรู้พื้นฐานซึ่งอาจเป็นความเชื่อ ข้อตกลง กฎ หรือบทนิยาม ซึ่งเป็นสิ่งที่รู้มาก่อน และยอมรับว่าเป็นความจริงเพื่อหาเหตุผลนำไปสู่ข้อสรุป เป็นการอ้างเหตุผลที่มีข้อสรุปตามเนื้อหาสาระที่อยู่ภายในขอบเขตของข้ออ้างที่กำหนด

ตัวอย่างที่ 1 เหตุ 1. สัตว์เลี้ยงทุกตัวเป็นสัตว์ไม่ดุร้าย

2. แมวทุกตัวเป็นสัตว์เลี้ยง

ผล แมวทุกตัวเป็นสัตว์ไม่ดุร้าย

ตัวอย่างที่ 2 เหตุ 1. นักเรียน ม.4 ทุกคนแต่งกายถูกระเบียบ

2. สมชายเป็นนักเรียนชั้น ม.4

ผล สมชายแต่งกายถูกระเบียบ

ตัวอย่างที่ 3 เหตุ 1. วันที่มีฝนตกทั้งวันจะมีท้องฟ้ามืดครึ้มทุกวัน

2. วันนี้ท้องฟ้ามืดครึ้ม

ผล วันนี้ฝนตกทั้งวัน

การให้เหตุผลแบบอุปนัย (Inductive Reasoning) คือ วิธีการสรุปเกิดจากค้นคว้า สังเกตหรือทดลองหลายครั้ง แล้วนำมาสรุปเป็นความรู้แบบทั่วไป เช่น สังเกตพระอาทิตย์ขึ้นทางทิศตะวันออกทุกวัน จึงสรุปว่าพระอาทิตย์ขึ้นทางทิศตะวันออก

การให้เหตุผลแบบอุปนัย เป็นวิธีการสรุปผลมาจากการค้นหาความจริงจากการสังเกตหรือการทดลองหลายครั้งจากกรณีย่อยๆ แล้วนำมาสรุปเป็นความรู้แบบทั่วไป

การหาข้อสรุปหรือความจริงโดยใช้วิธีการให้เหตุผลแบบอุปนัยนั้น ไม่จำเป็นต้องถูกต้องทุกครั้ง เนื่องจากการให้เหตุผลแบบอุปนัยเป็นการสรุปผลเกิดจากหลักฐานข้อเท็จจริงที่มีอยู่ ดังนั้นข้อสรุปจะเชื่อถือได้มากน้อยเพียงใดนั้นขึ้นอยู่กับลักษณะของข้อมูล หลักฐานและข้อเท็จจริงที่นำมาอ้างซึ่งได้แก่

1. จำนวนข้อมูล หลักฐานหรือข้อเท็จจริงที่นำมาเป็นข้อสังเกตหรือข้ออ้างมีมากพอกับการสรุปความหรือไม่ เช่น ถ้าไปทานส้มตำที่ร้านอาหารแห่งหนึ่งแล้วท้องเสีย แล้วสรุปว่า ส้มตำนั้นทำให้ท้องเสีย การสรุปเหตุการณ์นั้นอาจเกิดขึ้นเพียงครั้งเดียว ย่อมเชื่อถือได้น้อยกว่าการที่ไปรับประทานส้มตำบ่อยๆ แล้วท้องเสียเกือบทุกครั้ง

2. ข้อมูล หลักฐานหรือข้อเท็จจริง เป็นตัวแทนที่ดีในการให้ข้อสรุปหรือไม่ เช่น ถ้าอยากรู้ว่าคนไทยชอบกินข้าวเจ้าหรือข้าวเหนียวมากกว่ากัน ถ้าถามจากคนที่อาศัยอยู่ในภาคเหนือหรือภาค-อีสาน คำตอบที่ตอบว่าชอบกินข้าวเหนียวอาจจะมีมากกว่าชอบกินข้าวจ้าว แต่ถ้าถามคนที่อาศัยอยู่ในภาคกลางหรือภาคใต้ คำตอบอาจจะเป็นในลักษณะตรงกันข้าม

ตัวอย่างการให้เหตุผลแบบอุปนัย

ตัวอย่างที่ 1 จงหาว่า ผลคูณของจำนวนเต็มบวกสองจำนวนที่เป็นจำนวนคี่ จะเป็นจำนวนคู่หรือจำนวนคี่ โดยการให้เหตุผลแบบอุปนัย วิธีทำ ลองหาผลคูณของจำนวนเต็มบวกที่เป็นจำนวนคี่หลาย ๆ จำนวนดังนี้

$1 \times 3 = 3$	$3 \times 5 = 15$	$5 \times 7 = 35$	$7 \times 9 = 63$
$1 \times 5 = 5$	$3 \times 7 = 21$	$5 \times 9 = 45$	$7 \times 11 = 77$
$1 \times 7 = 7$	$3 \times 9 = 27$	$5 \times 11 = 55$	$7 \times 13 = 91$
$1 \times 9 = 9$	$3 \times 11 = 33$	$5 \times 13 = 65$	$7 \times 15 = 105$

จากการหาผลคูณดังกล่าว โดยการอุปนัย จะพบว่า ผลคูณที่ได้เป็นจำนวนคี่ สรุป ผลคูณของจำนวนเต็มบวกสองจำนวนที่เป็นจำนวนคี่ จะเป็นจำนวนคี่ โดยการให้เหตุผลแบบอุปนัย

ตัวอย่างที่ 2 จงหาพจน์ที่อยู่ถัดไปอีก 3 พจน์

1) 1, 3, 5, 7, 9, ...

2) 2, 4, 8, 16, 32, ...

