การวิเคราะห์โปรแกรม (Analysis of algorithm)

คพ.372/213 โครงสร้างข้อมูล

เยาวดี เต็มธนาภัทร์

Edited by: วิรัตน์ จารีวงศ์ไพบูลย์ และ ฐาปนา บุญชู

หัวข้อในวันนี้

- การวิเคราะห์อัลกอริทึม
- อัลกอริทึมสำหรับการค้นหา (Searching)
- อัลกอริทึมอย่างง่ายสำหรับการเรียงลำดับ (Sorting)

โปรแกรม

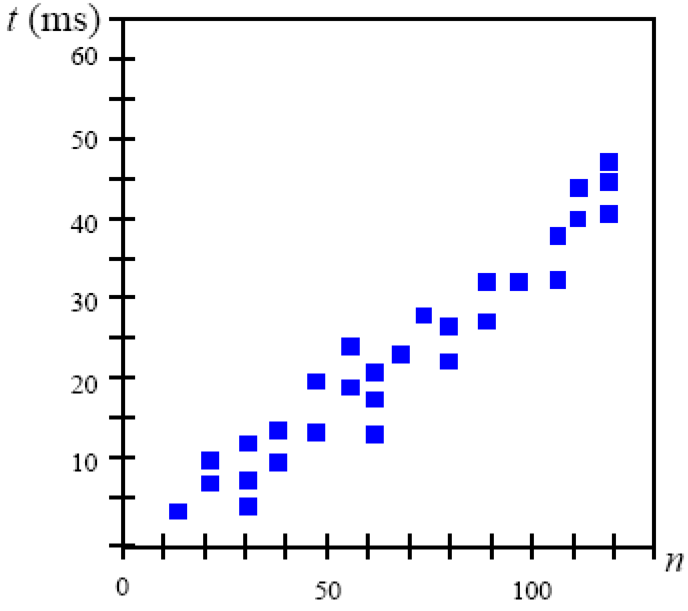
- โปรแกรมทั่วไปประกอบด้วย
  - ***Algorithm*** กลุ่มของขั้นตอนหรือ procedures การทำงาน
    - ขั้นตอนการทำงาน ประกอบจากกลุ่มคำสั่งโปรแกรม
  - ***โครงสร้างข้อมูล (Data structure)*** ใช้ในการเก็บข้อมูลเพื่อให้สามารถทำงานตามขั้นตอนการทำงานหรือ algorithm ที่กำหนด
- ตัวอย่างเช่น การค้นหาข้อมูล
  - **โครงสร้างที่ใช้เก็บเก็บชุดข้อมูล**: อาร์เรย์ หรือ linked list
  - **Algorithm**:
    - *Linear search algorithm*
    - *Binary search algorithm*

การวัดประสิทธิภาพโปรแกรม (1)

- เราวัดประสิทธิภาพโปรแกรมอย่างไร?
  - **ด้านเวลาการทำงาน (time):**
    - ทำงานได้เร็ว (running fast) → ***Time Complexity***
  - **ด้านพื้นที่ที่ใช้ในการทำงาน (memory space):**
    - ใช้พื้นที่ในการทำงานน้อย (using small spaces) → ***Space Complexity***
- โปรแกรมที่ดีควรใช้ทรัพยากร (resource) ของระบบน้อย แต่ทำงานได้รวดเร็ว
  - ในบ่อยครั้งที่พบว่า ต้อง tradeoff ระหว่างพื้นที่ที่ใช้กับความเร็วในการทำงาน

การวัดประสิทธิภาพโปรแกรม (2)

- วัดประสิทธิภาพด้านเวลาอย่างไร?
  - โดยการทดลอง (Experimental Study)
    - เขียนโปรแกรมที่ implements algorithm นั้น
    - ทดลอง run โปรแกรมกับชุดข้อมูลที่มีลักษณะและขนาดต่าง ๆ กัน
    - จับเวลาการทำงานเมื่อเริ่มต้น จนโปรแกรมทำงาน
    - Plot กราฟแสดงประสิทธิภาพด้านเวลา



การวัดประสิทธิภาพโปรแกรม (3)

- **ข้อจำกัดของ Experimental study**
  - ต้อง implement แล้วจึงจะทดสอบ algorithm ในด้านเวลาได้
  - การทดสอบอาจทำได้กับเพียงข้อมูลนำเข้าที่จำกัด ซึ่งอาจไม่สะท้อนค่าความเร็วเมื่อไปใช้กับข้อมูลนำเข้าชุดอื่น ๆ
  - ในการเปรียบเทียบ 2 algorithms ต้องใช้สิ่งแวดล้อมทั้งด้าน hardware และ software ที่เหมือนกัน
- **ต้องการวิธีการวิเคราะห์ประสิทธิภาพของ algorithm**
  - วิเคราะห์ได้แม้อยู่ในรูป high-level description โดยไม่จำเป็นต้องมี implementation ที่เสร็จแล้ว
  - สามารถใช้พิจารณากับชุดข้อมูลนำเข้าแบบต่าง ๆ ที่เป็นไปได้
  - ต้องไม่ขึ้นกับ hardware และ software environment

