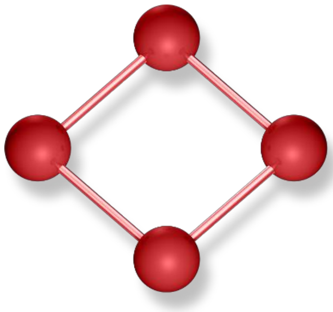
# บทที่ 17

## โครงสร้างข้อมูลเบื้องต้น

## (Basic Data Structures)



อัลกอริทึม (Algorithm) เป็นวิธีการดำเนินงานของโพรเซสบนระบบคอมพิวเตอร์ เพื่อให้สามารถแก้ปัญหาได้มีประสิทธิภาพสูงสุด (Optimal) บนทรัพยากรที่มีอย่างจำกัด ซึ่งทรัพยากรดังกล่าวมี 2 ประเภทคือ เวลาที่ใช้ในการประมวลผล (Processing Time) และพื้นที่หน่วยความจำ (Memory) ที่ใช้งาน ถ้าโพรเซสตั้งแต่ 2 โพรเซสขึ้นไป ดำเนินการแก้ปัญหาชนิดเดียวกันกับข้อมูลชุดเดียวกันแล้ว โพรเซสที่ใช้ทรัพยากรทั้ง 2 ชนิดน้อยที่สุดแสดงว่า อัลกอริทึมของโพรเซสนั้นๆ มีประสิทธิภาพที่สูงกว่านั่นเอง ในบทนี้จะกล่าวถึงวิธีที่ใช้วัดประสิทธิภาพของอัลกอริทึมในเบื้องต้นก่อน จากนั้นจะนำเอาวิธีการดังกล่าวไปประยุกต์ใช้กับอัลกอริทึมการจัดเรียง (Sorting) และการค้นหาข้อมูล (Searching) ต่อไป



### 1. การประเมินประสิทธิภาพอัลกอริทึมจากเวลาที่ใช้ทำงาน (Run time)

วิธีการแรกที่ใช้วัดประสิทธิภาพอัลกอริทึมคือ วัดจากระยะเวลาการทำงานตั้งแต่เริ่มต้นจนจบ โดยใช้สัญญาณนาฬิกาจริงจากเครื่องคอมพิวเตอร์ (Computer's clock) เรียกวิธีการนี้ว่า benchmarking หรือ profiling โดยออกแบบให้ชุดของข้อมูลที่ใช้สำหรับทดสอบมีความหลากหลาย เช่น ขนาดข้อมูลที่แตกต่างกัน และชนิดข้อมูลไม่เหมือนกันเป็นต้น ทำการทดสอบกับชุดข้อมูลดังกล่าวหลายๆ ครั้งแล้วจึงหาค่าเฉลี่ย จากตัวอย่าง ถ้าพิจารณาอัลกอริทึมการนับเลขจำนวนเต็มตั้งแต่ 1 – N โดย N คือขนาดของปัญหาเป็นเลขจำนวนเต็มที่ไม่เท่ากับ 0 และไม่เป็นค่าลบ ($\sum_{i=1}^{n} i$) กำหนดให้โปรแกรมทำงาน 5 รอบ โดยในแต่ละรอบ (Epoch) โปรแกรมจะเพิ่มขนาดของปัญหา (N) และจับเวลาเริ่มต้นและลบด้วยเวลาสิ้นสุด ผลลัพธ์ที่ได้คือ เวลาในการทำงานของอัลกอริทึมนั่นเอง ดังตัวอย่างโปรแกรมที่ 17.1

**Program Example 17.1: *Counting Runtime***

```
import time

problemSize = 10000000
```

```
4     print("%3s%14s%15s" %("Epoch","Problem Size","Seconds"))
⇨5    for count in range(5):
⇨6        start = time.time()
⇨7        work = 0
8         #Start of the algorithm
⇨9        for x in range(problemSize):
⇨10           work += x
11        #Stop of the algorithm
⇨12       elapsed = time.time() - start
⇨13       print("%5d%14d%15f" %(count+1, problemSize, elapsed))
⇨14       problemSize *= 2
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
Epoch  Problem Size        Seconds
    1      10000000       2.220476
    2      20000000       4.396939
    3      40000000       9.169125
    4      80000000      18.255202
    5     160000000      37.282913
```

จากตัวอย่างที่ 17.1 บรรทัดที่ 1 โปรแกรมนำเข้าโมดูล time เพื่อใช้สำหรับจับเวลาของโปรแกรมที่ประมวลผล บรรทัดที่ 3 กำหนดขนาดของปัญหา (problemSize) ให้มีขนาดใหญ่เท่ากับ 10,000,000 บรรทัดที่ 5 กำหนดจำนวนรอบในการทำงานให้กับโปรแกรม โดยใช้คำสั่ง for ในที่นี้โปรแกรมจะทำงานทั้งหมด 5 รอบ บรรทัดที่ 6 ทำการเรียกใช้เมธอด time() ซึ่งให้ผลลัพธ์มีหน่วยเป็นวินาที เก็บไว้ในตัวแปร start เพื่อทำหน้าที่เก็บเวลาเริ่มต้นก่อนอัลกอริทึมจะทำงาน บรรทัดที่ 9 และ 10 โปรแกรมจะทำอัลกอริทึมหาผลรวมจาก 0 ถึง problemSize -1 โดยเก็บผลลัพธ์สะสมไว้ในตัวแปร work เมื่อโปรแกรมทำงานเสร็จในรอบที่ 1 จะคำนวณหาเวลาเริ่มต้น – เวลาสิ้นสุดของอัลกอริทึม (บรรทัดที่ 12) และพิมพ์ผลลัพธ์ดังกล่าวออกจอภาพ (บรรทัดที่ 13) เมื่อพิมพ์ผลลัพธ์แล้ว โปรแกรมจะเพิ่มขนาดของปัญหาเป็น 2 เท่าหรือ 2N (โดยการคูณด้วย 2 ในบรรทัดที่ 14) ส่งผลให้เวลาประมวลผลของอัลกอริทึมเพิ่มขึ้นประมาณ 2 เท่าด้วย ดังแสดงในตัวอย่าง OUTPUT ด้านบน

จากตัวอย่างข้างบนเป็นอัลกอริทึมที่ใช้หาผลรวมของจำนวนนับตั้งแต่ 1 – N (problemSize) โดยอาศัยการทำงานของ for แบบชั้นเดียว แต่ถ้าอัลกอริทึมมีลักษณะในตัวอย่างต่อไปนี้ เวลาที่ใช้ประมวลผลจะมีค่าเป็นอย่างไร

```
for i in range(problemSize):
    for j in range(problemSise):
        work += j
```

คำตอบคือ อาจจะต้องใช้ตลอดทั้งคืนก็เป็นได้ เนื่องจากปัญหามีขนาดเพิ่มขึ้นเป็นการยกกำลังสอง ($N^2$) นั่นเอง สำหรับวิธีการวัดประสิทธิภาพอัลกอริทึมด้วยการจับเวลามีข้อจำกัด 2 ประการคือ

- ฮาร์ดแวร์ที่มีความแตกต่างกันจะส่งผลโดยตรงกับความเร็วในการวัดประสิทธิภาพ, ระบบปฏิบัติการที่ใช้งานไม่เหมือนกันจะมีผลกระทบเช่นกัน รวมไปถึงภาษาที่ใช้เขียน

โปรแกรมก็มีผลกระทบด้วย เช่น ภาษาซีจะทำงานเร็วกว่าภาษาไพธอน เป็นต้น ดังนั้น สรุปว่า ความแตกต่างของฮาร์ดแวร์และซอฟต์แวร์จะมีผลกระทบโดยตรงกับการประเมินประสิทธิภาพของอัลกอริทึม

- สำหรับในบางกรณี ข้อมูลที่ใช้ทดสอบมีขนาดใหญ่เกินกว่าที่เครื่องคอมพิวเตอร์จะสามารถคำนวณได้ เมื่อไม่สามารถคำนวณได้ก็ส่งผลให้การจับเวลาไม่บรรลุผลเช่นเดียวกัน

โปรดจำไว้ว่า การประเมินอัลกอริทึมโดยการจับเวลาจะใช้ได้กับบางกรณีเท่านั้น ดังนั้นนักคอมพิวเตอร์ทั้งหลายจึงได้พยายามคิดค้นวิธีการที่จะแก้ปัญหาการประเมินประสิทธิภาพอัลกอริทึมโดยไม่ขึ้นกับฮาร์ดแวร์และซอฟต์แวร์ ดังต่อไปนี้

### 2. การประเมินประสิทธิภาพอัลกอริทึมจากการนับจำนวนบรรทัดคำสั่ง (Counting instructions)

วิธีการนับจำนวนคำสั่งถูกคิดค้นขึ้นเพื่อจะแก้ปัญหาวิธีการแบบจับเวลา เนื่องจากวิธีการดังกล่าวจะขึ้นกับฮาร์ดแวร์และซอฟต์แวร์ โดยวิธีการนี้จะใช้วิธีการนับจำนวนบรรทัดของโปรแกรม (lines of code) ที่เขียนขึ้น โปรดจำไว้ว่าเป็นการนับจำนวนคำสั่งของโปรแกรมระดับสูง (High level code) ไม่ใช่คำสั่งระดับล่าง (Machine code) โดยหลักการนี้จะให้ความสนใจเป็นพิเศษกับการนับจำนวนบรรทัดในลูปชนิดซ้อน หรือ Nested loop หรือการเวียนเกิด (Recursion) ดังตัวอย่างต่อไปนี้

