บทที่ 2 อัลกอริทึม

บทเรียนย่อย

2.1 Algorithm Introduction

2.2 Algorithm Analysis

2.3 Algorithm Efficiency

2.4 Big–O Notation

วัตถุประสงค์

- มีความรู้ ความเข้าใจเกี่ยวกับอัลกอริทึมเบื้องต้น
- มีความรู้ ความเข้าใจเกี่ยวกับรูปแบบของการวิเคราะห์อัลกอริทึมเบื้องต้น
- มีความรู้ ความเข้าใจเกี่ยวกับคณิตศาสตร์พื้นฐานสำหรับการวิเคราะห์ประสิทธิภาพของอัลกอริทึม
- มีความรู้ ความเข้าใจเกี่ยวกับการทดสอบประสิทธิภาพของอัลกอริทึม

บทที่ 2 อัลกอริทึม

บทเรียนย่อย

2.1 Algorithm Introduction

Algorithm (อัลกอริทึม) คือ ลำดับขั้นตอนวิธีในการทำงานของโปรแกรมเพื่อแก้ปัญหาใดปัญหาหนึ่ง ซึ่งถ้าปฏิบัติตามขั้นตอนอย่างถูกต้องแล้ว จะสามารถช่วยแก้ปัญหา หรือประมวลผลตามต้องการได้สำเร็จ โดยอัลกอริทึมสามารถแบ่งออกเป็นตามประเภทหลัก ๆ ได้ดังนี้

- Brute force algorithm
- Divide and Conquer algorithm
- Decrease and Conquer algorithm
- Transform and Conquer algorithm

2.1 Algorithm Introduction [2]

- Greedy algorithm
- Dynamic programming algorithm
- Backtracking algorithm
- Branch and bound algorithms
- Recursive algorithm
- Randomized algorithms

Brute force algorithm

เป็นอัลกอริทึมสำหรับการแก้ไขปัญหาโดยมีรูปแบบในการสั่งให้ทำงานเรื่อย ๆ จนกระทั่งได้คำตอบของทุกปัญหา เช่น การค้นหาค่าที่ต้องการในตัวแปร array จากค่าทุกตัวที่ได้เก็บไว้

ข้อดี : เป็นวิธีคิดที่ง่าย และนำไปเขียนโปรแกรมได้สะดวก

ข้อเสีย : ประสิทธิภาพในการทำงานต่ำ ไม่เหมาะกับงานที่ซับซ้อน

Divide and Conquer algorithm

Divide and Conquer algorithm

เป็นอัลกอริทึมที่มีหลักการคิดด้วยการแยกปัญหาออกเป็นสองส่วน คือ ส่วนที่หนึ่งแบ่งปัญหาออกเป็นส่วนเล็ก ๆ แล้วแก้ปัญหาในส่วนเล็ก ๆ นั้นก่อน และอีกส่วนนำผลที่ได้จากการแก้ไขปัญหากลับมารวมกันใหม่ เช่น การจัดเรียงข้อมูลแบบ Quick sort และ Merge sort เป็นต้น

ข้อดี : มีประสิทธิภาพการใช้เวลาในการทำงานสูง เหมาะกับงานที่ซับซ้อน

ข้อเสีย : การเขียนโปรแกรมซับซ้อน และใช้หน่วยความจำค่อนข้างมาก

Decrease and Conquer algorithm

Decrease and Conquer algorithm

เป็นอัลกอริทึมที่มีแก้ปัญหาด้วยการลดขนาดของปัญหาลง และเลือกขนาดของกลุ่มปัญหาที่ต้องการแก้ไขปัญหา โดยละเว้นปัญหาบางส่วนไว้ก่อน เพื่อจะแก้ปัญาที่มีขนาดเล็กลงกว่าเดิมเนื่องจากสามารถแก้ไขได้ง่ายกว่า เช่น การค้นหาข้อมูลแบบไบนารี เป็นต้น