วิธีทำ 1) จากการสังเกตพบว่าแต่ละพจน์มีผลต่างอยู่ 2

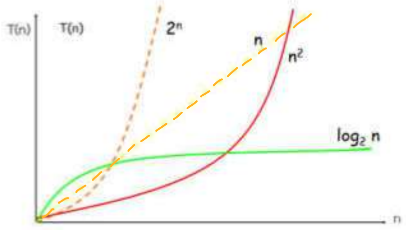
(3 – 1 = 2, 5 – 3 = 2, 9 - 7 = 2) ดังนั้น อีก 3 จำนวน คือ 11, 13, 15

2) จากการสังเกตพบว่าแต่ละพจน์จะมีการไล่ลำดับขึ้นไป ถ้าลองสังเกตดู จะอยู่ในรูปแบบ 2×2n โดยที่ n เป็นจำนวนเต็มบวกที่เริ่มตั้งแต่ 0 ดังนั้น พจน์ที่ 6 คือ 2×32 = 64, พจน์ที่ 7 คือ 128 และพจน์ที่ 8 คือ 256 หรืออาจมองในอีกแง่หนึ่ง จำนวนแต่ละจำนวนจะเพิ่มขึ้นเรื่อยๆ 2 เท่าก็ได้

11.5 อัตราการเจริญเติบโตของอินพุต (Order of growth)

การเติบโตของฟังก์ชันเวลา เมื่อ n เพิ่มขึ้น

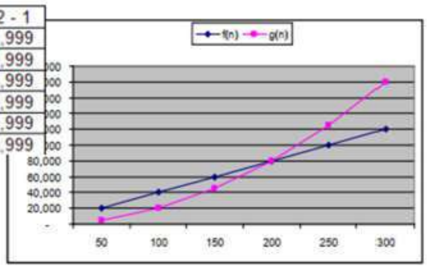
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



ตัวอย่างที่ 1

$f(n) = 400n + 23$ และ $g(n) = 2n^2 - 1$

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



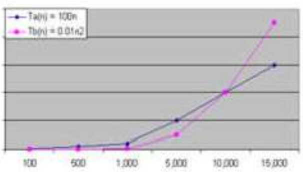
ตัวอย่างที่ 2

Effect of coefficient term พิจารณาฟังก์ชันเวลาของ algorithm ฟังก์ชันใดทำงานเติบโตได้รวดเร็วและช้ากว่ากัน

T1 = 100n T2 = $0.01n^2$ T3 = 1000 log n T4 = n

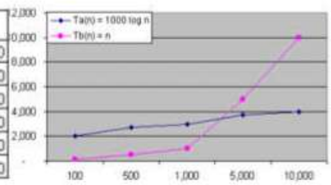
T2 โตเร็วกว่า T1

Input size	Ta(n) = 100n	Tb(n) = $0.01n^2$
100	10,000	100
500	50,000	2,500
1,000	100,000	10,000
5,000	500,000	250,000
10,000	1,000,000	1,000,000
15,000	1,500,000	2,250,000



T4 โตเร็วกว่า T3

Input size	Ta(n) = 1000 log n	Tb(n) = n
50	1,699	50
100	2,000	100
500	2,699	500
1,000	3,000	1,000
5,000	3,699	5,000
10,000	4,000	10,000



11.6 ทฤษฎีโลปีตาล (L'Hôpital's rule)

วิธีการที่ในการเปรียบเทียบลำดับการเติมโตของฟังก์ชันเวลา คือทฤษฎีลิมิตของโลปิตา กำหนดให้ f(n) และ g(n) เป็นฟังก์ชันเวลาอัลกอริทึม

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

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

จะเห็นว่าเราจะสนใจเปรียบเทียบเมื่อ input เข้าใกล้ infinity การหา derivative ของฟังก์ชัน คือมองข้อมูล ณ จุดที่เป็น infinity

ตัวอย่างการเปรียบเทียบ

T1 = 100n, T2 = 0.01n^2 (∞) เยอะกว่า n

ดังนั้น T1 < T2

$$\lim_{n\to\infty}\frac{100n}{0.01n^2}=10000\lim_{n\to\infty}\frac{n}{n^2}=10000\lim_{n\to\infty}\frac{1}{\infty}=0$$

T3 = 1000 log n, T4 = n (∞) เยอะกว่า log n

ดังนั้น T3 < T4

$$\lim_{n\to\infty}\frac{1000\times log_{10}e^{\frac{1}{n}}}{n}=1000\lim_{n\to\infty}\frac{1}{n}=1000\lim_{n\to\infty}\frac{1}{\infty}=0$$

T5 = 1/2 * n * (n-1), T6 = n^2

ดังนั้น T5(n) ≡ T6(n)

ใกล้เคียงกัน

$$\lim_{n\to\infty}\frac{0.5n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}$$

ทั้ง time and space efficiencies เป็นฟังก์ชันขึ้นอยู่กับ input size

Time efficiency พิจารณาจากการนับจำนวนครั้งการทำงานของ basic operation

Space efficiency พิจารณาจากการนับจำนวนหน่วยของ extra memory ที่ต้องใช้

Efficiencies ของบางขั้นตอนวิธีอาจแตกต่างกัน แม้อินพุทมีขนาดเท่ากัน ในกรณี นี้เราต้องแบ่งออกเป็น worst-case, average-case, และ best-case โดยส่วนใหญ่แล้วเราสนใจอัตราการเติบโตของเวลาที่ใช้ประมวลผลอัลกอริทึมเมื่ออินพุทมีขนาดเข้าสู่อินฟินิตี

11.7 สัญกรณ์เชิงเส้นกำกับ (Asymptotic Notation)

สัญกรณ์เชิงเส้นกำกับ คือสัญลักษณ์ที่บรรยายฟังก์ชันที่ยุ่งยาก ด้วยกลุ่มของฟังก์ชันที่อ่านง่ายกว่า เพื่อแสดงพฤติกรรมของฟังก์ชั่นนั้น ในบางครั้งประสิทธิภาพขั้นตอนวิธี อาจจะถูกอธิบายอยู่ในรูปแบบที่เข้าใจได้ง่ายผ่านการใช้สัญกรเชิงเส้นเพื่อประมาณเวลาที่ใกล้เคียงในการทำงานของขั้นตอนวิธี สัญกรเชิงเส้นกำกับ Asymptotic Notation โดยทั่วไปมี 3 ประเภทหลัก คือ Big-omega, Big-oh, Big-theta