การวิเคราะห์อัลกอริทึม (Analysis of Algorithm) (1)

- โดยการนับจำนวน primitive operation ที่ต้องถูก executed โดย algorithm
  - ตัวอย่างเช่น:

```
Algorithm findMax(A, n):
Input: An array A storing n integers.
Output: The maximum element in A.
currentMax = A[0]                         1 op
for i = 1 to n -1 do                      1 op
   if currentMax < A[i] then              1 op    n-1 rounds
      currentMax = A[i]                   1 op
return currentMax                         1 op
```

1+3(n-1)+1

การวิเคราะห์อัลกอริทึม (Analysis of Algorithm) (2)

- ตัวอย่างการวัดเมื่อ input มีขนาดต่าง ๆ กัน
  - Algorithm1:
    - ข้อมูลนำเข้ามีสมาชิก 1 ตัว ใช้เวลา 1 μs.
    - ข้อมูลนำเข้ามีสมาชิก 10 ตัว ใช้เวลา 10 μs.
    - ข้อมูลนำเข้ามีสมาชิก 100 ตัว ใช้เวลา 100 μs.
    - ข้อมูลนำเข้ามีสมาชิก 10,000 ตัว ใช้เวลา 10,000 μs.
    - ข้อมูลนำเข้ามีสมาชิก n ตัว ใช้เวลา **n** μs.
  - Algorithm2:
    - ข้อมูลนำเข้ามีสมาชิก 1 ตัว ใช้เวลา 1 μs.
    - ข้อมูลนำเข้ามีสมาชิก 10 ตัว ใช้เวลา 100 μs.
    - ข้อมูลนำเข้ามีสมาชิก 100 ตัว ใช้เวลา 10,000 μs.
    - ข้อมูลนำเข้ามีสมาชิก 10,000 ตัว ใช้เวลา 10^8 μs.
    - ข้อมูลนำเข้ามีสมาชิก n ตัว ใช้เวลา $\mathbf{n^2}$ μs.

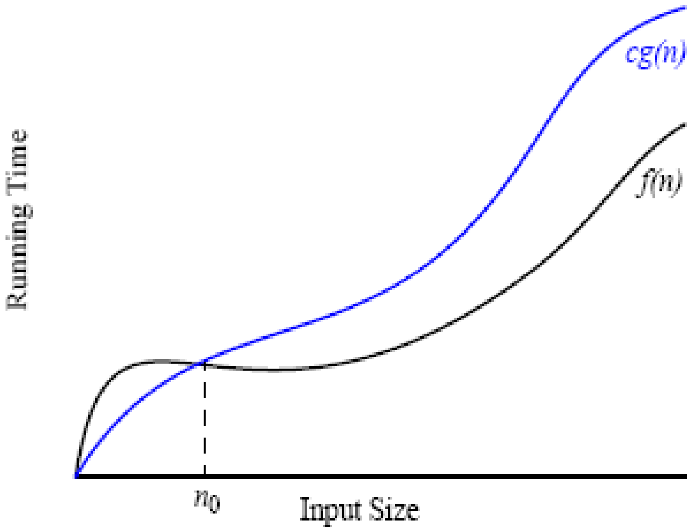
Asymptotic notation

- **จุดมุ่งหมาย:** เพื่อให้การวิเคราะห์เป็นไปได้ง่าย เรากำจัดรายละเอียดปลีกย่อยที่ไม่จำเป็น
  - เช่นการปัดเศษ: 1,000,010 → 1,000,000
- เมื่อค่าของ n มีขนาดใหญ่มาก ๆ ค่าส่วนใหญ่จะขึ้นกับค่าตัวเลขที่ยกกำลังสูงสุด ดังนั้น
  - 1,002,320 → 10^6, $4n^3 + 2n^2 + 3 \rightarrow n^3$

"Big-Oh" Notation

- กำหนดให้ functions f(n) และ g(n), เรากล่าวว่า
  f(n) = O(g(n)) if and only if $f(n) \le c \times g(n)$ สำหรับ $n > n_0$
  - เมื่อ c และ n_0 เป็นค่าคงที่, f(n) และ g(n) เป็นfunctions บน I^+