**Program Example 17.2: *Counting instructions (Interation)***

```
import time

problemSize = 1000
print("%3s%14s%15s" %("Epoch","Problem Size","Iterations"))
for count in range(5):
    number = 0
    work = 0
    #Start of the algorithm
    for x in range(problemSize):
        for y in range(problemSize):
            number += 1
            work += x
    #Stop of the algorithm
    print("%5d%14d%15d" %(count+1, problemSize, number))
    problemSize *= 2
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
Epoch   Problem Size      Iterations
    1           1000         1000000
    2           2000         4000000
    3           4000        16000000
    4           8000        64000000
```

```
        5          16000        256000000
```

จากตัวอย่างที่ 17.2 แสดงการนับจำนวนบรรทัดของโปรแกรม (lines of code) โดยบรรทัดที่ 3 โปรแกรมกำหนดค่าให้กับตัวแปร problemSize เท่ากับ 1,000 ซึ่งถ้ากำหนดค่าให้กับตัวแปรดังกล่าวใหญ่เกินไป จะส่งผลให้คอมพิวเตอร์ประมวลผลนานเกินไป บรรทัดที่ 6 กำหนดตัวแปร number เท่ากับ 0 เพื่อใช้สำหรับนับจำนวนบรรทัดของโปรแกรม บรรทัดที่ 9 และ 10 สร้างลูป for แบบซ้อน โดยวนลูปซ้ำจาก 0 ถึง problemSize (1,000) บรรทัดที่ 11 โปรแกรมทำการบวกค่าให้กับตัวแปร number ขึ้นทีละ 1 (number += 1 ➩ number = number + 1) ซึ่งหมายถึง โปรแกรมจะนับเป็น 1 บรรทัด โดยไม่สนใจว่าภายในลูปจะมีคำสั่งอื่นๆ ที่ทำงานอยู่ก็ตาม เช่นบรรทัดที่ 11 และ 12 สรุปคือ การวนลูป 1 ครั้ง อัลกอริทึมดังกล่าวจะนับเป็น 1 บรรทัด ตัวอย่างเช่น จาก Epoch 4 มีขนาดของปัญหาเท่ากับ 8,000 ส่งผลให้มีจำนวนบรรทัดในการทำงานเท่ากับ 64,000,000 บรรทัด เป็นต้น

ตัวอย่างโปรแกรมที่ 17.3 แสดงตัวอย่างการนับจำนวนบรรทัดของโปรแกรม Fibonacci โดยโปรแกรมดังกล่าวทำงานในลักษณะเวียนเกิด (Recursion) ดังนี้

**Program Example 17.3: *Counting instructions (Recursion)***

```python
class Counter(object):
    def __init__(self):
        self._number = 0

    def increment(self):
        self._number += 1

    def __str__(self):
        return str(self._number)

def Fibonacci(n, counter):
    counter.increment()
    if n < 3:
        return 1
    else:
        return Fibonacci(n-1, counter) + Fibonacci(n-2, counter)

problemSize = 2
print("%12s%15s" %("Problem Size", "Calls"))
for i in range(5):
    counter = Counter()
    Fibonacci(problemSize, counter)
    print("%12d%15s" %(problemSize, counter))
    problemSize *= 2
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

**OUTPUT**

```
Problem Size          Calls
           2              1
           4              5
           8             41
          16           1973
```

```
32          4356617
```

จากผลลัพธ์แสดงให้เห็นว่าขนาดของปัญหาเพิ่มขึ้นเป็น $2^n$ คือ 2, 4, 8, 16 และ 32 ส่งผลให้จำนวนบรรทัดเพิ่มขึ้นอย่างรวดเร็ว วิธีการนี้มีข้อดีคือไม่ขึ้นต่อฮาร์ดแวร์และซอฟต์แวร์ และยังเป็นพื้นฐานที่สำคัญในการออกแบบวิธีการประเมินประสิทธิภาพด้วยวิธีทางคณิตศาสตร์ ซึ่งจะกล่าวต่อไป

### 3. การประเมินประสิทธิภาพอัลกอริทึมจากการคำนวณหน่วยความจำ (Memory)

สำหรับการคำนวณหน่วยความจำแบ่งเป็น 2 ประเภทคือ การคำนวณหน่วยความจำที่เกิดจากการใช้งานจริงในขณะที่โพรเซสกำลังทำงาน และการคำนวณทางคณิตศาสตร์ ในส่วนแรกจะกล่าวถึงการคำนวณหน่วยความจำจากการใช้งานจริงก่อน ผู้เขียนแนะนำให้ใช้โมดูลชื่อ psutil ซึ่งสามารถดาวน์โหลดและติดตั้งได้จาก URL: https://pypi.python.org เมื่อติดตั้งแล้วสามารถเรียกใช้งานกับไพธอนได้ทันที จากโปรแกรมตัวอย่างที่ 17.4 แสดงการใช้โมดูล psutil ในการคำนวณขนาดของหน่วยความจำดังนี้

**Program Example 17.4: *Counting memory***

```
1   import psutil
2   import os
3
4   def memory_usage_psutil():
5       # return the memory usage in MB
6       process = psutil.Process(os.getpid())
7       mem = process.get_memory_info()[0] / float(2 ** 20)
8       return mem
9
10  mem = memory_usage_psutil()
11  print(str(mem) + " MB")
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
20.38 MB
```

จากตัวอย่างโปรแกรมที่ 17.4 แสดงการคำนวณหน่วยความจำที่โพรเซสกำลังใช้งาน โดยบรรทัดที่ 1 นำเข้าโมดูล psutil เพื่อใช้สำหรับแสดงข้อมูลทรัพยากรต่างๆ ของเครื่องคอมพิวเตอร์ บรรทัดที่ 2 นำเข้าโมดูล os เพื่อใช้ในการเข้าถึงข้อมูลระบบ เช่น หมายเลขโพรเซส, ชื่อไดเรคทรอรี่ปัจจุบัน เป็นต้น บรรทัดที่ 4 – 7 สร้างฟังชันชื่อ memory_usage_psutil() ทำหน้าที่คำนวณขนาดของหน่วยความจำ ค่าที่ส่งคืนกลับจากฟังชันดังกล่าว คือขนาดหน่วยความจำที่โพรเซสปัจจุบันกำลังใช้งานอยู่ มีหน่วยเป็นเม็กกะไบต์ (MB) โมดูล psutil ยังมีความสามารถในการแสดงข้อมูลอื่นๆ ของระบบได้อีกเป็นจำนวนมากดังนี้

| เมธอด | คำอธิบาย |
|---|---|
| psutil.cpu_times() | แสดงข้อมูลเกี่ยวกับหน่วยประมวลผลกลาง (ซีพียู) |
| psutil.virtual_memory() | แสดงข้อมูลเกี่ยวกับหน่วยความจำชนิดเวอร์ชวล |
| psutil.disk_partitions() | แสดงข้อมูลเกี่ยวกับหน่วยความจำสำรอง |
| psutil.net_io_counters() | แสดงข้อมูลเกี่ยวกับเน็ตเวิร์ค |
| psutil.users() | แสดงข้อมูลเกี่ยวกับผู้ใช้งาน |
| psutil.pids() | แสดงข้อมูลเกี่ยวกับโพรเซส |

**ส่วนที่สองแสดงการคำนวณหน่วยความจำด้วยวิธีทางคณิตศาสตร์**

องค์ประกอบของการวิเคราะห์หน่วยความจำที่ใช้โดยวิธีคณิตศาสตร์ มีดังต่อไปนี้

- Instruction Space คือ ขนาดหน่วยความจำที่จำเป็นต้องใช้ขณะคอมไพเลอร์ (Compiler) คอมไพล์โปรแกรมต้นฉบับ
- Data Space คือ ขนาดหน่วยความจำที่จำเป็นต้องใช้สำหรับเก็บข้อมูลค่าคงที่ และตัวแปรที่ใช้ในขณะประมวลผลโปรแกรม ซึ่งแยกออกได้ 2 ประเภท คือ
    - Static memory ขนาดของหน่วยความจำที่ถูกจองไว้ ซึ่งจะถูกประมวลผลแน่นอน เช่น หน่วยความจำที่ใช้เก็บค่าคงที่ หรือตัวแปรชนิดอาร์เรย์
    - Dynamic memory ขนาดหน่วยความจำที่ใช้ในการประมวลผลประเภทไม่แน่นอน คือ จะรู้ว่าต้องใช้หน่วยความจำเท่าไรก็ต่อเมื่อโปรแกรมต้องใช้งานนั้นเอง เช่น การประกาศตัวแปรพอยเตอร์ (Pointer) ในภาษา C หรือการเก็บข้อมูลในรูปแบบลิงค์ลิสต์ที่สามารถเพิ่มหรือลดขนาดการเก็บข้อมูลได้แบบอัตโนมัติโดยไม่ต้องจองพื้นที่หน่วยความจำไว้ก่อนใช้งาน เป็นต้น
- Environment Stack Space คือ หน่วยความจำที่ใช้สำหรับเก็บข้อมูลผลลัพธ์ที่ได้จากการประมวลผล เพื่อรอเวลาที่จะนำกลับไปใช้ใหม่ในโปรแกรม ซึ่งหน่วยความจำประเภทนี้จะเกิดขึ้นเมื่อมีการใช้งานเท่านั้น

ตัวอย่างการวิเคราะห์หน่วยความจำที่ใช้ด้วยวิธีทางคณิตศาสตร์ ดังต่อไปนี้