ข้อดี : มีประสิทธิภาพการในการทำงานดี เหมาะกับงานที่ซับซ้อน

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อน

Transform and Conquer algorithm

Transform and Conquer algorithm

เป็นอัลกอริทึมที่มีแก้ปัญหาด้วยการเปลี่ยนรูปแบบของปัญหาที่ต้องการแก้ไขให้อยู่ในรูปแบบอื่นก่อน ด้วยคาดหวังว่าเมื่อเปลี่ยนรูปแบบของปัญหาแล้วจะสามารถแก้ไขปัญหาได้ง่ายและรวดเร็วขึ้น เช่น การนำข้อมูลที่ต้องการค้นหามาจัดเรียงข้อมูลก่อนที่จะค้นหา

ข้อดี : มีประสิทธิภาพการในการทำงานดี เหมาะกับงานที่ซับซ้อนและมีความหลากหลายของข้อมูล

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนค่อนข้างมาก

Greedy algorithm

เป็นอัลกอริทึมที่มีลักษณะการแก้ปัญหาด้วยการเพิ่มประสิทธิภาพของการแก้ปัญหาให้เหมาะสมที่สุด (optimization problems) ซึ่งเป็นรูปแบบอัลกอริทึมที่พิจารณาคำตอบที่ดีที่สุด และคุ้มค่าที่สุดในการแก้ปัญหานั้น ๆ เช่น ปัญหาการทอนเหรียญ โดยจะทำการเลือกทอนเหรียญหน่วยที่มีขนาดใหญ่ที่สุดก่อน เป็นต้น

ข้อดี : มีประสิทธิภาพการในการทำงานสูง เหมาะกับงานที่ซับซ้อน

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนสูง

Dynamic programming algorithm

เป็นอัลกอริทึมที่มีลักษณะการแก้ปัญหาด้วยการแบ่งปัญหาโดยทำการแบ่งปัญหาออกเป็นส่วนเล็ก ๆ แล้วนำผลของปัญหาเล็ก ๆ ที่ดีที่สุดนำมาแก้ไขปัญหาใหญ่ ที่เรียกกันว่า การแก้ไขปัญหาจากล่างขึ้นบน (Bottom-up approach) เช่น การหาค่าตัวเลข Fibonacci เป็นต้น

ข้อดี : มีประสิทธิภาพการในการทำงานสูง เหมาะกับงานที่ซับซ้อน

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนสูง

Backtracking algorithm

เป็นอัลกอริทึมที่มีลักษณะการค้นหาเส้นทางทุกเส้นทางที่เป็นไปได้เพื่อหาคำตอบของปัญหาทีละส่วนย่อย หากคำตอบที่ได้ไม่ถูกต้องหรือไม่ใช่ส่วนหนึ่งของคำตอบจะถอยหลังกลับมาจุดเดิมเพื่อค้นหาคำตอบใหม่ เช่น การคิดความเป็นไปได้ทั้งหมดของการเดินหมากกระดาน เป็นต้น

ข้อดี : มีประสิทธิภาพการในการทำงานสูง เหมาะกับงานที่ซับซ้อนสูง

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนสูง

Branch and bound algorithm

Branch and bound algorithm

เป็นอัลกอริทึมที่เพิ่มประสิทธิภาพในการแก้ปัญหาด้วยการนำโครงสร้างแบบต้นไม้ (Tree) มาเก็บปัญหาย่อย ๆ โดยที่ปัญหาหลักจะอยู่ในตำแหน่งบนสุด และทำการแบ่งปัญหาย่อยออกทีละสองโหนด (ซ้ายและขวา) จนกว่าจะสามารถแก้ปัญหาย่อย ๆ นั้นได้ทุกโหนด เช่น การหาเส้นทางที่เหมาะสมให้กับพนักงานขายสินค้าให้สามารถเดินทางครบทุกที่ได้เร็วที่สุด

ข้อดี : มีประสิทธิภาพการในการทำงานสูง เหมาะกับงานที่ซับซ้อนสูง

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนสูงมาก

Recursive algorithm

Recursive algorithm

เป็นอัลกอริทึมแบบวนซ้ำ เป็นการแก้ไขปัญหาขั้นพื้นฐานด้วยการเรียกตัวเองซ้ำ ๆ โดยนำข้อมูลปัญหาบางส่วนย่อยของปัญหาทั้งหมดกลับมาเป็นข้อมูลในการแก้ปัญหา เช่น การหาค่า Factorial เป็นต้น