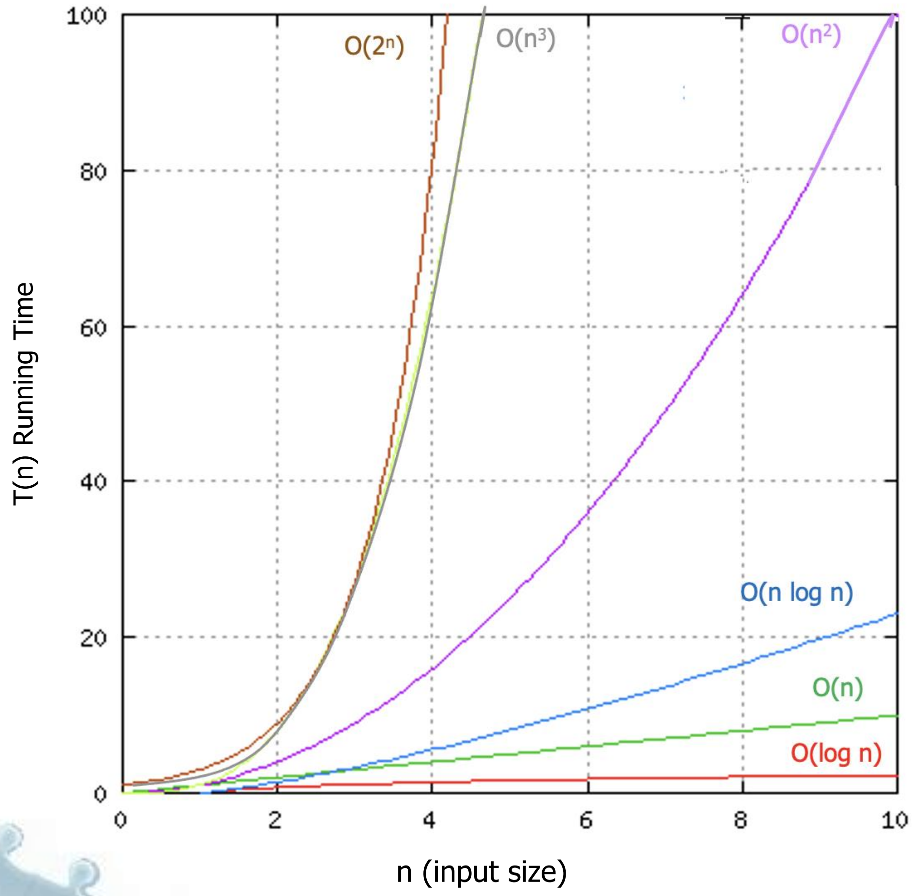
f(n) = O(g(n)) $\leftrightarrow$ **f(n) ≤ c** $\times$ **g(n)**
สำหรับ $n > n_0$



Big Oh

- ข้อสังเกต Big Oh เป็นค่าบ่งบอก *upper bound* หรือ *worse case* ของ algorithm
  - ถึงแม้ $6n - 3 = O(n^5)$ แต่ค่าประมาณนี้ควรใกล้เคียง (เล็กที่สุดเท่าที่เป็นไปได้)
- **กฎอย่างง่าย**คือ ทิ้งค่าคงที่ และค่าที่ยกกำลังต่ำ ๆ ไป เช่น
  - $6n - 3 = O(n)$
  - $8n^2\log n + 5n^2 + n = O(n^2\log n)$

กราฟเปรียบเทียบ input size และเวลา



constant	O(c)
logarithmic	O(log n)
linear	O(n)
linear log	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(2^n)$

Example:

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

$O(1)$	$T(n) = 1$ second
$O(\log_2 n)$	$T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds
$O(n)$	$T(n) = (1*16) / 8 = 2$ seconds
$O(n*\log_2 n)$	$T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds
$O(n^2)$	$T(n) = (1*16^2) / 8^2 = 4$ seconds
$O(n^3)$	$T(n) = (1*16^3) / 8^3 = 8$ seconds
$O(2^n)$	$T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

การคำนวณ Running Time T(n)

ตัวอย่างของส่วนของโปรแกรมที่คำนวณค่า $\sum_{i=1}^{n} i^3$

```
int sum(int n)
{
    int i, partial_sum;

    partial_sum = 0;
    for(i=0;i<n;i++)
        partial_sum += i*i*i;
    return(partial_sum);
}
```

T(n) = 7n+2

- จะใช้การคำนวณจำนวนคำสั่งที่ต้องทำงานแทนการจับเวลาจริง

กฎทั่วไปเพื่อประมาณค่า Big-oh

- **For Loop**
  - จำนวนคำสั่งใน for loop คูณด้วยจำนวนครั้งของการวน loop

```
i = 1;
sum = 0;
while (i <= n)
{
    i = i + 1;
    sum = sum + i;
}
```

```
for(i = 1, sum = 0;i <= n; i++)
{
    sum = sum + i;
}
```

$T(n) = c\,n \quad \Rightarrow O(n)$

กฎทั่วไปเพื่อประมาณค่า Big-oh

- **Nested For Loop**
  - จำนวนคำสั่งใน for loop ด้านใน คูณจำนวนรอบของ loop ในคูณกับจำนวนรอบของ loop นอก

```
i=1;
sum = 0;
while (i <= n) {
    j=1;
    while (j <= n) {
        sum = sum + i;
        j = j + 1;
    }
    i = i + 1;
}
```

```
for(i = 1, sum = 0;i <= n; i++)
{
   for(j = 1; j <= n; j++)
        sum = sum + i;
}
```

$$Tn = cn^2 \Rightarrow O(n^2)$$

กฎทั่วไปเพื่อประมาณค่า Big-oh

- **Consecutive Statement**
  - จำนวนคำสั่งของ statement มารวมกัน

```
for (i=0; i < n; i++)
     a[i] = 0;
```