```
def myFunc(data1, data2):
    temp = 0
    temp = data1 + data2
    return temp
```

จากตัวอย่างโปรแกรมข้างบนได้ประกาศตัวแปรทั้งหมด 3 ตัว คือ data1, data2 และ temp ซึ่งตัวแปรทั้งสามเป็นข้อมูลชนิด integer (ภาษาไพธอน integer มีขนาดโดยประมาณคือ 24 ไบต์ โดยใช้

คำสั่ง sys.getsizeof(i)) ดังนั้นเมื่อคำนวณหาพื้นที่หน่วยความจำที่ใช้ทั้งหมดของฟังชันดังกล่าวเท่ากับ 24*3 = 72 ไบต์ ตัวอย่างต่อไปแสดงการคำนวณพื้นที่หน่วยความจำที่ใช้งานในลักษณะการเวียนเกิด (Recursion) ดังนี้

```
def Factorial(n):
    if n == 0:
        return 1
    else:
        return n * Factorial(n - 1)
```

จากฟังชัน Factorial(n) ด้านบน เมื่อกำหนดให้ n มีค่าเท่ากับ 3 จะต้องทำการวนซ้ำเท่ากับ 3 ครั้ง ซึ่งจะใช้พื้นที่หน่วยความจำเท่ากับ 3 * 24 = 72 ไบต์ สรุปได้ว่าฟังชัน Factorial จะต้องใช้พื้นที่ของหน่วยความจำเท่ากับ n * 24 ไบต์นั่นเอง

### 4. การประเมินประสิทธิภาพอัลกอริทึมด้วยฟังก์ชันอัตราการเติบโต (Growth-rate functions)

การประเมินประสิทธิภาพอัลกอริทึมด้วยฟังก์ชันอัตราการเติบโตจะมุ่งเน้นการวิเคราะห์เวลาที่ใช้ในการประมวลผลเป็นหลัก หรือเรียกว่า Time complexity analysis โดยเวลาที่อัลกอริทึมใช้ทำงานมี 2 ประเภทคือ เวลาที่ใช้ในการตรวจไวยากรณ์ (Compile time) และเวลาที่เครื่องคอมพิวเตอร์ใช้ในการประมวลผลอัลกอริทึม (Execution time) ซึ่งขึ้นอยู่กับชนิดข้อมูล จำนวนตัวแปรที่ใช้ และจำนวนลูปเป็นต้น ตัวอย่างที่ 1: แสดงการวิเคราะห์เวลาที่ใช้ฟังก์ชันอัตราการเติบโต ดังต่อไปนี้

**ตัวอย่างที่ 1:**

```
n = 0                         ⇨ กำหนดค่า 1 ครั้ง ❶
total = 0                     ⇨ กำหนดค่า 1 ครั้ง ❷
while n <= 30:                ⇨ เปรียบเทียบ n + 1 ครั้ง ❸
    total = total + n         ⇨ คำนวณ n ครั้ง ❹
    n = n + 1                 ⇨ คำนวณ n ครั้ง ❺
print("Total = ", total)      ⇨ แสดงผล 1 ครั้ง ❻
```

กำหนดให้ f(n) แทนประสิทธิภาพในการวิเคราะห์เวลาที่ใช้ในการประมวลผล และ n แทนจำนวนรอบในการทำงาน เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้ดังนี้

f(n) = ❶ + ❷ + ❸ + ❹ + ❺ + ❻

$f(n) = 1 + 1 + (n + 1) + n + n + 1 = 1 + 1 + 1 + 1 + (n + n + n) = 3n + 4$

$f(n) = 3n + 4$ -------------------------①

**ตัวอย่างที่ 2:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิดเวียนเกิด ดังนี้

| | |
|---|---|
| `def Factorial(n):` | ⇨ ถูกเรียกใช้ n ครั้ง ❶ |
| `    if n == 0:` | ⇨ ตรวจสอบเงื่อนไข n ครั้ง ❷ |
| `        return 1` | ⇨ คืนค่า 1 ครั้ง ❸ |
| `    else: return n * Factorial(n - 1)` | ⇨ เรียกใช้ตัวเอง n ครั้ง ❹ |

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = ❶ + ❷ + ❸ + ❹ = n + n + 1 + n

f(n) = 3n + 1 ------------------------②

**ตัวอย่างที่ 3:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิดวนลูปข้ามลำดับ (งานจะเสร็จเร็วกว่ากำหนด $\frac{n}{2}$) ดังนี้

| | |
|---|---|
| `total = 0` | ⇨ กำหนดค่า 1 ครั้ง |
| `for i in range(1, 3, 5, 7, 9):` | ⇨ เปรียบเทียบ $\frac{n}{2} + 1$ ครั้ง |
| `    total = total + i` | ⇨ คำนวณ $\frac{n}{2}$ ครั้ง |
| `    total = total * 2` | ⇨ คำนวณ $\frac{n}{2}$ ครั้ง |
| `print("Total = ", total)` | ⇨ แสดงผล 1 ครั้ง |

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = $1 + (\frac{n}{2} + 1) + \frac{n}{2} + \frac{n}{2} + 1 = \frac{3n}{2} + 3$ ------------------------③

**ตัวอย่างที่ 4:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิด Logarithmic ฐานสองดังนี้

สำหรับค่าตัวแปรที่ทำหน้าที่เป็นเงื่อนไขของลูปจะมีการเพิ่มขึ้นหรือลดลงด้วยการคูณหรือการหารเป็นอัตราเท่าตัวหรือ $2^n$

| | |
|---|---|
| `total = 0` | ⇨ กำหนดค่า 1 ครั้ง |
| `for i in (2**x for x in range(10)):` | ⇨ เปรียบเทียบ $\log_2 n + 1$ ครั้ง |
| `    total = total + i` | ⇨ คำนวณ $\log_2 n$ ครั้ง |
| `    total = total * 2` | ⇨ คำนวณ $\log_2 n$ ครั้ง |
| `print("Total = ", total)` | ⇨ แสดงผล 1 ครั้ง |

ผลลัพธ์ของค่า i = 1, 2, 4, 8, 16, 32,…, $2^n$

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = 1 + ($\log_2 n$ + 1) + $\log_2 n$ + $\log_2 n$ + 1 = 3 $\log_2 n$ + 3 ------------------------④

**ตัวอย่างที่ 5:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิดลูปซ้อนดังนี้

```
for i in range(10):               ⇨ เปรียบเทียบ n + 1 ครั้ง
      for i in range(10):         ⇨ เปรียบเทียบ n(n + 1) = n² + n ครั้ง
          total = total + i       ⇨ คำนวณ n*n = n² ครั้ง
```

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = (n + 1) + ($n^2$ + n) + $n^2$ = $2n^2$ + 2n + 1 ------------------------⑤

**ตัวอย่างที่ 6:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิด Linear Logarithmic ดังนี้

```
for i in range(10):                          ⇨ n + 1 ครั้ง
      for i in (2**x for x in range(10)):    ⇨ n(log₂ n + 1) ครั้ง
          total = total + i                  ⇨ n log₂ n ครั้ง
```

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = (n + 1) + n ($\log_2 n$ + 1) + n $\log_2 n$ = n + 2n $\log_2 n$ + 2 ------------------------⑤

**ตัวอย่างที่ 7:** เป็นการเขียนสมการฟังก์ชันอัตราการเติบโตชนิด Dependent Quadratic ดังนี้