การพิจารณาฟังก์ชันนั้นจะพิจารณาจากกรณี n มีค่าเป็นอินฟินิตี และให้ค่าคงที่มีค่าน้อยที่สุด

สำหรับการเปรียบเทียบประสิทธิภาพสัญกรณ์เชิงเส้นกำกับ (Asymptotic Efficiency)

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

1 (constant) < log n < n (linear) < n log n < n^2 (quadratic)< n^3 (cubic) < 2^n (exponential) < n! (factorial)

1) Big-oh ♡

$O(g(n))$ คือ Big-Oh ช้าที่สุด ขอบเขตบนที่ต่ำที่สุด ขั้นตอนวิธีไม่สามารถโตเร็วได้มากกว่านี้

$4n + 7 \le 5n$ --> $O(n)$

$1000n+190 \le 1001n$ --> $O(n)$

$189 \le 189$ --> $O(1)$ // ค่าคงที่

$1009 \le 1009$ --> $O(1)$

$14n^2 + 56n + 18 \le 15n^2$ --> $O(n^2)$

$100n^2 + 14n - 59 \le 101n^2$ --> $O(n^2)$

$100\ 2^n + n^2 \le 101\ 2^n$ --> $O(2^n)$ // 2^n, n^2 เลือก 2^n เยอะกว่า

$n \in O(n^2)$ เป็นสมาชิก ✓

$100n + 5 \in O(n)$

$1/2\ n(n-1) \in O(n^2)$

$n^3 \notin O(n^2)$ → n^3 ใช้ bigO เป็น n^2 ไม่ได้

$0.00001n^3 \notin O(n^2)$ → n^3 มากกว่า n^2 จึงไม่เป็นสมาชิก

$n^3 + n + 1 \notin O(n^2)$ → n^3 มากกว่า n^2 จึงไม่เป็นสมาชิก

$1000n \in O(n^2)$ ได้แต่ไม่เหมาะสม

$\sqrt{n} \in O(n)$ ได้แต่ไม่เหมาะสม

$13n^2 + 4n - 5$ --> $O(n^3)$ ใช้ได้แต่ไม่เหมาะสม

$13n^2 + 4n - 5$ --> $O(n^4)$ ใช้ได้แต่ไม่เหมาะสม

$13n^2 + 4n - 5$ --> $O(n^2)$ เหมาะสมที่สุด มาจาก $14n^2$

ตัวอย่าง $100n + 5 \in O(n^2)$

$100n \le n^2$ จริง

ตัวอย่าง $23n + 9 = O(n)$

$23n + 9 = 24n = O(n)$ จริง

ตัวอย่าง $12n^3 + 90n^2 + 2n + 54 = O(n^3)$

$13n^3 = O(n^3)$ จริง

ตัวอย่าง $12\ 2^n + n^2 = O(2^n)$

$13\ 2^n = O(2^n)$ จริง

ตัวอย่าง $27n^3 + 9n^2 + 4 = O(n^5)$

$28n^3 = O(n^5)$ จริง แต่ไม่เหมาะสม

ตัวอย่าง $4n^3 + 11n^2 + 5n = O(n^2)$

$5n^3 = O(n^2)$ ไม่จริง

ex $n \in O(n!)$ ✓ เป็นได้

สูตร

+ จัดรูป

+ ตัด

+ ตอบ

\# นิยาม bigO = ช้าที่สุด อะไรก็ได้ที่ช้ากว่า ตัวด้านหน้า

\# ตัวหลัง ต้องช้ากว่า ตัวข้างหน้า \#

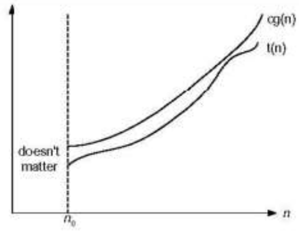


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

2) Big-omega

$\Omega(g(n))$ คือ omega เร็วที่สุด ขอบเขตล่าง ไม่สามารถเร็วกว่านี้ → ♡ ตรงข้ามกับ bigO = เร็วที่สุด

$n^3 \in \Omega(n^2)$

$1/2\ n(n-1) \in \Omega(n^2)$

$0.00001n^4 \in \Omega(n^3)$

$1000n + 5 \notin \Omega(n^2)$ → n^2 ช้ากว่า n จึงไม่เป็นสมาชิก

$n^2 + 50\log n \notin \Omega(n^3)$ n^3 ช้ากว่า n^2

$5n+17 \in \Omega(n)$

$9n^2 + 15n + 8 \in \Omega(n^2)$

$2000n^2 + 19n - 20 \in \Omega(n^2)$

$n/2\log(n/2) \in \Omega(n\log n)$

$13n^2 + 4n - 5 \in \Omega(1)$ ได้ ไม่เหมาะสม

$13n^2 + 4n - 5 \in \Omega(n)$ ได้ ไม่เหมาะสม

$13n^2 + 4n - 5 \in \Omega(n^2)$ เหมาะสมที่สุด

ตัวอย่าง $0.1n^3 \in \Omega(n^2)$ $0.1n^3 \le 0.1n^2$ จริง

ตัวอย่าง $11 * 2^n + 15n^2 + 7\log n \in \Omega(n^2)$ จริง

ตัวอย่าง $2n + 19 \in \Omega(n)$ จริง

ตัวอย่าง $18n^4 + 25n^2 + 4 \in \Omega(n^4)$ จริง

ตัวอย่าง $17n^3 + 97n^2 \in \Omega(n^5)$ ไม่จริง

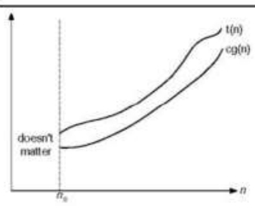
ตัวอย่าง $94n^3 + 5n^2 + 15n \in \Omega(n^2)$ จริง ไม่เหมาะสม