$\rightarrow O(n)$

```
  for (i=0; i< n; i++)
          for (j=0; j < n; j++)
              a[i] += a[j] + i + j;
```

$\rightarrow O(n^2)$

$O(n^2) + O(n)$

$\downarrow$

$\boxed{O(n^2)}$

```
count = count + 1;
sum = sum + count;
```

$\rightarrow \boxed{T(n) = c1+c2 => O(c)}$

กฎทั่วไปเพื่อประมาณค่า Big-oh

- **IF/Else**
  - นับคำสั่งที่ใช้ทดสอบรวมกับจำนวนคำสั่งที่มากกว่าระหว่าง if หรือ else

```
if (condition)
        S1
else

        S2
```

```
if (test >= 5)                       c1
    count = count + 1;               c2
else {
    count = count + 5;
    sum = sum + count;               c3
    count = 0;
}
```

$$T(n) = c1 + \max(c2,c3)$$

ชนิดของ Loop และค่า Big-oh

- Linear Loop
- Logarithmic Loop
- Linear Logarithmic Loop
- Quadratic Loop

Linear Loop :

```
i   =  1;
while ( i <= n  )
{
    //application code
    i = i + 1;
}
```

$$T(n) = n \quad => O(n)$$

```
i   =  1;
while ( i <= n  )
{
    //application code
    i = i + 2;
}
```

$$T(n) = \frac{n}{2} \quad => O(n)$$

Logarithmic Loop :

A loop in which the controlling variable (i) is multiplied or divided

multiplication

```
i   =  1;
while( i < n  )
{
  //application code

   i = i * 2;
}
```

```
i  = 1,  2,  4,  8,  16, .... N
```

division

```
i   =  n;
while ( i > 1  )
{
  //application code

     i = i / 2;
}
```

```
i  = 1000,  500,  250, ....
```

$$O\ (\log_2 n)$$

Linear Logarithmic Loop :

```
i   =  1;
while  ( i <= n  )
{
    j  =  1;

    while ( j <= n )
    {
        //application code

        j  = j * 2;
    }
    i  =  i  + 1;
}
```

n times

$\log_2 n$

$$\boxed{O(n \log_2 n)}$$

Quadratic Loop :

```
i   =  1;
while  ( i   <=   n  )
{
   j  =  1;
   while (   j  <=   n  )
   {
       //application code

       j = j + 1;
   }
   i  =  i  + 1;
}
```

n times

n times

$O(n^2)$

สรุปขั้นตอนการคำนวณ Big O

ขั้นตอนการพิจารณาหาค่า Big O

1. แบ่ง algorithm/function เป็น operations เดี่ยว
2. คำนวณ Big O ของแต่ละ operation
3. รวมค่า Big O ของ operation ทั้งหมดเข้าด้วยกัน
4. ไม่พิจารณาค่าคงที่ (ลบค่าคงที่)
5. ค่าเทอมที่มี order สูงสุด เป็นค่า Big O ของ algorithm/function

Time Complexity ของการค้นหา (1)

- **Time Complexity** ของ *Binary Search algorithm*
  - จำนวนการเปรียบเทียบสูงสุดในกรณีแย่ที่สุด (worst case) ซึ่งต้องทำในการค้นหา
  - ใน searched array ถูกแบ่งครึ่ง เพื่อเลือกเอาสมาชิกที่จะนำไปเปรียบเทียบค่ากับ search key
- ดังนั้น จำนวนการเปรียบเทียบสูงสุดที่วัดได้สำหรับอาร์เรย์ที่มีสมาชิก n ตัว
  - **$O(\log_2(n))$** เมื่อ n เป็นขนาดของอาร์เรย์
- ตัวอย่างเช่น:
  - ถ้าให้ sorted array มีสมาชิกจำนวน 1024 ตัว จำนวนสูงสุดที่ต้องดำเนินการเปรียบเทียบจะเป็น $\log_2(1024) = 10$ (นั่นคือเพียง 10 ครั้งก็พอเพียงในการค้นหา)

Time Complexity ของการค้นหา (2)

- ในทำนองเดียวกัน หากเปรียบเทียบกับ **Computational Complexity** ของ *Linear Search*
  - พบว่าจำนวนการเปรียบเทียบในกรณีที่แย่ที่สุด (worst case) จะเท่ากับ: **O(n)** (เมื่อ n เป็นขนาดของอาร์เรย์)
- ตัวอย่างเช่น:
  - ถ้าให้อาร์เรย์มีสมาชิก 1024 ตัว เราต้องเปรียบเทียบ (สูงสุด) ทั้งหมด: n = 1024 ครั้ง