```
Total = 0;                         ⇨ 1 ครั้ง
      for (i = 0;i < n; i++){      ⇨ n + 1
         for(j = 0; j < i; j++){   ⇨ n((n+1)/2 + 1)
             …                     ⇨ n((n+1)/2)
             …                     ⇨ n((n+1)/2)
         }
      }
```

เขียนเป็นสมการฟังก์ชันอัตราการเติบโตได้คือ

f(n) = 1 + (n + 1) + n($\frac{n+1}{2}$ + 1) + n($\frac{n+1}{2}$) + n($\frac{n+1}{2}$) = 3n($\frac{n+1}{2}$) + 2n + 2 -----------⑥

**Asymptotic notation**

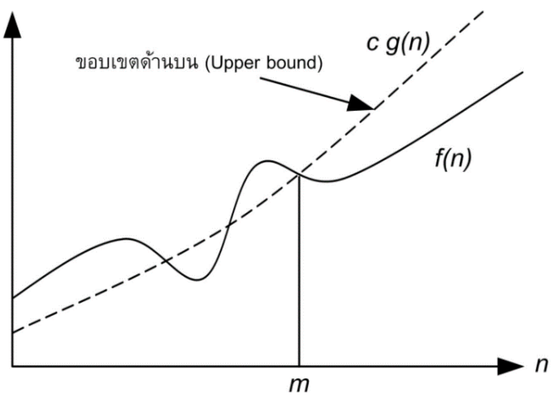
Asymptotic notation คือเครื่องหมายหรือสัญลักษณ์ที่ใช้อธิบายการเจริญเติบโตของฟังชัน หรืออัลกอริทึม (Algorithm Growth Rates) ซึ่งจะใช้ในการวัดประสิทธิภาพของอัลกอริทึม เช่น Big-O, Big-Omega, Big-Teta, Little-o และ Little-omga เป็นต้น แต่ในที่นี้ผู้เขียนจะอธิบายเฉพาะ Big-O เท่านั้น เพราะเป็นวิธีที่นิยมมากที่สุด ซึ่งมีรายละเอียดดังนี้

**อัตราการเติบโต Big-O**

Big-O เป็นขอบเขตบน (Upper bound) ของประสิทธิภาพที่แย่ที่สุดในการประมวลผลของอัลกอริทึม (Worst case) ซึ่งหมายความว่าอัลกอริทึมนี้จะทำงานไม่แย่ไปกว่า Big-O ของมันแล้ว ซึ่งก็เหมือนกับเป็นตัวบอกถึงประสิทธิภาพของอัลกอริทึมนั้นๆ นั่นเอง หรือเพื่อใช้ในการเปรียบเทียบว่าอัลกอริทึมใดดีกว่ากัน ซึ่ง Big-O มีคุณสมบัติดังนี้คือ

- เป็นการวัดประสิทธิภาพเชิงเวลาที่ใช้ในการประมวลผลของอัลกอริทึม
- เป็นฟังก์ชันเวลาขอบเขตบนที่ใช้ในการประมวลผล (Asymptotic upper bounds)
- สัญลักษณ์เป็นตัวโอใหญ่ (O)
- O(n) หมายถึง ฟังก์ชันนี้จะใช้เวลาในการประมวลผลน้อยกว่าหรือเท่ากับ n (<n) เสมอ

นิยาม Big-O: ฟังก์ชัน f(n) = O(g(n)) ก็ต่อเมื่อมีค่าคงที่ m, c ที่ทำให้ f(n) $\leq$ cg(n) เมื่อ n > m ดังรูป 17.1




รูปที่ 17.1 Big-O

จากตัวอย่าง กำหนดให้ $f(n) = 5n^4 - 37n^3 + 13n - 4$ สามารถแสดงการหาค่า Big-O และค่าคงที่ c กับ m ได้ดังนี้

จาก $f(n) = |5n^4 - 37n^3 + 13n - 4| \leq c|n4|$ โดยที่ $n > m$ (|…| คือค่าจำนวนเต็มบวกเท่านั้น) จะได้ว่า

เมื่อกำหนดให้ $m = 1$ จะทำให้ $n > 1$ สามารถทำให้การดำเนินการถูกต้องได้ดังนี้

$|5n^4 - 37n^3 + 13n - 4| \leq |5n^4 + 37n^4 + 13n^4 + 4n^4|$ (เนื่องจาก $37n^4 > 37n^3$)

$\leq 59|n^4|$

เพราะฉะนั้น $|5n^4 - 37n^3 + 13n - 4| \leq 59|n^4|$ เมื่อ $n > 1$

ดังนั้น $f(n) = 5n^4 - 37n^3 + 13^n - 4 \in O(n^4)$

จึงได้ว่า $c = 59$, $n = 1$ และ Big-O คือ $O(n^4)$

**Note**: สรุปแนวคิดการหา Big-O จะพิจารณาค่าที่มีผลกระทบมากที่สุดเพียงค่าเดียว ซึ่งค่าคงที่ c มีผลน้อยกว่าค่าที่มีผลกระทบมากที่สุด ดังนั้น จึงนำค่าที่มีผลกระทบมากที่สุดเป็นค่าของตัววัดประสิทธิภาพของ Big-O

**<u>สรุปการหาค่า Big-O แบบง่าย ๆ</u>**

1. ตัดสัมประสิทธิ์ของแต่ละเทอมทิ้ง (ตัดค่าคงที่ทิ้ง)
2. เลือกเทอมที่ใหญ่ที่สุดเก็บไว้เป็นคำตอบ

**ตัวอย่างที่ 1:** จากตัวอย่างที่ ① $f(n) = 3n + 4$

$f(n) = n$ ⇨ ตัดสัมประสิทธิ์ของแต่ละเทอมทิ้ง

$O(f(n)) = n = O(n)$ ⇨ เลือกเทอมใหญ่ที่สุด

**ตัวอย่างที่ 2:** $f(n) = 3n^4 + 2n^2 + n$

$f(n) = n^4 + n^2 + n$ ⇨ ตัดสัมประสิทธิ์

$O(f(n)) = n^4 = O(n^4)$ ⇨ เลือกเทอมใหญ่ที่สุด

**ตัวอย่างที่ 3:** $f(n) = 10n^3 + n^3 + 10$

$f(n) = n^3 + n^3$ ⇨ ตัดสัมประสิทธิ์

$O(f(n)) = n^3 = O(n^3)$ ⇨ เลือกเทอมใหญ่ที่สุด

**ตัวอย่างที่ 4:** $f(n) = 100$

$O(f(n)) = 1 = O(1)$

**ตัวอย่างที่ 5:** $f(n) = 100n + 1$

$f(n) = n$

$O(f(n)) = n = O(n)$

**ตัวอย่างที่ 6:** $f(n) = 20n\log_n + 5n$

$f(n) = n\log_n + n$ ⇨ ตัดสัมประสิทธิ์

$O(f(n)) = O(n\log_n)$ ⇨ เลือกเทอมใหญ่ที่สุด

**ตัวอย่างที่ 7:** กำหนดให้แต่ละโปรแกรมทำงานดังนี้คือ โปรแกรม 1 ⇨ $f(n) = 3n^2 + 2n$, โปรแกรม 2 ⇨ $f(n) = 2\log_2 n + 6n + n$, โปรแกรมที่ 3 ⇨ $f(n) = n + n\log_2 n + 4n + 9$ จงหาค่า Big-O และประเมินว่าโปรแกรมใดมีประสิทธิภาพการทำงานจากดีที่สุดไปยังแย่ที่สุด

โปรแกรมที่ 1 ⇨ $f(n) = 3n^2 + 2n = n^2 + n$

$= O(n^2)$

โปรแกรมที่ 2 ⇨ $f(n) = 2\log_2 n + 6n + n = \log_2 n + n + n$

$= O(n)$

โปรแกรมที่ 3 ⇨ $f(n) = n + n\log_2 n + 4n + 9 = n + n\log_2 n + n$

$= O(n\log_2 n)$

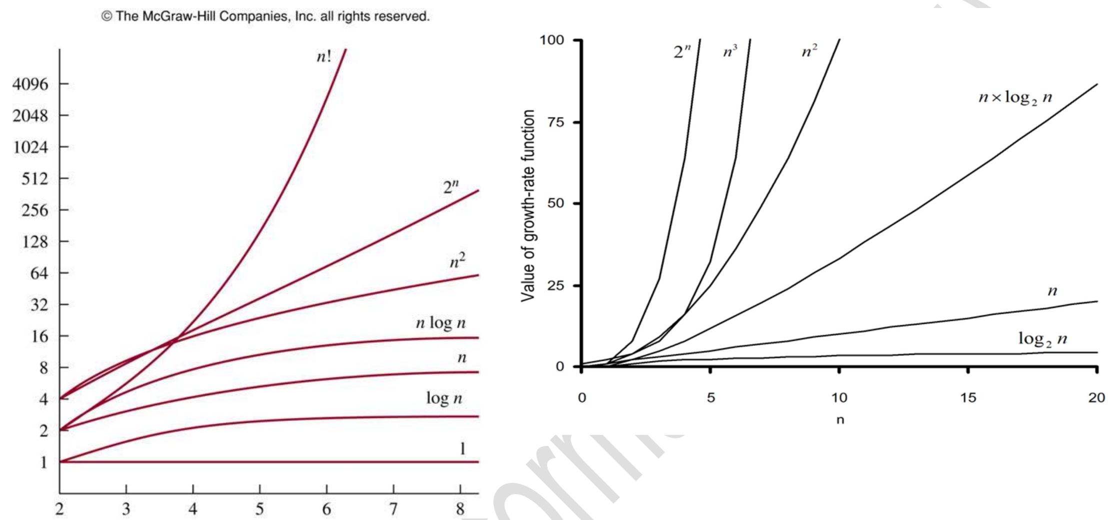
เพราะฉะนั้น $O(n)$ (โปรแกรมที่ 2 ดีที่สุด) < $O(n\log_2 n)$ (โปรแกรมที่ 3) < $O(n^2)$ (โปรแกรมที่ 1 แย่ที่สุด)

ตารางที่ 17.1 แสดงบทสรุปของอัตราการเติบโตตามการวัดประสิทธิภาพของอัลกอริทึม

| f(n) | แบบการนับตัวดำเนินการ | ประสิทธิภาพ |
|---|---|---|
| c | ค่าคงที่ (Constant) | เร็วที่สุด |
| $\log_2 n$ | ฟังชันลอการิทึม (Logarithm loops) | ค่อนข้างเร็ว |
| n | ฟังชันเชิงเส้น (Linear loops) | |

| | | |
|---|---|---|
| n $\log_2 n$ | ฟังชันลอการิทึมเชิงเส้น (Linear Logarithm) | |
| $n^2$ | ฟังชันกำลังสอง (Quadratic) | ปานกลาง |
| $n^3$ | ฟังชันกำลังสาม (Cubic) | |
| $n^k$ | ฟังชันโพลีโนเมียล (Polynomial) | |
| $2^n$ | ฟังชันเอ็กโพแนนเชียล (Exponential) | ค่อนข้างช้า |
| n! | ฟังชันแฟคทอเรียล (Factorial) | ช้าที่สุด |

จากตารางที่ 7.1 สามารถแสดงเป็นกราฟเชิงเส้นได้ดังรูปที่ 17.2




รูปที่ 17.2 แสดงอัตราการเติบโตตามการวัดประสิทธิภาพของอัลกอริทึม

**5. การวิเคราะห์ Best-case, Worst-case และ Average-case**

**Base-case:** การวิเคราะห์หาประสิทธิภาพที่ดีที่สุดในการประมวลผลของอัลกอริทึม เช่น การค้นหาข้อมูลในอาร์เรย์ แล้วเจอข้อมูลทันทีเมื่อทำการตรวจสอบในครั้งแรก

**Worst case:** การวิเคราะห์หาประสิทธิภาพที่แย่ที่สุดในการประมวลผลของอัลกอริทึม เช่น การค้นหาข้อมูลในอาร์เรย์ แล้วเจอข้อมูลในการตรวจสอบครั้งสุดท้าย เป็นต้น แสดงว่ากรณีนี้เป็นกรณีที่แย่ที่สุดเพราะต้องตรวจสอบข้อมูลจนถึงครั้งสุดท้ายจึงจะพบข้อมูลที่ต้องการ

**Average-case:** การหาค่าเฉลี่ยของเวลาที่ใช้ประมวลผลของอัลกอริทึม โดยนำเวลาที่ดีที่สุด + เวลาที่แย่ที่สุด แล้วหารด้วย 2

**6. การจัดเรียงข้อมูล (Sorting)**

Sorting หมายถึง การจัดเรียงข้อมูลให้มีการเรียงลำดับตามที่ผู้ใช้งานต้องการ โดยจะทำการเรียงข้อมูลจากค่าที่น้อยไปมาก หรือเรียงข้อมูลจากมากไปน้อยก็ได้ เช่น การเรียงลำดับตามความสูง จัด

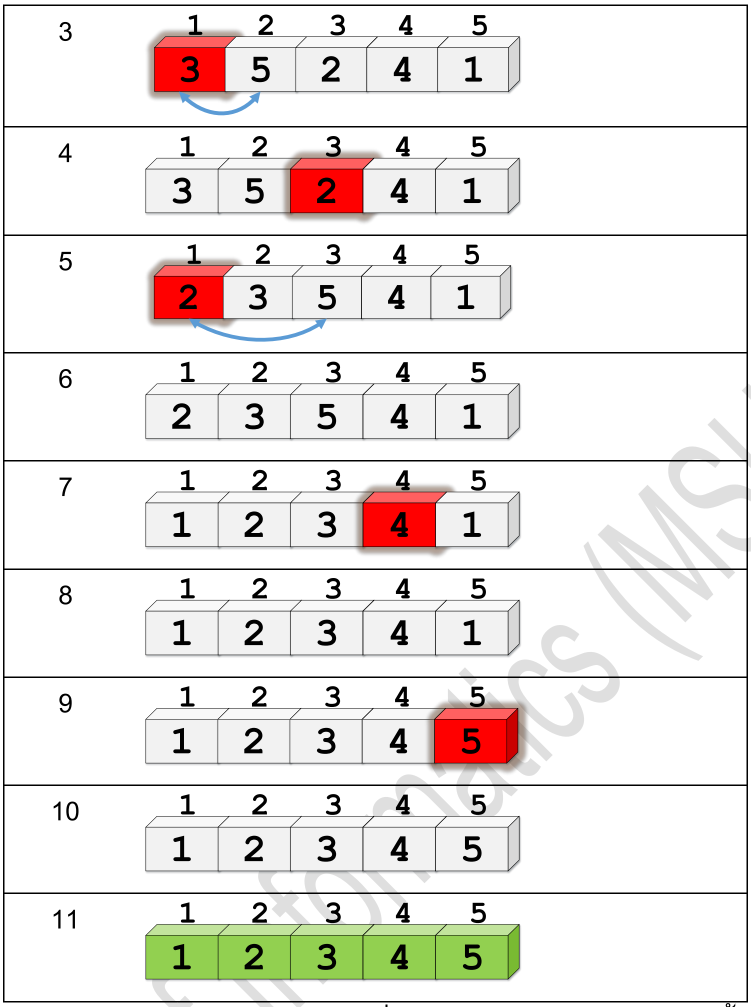
เรียงลำดับตามรหัสนักศึกษา การเรียงรายนามผู้ใช้โทรศัพท์ตามลำดับข้อมูลตัวอักษร การจัดเรียงตัวอักษรในพจนานุกรม การเรียงลำดับรายชื่อที่มีคะแนนสูงสุดไปต่ำสุด เป็นต้น สาเหตุที่ทำให้ต้องมีการจัดเรียงข้อมูลก็เพราะว่าง่ายต่อการค้นหา ถ้าหากมีข้อมูลจำนวนมากๆ แล้วไม่ทำการเรียงข้อมูล ก็จะทำให้เสียเวลาในการค้นหาข้อมูลเป็นอย่างมาก วิธีการจัดเรียงข้อมูลในระบบคอมพิวเตอร์แบ่งเป็น 2 ประเภทใหญ่ๆ คือ

- การเรียงข้อมูลภายใน (Internal sorting) จะใช้กับข้อมูลที่มีขนาดไม่เกินพื้นที่หน่วยความจำที่มีอยู่ในระบบ โดยเรียงข้อมูลภายในหน่วยความจำของเครื่องคอมพิวเตอร์ โดยไม่ต้องใช้หน่วยความจำสำรอง เช่น ดิสก์ เทป เป็นต้น ซึ่งมีวิธีการจัดเรียงลำดับข้อมูลภายในมีหลายวิธี ได้แก่ Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Shell Sort, Heap Sort และ Radix Sort เป็นต้น
- การเรียงข้อมูลภายนอก (External sorting) จะใช้กับข้อมูลที่มีขนาดใหญ่เกินกว่าที่จะเก็บลงในหน่วยความจำได้หมดภายในครั้งเดียว และจะใช้หน่วยความจำจากภายนอกแทน เช่น ดิสก์ เทป สำหรับเก็บข้อมูลชั่วคราวที่ได้รับการเรียงข้อมูลแล้ว ซึ่งมีวิธีการจัดเรียงลำดับข้อมูลภายนอกหลายวิธี ได้แก่ Merge Sort, Run List, การเรียงข้อมูลบนดิสก์ และการเรียงข้อมูลบนเทป เป็นต้น

### 1. การจัดเรียงข้อมูลแบบ Insertion Sort

**หลักการ:** มีลักษณะคล้ายกับการจัดไพ่ในมือของผู้เล่น คือ เมื่อผู้เล่นได้ไพ่ใบใหม่เพิ่มขึ้นมา จะนำไพ่ใบนั้นไปแทรกในตำแหน่งที่เหมาะสม ทำให้ไพ่ในมือบางส่วนต้องขยับตำแหน่งออกไป ซึ่งการจัดเรียงลำดับข้อมูลแบบแทรกนี้ จะเริ่มพิจารณาข้อมูลในตำแหน่งที่ 2 เป็นต้นไป เช่น ผู้เล่นมีไพ่หมายเลข 3, 5 และ 9 อยู่ในมือ เมื่อได้ไพ่ใบใหม่มาเป็นเลข 4 ผู้เล่นจะแทรกไพ่ดังกล่าวระหว่างไพ่หมายเลข 3 และ 5 เป็นต้น จากรูปด้านล่างแสดงการจัดเรียงข้อมูลจากน้อยไปมากโดยใช้อัลกอริทึมแบบ Insertion sort

| ลำดับที่ | แสดงการจัดเรียงลำดับ |
|---|---|
| 1 | 1 2 3 4 5<br>5 3 2 4 1 |
| 2 | 1 2 3 4 5<br>5 **3** 2 4 1 |




จากรูปด้านบนแสดงการทำงานของ Insertion Sort ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.5:** ***Insertion Sort***