ex

$\log n \in \Omega(1)$ ✓

$\log_2 n \in \Omega(1)$ ✓

\# ตัวหลัง ต้องเร็วกว่า ตัวหน้า



3) Big-theta

$\Theta(g(n))$ คือ theta เท่ากับ g(n) ระหว่างขอบเขตบนและขอบเขตล่าง

$9n+8 \in \Theta(n)$

$9n \le f(n) \le 10n$

$11n^2 + 4n + 22 \in \Theta(n^2)$

$11n^2 \le f(n) \le 12n^2$

$1/2\ n(n-1) \in \Theta(n^2)$

$n^2 + \sin n \in \Theta(n^2)$

$n^2 + \log n \in \Theta(n^2)$

$1500n^2 \notin \Theta(n^3)$

$5(2^n) + n^2 + 4n \in \Theta(2^n)$

$21n + 56 \in \Theta(n)$

$4n^4 + 5n^2 + 4n \in \Theta(n^4)$

$8n^3 + 7n^2 + 23 \in \Theta(n^2)$ ไม่จริง

$2n^2 + 500n + 1000\log n \in \Theta(n^2)$

$99n^3 + 2n + 167 \in \Theta(n^3)$

$53n^3 + 4n + 110n^2 + 29n^3 + 27 \in \Theta(n^3)$

ตัวอย่าง $(1/2)(n)(n-1) \in \Theta(n^2)$

$0.5n^2 - 0.5n \le n^2$ จริง

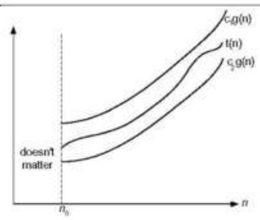


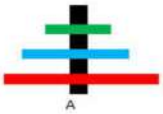
Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

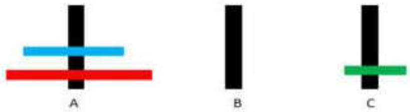
11.8 ปริศนาหอคอยฮานอย

EDOUARD LUCAS คือ นักคณิตศาสตร์ชาวฝรั่งเศส เป็นผู้คิดค้นปริศนาหอคอยฮานอย (The Tower of Hanoi) โดยปริศนาหอคอยฮานอย นั้นจะมีแผ่นจานไม้ 8 แผ่น รัศมีแตกต่างกัน แต่ละแผ่นมีรูตรงกลาง นำมาใส่ไว้ในหลักเป็นกองซ้อน โดยให้แผ่นที่เล็กกว่าทับแผ่นที่ใหญ่กว่า และมีหลักเปล่าสองหลัก ดังรูป

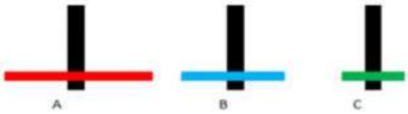
ปริศนาหอคอยฮานอยคือ ให้ย้ายแผ่นจานทั้งหมดไปกองไว้ที่หลักเปล่าหลักหนึ่ง โดยมีเงื่อนไขว่า เคลื่อนย้ายได้คราวละแผ่น และต้องนำไปไว้ที่หลักใดหลักหนึ่ง และห้ามแผ่นที่มีขนาดใหญ่กว่าวางทับแผ่นที่มีขนาดเล็กกว่า ต่อไปนี้แสดงขั้นตอนการเคลื่อนย้ายแผ่นจานจำนวน 3 แผ่นจากแผ่น A ไปแผ่น C

ตัวอย่างการเคลื่อนย้าย

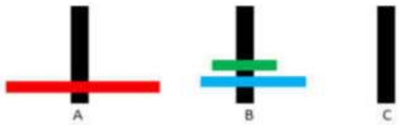




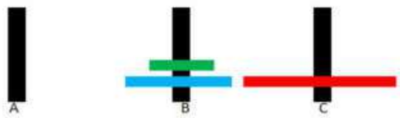
ขั้นตอนที่ 1 เคลื่อนย้ายจากแท่น A ไป C



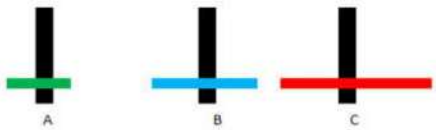
ขั้นตอนที่ 2 เคลื่อนย้ายจากแท่น A ไป C



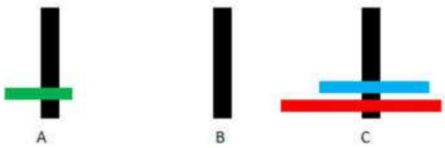
ขั้นตอนที่ 3 เคลื่อนย้ายจากแท่น C ไป B



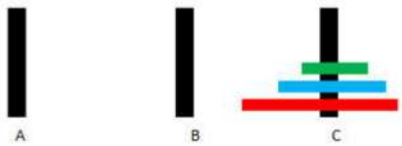
ขั้นตอนที่ 4 เคลื่อนย้ายจากแท่น A ไป C



ขั้นตอนที่ 5 เคลื่อนย้ายจากแท่น B ไป A



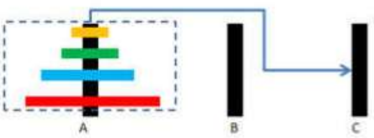
ขั้นตอนที่ 6 เคลื่อนย้ายจากแท่น B ไป C



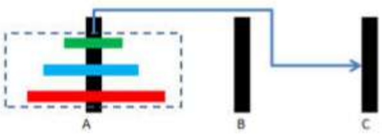
ขั้นตอนที่ 7 เคลื่อนย้ายจากแท่น A ไป C

ปริศนาหอคอยฮานอยนี้สามารถแก้ได้ แต่ที่สนใจคือ จำนวนครั้งของการเคลื่อนย้ายแผ่นจานน้อยที่สุดเป็นเท่าไร ให้ a_n เป็นจำนวนครั้งน้อยสุดในการเคลื่อนย้ายแผ่นจาน n แผ่นจากหลักหนึ่งไปยังอีกหลักหนึ่ง