สรุปการวิเคราะห์อัลกอริทึม: Asymptotic

- แนวคิด:
  - การวิเคราะห์อัลกอริทึม เพื่อให้สามารถประมาณหรือคำนวณหาความเร็วในการทำงานได้อย่างง่าย ๆ โดยการพิจารณาลักษณะของอัตราการเติบโตสัมพัทธ์ (Relative rate of growth) เมื่อจำนวนของข้อมูลมีขนาดโตขึ้น
- ข้อสมมติของ Asymptotic Analysis
  - จำนวนของข้อมูล input มีขนาดใหญ่
  - ไม่คิดปัจจัยที่เป็นค่าคงที่

Searching (การค้นหา)

- กระบวนการในการหาสมาชิกที่ต้องการ
- 2 เทคนิคพื้นฐานในการค้นหา:
  - Linear search
  - Binary search.

Linear search

- ค้นหาค่าข้อมูลแหล่งเก็บโดยระบุ แหล่งเก็บข้อมูล เช่น อาร์เรย์ และ *ค่าที่ต้องการค้น (search key)*
  - ถ้า*ค่าที่ต้องการค้น* ถูกค้นพบในแหล่งเก็บ คืนค่าตำแหน่งที่พบ ถ้าไม่เช่นนั้นคืนค่าพิเศษเพื่อบ่งบอกการไม่พบ

ตัวอย่างโปรแกรม Linear Search

```
int LinearSearch (int b[ ], int skey)
{
   int i;
   for (i=0; i<= SIZE-1; i++) {
       if (b[i] == skey)
              return i;
   }
   return -1;
}
```

- Big-oh: ????

- ถ้าอาร์เรย์มีข้อมูลเรียงลำดับ เราสามารถปรับปรุงให้ฟังก์ชัน Linearsearch ทำงานได้มีประสิทธิภาพที่ดีขึ้นได้อย่างไร?

Binary Search

- สมมติให้อาร์เรย์ข้อมูลมีค่าเรียงลำดับ เราสามารถใช้ Binary Search algorithm เพื่อช่วยให้สามารถค้นหาได้รวดเร็วขึ้น
- เช่น ค้นหา search key ในอาร์เรย์

  (1, 4, 5, 7, 9, 15, 22, 45, 78, 96)

ตัวอย่างโปรแกรม Binary Search (2)

```
int BinarySearch (int A[], int skey, int low, int high)
{
   int middle;

   while(low <= high){
       middle = (low + high)/2;
       if(skey == A[middle])
              return middle;
       else if(skey < A[middle])
              high = middle-1;
       else
              low = middle+1;
   }
   return -1;
}
```

- Big-oh: ????

สมมติต้องการหาค่า x

Step 1

อัลกอริทึมนำค่า **x** เปรียบเทียบกับค่าที่อยู่ในตำแหน่งตรงกลางของอาร์เรย์

Step 2

ถ้า ค่า **x** พบที่ตำแหน่งนี้ ยุติการค้นหาคืนค่าตำแหน่งที่พบ

ถ้า ค่า **x** เป็นค่าที่อยู่ก่อนหน้าค่าที่ตำแหน่งกลางของอาร์เรย์

ค้นหาต่อเหมือนกับใน Step 1 แต่ค้นกับเฉพาะส่วนของอาร์เรย์ที่อยู่ *ก่อนหน้า*ค่าตำแหน่งกลาง (อาร์เรย์ที่ใช้ค้นหามีขนาดเล็กลง)

ถ้า ค่า **x** เป็นค่าที่อยู่หลังจากที่ตำแหน่งกลางของอาร์เรย์

ค้นหาต่อเหมือนกับใน Step 1 แต่ค้นกับเฉพาะส่วนของอาร์เรย์ที่อยู่ *หลังจาก*ค่าตำแหน่งกลาง (อาร์เรย์ที่ใช้ค้นหามีขนาดเล็กลง)

Stop เมื่อพบสมาชิกที่มีค่า **17** หรือพบว่าไม่ปรากฏสมาชิกที่มีค่า 17 ในอาร์เรย์

แต่ละขั้นตอน ทำให้ขนาดของส่วนอาร์เรย์ที่ต้องค้นหาเล็กลงครึ่งหนึ่งเสมอ

ให้ค้นหาค่า **78** ในอาร์เรย์ตัวอย่าง:

0	1
1	4
2	5
3	7
4	9
5	15
6	22
7	45
8	78
9	96

Middle position:

$$\left\lfloor \frac{0+9}{2} \right\rfloor = \left\lfloor 4\frac{1}{2} \right\rfloor = 4$$

value 9

78 เป็นค่าที่อยู่หลังค่า 9 ไม่พิจารณา อาร์เรย์ลำดับที่ 0-4

ทำให้เหลือเพียงอาร์เรย์ตำแหน่งที่ 5 ถึง 9

Middle position :

$$\left\lfloor \frac{5+9}{2} \right\rfloor = 7$$

value 45

78 เป็นค่าที่อยู่หลังค่า 45 ไม่พิจารณา อาร์เรย์ลำดับที่ 5-7

ทำให้เหลือเพียงอาร์เรย์ตำแหน่งที่ 8 ถึง 9

Middle position :

value 78

$$\left\lfloor \frac{8+9}{2} \right\rfloor = \left\lfloor 8\frac{1}{2} \right\rfloor = 8$$

พบสมาชิกที่ต้องการ!

