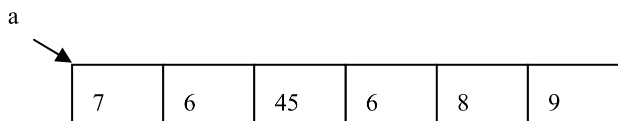
บทที่ 2

อาร์เรย์และการจัดเรียงข้อมูล

บทนี้เราจะมาเรียนโครงสร้างข้อมูลที่ใช้กันบ่อยที่สุด นั่นคือ อาร์เรย์ (array) อาร์เรย์ก็คือ ลิสต์ของข้อมูลชนิดเดียวกัน ข้อมูลแต่ละตัวในอาร์เรย์นี้จะหาได้จากการใช้ index (หรือที่เรียกว่า subscript) รูปที่ 2.1 เป็นตัวอย่างของอาร์เรย์ a ซึ่งเก็บจำนวนเต็มไว้หกจำนวน



รูป 2.1 อาร์เรย์ของจำนวนเต็ม

a คือตัวแปรที่เป็นชื่อของอาร์เรย์นี้ ในภาษาจาวานั้น อาร์เรย์เป็น object ชนิดหนึ่ง เพราะฉะนั้น a ในที่นี้คือ object reference นั่นเอง พูดง่ายๆคือ a เป็นตัวลูกศรที่จะชี้ไปยังส่วนที่เก็บค่าต่างๆของอาร์เรย์นั่นเอง

เราใช้ a[index] อ่านตำแหน่งต่างๆของอาร์เรย์ จากรูปที่ 2.1 a[0] จะหมายถึง 7 และ a[1] จะหมายถึง 6 ค่า index จะเริ่มต้นที่ 0 (สำหรับภาษาจาวา) และทุกค่า index จะต้องเป็นจำนวนเต็มเสมอ รวมทั้งเป็น expression ที่คำนวณแล้วได้จำนวนเต็มก็ได้ เช่น ถ้า b=2 กับ c=3 แล้ว a[b+c] ถือว่าเป็นการกล่าวถึงสมาชิกของอาร์เรย์ได้ถูกต้อง ตัว a[b+c] นี้ก็สามารถถูกนำไปใช้ได้เหมือนตัวแปรธรรมดา เช่น a[b+c] += 5;

สำหรับการหาขนาดของอาร์เรย์ (จำนวนช่องที่เก็บค่า) นั้น ในภาษาสมัยใหม่ทำได้ง่ายมาก ในจาวาเอง ทุกๆ อาร์เรย์ จะมีตัวแปร length ที่จะถูกตั้งค่าไว้โดยอัตโนมัติตอนที่อาร์เรย์ถูกสร้างขึ้น เราสามารถใช้ a.length หาขนาดของอาร์เรย์ a ได้ทันที

การสร้างอาร์เรย์ในภาษาจาวา

การสร้างอาร์เรย์ขึ้นมาใช้งานนั้นประกอบด้วยสองขั้นตอนใหญ่ๆ คือ

1. Array Declaration: คือการนิยามตัวแปรอาร์เรย์ขึ้นมา
2. Array Allocation: คือการสร้างออบเจ็กต์ที่เป็นอาร์เรย์ขึ้นมาจริงๆ แล้วใช้ assignment operator ทำให้ตัวแปรในข้อ 1 ชี้มายังตัวออบเจ็กต์นี้

ตัวอย่างที่ 2-1

โค้ดที่สร้างอาร์เรย์ของจำนวนเต็มเป็นดังนี้

```
1:    int x[]; //Declaration -นี่คือการนิยามตัวแปร x ให้เป็นอาร์เรย์ของจำนวนเต็ม
2:            // แต่ยังไม่มีออบเจ็กต์จริงๆเกิดขึ้น
3:    x=new int[5]; //Allocation - นี่คือการสร้างอาร์เรย์ออบเจ็กต์ขนาด 5 ช่อง
4:          // เมื่อทำแล้วจะได้อาร์เรย์ที่มีค่าในแต่ละช่องเป็นค่า default ของข้อมูลชนิดนั้น
5:          // ในที่นี้จะมี 0 อยู่ทั้ง 5 ช่อง
```

ในการสร้างออบเจ็กต์นั้น ต้องมีการกำหนดจำนวนช่องของอาร์เรย์ให้แน่นอนด้วย นอกจากนี้ ยังมีอีกวิธีหนึ่งในการทำ Allocation นั่นคือการสร้างอาร์เรย์โดยใช้ initializer list

ตัวอย่างที่ 2-2

เราสามารถสร้างอาร์เรย์ x ของจำนวนเต็ม 1 2 และ 3 ได้ด้วย

```
1:    int x[] = {1,2,3};
```

โดย {1,2,3} คือ initializer list นั่นเอง

อาร์เรย์ของออบเจ็กต์

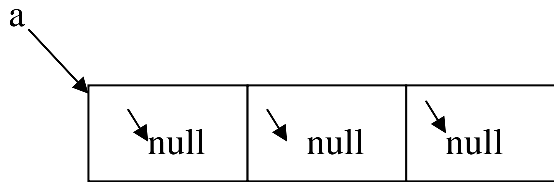
เราสามารถสร้างอาร์เรย์ของออบเจ็กต์ได้โดยใช้หลักเดียวกับการสร้างอาร์เรย์ธรรมดา

ตัวอย่างที่ 2-3

ถ้ามีออบเจ็กต์ type MyObject เราก็สร้างอาร์เรย์ขนาด 3 ช่องของมันได้

```
1:    MyObject a[] = new MyObject[3];
```

โดยตัวอาร์เรย์ที่ถูกสร้างจะมีลักษณะดังรูปที่ 2.2 คือมีค่าเริ่มต้นของสมาชิกเป็น null



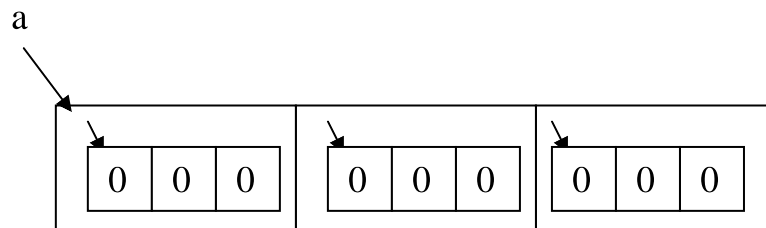
รูป 2.2 อาร์เรย์ของออบเจ็กต์เมื่อถูกสร้าง