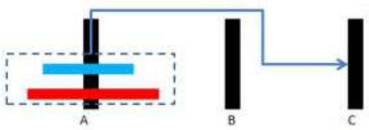
เมื่อพิจารณาการจำนวนการเคลื่อนย้ายแผ่นทั้งหมดจาก A ไป C ผลจำนวนที่เคลื่อนย้ายเป็นดังนี้



จำนวนที่เคลื่อนย้ายจาก A ไป C คือ a_n โดย n คือ 4

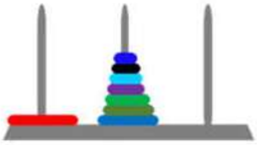


จำนวนที่เคลื่อนย้ายจาก A ไป C คือ a_{n-1} โดย n คือ 3

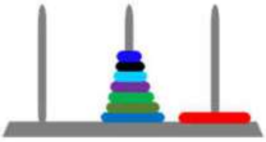


จำนวนที่เคลื่อนย้ายจาก A ไป C คือ a_{n-2} โดย n คือ 2

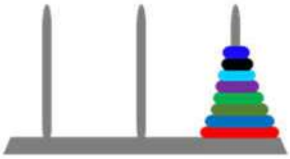
1. ต้องย้ายแผ่นที่เล็กกว่าทั้งหมด n-1 แผ่น ไปยังหลักที่ว่างก่อน จำนวนการเคลื่อนย้าย คือ $a_{n 1}$ ครั้ง



2. ย้ายแผ่นใหญ่ที่สุดไปยังหลักเป้าหมาย จำนวนการเคลื่อนย้ายคือ 1 ครั้ง



3. ย้ายแผ่นจานทั้งหมดในข้อ 1. ไปยังหลักเป้าหมาย จำนวนการเคลื่อนย้าย คือ $a_{n 1}$ ครั้ง



จะได้ $a_n = a_{n 1} + 1 + a_{n 1}$

$= 2a_{n 1} + 1$

5) จงหาผลเฉลยของความสัมพันธ์เวียนเกิดปริศนาหอคอยฮานอย

ถ้า n = 1 คือมีแผ่นจานเพียง 1 แผ่น จำนวนการเคลื่อนย้ายเป็น 1 ครั้ง และ n = 0 ไม่มีแผ่นจาน จำนวนการเคลื่อนย้ายเป็น 0 ครั้ง ความสัมพันธ์เวียนบังเกิดของปริศนาหอคอยฮานอย คือ $a_n = 2a_{n 1} + 1 \; ; n \geq 2$

เงื่อนไขเริ่มต้น $a_0 = 0$ และ $a_1 = 1$

จากปริศนาหอคอยฮานอย ทำแปลงรูปจากสูตรการแปลง $a_n = ra_{n 1} + d$ เป็นรูปแบบนี้ $a_n = Ar^n + B$

ผลเฉลยอยู่ในรูป $a_n = Ax2^n + B$ เมื่อ $r = 2$

จะได้ $a_0 = Ax2^0 + B$ แล้ว $0 = A+B$

$a_1 = Ax2^1 + B$ แล้ว $1 = 2A+B$

แก้สมการ $A = 1,\ B = -1$

ผลเฉลยคือ $a_n = 2^n - 1 \; ; n \geq 2$

ถ้าจานมีจำนวน 3 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 7
ถ้าจานมีจำนวน 4 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 15
ถ้าจานมีจำนวน 5 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 31
ถ้าจานมีจำนวน 6 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 63
ถ้าจานมีจำนวน 7 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 127
ถ้าจานมีจำนวน 8 อัน ต้องใช้จำนวนรอบทั้งหมดน้อยสุด = 255

หมายเหตุ รูปแบบของความสัมพันธ์เวียนบังเกิด คือ จะนำพจน์ก่อนหน้ามาคำนวณ ซึ่งถ้าต้องการทราบพจน์ที่ n ต้องทราบพจน์ที่ n-1 ก่อน จากตัวอย่างปริศนาหอคอยฮานอย คือ $a_n = 2a_{n 1} + 1$ ถ้าต้องการทราบพจน์ที่ 8 ต้องหาพจน์ที่ 7 ก่อนแล้วนำมาเข้าสูตร คือ $a_8 = 2(127) + 1 = 255$ จึงจะสามารถหาพจน์ที่ 8 แต่ถ้ารูปแบบผลเฉลยของความสัมพันธ์เวียนเกิดนั้นไม่มีความจำเป็นต้องรู้ค่าพจน์ก่อนหน้า จากตัวอย่างปริศนาหอคอยฮานอย คือ $a_n = 2^n - 1$ ถ้าต้องการทราบพจน์ที่ 8 สามารถแทนค่า n ด้วย 8 เข้าไปได้ ตัวอย่างเช่น $a_8 = 2^8 - 1 = 255$ จึงไม่มีความจำเป็นต้องหาพจน์ก่อนหน้าจึงสะดวกในการใช้งานมากกว่า

11.9 ฟีโบนักชี

ลำดับฟีโบนัชชี (Fibonacci Sequence) มีนิยามของความสัมพันธ์ว่า จำนวนถัดไปเท่ากับผลบวกของจำนวนสองจำนวนก่อนหน้า และสองจำนวนแรกก็คือ 0 และ 1 ตามลำดับ หากเขียนให้อยู่ในรูปของสัญลักษณ์ ลำดับ F_n ของฟีโบนัชชี สามารถเขียนความสัมพันธ์เวียนเกิดได้ดังนี้

$F_n = F_{n 1} + F_{n 2}$

โดยกำหนดค่าเริ่มแรกให้

$F_0 = 0$ และ $F_1 = 1$

ลำดับฟีโบนัชชี ประกอบด้วยเทอมต่างๆ เรียงตามลำดับดังนี้

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ,... จากตัวเลขอนุกรมดังกล่าว สามารถแสดงวิธีการหาค่าเทอมต่างๆ ได้ดังนี้