ให้ค้นหาค่า **6** ในอาร์เรย์ตัวอย่าง:

0	1
1	4
2	5
3	7
4	9
5	15
6	22
7	45
8	78
9	96

Middle position:

$$\left\lfloor \frac{0+9}{2} \right\rfloor = \left\lfloor 4\frac{1}{2} \right\rfloor = 4$$

value 9

6 เป็นค่าที่อยู่ก่อนค่า 9 — ไม่พิจารณา อาร์เรย์ลำดับที่ 4-9

ทำให้เหลือเพียงอาร์เรย์ตำแหน่งที่ 0 ถึง 3

Middle position :

$$\left\lfloor \frac{0+3}{2} \right\rfloor = \left\lfloor 1\frac{1}{2} \right\rfloor = 1$$

value 4

6 เป็นค่าที่อยู่หลังค่า 4 — ไม่พิจารณา อาร์เรย์ลำดับที่ 0-1

ทำให้เหลือเพียงอาร์เรย์ตำแหน่งที่ 2 ถึง 3

Middle position :

$$\left\lfloor \frac{2+3}{2} \right\rfloor = \left\lfloor 2\frac{1}{2} \right\rfloor = 2$$

value 5

6 เป็นค่าที่อยู่หลังค่า 5 — ไม่พิจารณา อาร์เรย์ลำดับที่ 2-2

ทำให้เหลือเพียงอาร์เรย์ตำแหน่งที่ 3 ถึง 3

6 เป็นค่าที่อยู่ก่อนค่า 7 — ไม่พิจารณา อาร์เรย์ลำดับที่ 3-3

ไม่พบสมาชิกที่ต้องการ!

การเรียงลำดับข้อมูล

คพ.372/213 โครงสร้างข้อมูล

เยาวดี เต็มธนาภัทร์

Edited by: วิรัตน์ จารีวงศ์ไพบูลย์ และ ฐาปนา บุญชู

หัวข้อ

- การเรียงลำดับข้อมูล (Sorting)
  - ความหมาย
  - การเรียงลำดับแบบ Selection
  - การเรียงลำดับแบบ Bubble
  - การเรียงลำดับแบบ Insertion

การเรียงลำดับข้อมูล (Sorting)

- **การเรียงลำดับข้อมูล**
  - การจัดเรียงข้อมูลใหม่ให้อยู่ในลำดับมากไปน้อย หรือน้อยไปมาก
  - Sort นับเป็นการดำเนินการที่ถูกใช้งานมากที่สุด ในการแก้ปัญหาทางคอมพิวเตอร์
  - algorithms สำหรับ Sort มีอยู่หลากหลาย โดยบาง algorithm ทำงานได้เก่งกว่าบาง algorithm
  - เวลาที่ใช้ในการทำงานของ algorithm ทั่วไปแตกต่างกัน โดยจะอยู่ระหว่าง $O(n \log n)$ และ $O(n^2)$

วิธีการเรียงลำดับข้อมูล

- Sorting algorithms พื้นฐานที่นิยมใช้มี 5 แบบ:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort (จะพูดถึงอีกครั้งในเรื่อง recursion)
  - Quick Sort (จะพูดถึงอีกครั้งในเรื่อง recursion)
- ตัวอย่าง algorithms ทั้งหมดเรียงผลลัพธ์จากน้อยไปมาก

Bubble Sort

- หนึ่งใน sorting algorithm ที่ง่ายที่สุดแต่ก็แย่ที่สุด
  - เรียนรู้เพราะเป็น algorithm ที่ง่ายที่จะเข้าใจและวิเคราะห์
- Bubble Sort ใช้ 2 operations พื้นฐานคือ
  - **การเปรียบเทียบ**
    - หลักการ: พิจารณาสมาชิกคู่ที่อยู่ติดกันใน list ตัวใดควรอยู่ก่อน?
  - **การสลับ (Swap)**
    - กำหนดสมาชิกคู่ที่อยู่ติดกันใน list สลับที่สมาชิกคู่นั้น
- หมายเหตุ Bubble Sort ทำ operations ข้างต้นนี้กับ items ทีละคู่

Bubble Sort

- การทำงานของ bubble sort ในแต่ละรอบ (pass)
  - พิจารณา สมาชิกทีละคู่ เปรียบเทียบกัน ถ้าสมาชิกคู่นั้นอยู่สลับตำแหน่ง ให้สลับที่สมาชิกคู่นั้น
  - หลังผ่านรอบหนึ่ง สมาชิกที่มีค่าเล็กที่สุดจะเลื่อน (ลอย) ไปอยู่ที่ตำแหน่งแรกสุด
  - ดังนั้นในรอบถัดไปจึงไม่จำเป็นต้องเปรียบเทียบทั้ง list