อาร์เรย์ของอาร์เรย์

ตัวอย่างที่ 2-4

เราสามารถสร้างอาร์เรย์ที่เก็บอาร์เรย์อื่นไว้ข้างในได้ดังนี้

```
1:    int x[][] = new int[3][3];
```



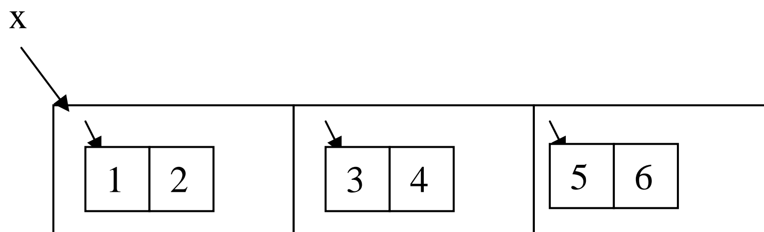
รูป 2.3 อาร์เรย์ของอาร์เรย์ของจำนวนเต็มเมื่อถูกสร้าง

ตัวอย่างที่ 2-5

เราสามารถใช้ initializer list ในการสร้างอาร์เรย์ของอาร์เรย์ได้เช่นเดียวกัน จากโค้ดต่อไปนี้

```
1:    int x[] = {{1,2},{3,4},{5,6}};
```

จะทำให้เกิดอาร์เรย์ดังรูปที่ 2.4



รูป 2.4 อาร์เรย์ของอาร์เรย์ของจำนวนเต็ม สร้างด้วย initializer list

การเข้าถึงข้อมูลต่างๆของอาร์เรย์ที่เก็บอาร์เรย์นั้น ถ้าดูจากรูปที่2.4 เราสามารถเข้าถึงข้อมูลได้ดังนี้

- x[0][0] คือ 1
- x[0][1] คือ 2
- x[0][2] คือ 3

นั่นก็คือค่า index แรก บอกว่าเราควรดูที่ช่องไหนของอาร์เรย์ชั้นนอกสุด ค่า index ที่สองจะบอกว่าเราควรดูช่องไหนของอาร์เรย์ย่อยอีกที ดังนั้นเราสามารถเข้าถึงข้อมูลในอาร์เรย์หลายชั้นได้ไม่ยาก

สำหรับอาร์เรย์ 2 ชั้นนั้นเราสามารถดูให้เป็นเมตริกซ์ได้ โดยให้ index ตัวแรกแทนแถวและตัวที่ 2 แทนหลัก ดังนั้นอาร์เรย์ในรูปที่2.4 จึงสามารถเขียนได้ในรูปของ

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

มีข้อสังเกตว่าในการ allocate อาร์เรย์สำหรับภาษาจาวานั้น เราสามารถเว้นการสร้างค่าเริ่มต้นของบางตัวไว้ได้ เช่น

ตัวอย่างที่ 2-6

```
1:     int y [][] = new int [2][];
```

ตัว index ตัวหลังนั้นเราเว้นไว้ได้ อาร์เรย์ในชั้นที่ 2 ก็จะเป็น null ไปก่อน แต่อย่างไรก็ตาม ต้องมีการสร้างค่าเริ่มต้นของอาร์เรย์ชั้นนี้ให้เรียบร้อยก่อนที่จะมีการนำสมาชิกภายในไปใช้ ในตัวอย่างนี้เราอาจทำการสร้างค่าเริ่มต้นได้ดังนี้

```
2:     y[0] = new int[2];
3:     y[1] = new int[3];
```

จะเห็นได้ว่าอาร์เรย์ย่อยไม่จำเป็นต้องมีขนาดเท่ากัน นี่คือประโยชน์ของการเว้นการสร้างค่าเริ่มต้นในตอนแรกนั่นเอง

แบบฝึกหัด

1. สมมติว่ามีฟังก์ชั่น rand(i,j) ซึ่ง return random number จาก i ถึง j จงเขียนโปรแกรมโดยไม่สร้างโครงสร้างข้อมูลเพิ่มเติม เพื่อสร้างอาร์เรย์ขนาด n ของ random number โดยที่ภายในอาร์เรย์นี้ต้องไม่มีเลขซ้ำกัน นอกจากนี้ จำนวนในอาร์เรย์ต้องมีค่าตั้งแต่ 0 ถึง n-1 ทุกจำนวนด้วย โปรแกรมนี้จะมีค่า big O เป็นเท่าไร
2. จากข้อที่แล้ว ถ้าเรามีอาร์เรย์ของ Boolean เพื่อบอกว่าค่า random number ตัวไหนใช้ไปแล้วบ้าง จงเขียนโปรแกรมนี้ใหม่และบอกว่าค่า big O เป็นเท่าไร
3. จงเขียนโปรแกรมเพื่อหาว่าในอาร์เรย์ของจำนวนเต็มอาร์เรย์หนึ่ง มีเลขที่อยู่ในอาร์เรย์ซ้ำกันเกินครึ่งหนึ่งของอาร์เรย์หรือไม่ บอกค่า big O ของโปรแกรมนี้ด้วย
4. จงเขียนโปรแกรมเพื่อหาว่าในอาร์เรย์ของจำนวนเต็มอาร์เรย์หนึ่ง มีตัวเลขสองตัวซึ่งบวกกันได้ค่ามากที่สุดเป็นเท่าใด พิมพ์ index ของเลขสองจำนวนนี้ออกมาด้วย รวมทั้งบอกค่า big O ของโปรแกรมนี้
5. ถ้ามีอาร์เรย์ของจำนวนเต็มและจำนวนเต็มเหล่านั้นเรียงจากน้อยไปมาก จงเขียนโปรแกรมเพื่อหาว่า มี k ที่ a[k] = = k ซักตัวหรือไม่ โปรแกรมนี้จะมีค่า big O เป็นเท่าไร

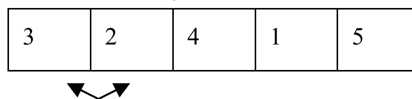
6. ถ้าเรามีอาร์เรย์ซึ่งเก็บจำนวนเต็ม n-1 จำนวน ให้จำนวนเหล่านั้นมีค่าตั้งแต่ 0 ถึง n-1 นั่นคือ จะมีตัวเลขตัวหนึ่งที่ไม่อยู่ในอาร์เรย์นี้ จงเขียนโปรแกรมหาตัวเลขนี้ โดยให้โปรแกรมมี big O เป็น O(n) (สามารถเก็บตัวเลขได้อีกหนึ่งที่นอกจากในอาร์เรย์เท่านั้น)
7. ถ้ามีเมทริกซ์ของจำนวนเต็มและจำนวนเต็มเหล่านั้นเรียงจากน้อยไปมาก ทั้งจากซ้ายไปขวาและบนลงล่าง จงเขียนโปรแกรมเพื่อหาว่ามีค่า x อยู่ในเมทริกซ์นี้หรือไม่ บอกค่า big O ด้วย
8. ถ้ามีเมทริกซ์ของ 0 และ 1 ถ้าในแถวๆหนึ่ง เลข 0 จะต้องอยู่ก่อนเลขหนึ่งเสมอ จงเขียนโปรแกรมที่มี big O เป็น O(n) เพื่อหาว่าแถวไหนมีจำนวน 0 มากที่สุด