ข้อดี : มีประสิทธิภาพการในการทำงานดี

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนเล็กน้อย

Randomized algorithm

Randomized algorithm

เป็นอัลกอริทึมแบบสุ่ม โดยใช้หลักการสุ่มข้อมูลแล้วนำข้อมูลที่สุ่มเลือกขึ้นมาไปดำเนินการต่อเพื่อให้ได้ผลลัพธ์ตามที่ต้องการ

ข้อดี : มีประสิทธิภาพการในการทำงานดี

ข้อเสีย : การเขียนโปรแกรมมีความซับซ้อนเล็กน้อย

บทที่ 2 อัลกอริทึม

บทเรียนย่อย

2.1 Algorithm Introduction

2.2 Algorithm Analysis

2.3 Algorithm Efficiency

2.4 Big–O Notation

2.2 Algorithm Analysis

Algorithm Analysis เป็นการวิเคราะห์และวัดประสิทธิภาพของอัลกอริทึม จากวิธีการทำงานของอัลกอริธึม โดยมีองค์ประกอบสำคัญที่ต้องทำการวิเคราะห์ ดังนี้

- การวิเคราะห์ Space Complexity หรือที่เรียกว่า การวิเคราะห์หน่วยความจำที่ต้องใช้ในการประมวลผล (ใช้เนื้อที่หน่วยความจำมากน้อยแค่ไหน)

- การวิเคราะห์ Time Complexity หรือที่เรียกว่า การวิเคราะห์เวลาที่จะต้องใช้ในการประมวลผล (เวลาที่ใช้ในการประมวลผลเท่าไหร่)

องค์ประกอบของ SPACE COMPLEXITY

- Instruction Space เป็นจำนวนของหน่วยความจำที่คอมไพเลอร์จำเป็นต้องใช้ขณะทำการคอมไพล์โปรแกรม
- Data Space เป็นจำนวนหน่วยความจำที่ต้องใช้สำหรับเก็บค่าคงที่และตัวแปรทั้งหมดที่ต้องใช้ในการประมวลผลโปรแกรม ซึ่งจะสามารถแยกออกได้ 2 ประเภทคือ
  - Static memory เช่น ตัวแปรชนิด Array
  - Dynamic memory เช่น ตัวแปรชนิด Pointer
- Environment Stack Space เป็นจำนวนหน่วยความจำที่ต้องใช้ในการเก็บผลลัพธ์ของข้อมูลเอาไว้ เพื่อรอเวลาที่จะนำผลลัพธ์นั้นกลับไปประมวลผลอีกครั้ง (พบใน recursive function)

องค์ประกอบของ Time Complexity

ในการวิเคราะห์เวลาที่จะต้องใช้ในการประมวลผลจะใช้คณิตศาสตร์เป็นเครื่องมือสำหรับการวิเคราะห์ของอัลกอริทึม ซึ่งมีคณิตศาสตร์พื้นฐานสำหรับการวิเคราะห์ดังนี้

- ลอการิทึม (Logarithms)
- ผลรวม (Summation)
- เลขยกกำลัง (Logarithm)

รูปแบบการวิเคราะห์อัลกอริทึม

- Best – case การวิเคราะห์หาประสิทธิภาพที่ดีที่สุดในการประมวลผลของอัลกอริทึม
- Worst – case การวิเคราะห์หาประสิทธิภาพที่แย่ที่สุดในการประมวลผลของของอัลกอริทึม
- Average – case การหาค่าเฉลี่ยของเวลาที่ใช้ในการประมวลผลของของอัลกอริทึม

บทที่ 2 อัลกอริทึม

บทเรียนย่อย

2.1 Algorithm Introduction

2.2 Algorithm Analysis

2.3 Algorithm Efficiency

2.4 Big–O Notation

2.3 Algorithm Efficiency

ประสิทธิภาพของอัลกอริทึม มักถูกกำหนดมาในรูปแบบของฟังก์ชัน ด้วยการพิจารณาจากจำนวนของ element ที่ถูกโปรเซส และชนิดของลูปที่ใช้งาน ซึ่งแทนด้วยฟังก์ชัน f(n) = efficiency โดยประสิทธิภาพของลูปประเภทต่าง ๆ เป็นดังนี้