```
def insertion(data):
    for i in range(1, len(data)):
        temp = data[i]
        j = i
        while (temp < data[j-1] and j>0):
            data[j] = data[j-1]
            j -= 1
        data[j] = temp

data = [6, 1, 7, 9, 2, 8, 5, 4, 3]
print("The data before sorting = ", data)
insertion(data)
print("The data after sorting = ", data)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
The data before sorting =  [6, 1, 7, 9, 2, 8, 5, 4, 3]
The data after sorting =  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

การจัดเรียงลำดับโดยการแทรก จะมีการจัดเรียงลำดับข้อมูลทั้งหมด n - 1 รอบ โดยแต่ละรอบนั้นจำนวนการเปรียบเทียบจะไม่แน่นอน เพราะในแต่ละรอบการเปรียบเทียบจะสิ้นสุดเมื่อไม่มีการสลับตำแหน่งของข้อมูล เมื่อพิจารณาจำนวนการเปรียบเทียบข้อมูลแบ่งเป็น 3 กรณีคือ

1. กรณีดีที่สุด (Base-case) ข้อมูลถูกจัดเรียงลำดับเรียบร้อยแล้ว การเปรียบเทียบในกรณีนี้แต่ละรอบจะมีการเปรียบเทียบข้อมูลเพียงครั้งเดียวเท่านั้น เพราะฉะนั้นจำนวนการเปรียบเทียบข้อมูลจะเท่ากับ n – 1 ครั้ง ดังนั้น BigO = O(n)
2. กรณีแย่ที่สุด (Worst-case) ข้อมูลถูกจัดเรียงลำดับในตำแหน่งที่สลับกัน เช่น เรียงลำดับค่าข้อมูลจากมากไปหาน้อย (ในกรณีที่ต้องการจัดเรียงลำดับจากน้อยไปหามาก) ในกรณีนี้แต่ละรอบจะมีจำนวนการเปรียบเทียบข้อมูลดังนี้

   รอบที่ 1 เปรียบเทียบทั้งหมดจำนวน 1 ครั้ง

   รอบที่ 2 เปรียบเทียบทั้งหมดจำนวน 2 ครั้ง

   รอบที่ 3 เปรียบเทียบทั้งหมดจำนวน 3 ครั้ง

   ........