การจัดเรียงข้อมูลในอาร์เรย์

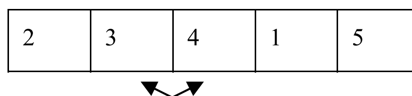
การจัดเรียงข้อมูล (ในที่นี้สมมติให้เป็นจำนวนเต็ม และเราต้องการเรียงจากน้อยไปมาก) ในอาร์เรย์นั้นมีหลายวิธี เราจะนำเสนอไปตามลำดับดังนี้

บับเบิ้ลซอร์ท (Bubble Sort)

ทำได้โดยเปรียบเทียบสองจำนวนแรกในอาร์เรย์ สลับที่สองจำนวนนี้ถ้าจำนวนขวามีค่าน้อยกว่าจำนวนซ้าย (ในที่นี้เราตั้งใจจะเรียงข้อมูลจากน้อยไปมาก)



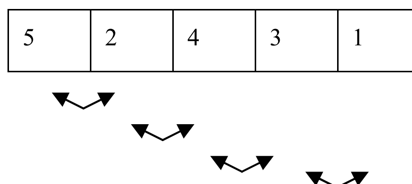
ต่อจากนั้นก็มาเปรียบเทียบตัวเลขคู่ถัดมาในอาร์เรย์นั้น(ซึ่งตัวแรกอาจเคยถูกสลับที่มาแล้ว) แล้วทำการสลับที่ในกรณีเดียวกัน



ทำไปเรื่อยๆจนถึงคู่สุดท้าย แล้วก็กลับมาเปรียบเทียบคู่แรกใหม่ ทำไปเรื่อยๆจนได้อาร์เรย์ที่มีสมาชิกเรียงกันทั้งหมด แต่จะต้องเปรียบเทียบและสลับที่กี่ครั้งดี เราลองมาคิดกันดู

การสลับที่ต้องมีจำนวนครั้งที่เพียงพอที่จะเคลื่อนค่าที่มากที่สุดจากช่องซ้ายสุดของอาร์เรย์ไปยังช่องขวาสุด

การสลับที่ต้องมีจำนวนครั้งที่เพียงพอที่จะเคลื่อนค่าที่น้อยที่สุดจากช่องขวาสุดของอาร์เรย์ไปยังช่องซ้ายสุด



รูป 2.5 bubble sort ผ่านอาร์เรย์ 1 รอบ

จากรูปที่2.5 เมื่ออาร์เรย์มีขนาด n การเปรียบเทียบ (และสลับที่) n-1 ครั้งแรกจะเพียงพอทำให้ค่าที่อยู่ช่องซ้ายสุดของอาร์เรย์เคลื่อนไปยังช่องขวาสุดได้ แต่ข้อมูลในช่องขวาสุดนั้นจะเลื่อนมาได้เพียงหนึ่งช่องเท่านั้น ถ้าจะให้เพียงพอที่จะเลื่อนข้อมูลในช่องขวาสุดมาช่องซ้ายสุดจะต้องทำการเปรียบเทียบ (และสลับที่) แบบเดียวกันนี้อีก (n-1) รอบ นั่นคือ เราต้องใช้ลูปสองชั้น โค้ดในรูปที่ 2.6 คือ bubble sort ของการเรียงสมาชิกอาร์เรย์จากน้อยไปมาก

```
1:    public static void bubblesort(int[] array){
2:     for (int pass = 1; pass<=array.length-1; pass++)
3:        for(int element=0; element<= array.length –2; element++)
4:            if(array[element] > array[element+1])
5:              swap(array, element, element +1);
6:    }
```

รูป 2.6 โค้ดของ bubble sort

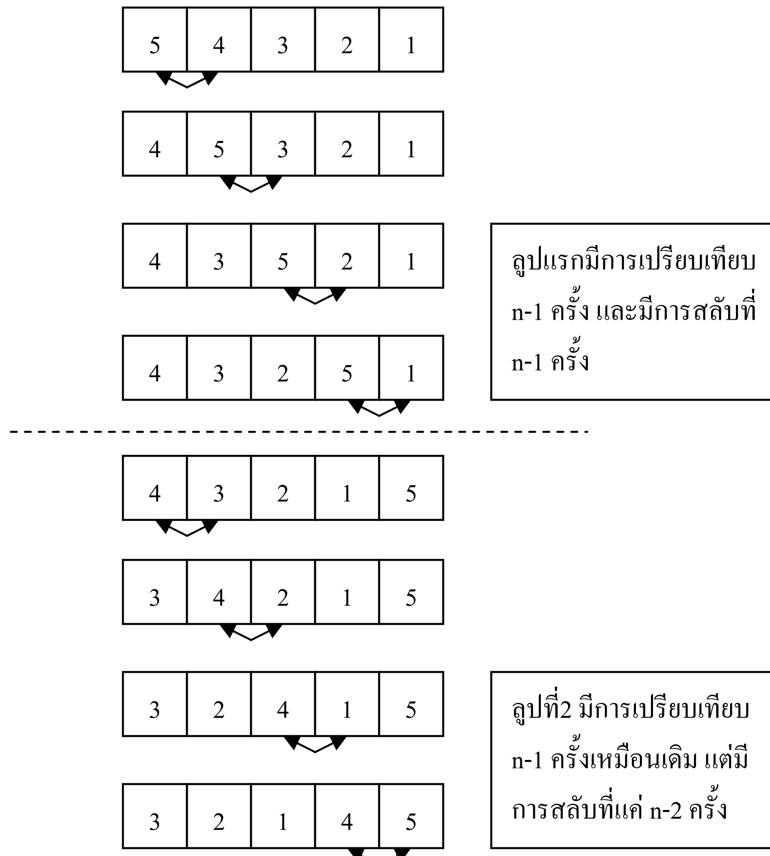
โดยมีโค้ดของ swap ดังรูปที่ 2.7

```
1:    public static void swap(int[] array, int a, int b){
2:       int temp = array[a];
3:       array[a]= array[b];
4:       array[b]= temp;
5:    }
```