- ประสิทธิภาพของ Linear Loop คือ f(n) = n
- ประสิทธิภาพของ Logarithm Loop คือ f(n) = log n
- ประสิทธิภาพของ Linear Logarithm Loop คือ f(n) = n(log n)
- ประสิทธิภาพของ Quadratic Loop คือ f(n) = n^2
- ประสิทธิภาพของ Dependent Quadratic Loop คือ f(n) = n(n+1)/2

ประสิทธิภาพของ Linear Loops

Linear Loop คือ การวนลูปที่มีการเพิ่มหรือลดค่าให้กับตัวแปรที่ควบคุมลูป

ตัวอย่าง for (i=0; i<1000; i++)

program code ...

ประสิทธิภาพ f(n) = n

หมายเหตุ : ปัจจัย (n) ยิ่งมาก จำนวนรอบของการวนลูปก็จะสูงด้วย

ประสิทธิภาพของ Logarithmic Loops

Logarithmic loops คือ การวนลูปที่มีการคูณ หรือหารเป็นการควบคุมลูป ซึ่งประสิทธิภาพของ Logarithm Loop คือ f(n) = log n

ตัวอย่าง Multiply Loops

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

ตัวอย่าง Divide Loops

```
for (i=1; i<=1000; i/=2)
    program code ...
```

ตัวอย่างการวิเคราะห์ Multiply Loops และ Divide Loops

Multiply		Divide	
Iteration	value of i	Iteration	value of i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

$$f(n) = \log_2 n$$

ประสิทธิภาพของ Nested Loops

Nested Loops คือ การวนลูปที่มีการซ้อนลูป ซึ่งสามารถแบ่งได้ 3 ประเภทย่อย ดังนี้

1. ประสิทธิภาพของ Linear Logarithm Loop คือ f(n) = n(log n)

ตัวอย่าง

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

ประสิทธิภาพของ Nested Loops [2]

2. ประสิทธิภาพของ Quadratic Loop คือ f(n) = n^2

ตัวอย่าง

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

ประสิทธิภาพของ Nested Loops [3]

2. ประสิทธิภาพของ Dependent Quadratic Loop

คือ $f(n) = n(n+1) / 2$

ตัวอย่าง

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

การวิเคราะห์ความเร็วของอัลกอริทึม

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1,024	32,768	4,294,967,296

บทที่ 2 อัลกอริทึม

บทเรียนย่อย

2.1 Algorithm Introduction

2.2 Algorithm Analysis

2.3 Algorithm Efficiency

2.4 Big–O Notation

2.4 Big–O Notation

การวิเคราะห์อัลกอริธึมจะใช้วิธีหาจำนวนครั้งของการทำงานของโปรแกรม โดยมักจะสนใจค่าโดยประมาณเท่านั้น ซึ่งจะใช้สัญลักษณ์ว่า O เรียกว่า บิ๊กโอ (big O) ซึ่งเป็นสัญลักษณ์ทางคณิตศาสตร์ที่มาจากคำว่า Order of Magnitude ซึ่งประกอบด้วย 7 สัญลักษณ์ ดังนี้