   รอบที่ n – 2 เปรียบเทียบทั้งหมดจำนวน n - 2 ครั้ง

   รอบที่ n – 1 เปรียบเทียบทั้งหมดจำนวน n - 1 ครั้ง

   ดังนั้น จำนวนการเปรียบเทียบทั้งหมดเท่ากับ

   $1 + 2 + 3 + \ldots + (n - 2) + (n - 1) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$

   ฉะนั้น BigO ในกรณี West-case ของ Insertion Sort เท่ากับ $O(n^2)$

3. กรณีเฉลี่ย (Average-case) จำนวนการเปรียบเทียบข้อมูลทั้งหมด สามารถคำนวณได้จากนำเอาจำนวนการเปรียบเทียบในกรณีที่ดีที่สุดและกรณีที่แย่ที่สุดมารวมกันและหารด้วย 2 ดังนี้

   $$\frac{(n-1) + \frac{n^2 - n}{2}}{2} = \frac{\frac{(2n-2) + n^2 - n}{2}}{2} = \frac{\frac{n^2 + n - 2}{2}}{\frac{2}{1}} = \frac{n^2 + n - 2}{2} x \frac{1}{2} = \frac{n^2 + n - 2}{4}$$

   ฉะนั้น BigO ในกรณี Average-case ของ Insertion Sort เท่ากับ $O(n^2)$

**2. การจัดเรียงข้อมูลแบบ Selection Sort**

**หลักการ:** จดจำตำแหน่งข้อมูลที่มีค่าน้อยที่สุด (กรณีเรียงข้อมูลจากน้อยไปหามาก) ในแต่ละรอบและนำข้อมูลในตำแหน่งดังกล่าวมาแลกกับข้อมูลในตำแหน่งที่ 1, 2, 3, …, n - 1 ตามลำดับ

จากรูปด้านล่างแสดงการจัดเรียงข้อมูลจากน้อยไปมากโดยใช้อัลกอริทึมแบบ Selection Sort

| ลำดับที่ | แสดงการจัดเรียงลำดับ |
|---|---|
| 1 | 1 2 3 4 5<br>5 1 2 4 3 |
| 2 | 1 2 3 4 5<br>5 1 2 4 3 |
| 3 | 1 2 3 4 5<br>1 5 2 4 3 |
| 4 | 1 2 3 4 5<br>1 5 2 4 3 |
| 5 | 1 2 3 4 5<br>1 5 2 4 3 |
| 6 | 1 2 3 4 5<br>1 2 5 4 3 |
| 7 | 1 2 3 4 5<br>1 2 5 4 3 |
| 8 | 1 2 3 4 5<br>1 2 5 4 3 |
| 9 | 1 2 3 4 5<br>1 2 3 4 5 |
| 10 | 1 2 3 4 5<br>1 2 3 4 5 |
| 11 | 1 2 3 4 5<br>1 2 3 4 5 |
| 12 | 1 2 3 4 5<br>1 2 3 4 5 |
| 13 | 1 2 3 4 5<br>1 2 3 4 5 |
| 14 | 1 2 3 4 5<br>1 2 3 4 5 |

จากรูปด้านบนแสดงการทำงานของ Selection Sort ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.6: *Selection Sort***

```
def selection(data):
    for i in range(0, len(data)-1):
        indexOfMin = i
        for j in range(i+1, len(data)):
            if (data[j] < data[indexOfMin]):
                indexOfMin = j
        temp = data[i]
        data[i] = data[indexOfMin]
        data[indexOfMin] = temp

data = [6, 1, 7, 9, 2, 8, 5, 4, 3]
print("The data before sorting = ", data)
selection(data)
print("The data after sorting = ", data)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
The data before sorting =  [6, 1, 7, 9, 2, 8, 5, 4, 3]
The data after sorting =  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

ในกรณีที่มีข้อมูล N ตัว การเรียงลำดับข้อมูลแบบ Selection Sort จะมีการค้นหาทั้งหมด N - 1 ครั้ง และมีการสลับที่กันจริงไม่เกิน N - 1 ครั้ง โดยในแต่ละรอบจะมีการเปรียบเทียบข้อมูลดังนี้

1. กรณีดีที่สุด (Base-case) มีค่า BigO = $O(n^2)$
2. กรณีแย่ที่สุด (Worst-case) ข้อมูลถูกจัดเรียงลำดับในตำแหน่งที่สลับกัน เช่น เรียงลำดับค่าข้อมูลจากมากไปหาน้อย (ในกรณีที่ต้องการจัดเรียงลำดับจากน้อยไปหามาก) ในกรณีนี้แต่ละรอบจะมีจำนวนการเปรียบเทียบข้อมูลดังนี้

   รอบที่ 1 เปรียบเทียบทั้งหมดจำนวน n - 1 ครั้ง

   รอบที่ 2 เปรียบเทียบทั้งหมดจำนวน n - 2 ครั้ง

   รอบที่ 3 เปรียบเทียบทั้งหมดจำนวน n - 3 ครั้ง

   ........

   รอบที่ n – 2 เปรียบเทียบทั้งหมดจำนวน 2 ครั้ง

   รอบที่ n – 1 เปรียบเทียบทั้งหมดจำนวน 1 ครั้ง

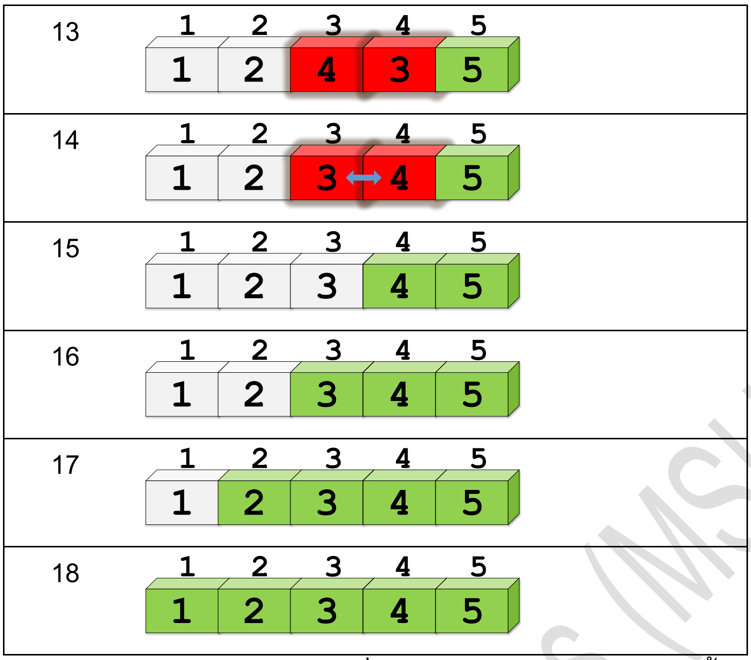
   ดังนั้น จำนวนการเปรียบเทียบทั้งหมดเท่ากับ

   $$1 + 2 + 3 + \ldots + (n - 2) + (n - 1) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = O(n^2)$$
3. กรณีเฉลี่ย (Average-case) มีค่า BigO = $O(n^2)$

### 3. การจัดเรียงข้อมูลแบบ Bubble Sort

**หลักการ:** เปรียบเทียบข้อมูลในตำแหน่งที่อยู่ติดกันทีละคู่ ถ้าข้อมูลที่เปรียบเทียบไม่อยู่ในตำแหน่งที่ต้องการแล้วให้ทำการสลับที่กันระหว่างข้อมูล 2 ตัวนั้น ทำเช่นนี้จนกระทั่งเปรียบเทียบครบทุกตัว ซึ่งคือ N - 1 ครั้ง จากรูปด้านล่างแสดงการจัดเรียงข้อมูลจากน้อยไปมากโดยใช้อัลกอริทึมแบบ Bubble Sort

| ลำดับที่ | แสดงการจัดเรียงลำดับ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| 1 | 5 | 1 | 2 | 4 | 3 |
| 2 | 5 | 1 | 2 | 4 | 3 |
| 3 | 1 | 5 | 2 | 4 | 3 |
| 4 | 1 | 5 | 2 | 4 | 3 |
| 5 | 1 | 2 | 5 | 4 | 3 |
| 6 | 1 | 2 | 5 | 4 | 3 |
| 7 | 1 | 2 | 4 | 5 | 3 |
| 8 | 1 | 2 | 4 | 5 | 3 |
| 9 | 1 | 2 | 4 | 3 | 5 |
| 10 | 1 | 2 | 4 | 3 | 5 |
| 11 | 1 | 2 | 4 | 3 | 5 |
| 12 | 1 | 2 | 4 | 3 | 5 |




จากรูปด้านบนแสดงการทำงานของ Bubble Sort ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.7: *Bubble Sort***

```python
def bubble(data):
    for i in range(0, len(data)-1):
        for j in range(len(data)-1, i, -1):
            if (data[j] < data[j-1]):
                temp = data[j]
                data[j] = data[j-1]
                data[j-1] = temp

data = [6, 1, 7, 9, 2, 8, 5, 4, 3]
print("The data before sorting = ", data)
bubble(data)
print("The data after sorting = ", data)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้