รูป 2.7 โค้ดของการ swap

เมื่อลองดูจากจำนวนลูปจะเห็นได้ว่ามี big O คือ n^2

ทีนี้เรามาลองหา worst case เพื่อเอาไว้เปรียบเทียบกับการจัดเรียงแบบอื่น กรณีที่ต้องมีการสลับที่บ่อยที่สุดจะเกิดขึ้นเมื่ออาร์เรย์นั้นเรียงจากมากไปน้อย รูปที่ 2.8 แสดงการ bubble sort สองลูปแรก โดยทำกับจำนวนเต็ม 5 จำนวนซึ่งเรียงจากมากไปน้อยในตอนเริ่มต้น



รูปที่ 2.8 bubble sort 5,4,3,2,1 (สองลูปแรก)

จะเห็นได้ว่ามีการเปรียบเทียบข้อมูล n-1 ครั้งในแต่ละลูป และโค้ดเรามีทั้งหมด n-1 ลูป เพราะฉะนั้นจะมีการเปรียบเทียบข้อมูลในอาร์เรย์ $(n-1)^2$ ครั้ง (ผมจะคิด unit time จากการเปรียบเทียบข้อมูลในอาร์เรย์และการ assign ข้อมูลเข้าหรือออกจากอาร์เรย์เท่านั้น จะได้ไม่มีรายละเอียดมากเกินไป) และจะมีการสลับที่ของข้อมูลในอาร์เรย์ (n-1) + (n-2) + (n-3) +…+ 1 = n(n-1)/2 ครั้ง ดังนั้นถ้าคิดเวลาเป็น unit time ของเรา bubble sort (worst case) จะกินเวลาทั้งสิ้น $= (n-1)^2 + n(n-1)/2$ * unit time ของการสลับที่

$= (n-1)^2 + n(n-1)/2 * 3$

$= (5n^2 - 7n + 2)/2$

แบบฝึกหัด

1. จงปรับปรุงโค้ดของ bubble sort ที่ให้มา ให้สามารถ run ได้เร็วกว่าเดิม และจงอธิบายแนวคิดในการปรับปรุงนี้ด้วย
2. จงหาเวลาของ best case และ average case ของ bubble sort

ซีเลคชั่นซอร์ท (Selection Sort)

เราสามารถเรียงสมาชิกของอาร์เรย์ a จากน้อยไปมากด้วยวิธีนี้โดย

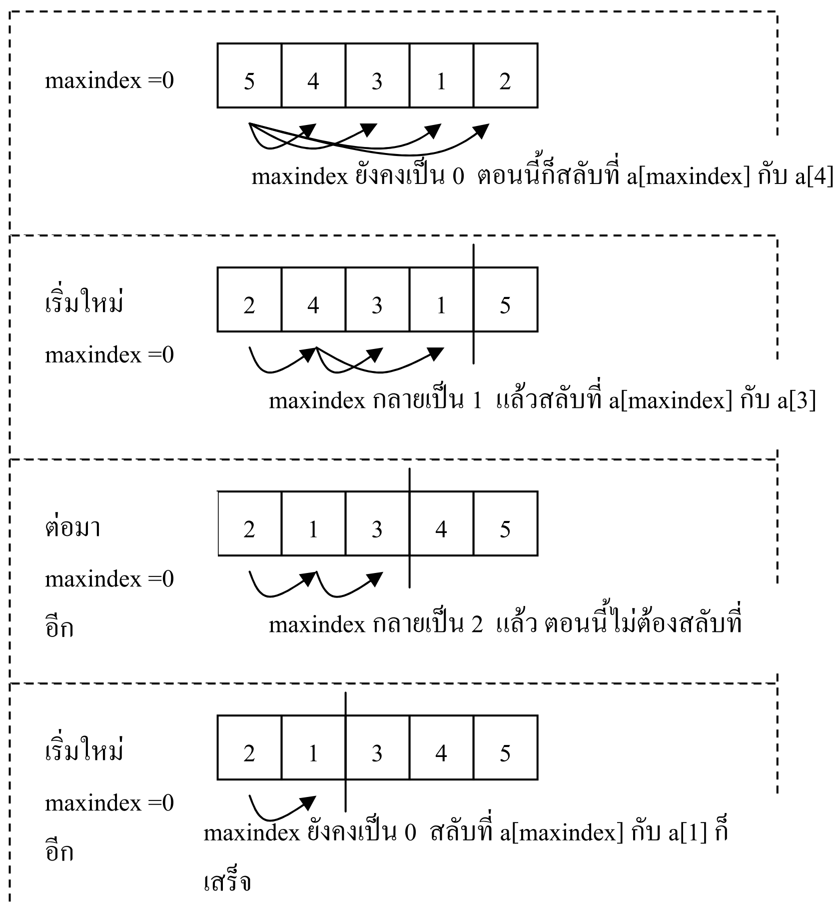
1. เก็บค่า index ของสมาชิกตัวแรกของอาร์เรย์ a เอาไว้ในตัวแปร maxindex
2. ตรวจสอบสมาชิกของอาร์เรย์ที่ละตัวตามลำดับ ถ้าเจอตัวที่มีค่ามากกว่า a[maxindex] ก็ให้เปลี่ยนค่า maxindex ให้เก็บ index ค่าใหม่ ตรวจจนหมดอาร์เรย์
3. สลับที่สมาชิกตัวสุดท้ายของอาร์เรย์กับ a[maxindex] (ถ้าสองตัวนี้ไม่ใช่ตัวเดียวกัน) เพื่อให้ค่าที่มากที่สุดไปอยู่ขวาสุด
4. ทำข้อ 1 – 3 ใหม่ กับสมาชิก n-1 ตัวที่เหลือ แล้วทำต่อไป โดยสมาชิกจะลดลงเรื่อยๆ