ตัวอย่างเช่น รอบแรก

	42	**20**	**17**	**13**	**28**	**14**	**23 ↔**	**15**
	42	**20**	**17**	**13**	**28**	**14 ↔**	**15**	**23**
	42	**20**	**17**	**13**	**28 ↔**	**14**	**15**	**23**
	42	**20**	**17**	**13 ↔**	**14**	**28**	**15**	**23**
	42	**20**	**17 ↔**	**13**	**14**	**28**	**15**	**23**
	42	**20 ↔**	**13**	**17**	**14**	**28**	**15**	**23**
	42 ↔	**13**	**20**	**17**	**14**	**28**	**15**	**23**
	13 |	**42**	**20**	**17**	**14**	**28**	**15**	**23**
รอบ 2	**13 |**	**42**	**20**	**17**	**14**	**28**	**15 ↔**	**23**

:

```
bubble_sort(int a[], int N)
   /* sort a[0..N-1] */
{
   int i, j;
   /* some checking for special conditions (omitted)*/

   for(i=0; i < N-1; i++)
   {
      for(j=N-1; j > i; j--) {
         if (a[j] < a[j-1])
             swap(a[j-1], a[j]);
      }
   }
}/*bubble_sort*/
```

วิเคราะห์การเรียงลำดับแบบ Bubble Sort

- ประสิทธิภาพ ☹
  - Time Complexity เป็น $O(n^2)$
- ความต้องการในชุดข้อมูล ☺
  - Bubble Sort ไม่จำเป็นต้องใช้การเข้าถึงสมาชิกแบบสุ่ม (random-access) และยังทำงานได้กับชุดข้อมูลแบบ Linked list
  - Operations ที่จำเป็น: การเปรียบเทียบ, การสลับที่ (Swap) ของสมาชิกคู่ที่อยู่ติดกัน
- พื้นที่ที่ต้องการ (Space Usage) ☺
  - Bubble Sort สามารถทำงานได้แบบ in-place
- Stability ☺
  - Bubble Sort เป็น stable
- ประสิทธิภาพในกรณีที่ชุดข้อมูลเกือบเรียงลำดับแล้ว ☺/☹
  - สามารถเขียน Bubble Sort ที่ทำงานใน $O(n)$ ถ้าหากไม่มีสมาชิกที่อยู่ผิดตำแหน่ง ☺
  - Bubble Sort ใช้เวลา $O(n^2)$ แม้ในกรณีที่มีเพียงสมาชิกเดียวที่อยู่ผิดตำแหน่ง ☹

Selection Sort

- Selection Sort หนึ่งใน sorting algorithm ที่ง่าย แต่ไม่มีประสิทธิภาพ
- Selection Sort ใช้ swap operation จำนวนน้อยกว่า Bubble แต่ยังคงใช้การเปรียบเทียบจำนวนมากอยู่ดี
- ดังนั้นจึงไม่ดีกว่า Bubble Sort อย่างเด่นชัด *แต่*กลับมีข้อด้อย
  - ทำงานไม่ได้รวดเร็วแม้ในกรณีที่ชุดข้อมูลเกือบเรียงลำดับ
  - ให้ผลลัพธ์ที่ไม่ Stable

Selection Sort

- แทนการไล่ฟองให้ลอยขึ้น (Bubble) Selection sort
  - ใช้ค้นหาสมาชิกที่เล็กที่สุด แล้วทำ swap ระหว่างสมาชิกนั้นกับสมาชิกตัวแรก
  - หลังผ่านรอบหนึ่ง สมาชิกที่มีค่าเล็กที่สุดจะเปลี่ยนไปอยู่ที่ตำแหน่งแรกสุด
  - ดังนั้นในรอบถัดไปจึงไม่จำเป็นต้องเปรียบเทียบทั้ง list

ตัวอย่างเช่น

	|42	20	17	13	28	14	23	15
i= 1	13 |	20	17	42	28	14	23	15
i= 2	13	14|	17	42	28	20	23	15
i= 3	13	14	15|	42	28	20	23	17
i= 4	13	14	15	17|	28	20	23	42
i= 5	13	14	15	17	20|	28	23	42
i= 6	13	14	15	17	20	23|	28	42
i= 7	13	14	15	17	20	23	28|	42

```
selection_sort(int a[], int N)
   /* sort a[0..N-1] */
{
   int i, j, min;
   /* some checking for special conditions (omitted)*/

   for(i=0; i < N-1; i++)
   {
      min = i;
      for(j=i+1; j < N; j++) {
         if (a[j] < a[min]) {
            min = j;
         }
      }
      swap(a[i], a[min]);
   }
}/*selection_sort*/
```