1) $F_0 = 0$
2) $F_1 = 1$
3) $F_n = F_{n\ 1} + F_{n\ 2}$

พจน์ที่ 3 มาจาก พจน์ที่ 1 + พจน์ที่ 2 = 0 +1 = 1 โดยเริ่มจากพจน์ที่ 3 เป็นต้นไป

พจน์ที่ 4 มาจาก พจน์ที่ 2 + พจน์ที่ 3 = 1 + 1 = 2

พจน์ที่ 5 มาจาก พจน์ที่ 3 + พจน์ที่ 4 = 1 + 2 = 3

ดังนั้น ความสัมพันธ์เวียนเกิด คือ $a_n = a_{n\ 1} + a_{n\ 2}$; $n \geq 2$ และเงื่อนไขเริ่มต้น $a_0 = 0$ และ $a_1 = 1$

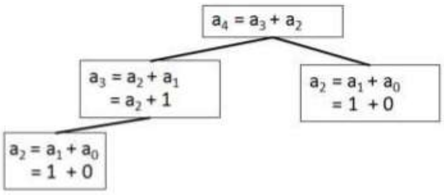
จากนิยามสามารถสรุปการหาคำตอบออกเป็น 2 ทางคือ

ถ้า n มีค่าเป็น 0 หรือ 1 คำตอบที่ได้คือ $F_n = n$

ถ้า n มีค่ามากกว่า 1 คำตอบที่ได้คือ $F_n = F_{n\ 1} + F_{n\ 2}$

จงคำนวณหาค่าอนุกรมไฟโบเนชชีที่ 4

วิธีคิดแบบ ต้นไม้แตกกิ่งก้านสาขาได้ ดังนี้



วิธีคิดแบบปกติ

F(4) = F(3) + F(2)
= (F(2) + F(1)) + (F(1) + F(0))
= ((F(1) + F(0)) + F(1)) + (F(1) +F(0))
= ((1+0) + 1) + (1 + 0)
= 3

แล้วคิดเฉพาะตัวเลข คำตอบที่ได้คือ a_4 =1+0+1+1+0=3

ผลเฉลยฟีโบนักชี

ความสัมพันธ์เวียนเกิด คือ $a_n = a_{n\ 1} + a_{n\ 2}$; $n \geq 2$ และเงื่อนไขเริ่มต้น $a_0 = 0$ และ $a_1 = 1$

จงหาผลเฉลยของความสัมพันธ์เวียนเกิด $a_n = a_{n\ 1} + a_{n\ 2}$; $n \geq 2$ โดยที่ $a_0 = 0$ และ $a_1 = 1$

ขั้นตอนที่ 1 จากสูตรการแปลง $a_n = c_1a_{n\ 1} + c_2a_{n\ 2}$ เป็นรูปแบบนี้ $a_n = Ar^n + Bnr^n$

กรณีนี้ คือ r เท่ากัน ตัว c ทั้ง 2 ตัวจะเท่าหรือไม่เท่าก็ได้ แต่ r ได้คำตอบเดี่ยวกัน

$a_n = x^2$

$a_{n\ 1} = x^1$

$a_{n\ 2} = x^0$

แก้สมการ $x^2 - x - 1 = 0$

$$x = \frac{1+\sqrt{5}}{2}, x = \frac{1-\sqrt{5}}{2} \text{ จาก } x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

ผลเฉลยอยู่ในรูป $a_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$ จากสูตรจะเห็นว่าเป็น 2^n แตกเป็น Tree

CODE FIBONACCI

```
F(n)
  if n <= 1
     return n
  else
     return F(n-1) + F(n-2)
```

การวิเคราะห์เวลาของขั้นตอนวิธีรูปแบบที่ไม่เป็น recursive	**การวิเคราะห์เวลาของขั้นตอนวิธีรูปแบบที่เป็น recursive**
• ระบุขนาดของข้อมูล (input size) และ basic operation ของอัลกอริทึม • Basic operation มักที่อยู่ในลูปชั้นในสุด • พิจารณาว่าอัลกอริทึมมี worst, average, and best case หรือไม่ • จำนวนครั้งการทำงานของ basic operation แตกต่างกันเมื่อข้อมูลต่างกัน แม้ขนาดข้อมูลเท่ากัน • ตั้งสมการ summation ของ basic operation • แก้สมการเพื่อหา Order of growth โอกาสที่เป็นได้ $\log n$, n , n^2 , n^3	• ระบุขนาดของข้อมูล(input size) และ basic operation ของ อัลกอริทึม • Basic operation มักที่อยู่ในลูปชั้นในสุด • พิจารณาว่าอัลกอริทึมมี worst, average, and best case หรือไม่ • จำนวนครั้งการทำงานของ basic operation แตกต่างกันเมื่อข้อมูล ต่างกัน แม้ขนาดข้อมูลเท่ากัน • ตั้งสมการ Recurrence relation ของ basic operation • ต้องระบุ initial condition ด้วยเสมอ • แก้สมการเพื่อหา Order of growth วิธีการแก้สมการ recurrence Forward substitutions Backward substitutions Linear second-order with constant coefficients Homogeneous Inhomogeneous โอกาสที่เป็นได้ $\log n$, n , 2^n , $n!$