จากโค้ด (รูป 2.9) เราจะสังเกตได้ว่า selection sort นั้นมี big O เป็น n^2 (เมื่อ n เป็นขนาดของอาร์เรย์) เหมือนกับ bubble sort แต่ถ้าเรามาดูให้ละเอียดจะสังเกตได้ว่า ในการตรวจสอบอาร์เรย์ 1 รอบใดๆ จะมีการสลับที่สมาชิกของอาร์เรย์เพียงหนึ่งครั้งเท่านั้น ซึ่งน้อยกว่าจำนวนการสลับที่สมาชิกของ bubble sort (รูป 2.10 เป็นตัวอย่าง selection sort ของ 5,4,3,1,2)

worst case (จากการเปรียบเทียบข้อมูลในอาร์เรย์และการ assign ข้อมูลเข้าหรือออกจากอาร์เรย์เท่านั้น) จะเกิดเมื่อมีการสลับที่ทุกรอบการใช้ลูป นั่นคือ ข้อมูลเกือบจะเรียงกัน แต่ตัวที่น้อยที่สุดอยู่ท้ายสุด เช่น 2,3,4,5,1 เราสามารถนับได้ดังนี้

```
1:    public static void selectionSort(int[] a){
2:      int maxindex; //index of the largest value
3:      for(int unsorted= a.length; unsorted > 1;
unsorted--){
4:        maxindex = 0;
5:        for(int index= 1; index < unsorted; index++){
6:           if(array[maxindex] < array[index])
7:             maxindex = index;
8:        }
9:        if(a[maxindex] != a[unsorted -1])
10:         swap(array, maxindex, unsorted -1);
11:     }
12:   }
```

รูป 2.9 โค้ดของ selection sort



รูป 2.10 ตัวอย่าง selection sort

1. การเปรียบเทียบข้อมูลในอาร์เรย์ (n-1) + (n-2) + … + 1 ครั้ง (บรรทัดที่ 6 จากโค้ด)

2. การเปรียบเทียบข้อมูลในอาร์เรย์เพื่อตัดสินการสลับที่ (n-1) ครั้ง (บรรทัดที่ 9 จากโค้ด)

3. การสลับที่สมาชิกของอาร์เรย์ n-1 ครั้ง (บรรทัดที่ 10 จากโค้ด)

เพราะฉะนั้น จะเป็นเวลารวม

= n(n-1)/2 + (n-1) + (n-1)(unit time ของการสลับที่)

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

= $(n^2 + 7n - 8)/2$

เราลองมาเทียบกับเวลา worst case ของ bubble sort โดยสมมติว่า bubble sort นั้นช้ากว่า จะได้ว่า

$$(5n^2 - 7n + 2)/2 >= (n^2 + 7n - 8)/2$$
$$5n^2 - 7n + 2 >= n^2 + 7n - 8$$
$$4n^2 - 14n + 10 >= 0$$
$$2n^2 - 7n + 5 >= 0$$
$$(2n-5)(n-1) >= 0$$

เพราะฉะนั้นเราได้ n = 1,2.5 เนื่องจาก n เป็นจำนวน input ดังนั้นต้องมีค่าตั้งแต่ 1 ขึ้นไป เรารู้จากคำตอบของสมการนี้ว่า n เป็น 1 กับ 2.5 ไม่ได้ อยู่ระหว่างสองจำนวนนี้ก็ไม่ได้เพราะจะทำให้สมการไม่เป็นจริง ฉะนั้น n จึงต้องมากกว่า 2.5 เท่านั้น สรุปคือ เมื่อ n เป็น 3 ขึ้นไป เวลา worst case unit time ของ bubble sort ก็มากกว่า selection sort แล้ว

อินเซอร์ชั่นซอร์ท (Insertion Sort)

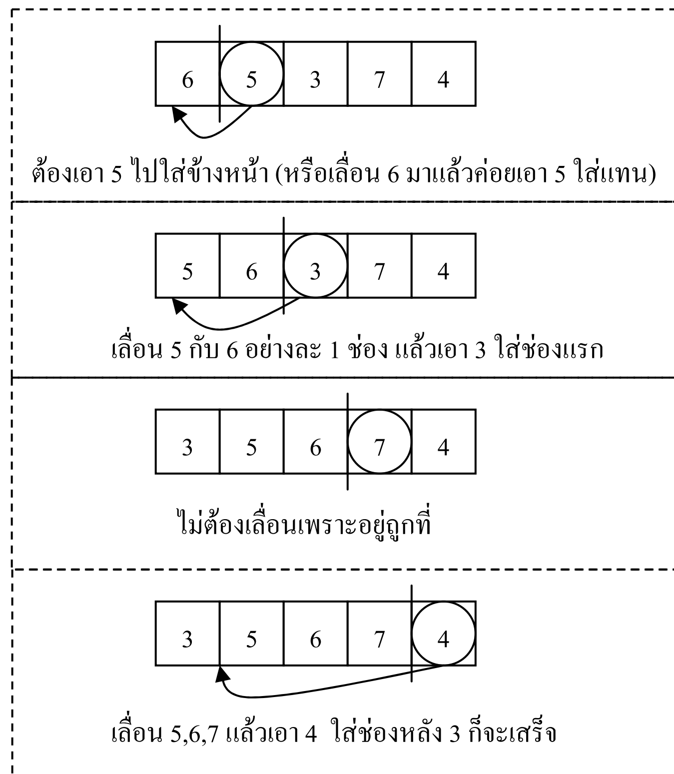
เราสามารถเรียงสมาชิกของอาร์เรย์ a จากน้อยไปมากด้วยวิธีนี้โดย

1. แบ่งอาร์เรย์เป็นข้างซ้ายและขวา ให้ข้างซ้ายเป็นข้างที่ถือว่าจัดเรียงแล้ว (แน่นอนว่าในตอนเริ่มต้องมีจำนวนสมาชิกในข้างนี้แค่ตัวเดียว)

2. ตรวจข้างขวาของอาร์เรย์โดยไล่ตรวจทีละตัว ถ้าเจอตัวที่มีค่าน้อยกว่าตัวสุดท้ายของข้างซ้าย ให้เอาตัวนั้นไปใส่ในข้างซ้ายให้ถูกที่

3. ทำซ้ำข้อ 1 โดยขยายขนาดของข้างซ้ายเพิ่มขึ้น 1 ตัว

รูป 2.11 เป็นตัวอย่าง insertion sort ของ 6,5,3,7,4 ในอาร์เรย์



รูป 2.11 Insertion sort ของ 6,5,3,7,4

โค้ดอยู่ในรูปที่ 2.12 จากโค้ดเราจะสังเกตได้ว่าBig O จะเป็น O(n^2) เช่นเดียวกับ bubble และ selection sort ส่วนกรณีของ worst case เกิดเมื่อต้องมีการเลื่อนสมาชิกของอาร์เรย์มากที่สุด นั่นคืออาร์เรย์ต้องเรียงจากมากไปน้อย ในกรณีนี้ ในการตรวจอาร์เรย์แต่ละรอบ ทุกสมาชิกของข้างซ้ายของอาร์เรย์จะต้องถูกเลื่อนไป 1 ตำแหน่ง ตารางที่ 2.1 แสดงจำนวนครั้งที่เกิดขึ้นของเหตุการณ์ต่างๆ ในแต่ละรอบของลูปด้านนอก