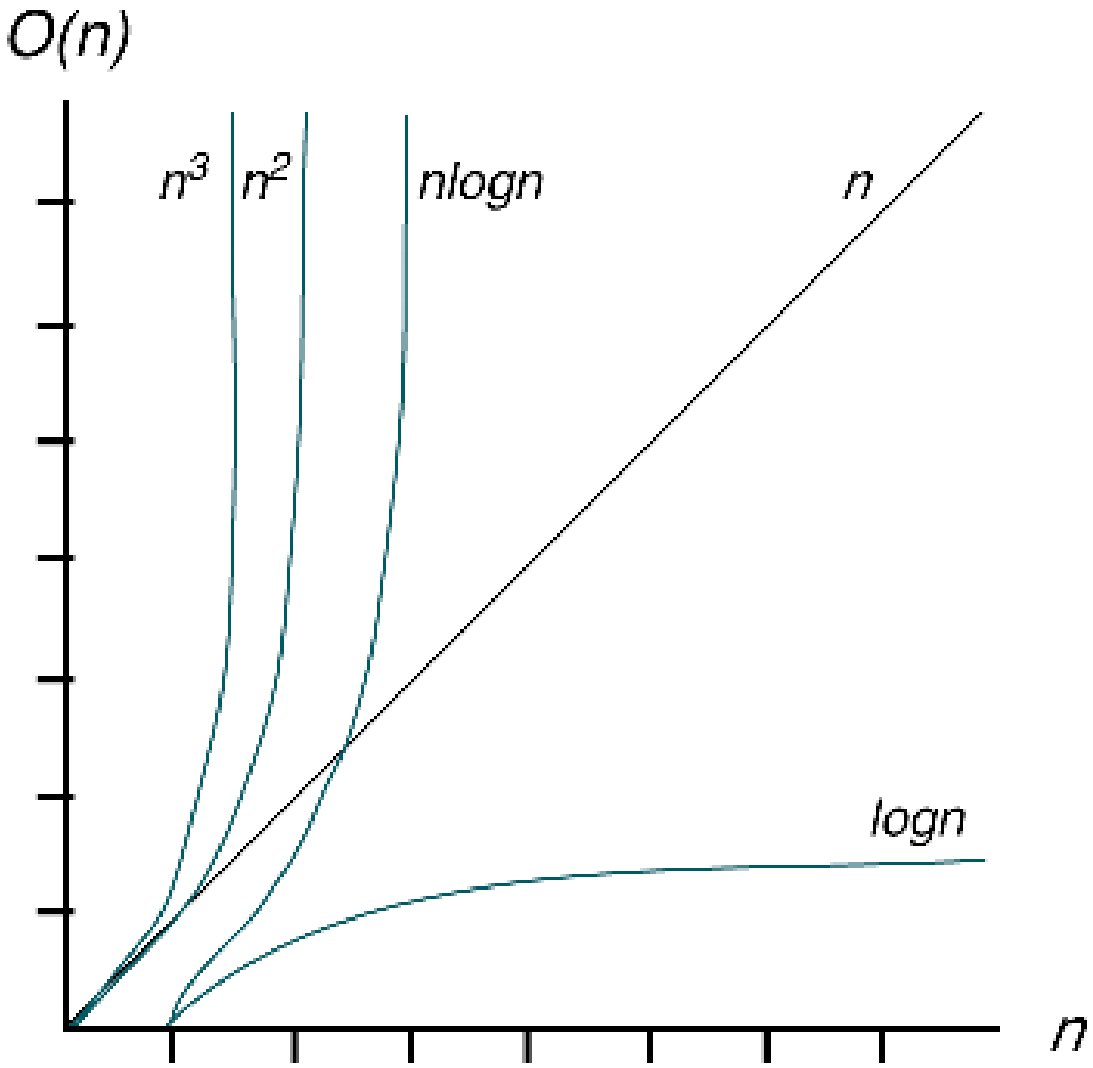
1. O(log n)
2. O(n)
3. O(n(log n))
4. O(n^2)
5. O(n^k)
6. O(c^n)
7. O(n!)

ประสิทธิภาพของ Big - O

เมื่อกำหนดค่าของ n = 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

กราฟแสดงการเติบโตของ Big - O



ตัวอย่างการหา Big – O แบบ Linear loops

เมื่อกำหนดค่าของ n = 3

```
1  Total = 0;                    ←1
2  for (i = 0;i <= n; i++){      ←n+2
3        Total = Total+i;        ←n+1
4  }
```

ค่า i	ตรวจสอบเงื่อนไข i < n	บรรทัด Total = Total + i
0	✓	✓
1	✓	✓
2	✓	✓
3	✓	✓
4	✓	✗
5	✗	✗
จำนวนครั้งที่ทำ	5	4