OUTPUT

```
The data before sorting =  [6, 1, 7, 9, 2, 8, 5, 4, 3]
The data after sorting =  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

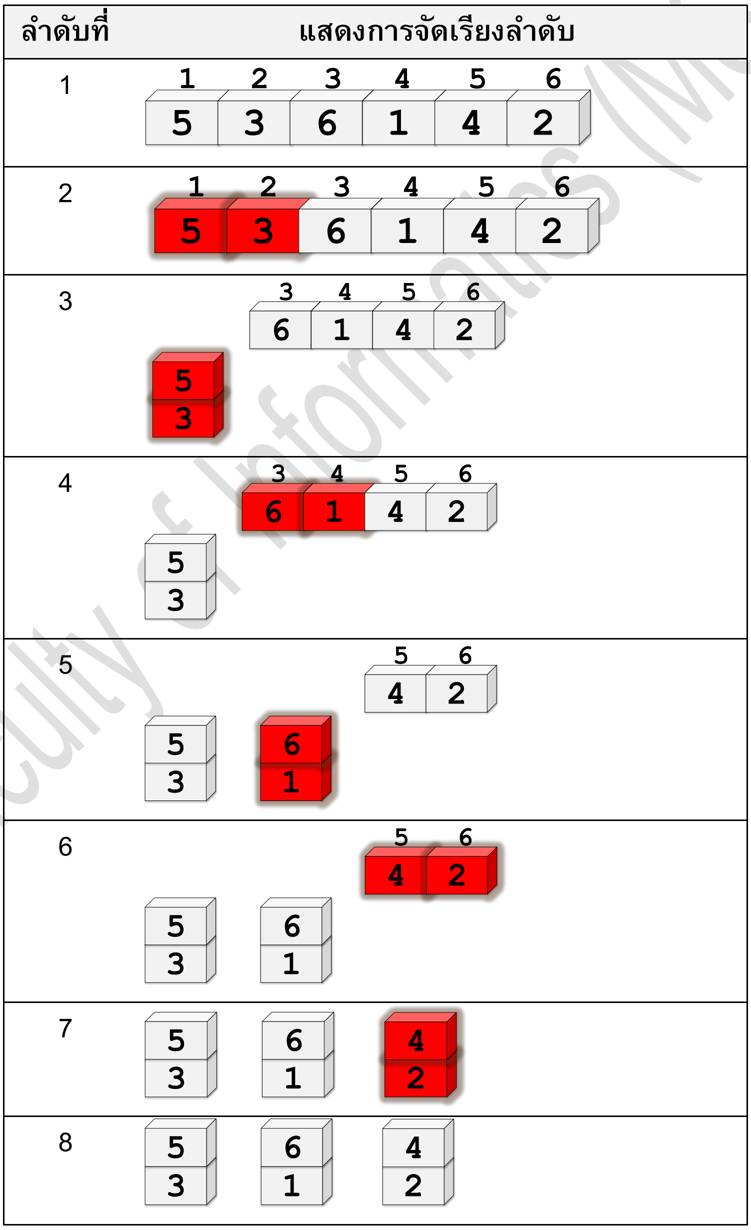
จากโปรแกรมสังเกตุว่าข้อมูลมีการสลับที่กันไปเรื่อยๆ จนได้ข้อมูลที่มีการเรียงลำดับเป็นที่เรียบร้อยแล้ว ซึ่งมีการจัดเรียง n - 1 รอบ โดยในแต่ละรอบจะมีการเปรียบเทียบข้อมูลเป็น n - 1, n - 2, n – 3 ,...., 3, 2, 1 ดังนั้น จำนวนการเปรียบเทียบทั้งหมดคือ

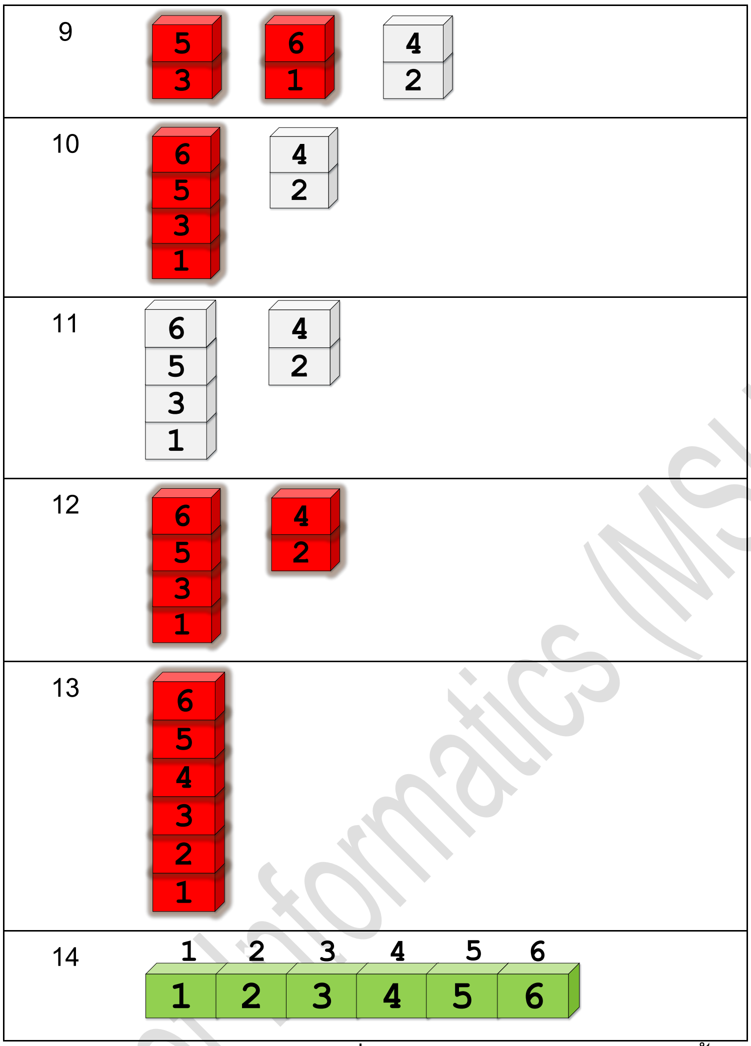
1. กรณีดีที่สุด (Base-case) มีค่า BigO = O(n)
2. กรณีแย่ที่สุด (Worst-case) มีค่า BigO เท่ากับ

   $$1 + 2 + 3 + \dots + (n - 2) + (n - 1) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = O(n^2)$$
3. กรณีเฉลี่ย (Average-case) มีค่า BigO = $O(n^2)$

## 4. การจัดเรียงข้อมูลแบบ Merge Sort

**หลักการ:** เป็นวิธีการเรียงข้อมูลที่มีความสำคัญมาก เพราะเป็นวิธีสำคัญในการเรียงข้อมูลขนาดใหญ่ (การเรียงแบบภายนอก) วิธีการเรียงจะเริ่มกระทำครั้งละ 2 ค่า ซึ่งจะได้ลิสต์ย่อยจำนวน n/2 ลิสต์ แต่ละลิสต์มี 2 ค่า จากนั้นจะทำการ merge ต่ออีกครั้งละ 2 ลิสต์แล้วจะได้ลิสต์ที่เรียงแล้ว จำนวน n/4 ลิสต์ แต่ละลิสต์มี 4 ค่า ทำเช่นนี้ไปเรื่อยๆ จนในที่สุดจะทำการ merge 2 ลิสต์สุดท้ายเข้าด้วยกัน จะได้ลิสต์ที่เรียงเรียบร้อยแล้ว จากรูปด้านล่างแสดงการจัดเรียงข้อมูลจากน้อยไปมากโดยใช้อัลกอริทึมแบบ Bubble Sort






จากรูปด้านบนแสดงการทำงานของ Merge Sort ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.8: *Merge Sort***

```
def mergeSort(data):
    if len(data) < 2: return data
    m = len(data) // 2
    return merge(mergeSort(data[:m]), mergeSort(data[m:]))

def merge(l, r):
    result = []
    i = j = 0
    while i < len(l) and j < len(r):
        if l[i] < r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
    result.extend(l[i:])
    result.extend(r[j:])
    return result
```

```
data = [6, 1, 7, 9, 2, 8, 5, 4, 3]
print("The data before sorting = ", data)
data = mergeSort(data)
print("The data after sorting = ", data)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
The data before sorting =  [6, 1, 7, 9, 2, 8, 5, 4, 3]
The data after sorting =  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

จากโปรแกรมจำนวนการเปรียบเทียบทั้งหมดคือ

1. กรณีดีที่สุด (Base-case) มีค่า BigO = $O(n\log_{10}(n))$
2. กรณีแย่ที่สุด (Worst-case) มีค่า BigO = $O(n\log_{10}(n))$
3. กรณีเฉลี่ย (Average-case) มีค่า BigO = $O(n\log_{10}(n))$

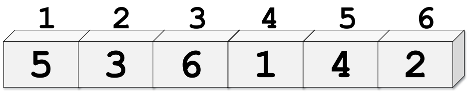
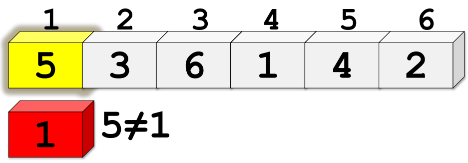
## 7. การค้นหาข้อมูล (Searching)