<table>
<tr><th>CODE</th><th>อธิบาย</th></tr>
<tr><td>MaxElement (A[1...n])
Maxval:= A[1]
for i:= 2 to n do
if A[i] > maxval then
maxval:= A[i]
return maxval
→ loop ปกติ : n</td><td>FIND MAXIMUM
ดู Loop = n
Θ (n)</td></tr>
<tr><td>MatrixMultiplication (A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
for i ⟵ 0 to n-1 do
for j ⟵ 0 to n-1 do
C[i, j] ⟵ 0.0
for k ⟵ 0 to n-1 do
C[i, j] ⟵ C[i, j] + A[i, k] * B[k, j]
return C
→ n^3</td><td>Matrix multiplication
ดู Loop = n
Θ (n^3)</td></tr>
<tr><td>Algorithm SelectionSort(A[0..n-1])
for i←0 to n-2 do
min←i
for j←i+1 to n-1 do
if A[j] < A[min]
min ← j
swap A[i] and A[min]
→ n^2</td><td>Selection sort
ดู Loop = n
Θ (n^2)</td></tr>
<tr><td>Algorithm InsertionSort(A[0 .. N-1])
for i←1 to n-1 do
v ← A[i]
j ← i-1
while j≥0 and A[j]>v do
A[j+1] ← A[j]
j ← j-1
A[j+1] ← v
→ n^2</td><td>Insertion sort
ดู Loop = n
อาจจะมีโอกาสเป็น Θ (n) ถึง Θ (n^2)</td></tr>
<tr><td>algorithm $BitCount(m)$
// Input: A positive integer m
// Output: The number of bits to encode m
$count \leftarrow 1$
while $m > 1$ do
$count \leftarrow count + 1$
$m \leftarrow \lfloor m/2 \rfloor$
return $count$
→ $\log_2 n$</td><td>BitCount
ดู Loop = $\log_2 n$
Θ ($\log_2 n$)</td></tr>
<tr><td>algorithm $Factorial(n)$
// Computes $n!$ recursively
// Input: A nonnegative integer n
// Output: The value of $n!$
if $n = 0$ then return 1
else return $Factorial(n-1) * n$
Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: multiplication

Let $M(n)$ = multiplication count to compute $Factorial(n)$.
$M(0) = 0$ because no multiplications are performed to compute $Factorial(0)$.
If $n > 0$, then $Factorial(n)$ performs recursive call plus one multiplication.
$M(n) = \underbrace{M(n-1)}_{\text{to compute } Factorial(n-1)} + \underbrace{1}_{\text{to multiply } Factorial(n-1) \text{ by } n}$</td><td>Factorial Function
ถ้าเขียนแบบนี้ เป็น recurisive ไม่ใช่เป็น 2^n กับ n!
5! --> 5 x 4 x 3 x 2 x 1 เวลาที่ใช้ คือ n คือ input 5 ทำ 5 ครั้ง</td></tr>
<tr><td>algorithm $BitCount(n)$
// Input: A positive integer n
// Output: The number of bits to encode n
if $m = 1$ then return 1
else return $BitCount(\lfloor n/2 \rfloor) + 1$

Input Size: Use number n (actually n has about $\log_2 n$ bits)
Basic Operation: division by 2
Let $D(n)$ = division count to compute $BitCount(n)$.
$D(1) = 0$ because no divisions are performed to compute $BitCount(1)$.
If $n > 1$, then $BitCount(n)$ performs recursive call on $\lfloor n/2 \rfloor$ plus one division.
$D(n) = \underbrace{D(\lfloor n/2 \rfloor)}_{\text{to compute } BitCount(\lfloor n/2 \rfloor)} + \underbrace{1}_{\text{to compute } \lfloor n/2 \rfloor}$</td><td>BitCount แบบ recurisive
กรณีแบ่งข้อมูลเป็น 2 ส่วน divide and conquer
ทำให้เวลาค้นหาเร็วขึ้น เป็น $\log_2 n$</td></tr>
<tr><td>algorithm findmin (A[], i, j)
if i=j then
return A[i];
mid = (i+j)/2;
m1 = findmin(A , i , mid);
m2 = findmin(A , mid+1 , j);
return m1<m2? m1:m2;

ดูที่ parameter (ถ้าแบ่งเป็นส่วนๆ (merge) → $\log_2 n$)</td><td>minimun แบบ recurisive
กรณีแบ่งข้อมูลเป็น 2 ส่วน divide and conquer
และเอาข้อมูลมาเปรียบเทียบตอน merge กลับใครน้อยสุดอยู่ต่อไป
ทำให้เวลาค้นหาเร็วขึ้น เป็น $\log_2 n$
↳ เพราะ data ถูกเอาออกครึ่งนึง</td></tr>
</table>

ถ้าแยกเป็น m1, m2, m3 } จะได้เป็น $\log_3 n$

```
ALGORITHM MoveDisk (N, fromPeg, toPeg)            call MoveDisk(4, 1, 3);
  if N = 1 then
    PRINT fromPeg + "=>" + toPeg;
 else
    help = 6-fromPeg-toPeg;          // fromPeg + help + toPeg = 6
    MoveDisk(N-1, fromPeg, help);    // Move n-1 disks to helpPeg
    PRINT fromPeg + "=>" + toPeg;    // Move the last disk to toPeg
    MoveDisk(N-1, help, toPeg);      //  Move n-1 disks to toPeg
```

swap

2^n

Recurrence of MoveDisk algorithm

$$T(n) = 2T(n-1)+1$$
$$T(n) = 2.[2T(n-2)+1]+1 = 2^2T(n-2)+2^1+1$$
$$T(n) = 2^2[2T(n-3)+1]+2^1+1$$
$$T(n) = 2^3T(n-3)+2^2+2^1+1$$
$$T(n) = 2^{n-1}T(n-(n-1))+2^{n-2}+2^{n-3}+....+2^1+2^0$$
$$T(n) = 2^{n-1}T(1)+2^{n-2}+2^{n-3}+....+2^1+2^0$$
$$T(n) = \sum_{i=0}^{n-1} 2^i = \sum_{i=0}^{n} 2^i - 2^n = \frac{2^{n+1}-1}{2-1} - 2^n$$
$$T(n) = 2^n[2-1]-1 = 2^n - 1 \in \Theta(2^n)$$ Exponential !!

Movedisk แบบ recurisive

จาก code มีการ แตก เป็น 2 ส่วน <u>เหมือน tree</u>

แบบนี้จะเป็นลักษณะ 2^n

T(1) = 1   n = 1

จาก Code

if N = 1 then

print (..............)