$f(n) = 1 + n + 2 + n + 1 = 2n + 4$ จึงได้ว่า Big-O = O(n)

ตัวอย่างการหา Big – O แบบ Logarithmic loops

```
1  Total = 0;                          ←1
2  for(i=1;i<10;i=i*2){                ←log2 n + 1
3    …                                 ←log2 n
4    …                                 ←log2 n
5  }
```

	เพิ่มขึ้นด้วยการคูณ		
รอบที่	ค่า i	บรรทัดที่ 2	บรรทัดที่ 3
1	1	✓	✓
2	2	✓	✓
3	4	✓	✓
4	8	✓	✓
5	16	✓	×
จำนวนครั้งที่ทำ		5	4

$f(n) = 1 + \log_2 n + 1 + \log_2 n + \log_2 n$

$= 3 \log_2 n + 2$

จึงได้ว่า Big-O = $O(\log_2 n)$

ตัวอย่างการหา Big – O แบบ Linear Logarithm Loops

```
1  Total = 0;                          ← 1
2  for (i = 0;i < n; i++){             ← n + 1
3     for (j = 1; j < n; j=j*2){       ← n(log2 n + 1)
4        …                             ← n*log2 n
5        …                             ← n*log2 n
6     }
7  }
```

ค่า i	j = j * 2	เงื่อนไข i < n	เงื่อนไข j < n	บรรทัดที่ 4
0	1	✓	✓	✓
0	2	✗	✓	✗
1	1	✓	✓	✓
1	2	✗	✓	✗
2	1	✓	✗	✗
จำนวนครั้งที่ทำ		3 = $n+1$	4 = $n*(\log_2 n+1)$	2 = $n*\log_2 n$

$f(n) = 1 + n + 1 + n(\log_2 n+1) + n\log_2 n + n\log_2 n = 3n\log_2 n+n+2$

จึงได้ว่า Big-O = $O(n\log_2 n)$

ตัวอย่างการหา Big – O แบบ Quadratic Loops

```
1  Total = 0;                          ← 1
2  for (i = 0;i < n; i++){             ← n + 1
3     for (j = 0; j < n; j++){         ← n(n + 1) = n^2+n
4           …                          ← n*n = n^2
5           …                          ← n*n = n^2
6     }
7  }
```

ค่า i	ค่า j	เงื่อนไข i < n	เงื่อนไข j < n	บรรทัดที่ 4
0	0	✓	✓	✓
0	1	×	✓	✓
0	2	×	✓	×
1	0	✓	✓	✓
1	1	×	✓	✓
1	2	×	✓	×
2	0	✓	×	×
จำนวนครั้งที่ทำ		n+1 = 3	n*(n+1) = 6	n*n = 4

$f(n) = 1 + n + 1 + n(n+1) + n2 + n2$

$= 3n^2 + 2n + 2$

จึงได้ว่า Big-O = $O(n^2)$

ตัวอย่างการหา Big – O แบบ Dependent Quadratic Loops

```
1  Total = 0;                          ← 1
2  for (i = 0;i < n; i++){             ← n + 1
3     for(j = 0; j < i; j++){          ← n((n+1)/2 + 1)
4        …                             ← n((n+1)/2)
5        …                             ← n((n+1)/2)
6     }
7  }
```

ค่า i	ค่า j	เงื่อนไข i < n	เงื่อนไข j < i	บรรทัดที่ 4
0	0	✓	✓	✓
0	1	×	✓	×
1	0	✓	✓	✓
1	1	×	✓	✓
1	2	×	✓	×
2	0	✓	×	×
จำนวนครั้งที่ทำ		3 = n+1	$5 = n\left(\frac{n+1}{2}+1\right)$	$3 = n\left(\frac{n+1}{2}\right)$

$$f(n) = 1 + n + 1 + n\left(\frac{n+1}{2}+1\right) + n\left(\frac{n+1}{2}\right) + n\left(\frac{n+1}{2}\right)$$

$$= 3n\left(\frac{n+1}{2}\right) + 2n + 2$$

จึงได้ว่า $\text{Big-O} = O\left(n\left(\frac{n+1}{2}\right)\right)$