จากตารางจะเห็นได้ว่า unit time ของ worst case insertion sort

$=(1+2+..+n-1)*2 + n-1$

$= n(n-1) + n-1$

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

```
1:    public static void insertionSort(int[] a){
2:     int index;
3:     for(int numSorted = 1; numSorted < a.length;
numSorted++){
4:       int temp = a[numSorted];
5:       for(index = numSorted; index >0; index--){
6:         if(temp< a[index-1]){
7:           a[index] = a[index -1];
8:         } else{
9:           break;
10:        }
11:      }
12:      a[index] = temp;
13:    }
14:   }
```

รูป 2.12 โค้ดของ Insertion sort

ตาราง 2.1 จำนวนเหตุการณ์ ของ worst case insertion sort ในแต่ละรอบของลูปนอกสุด

ลูปที่	temp< a[index-1]	a[index] =a[index-1]	a[index]= temp
1	1	1	1
2	2	2	1
...	…	…	1
n-1 (ลูปสุดท้าย)	n-1	n-1	1

ซึ่งถ้าสมมติว่าเร็วกว่า selection sort แล้ว เราจะได้สมการ

$$(n^2 + 7n - 8)/2 \quad >= \quad n^2 - 1$$

$$0 \quad >= \quad n^2 - 7n + 6$$

$$0 \quad >= \quad (n-6)(n-1)$$

ซึ่งจะเป็นจริงก็ต่อเมื่อ n มีค่าระหว่าง 1 ถึง 6 เท่านั้น แสดงว่า ถ้า n มีค่าเกิน 6 insertion sort จะช้ากว่า selection sort (อย่าลืมว่าตอนนี้เราดูแค่ worst case เท่านั้น) ในทำนองเดียวกัน insertion sort จะเร็วกว่า bubble sort เมื่อ n มีค่าเป็น 2 ขึ้นไป ส่วนกรณีที่ insertion sort จะมีเวลาเร็วที่สุด จะเกิดจากการที่ต้องมี break เกิดขึ้นบ่อยครั้งที่สุด(ดูจากโค้ด) นั่นคืออาร์เรย์ต้องจัดเรียงจำนวนอยู่แล้วนั่นเอง โดยจะมีเวลาที่ใช้เป็น O(n) ดังนั้น insertion sort จึงมักถูกใช้ในการจัดเรียงข้อมูลที่เกือบเรียงกันดีแล้ว

อาจมีคนสงสัยว่า แล้วเวลาโดยเฉลี่ยของ insertion sort จะเป็นเท่าไร เข้าใกล้ O(n) หรือ $O(n^2)$ ถ้าเราอยู่ที่ลูปนอกสุดลูปที่ i

1. ถ้า a[i] อยู่ในที่ที่ไม่ต้องมีการเลื่อน จะเกิดการเปรียบเทียบ temp< a[index-1] เพียงครั้งเดียวเท่านั้น(โค้ดบรรทัดที่ 6 ของรูป 2.12) สำหรับลูปนั้น การเปรียบเทียบครั้งเดียวถือว่าน้อยที่สุดเท่าที่เป็นไปได้ เพราะโค้ดกำหนดไว้อย่างนั้น
2. ถ้า a[i] อยู่ในที่ที่ต้องมีการเลื่อนตำแหน่ง จะมีการเปรียบเทียบเกิดขึ้นได้ตั้งแต่หนึ่งครั้งจนถึง i ครั้ง เช่นถ้า i มีค่าเป็น 2 จะต้องมีการเปรียบเทียบ a[2] กับ a[1] (นี่นับเป็นครั้งที่หนึ่ง)และจากนั้นถ้าเกิดการเลื่อน a[1] ไปใส่ a[2] จะต้องเปรียบเทียบค่าเดิมของ a[2] กับ a[0] ด้วย (นี่นับเป็นครั้งที่สองและครั้งสุดท้ายเพราะลูปด้านในจะทำงานจบแล้ว) เพื่อดูว่าจะมีการเลื่อน a[0] ไปใส่ a[1] ได้ด้วยหรือไม่

ฉะนั้น ถ้าเราดูแต่จำนวนครั้งของการเปรียบเทียบ จะได้ว่าในลูปที่ i มีจำนวนการเปรียบเทียบโดยเฉลี่ย $\frac{1+2+...+i}{i} = \frac{\frac{1}{2}i(i+1)}{i} = \frac{i+1}{2}$ ครั้ง

ดังนั้นถ้าคิดทุกลูป ค่าจำนวนครั้งของการเปรียบเทียบที่เกิดขึ้นจะเป็น

$\sum_{i=1}^{n-1}\frac{i+1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1}\frac{1}{2} = \frac{n^2+n-2}{4} = O(n^2)$ ครั้ง ดังนั้นจึงสรุปได้ว่าค่าเวลาเฉลี่ยจะเข้าใกล้ค่าเวลาของ worst case

เมิร์จซอร์ท (Merge Sort)

หลักของการจัดเรียงข้อมูลด้วยวิธีนี้ คือการ

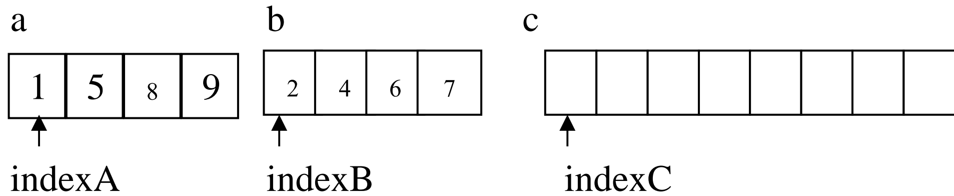
1. แบ่งอาร์เรย์เป็นสองส่วน ไปจัดเรียงข้อมูลในแต่ละส่วนให้เรียบร้อย (แต่ละอาร์เรย์ก็แบ่งสองส่วนได้อีก ทำเป็น recursion ได้)
2. รวมอาร์เรย์ที่จัดเรียงเรียบร้อยเข้าด้วยกัน

ก่อนจะไปดูว่าจัดเรียงอย่างไร เราควรรู้ว่า อาร์เรย์ที่เรียงสมาชิกแล้ว สองอาร์เรย์รวมกันได้อย่างไร โดยดูจากตัวอย่าง 2-7