วิเคราะห์การเรียงลำดับแบบ Selection Sort

- ประสิทธิภาพ ☹
  - ใช้ Time Complexity เป็น $O(n^2)$
- ความต้องการในชุดข้อมูล ☺
  - Selection Sort ไม่จำเป็นต้องใช้การเข้าถึงสมาชิกแบบสุ่ม (random-access) และยังทำงานได้กับชุดข้อมูลแบบ Linked list
  - Operations: การเปรียบเทียบ, การสลับที่ (Swap) ของสมาชิกคู่ค่าน้อยสุดกับตัวที่ต้องเรียงถัดไป
- พื้นที่ที่ต้องการ (Space Usage) ☺
  - Selection Sort สามารถทำงานได้แบบ in-place
- Stability ☹
  - Selection Sort **ไม่**เป็น stable!
- ประสิทธิภาพในกรณีที่ชุดข้อมูลเกือบเรียงลำดับแล้ว ☹
  - ไม่ได้ทำงานเร็วขึ้นมากนัก สำหรับกรณีที่มีบางสมาชิกอยู่ผิดตำแหน่ง ☹

Insertion Sort

- เพิ่มทีละสมาชิกเข้าไปในชุดข้อมูลที่เรียงลำดับ (Sorted List) แล้ว
- เริ่มต้นจาก sorted list ที่มีเพียง 1 สมาชิก
- จากนั้นเพิ่มสมาชิกตัวถัดไปเข้าไปในชุดข้อมูลในตำแหน่งที่เหมาะสม

ตัวอย่างเช่น

	42 |	20	17	13	28	14	23	15
i= 1	20	42 |	17	13	28	14	23	15
i = 2	17	20	42 |	13	28	14	23	15
i = 3	13	17	20	42 |	28	14	23	15
i = 4	13	17	20	28	42 |	14	23	15
i = 5	13	14	17	20	28	42 |	23	15
i = 6	13	14	17	20	23	28	42 |	15
i = 7	13	14	15	17	20	23	28	42

```
insertion_sort(int a[], int N)
   /* sort a[0..N-1] */
{
   int i, j, nextElement;
   /* some checking for special conditions (omitted)*/

   for(i=1; i < N; i++)
   {
      nextElement = a[i]; /* remember the next element */

      /* find appropriate position for the next element */
      for(j=i-1; j >= 0 && a[j] > nextElement; j--) {
         a[j+1] = a[j];
      }

      a[j+1] = nextElement;
   }

}/*insertion_sort*/
```

วิเคราะห์การเรียงลำดับแบบ Insertion

- ประสิทธิภาพ: ☹
  - ใช้ Time Complexity เป็น $O(n^2)$
  - กรณีแย่สุดเกิดเมื่อชุดข้อมูลเรียงลำดับกลับด้าน (reverse order)
- ความต้องการในชุดข้อมูล ☺
  - Insertion Sort ไม่จำเป็นต้องใช้การเข้าถึงสมาชิกแบบสุ่ม (random-access)
  - Operations ที่จำเป็น: การหาใน sorted list, การลบ (remove) และ insert ลงใน sorted list
- พื้นที่ที่ต้องการ (Space Usage) ☺
  - Insertion Sort สามารถทำงานได้แบบ in-place
- Stability ☺
  - Insertion Sort เป็น stable
- ประสิทธิภาพเมื่อชุดข้อมูลเข้า เกือบเรียงลำดับ ☺
  - Insertion Sort สามารถทำงานได้ใน $O(n)$ ถ้ามีสมาชิกไม่กี่ตัวที่ไม่อยู่ในลำดับ
  - Insertion Sort สามารถทำงานได้ใน $O(n)$ ถ้าตำแหน่งของสมาชิกมีระยะห่างจากที่ที่ควรอยู่ไม่ไกลกว่าค่าคงที่หนึ่ง

การวิเคราะห์ Insertion Sort เพิ่มเติม

- จาก sorting algorithm ที่เราดูมาทั้งหมด Insertion Sort เป็น algorithm เป็นอันแรกที่มีประโยชน์
- Insertion Sort อาจไม่ดีสำหรับกรณีทั่วไปทั้งหมด แต่ algorithm นี้มีประสิทธิภาพ สำหรับกรณีที่ชุดข้อมูลเกือบเรียงลำดับหมดแล้ว ดังนั้นจึงมีประโยชน์:
  - สำหรับกรณีชุดข้อมูลขนาดเล็ก
    - "เล็ก" นั้นขึ้นอยู่กับระบบและแอพพลิเคชัน (ตัวอย่างเช่น กรณีที่ชุดข้อมูลมีขนาด น้อยกว่า 32 items
    - สำหรับกรณีที่ข้อมูลเกือบเรียงลำดับแล้ว หรือ กรณีที่ items ทั้งหมดอยู่ในตำแหน่งใกล้เคียงกับตำแหน่งที่มันควรอยู่อยู่แล้ว หรือมีเพียงไม่กี่ item ที่อยู่ผิดที่
    - ทั้ง 2 กรณีนี้จะใช้เวลาในการเรียงลำดับเป็นแบบ linear