T(n) = T(n-1) + 1 + T(n-1)   n>1

จาก code ข้างล่างนี้

T(n-1) --> MoveDisk(N=1,F,H);

PRINT(...........) --> 1

T(n-1) --> MoveDisk(N=1,H,T);

*สังเกตุ parameter ไม่ได้มีการแยกออก
แต่แบบนี้เป็นเพียงแบบสลับที่

```
int size = 5;
int Array[100];
void permute(int j)
{
   if (j == size)
   {
      for(int i=0 ; i < size ; i++)
      {
         cout<<Array[i]<<" "; }  cout<<endl;
      }
      else
      {
         for (int i = j; i < size; i++)
         {
            int T = Array[j];
            Array[j] = Array[i];
            Array[i] = T;
            permute(j+1);
            T = Array[j];
            Array[j] = Array[i];
            Array[i] = T;
         }
      }
   }
}

for(int i=0;i<100;i++){ Array[i] = i+1;}
permute(0);
```

Permutation

จาก Code มีการสลับที่ เป็นลักษณะของ n!

ลักษณะที่เป็น Permutation

ตรงนี้จะมีการสลับที่

คือมี LOOP แล้วแตกด้วย ฟังก์ชั่นตัวเอง for (.....) โดย j จะเปลี่ยนแปลงตลอด

จากตรงนี้ จะเห็นว่า

ตัวแรกทำงาน 3 ครั้ง และในแต่ละครั้งทำงาน 2 ครั้ง

รอบต่อมาทำงาน 2 ครั้ง และในแต่ละครั้งทำงาน 1 ครั้ง

รอบต่อมาทำงาน 1 ครั้ง

คือ 3 x 2 x 1 = 6

ถ้าไม่เขียน recursive

For(1..3)

For(1..2)

For(1..1)

ตัว recursive ทำหน้าที่แทน For แต่ถ้า For มี 100 ตัวเขียนคงไม่ไหว

ถ้ามี loop และใน loop เรียก funtion ตัวเอง => factorial => # เลวร้ายที่สุด

⊕, ⊖ : 2n

⊗, ⊘ } $\log_2 n$

แบบฝึกหัด

1) จงเขียนค่า Big O ของโค้ดโปรแกรมข้างล่างนี้

1.1) i = i * i
i = i + 2
j = j + 1
j = j - 3
if (j > 5)
print "Love"
else
print "No"
C
	1.2) for i = 1 to n
	print "Loop"
	n

1.3) i = 1 loop (i <= n) i = i+2 n	1.4) i = 1 loop (i <= n) i = i * 2 $\log_2 n$
1.5) for i = 1 to n print "loop1" for j = i to n print "loop2" n^2	1.6) for i = 1 to n for j = 1 to n x = x+1 n^2
1.7) for i = 1 to n for j = 1 to n for k = 1 to n x = x+1 n^3	1.8) for i = 1 to n for j = 1 to i print "loop2" $\frac{(n)(n+1)}{2} = n^2$
1.9) for i = 1 to n if(a % 2 == 0) print "loop2" n	1.10) i = n loop (i >= 1) i = i-2 n

1.11) i = n
loop (i >= 1)
i = i / 2
$\log_2 n$ ✓
	1.12) i = 1
	loop (i <= n)
	if(i % 2 == 0)
	i = i + 2
	else
	i = i +1
	n

1.13)
for(i = 0; i < n; i++)
{
print E
}
for(j = 0; j < n; j++)
{
print ***
}
for(j = 0; j < n; j++)
{
print ann
}
# ไม่ใช่ for ซ้อน for 	
n ✓
	1.14)
	If(n < 2)
	{
	n = n + 2
	}
	for (i = 1; i < n ; j++) → i ไม่มีการเปลี่ยนแปลงค่า
	(ไม่มีโอกาสหลุดจาก loop)
	{
	j = j * 3
	print dead
	}
	∞

1.15)
for (i = 1; i < n ;)
{
i = i * 10 ✓
print dead
}
$\log_{10} n$ ✓
	1.16)
	for (i = 0 ; i < n ; i++)
	{
	for (j = 0 ; j < n ; j++)
	{
	for (k = 0 ; k < n ; k++)
	{
	print work
	}
	}
	}
	n^3

1.17) i ไม่มีการเปลี่ยนแปลง

```
for (i = 0 ; i < n ; )
{
        for (j = 0 ; j < n ; j++)
        {
      print no_name
        }
}
```

∞

1.18)

```
for (i = n; i ≥ 1;)
{
    i = i / 5;
        for (j = 0; j <n; j++)
        {
            print love me
        }
}
```

$i = i / 5 \rightarrow \log_5 n$

n

$n \log_5$

1.19) fix ค่า

```
for (i = 0 ; i < 100000 ; i++)
 {
     for (j = 0 ; j < 1000000 ; j++)
      {
         print work
      }
  }
```

C (ค่าคงที่)

1.20)

```
for (i = 0; i < N; i++)
{
   print love
}
for (j = 0; j < N; j++)
{
   print love
}
```

แยก loop ค.ช้า = n

n

1.21) i เงื่อนไขการหยุด ไม่มี

```
for (i = 1; ; i++)
 {
    i = i * 2;
}
```

∞

1.22)

```
Function test (int N)
{
   print chanida
   if (N<1)
   {
      return;
   }
   if (N>1)
   {
      test (N-1)
   }
   if (N>1)
   {
      test (N-1);
   }
}
```

เรียกใช้ function test

recursion

2^n

2) จงหา Big O (ช้าสุด) ของฟังก์ชัน $f(n) = 3n^2-7n$

n^2

3) จงหา Big O ของฟังก์ชัน $f(n) = 6n^3+5n$

n^3

4) จงหา Big O ของฟังก์ชัน $f(n) = 2n^2+n-n\log_2 n$

n^2

5) จงหา Big O ของฟังก์ชัน $f(n) = 2^n(n^2-2n)$

2^n

6) จงหา Big O ของฟังก์ชัน $f(n) = 2n^4+3n^3+5$

n^4

7) จงหา Big O ของฟังก์ชัน $f(n) = n^2-3n+3$

n^2

8) จงหา Big O ของฟังก์ชัน $f(n) = \frac{n(n+1)}{2}$

n^2