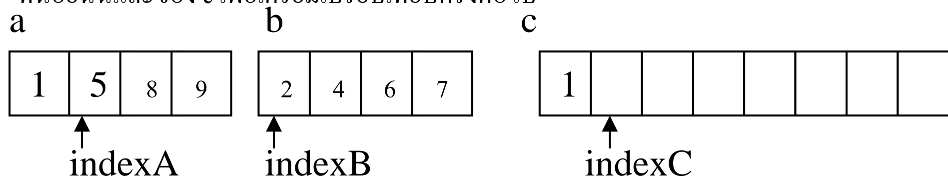
ตัวอย่างที่ 2-7

การรวมอาร์เรย์ a (1,5,8,9) กับ b (2,4,6,7)

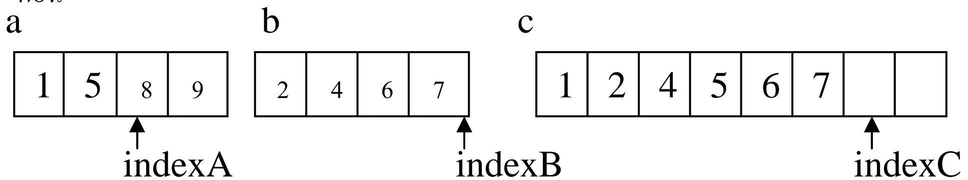
ให้มี counter อยู่ที่ index แรกของอาร์เรย์ทั้งสอง แล้วสร้างอาร์เรย์สำหรับเก็บผลการรวมขึ้นมา



เปรียบเทียบ a[indexA] กับ b[indexB] เอาค่าที่น้อยกว่า ใส่ c[indexC] แล้วเลื่อน counter ของตัวที่น้อยนั้นและของ c เพื่อเตรียมเปรียบเทียบครั้งต่อไป



เปรียบเทียบ a[indexA] กับ b[indexB] แบบนี้ไปเรื่อยๆจนหมดหนึ่งอาร์เรย์ ในที่นี้ b จะหมดก่อน



หลังจากนี้ก็แค่ก็อปปี้ส่วนที่เหลือของ a ใส่ c ให้หมดก็จะได้อาร์เรย์ผลลัพธ์ที่สมบูรณ์ (ไม่วาดให้ดูต่อแล้วนะ)

Worst case ของการรวมอาร์เรย์นี้ เกิดเมื่อต้องมีการเปรียบเทียบสมาชิกอาร์เรย์ทั้งสอง ไปจนถึงสมาชิกตัวสุดท้ายของทั้งคู่ ซึ่งจะมีการเปรียบเทียบทั้งหมด n-1 ครั้ง (n คือขนาดของอาร์เรย์ผลลัพธ์) เพราะฉะนั้น เวลาในการทำงานของการรวมอาร์เรย์จึงเป็น O(n) โค้ดของการรวมอาร์เรย์อยู่ในรูปที่ 2.13 จะเห็นว่าเวลาในการรวมอาร์เรย์นั้นเป็น O(n) จริงๆ

คราวนี้มาถึงการแบ่งอาร์เรย์ออกเป็นส่วนๆบ้าง จริงๆแล้วเราไม่ต้องมาจัดเรียงข้อมูลในตอนนี้เลยด้วยซ้ำ ลองคิดดูว่าถ้าเราแบ่งอาร์เรย์ออกเป็นสองส่วนไปเรื่อยๆ อะไรจะเกิดขึ้น แน่นอนว่า

```
1:    public static int[] merge(int[] a, int[] b){
2:        int aIndex = 0;
3:        int bIndex = 0;
4:        int cIndex = 0;
5:        int aLength = a.length;
6:        int bLength = b.length;
7:        int cLength = aLength + bLength;
8:        int[] c = new int[cLength];
9:        //ก่อนอื่นเปรียบเทียบ a กับ b แล้วเอาเลขออกใส่ c จนอาร์เรย์ตัวใดตัวหนึ่งหมด
10:       while((aIndex < aLength) && (bIndex < bLength){
11:           if(a[aIndex]<=b[bIndex]){
12:               c[cIndex] = a[aIndex];
13:               aIndex++;
14:           }else{
15:               c[cIndex] = b[bIndex];
16:               bIndex++;
17:           }
18:           cIndex++;
19:       }
20:       //ต่อไปก็ก็อปปี้อาร์เรย์ที่เหลือใส่ c ให้หมด
21:       if(aIndex == aLength){  //ถ้า a ถูกใช้หมดก่อน
22:            while(bIndex<bLength){
23:               c[cIndex] = b[bIndex];
24:               bIndex++;
25:               cIndex++;
26:            }
27:       }else{  //ถ้า b ถูกใช้หมดก่อน
28:            while(aIndex<aLength){
29:               c[cIndex] = a[aIndex];
30:               aIndex++;
31:               cIndex++;
32:            }
33:       }
34:       return c;
35:   }
```

รูป 2.13 การรวมอาร์เรย์ใน merge sort

เมื่อเราแบ่งอาร์เรย์ไปเรื่อยๆ ผลสุดท้ายจะได้อาร์เรย์ที่มีขนาดเพียงหนึ่งมารวมกัน ตอนรวมอาร์เรย์ขนาดหนึ่งเข้าด้วยกันก็จะเป็นการจัดเรียงไปในตัวเลย ดังนั้น จากการรวมอาร์เรย์ขนาด 1 จนใหญ่ขึ้นเรื่อยๆ ก็จะเป็นการเรียงข้อมูลไปในตัวเลย โค้ดของการแบ่งอาร์เรย์จึงเป็นดังรูปที่ 2.14 โดยตอนใช้งาน ก็แค่เรียกใช้ mergesort(array, 0, array.length-1);

```
1:    public static int[] mergeSort(int[] unsort, int left, int right){
2:     if(left == right){//ถ้าไม่เหลืออะไรให้จัดแล้ว ต้องตอบเป็นอาร์เรย์ขนาด 1
3:       int[] x = new int[1];
4:       x[0] = unsort[left];
5:       return x;
6:     }
7:     else if(left<right){//ถ้ายังจัดได้ ก็แบ่งอาร์เรย์ต่อไป
8:    int center = (left+right)/2;
9:      int[] result1 = mergeSort(unsort,left,center);
10:     int[] result2 = mergeSort(unsort,center+1,right);
11:     return merge(result1,result2);
12:    }
13:   }
```