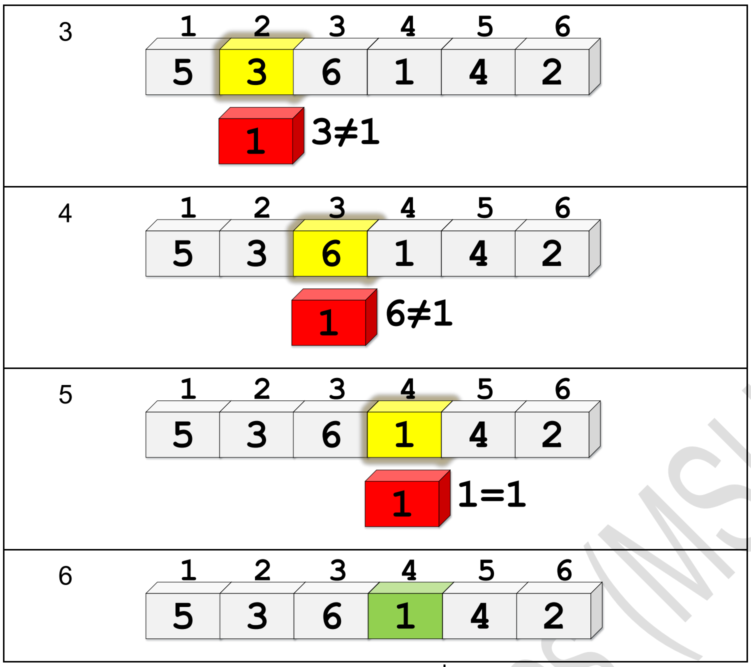
การค้นหาข้อมูลมีอยู่หลายวิธีด้วยกัน แต่ที่นิยมใช้และควรที่จะต้องทำความเข้าใจได้แก่

- การค้นหาข้อมูลแบบลำดับ (Sequential Search)
- การค้นหาข้อมูลแบบพับครึ่ง (Binary Search)

### 1. การค้นหาข้อมูลแบบลำดับ (Sequential Search)

**หลักการ:** การค้นหาข้อมูลแบบลำดับหรือเรียกอีกชื่อหนึ่งว่า Linear Search เป็นการค้นหาข้อมูลที่มีลักษณะการทำงานแบบเรียงตามลำดับ เป็นวิธีที่ง่ายที่สุด ในการค้นหาข้อมูลแบบนี้จะทำการตรวจสอบข้อมูลที่ต้องการโดยไล่ไปทีละ 1 ข้อมูลตามลำดับ ทำอย่างนี้ไปเรื่อยๆ จนกว่าจะพบข้อมูล (Key) ตามที่ต้องการหรือหมดข้อมูลที่ต้องการค้นหา จากรูปด้านล่างแสดงการค้นหาข้อมูลโดยใช้อัลกอริทึมแบบ Sequential Search

| ลำดับที่ | แสดงการจัดเรียงลำดับ |
|---|---|
| 1 |   |
| 2 |   |




จากรูปด้านบนแสดงการทำงานของ Sequential Search ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.9:** ***Sequential Search***

```
def sequentialSearch(data, key):
    for i in range(len(data)):
        if key == data[i]:
            return i
    return -1

data = [6, 1, 7, 9, 2, 8, 5, 4, 3]
print("Data source = ", data)
print("Would like to search the 8 number.")
position = sequentialSearch(data, 8)
print("The position of the 8 number is ", position)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

**OUTPUT**

```
Data source =  [6, 1, 7, 9, 2, 8, 5, 4, 3]
Would like to search the 8 number.
The position of the 8 number is  5
```

การทำงานของการค้นหาข้อมูลแบบนี้ จะทำการรับค่าข้อมูล (Key) สำหรับค้นหาแล้วนำไปค้นหากับข้อมูลที่เก็บอยู่ สำหรับวิธีการนี้สามารถสรุปประสิทธิภาพในการค้นหาได้เป็น 3 กรณี คือ

1. กรณีที่ดีที่สุด โปรแกรมทำการเปรียบเทียบข้อมูลเพียง 1 ครั้งเท่านั้น ถ้าข้อมูลที่ต้องการค้นหาอยู่อันดับแรกของข้อมูลทั้งหมด ฉะนั้นค่า BigO = O(1)

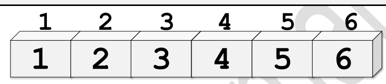
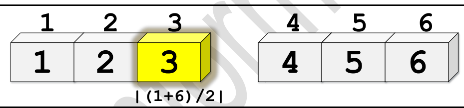
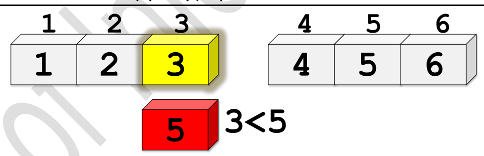
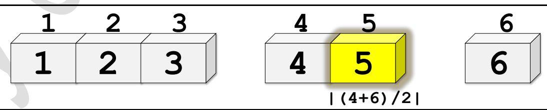
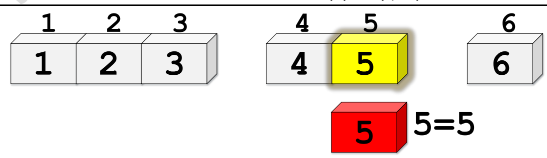
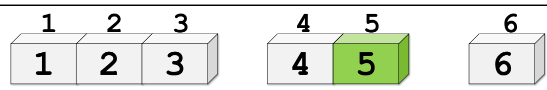
2. กรณีที่แย่ที่สุด โปรแกรมทำการเปรียบเทียบข้อมูลจำนวน n ครั้ง (n = จำนวนข้อมูลทั้งหมด) ถ้าข้อมูลที่ต้องการค้นหาอยู่อันดับสุดท้ายของข้อมูลทั้งหมด ฉะนั้นค่า BigO = O(n)

3. กรณีเฉลี่ยทั่วไป โปรแกรมทำการเปรียบเทียบข้อมูลประมาณ n/2 ครั้ง

## 2. การค้นหาข้อมูลแบบพับครึ่ง (Binary Search)

**หลักการ:** การค้นหาข้อมูลแบบพับครึ่งนี้ได้ถูกคิดค้นขึ้นมาเพื่อแก้ไขข้อเสียของการค้นหาข้อมูลแบบลำดับในกรณีที่ข้อมูลที่ต้องการค้นหาอยู่ตัวท้ายๆ ของข้อมูล แต่ข้อกำหนดของการค้นหาข้อมูลแบบพับครึ่งนี้จะสามารถทำงานได้กับข้อมูลที่มีการจัดเรียงเรียบร้อยแล้ว (เรียงจากมากไปน้อยหรือเรียงจากน้อยไปหามากก็ได้)

หลักการค้นหาข้อมูลแบบพับครึ่งจะทำงานโดยการแบ่งข้อมูลออกเป็น 2 ส่วน แล้วนำค่ากลางมาเปรียบเทียบกับข้อมูล (Key) ที่ต้องการค้นหา ถ้าข้อมูลที่ทำการค้นหาเรียงลำดับจากน้อยไปหามาก เมื่อเปรียบเทียบแล้วข้อมูลที่ต้องการค้นหามีค่ามากกว่า แสดงว่าต้องไปค้นหาข้อมูลต่อในส่วนข้อมูลครึ่งหลังต่อ จากนั้นให้นำข้อมูลครึ่งหลังพับครึ่งหาค่ากลางอีก ทำอย่างนี้ไปเรื่อยๆ จนกว่าจะได้ข้อมูลที่ต้องการ ดังตัวอย่างต่อไปนี้

| ลำดับที่ | แสดงการจัดเรียงลำดับ |
|---|---|
| 1 |  |
| 2 |   |
| 3 |   |
| 4 |   |
| 5 |   |
| 6 |  |

จากรูปด้านบน แสดงการทำงานของ Binary Search ซึ่งสามารถเขียนโปรแกรมได้ดังนี้

**Program Example 17.10: *Binary Search***

```
def binarySearch(data, key):
    min = 0
    max = len(data) - 1

    while True:
```

```
        if max < min:
            return -1
        m = (min + max) // 2
        if data[m] < key:
            min = m + 1
        elif data[m] > key:
            max = m - 1
        else:
            return m + 1

data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print("Data source = ", data)
print("Would like to search the 8 number.")
position = binarySearch(data, 8)
print("The position of the 8 number is ", position)
```

ผลลัพธ์ที่เกิดขึ้นจากการรันโปรแกรมดังนี้

OUTPUT

```
Data source =  [1, 2, 3, 4, 5, 6, 7, 8, 9]
Would like to search the 8 number.
The position of the 8 number is  8
```

การทำงานของการค้นหาข้อมูลแบบนี้ จะทำการรับค่าข้อมูล (Key) สำหรับค้นหาแล้วนำไปค้นหากับข้อมูลที่เก็บอยู่ในลักษณะพับครึ่ง ทำให้ประสิทธิภาพดีกว่าแบบ Sequential Search สำหรับวิธีการนี้สามารถสรุปประสิทธิภาพในการค้นหาได้เป็น 3 กรณี คือ

1. กรณีที่ดีที่สุด โปรแกรมทำการเปรียบเทียบข้อมูลเพียง 1 ครั้งเท่านั้น ถ้าข้อมูลที่ต้องการค้นหาอยู่ในตำแหน่งที่พับครึ่งแล้วพบในครั้งแรก ฉะนั้นค่า BigO = O(1)

2. กรณีที่แย่ที่สุด โปรแกรมทำการเปรียบเทียบข้อมูลจำนวน log(n) ครั้ง ฉะนั้นค่า BigO = O(log(n))

3. กรณีเฉลี่ยทั่วไป โปรแกรมทำการเปรียบเทียบข้อมูลประมาณ log(n) ครั้ง



**จบบทที่ 17**