รูป 2.14 โค้ดของการแบ่งอาร์เรย์เป็นส่วนๆในการ merge sort

เราลองมาวิเคราะห์เวลาในการทำงานของ merge sort กัน(สมมติว่าขนาดของอาร์เรย์จะแบ่งเป็นสองข้างได้เท่ากันเสมอ เพื่อความง่ายในการคำนวณ) ถ้ามีสมาชิกตัวเดียวเราเห็นได้ชัดว่าเวลาที่ใช้นั้นคงที่ (ให้เป็น 1 ได้) ส่วนถ้ามีสมาชิกหลายตัว เวลาที่ใช้จะเท่ากับเวลาที่ทำงานครึ่งซ้ายรวมกับครึ่งขวา แล้วรวมกับเวลาที่ใช้ในการรวมอาร์เรย์ทั้งสองอีกทีหนึ่ง (ซึ่งเป็น O(n)) ดังนั้นถ้าเราให้เวลาในการทำงานเมื่อมีข้อมูลเข้าขนาด n เป็น T(n) เราจะสามารถเขียนได้ว่า

$$T(1) = 1$$

$$T(n) = 2T(\frac{n}{2}) + n$$

เอา n หารตลอด ได้

$$\frac{T(n)}{n} = \frac{T(\frac{n}{2})}{\frac{n}{2}} + 1$$

จะเห็นว่า นี่มีรูปแบบเดียวกับการแก้ปัญหา maximum subsequence sum โดยใช้วิธีแบ่งแยกและเอาชนะ ดังที่สอนไว้ในบทแรก ดังนั้นเราสามารถเขียนสมการเพิ่มได้เช่นเดียวกัน

$$\frac{T(\frac{n}{2})}{\frac{n}{2}} = \frac{T(\frac{n}{4})}{\frac{n}{4}} + 1$$

$$\frac{T(\frac{n}{4})}{\frac{n}{4}} = \frac{T(\frac{n}{8})}{\frac{n}{8}} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

....

เอา T(n)/n มาบวกไปจนถึง T(2)/2 จะได้

$$\frac{T(n)}{n} = T(1) + \log n$$

$$\therefore T(n) = n + n \log n = O(n \log n)$$

ดังนั้นจะเห็นว่า merge sort นั้นมีเวลาเร็วกว่าการจัดเรียงข้อมูลแบบอื่นๆที่กล่าวมา แต่การจัดเรียงข้อมูลด้วยวิธีนี้มีข้อเสียคือต้องใช้พื้นที่ในการสร้างอาร์เรย์ที่เป็นผลรวมของสองอาร์เรย์ย่อย

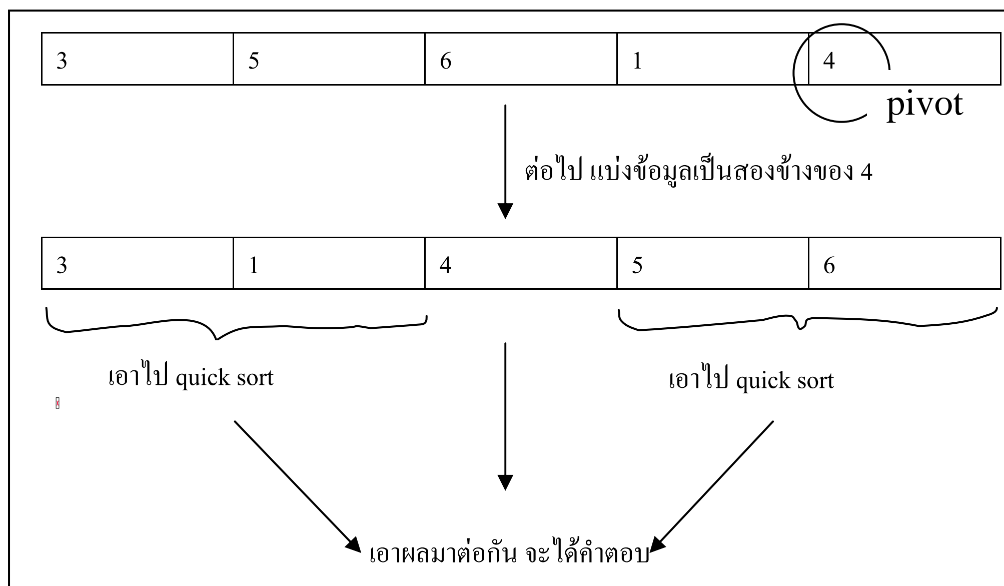
ควิกซอร์ท(Quick Sort)

มีหลักการคล้ายกับ merge sort แต่ว่าประหยัดเนื้อที่กว่ามาก นี่ถือเป็นการจัดเรียงข้อมูลทั่วไปที่เร็วที่สุดในทางปฏิบัติ วิธีนี้เป็นวิธีที่ใช้การแบ่งแยกและเอาชนะเช่นเดียวกับ merge sort โดยมีวิธีทำดังนี้

1. ถ้าไม่มีสมาชิกในอาร์เรย์หรือมีสมาชิกแค่ตัวเดียวให้ตอบเป็นอาร์เรย์นั้นเลย เพราะถือว่าแบบนี้ข้อมูลจัดเรียงในตัวเองอยู่แล้ว

2. เลือกตัวเลขใดตัวเลขหนึ่งในอาร์เรย์มา ให้จำนวนนั้นเป็นจุดหลัก (ต่อไปนี้จะเรียกว่า pivot)
3. ใช้ pivot เป็นหลัก เลื่อนสมาชิกทุกตัวที่น้อยกว่า pivot ไปอยู่ด้านซ้ายของ pivot และเลื่อนสมาชิกทุกตัวที่มากกว่า pivot ไปอยู่ด้านขวาของ pivot (ส่วนสมาชิกที่มีค่าเท่ากับpivot ก็แล้วแต่จะจัดการ วิธีที่ดีที่สุดน่าจะเป็นการกระจายจำนวนเหล่านี้ไปยังทั้งสองข้างเท่าๆกัน) ขั้นตอนนี้เรียกว่าการ partition
4. ตอนนี้ตัว pivot ก็จะได้อยู่ในที่ที่ถูกต้องเรียบร้อยแล้ว ที่เราเหลือต้องทำก็คือ ทำquick sort กับอาร์เรย์ย่อยข้างซ้ายและข้างขวาของ pivot
5. คำตอบคืออาร์เรย์ที่เกิดจากการต่อ quicksort(ด้านซ้าย) กับ pivot กับ quicksort(ด้านขวา)

รูป 2.15 แสดงหลักการทำ quick sort โดยเลือก pivot เป็นค่ามัธยฐานของค่าที่อยู่ในช่องแรก ช่องกลาง (คำนวณ index จาก (indexแรก+indexสุดท้าย)/2) และช่องสุดท้ายของอาร์เรย์



รูป 2.15 หลักการของ quick sort

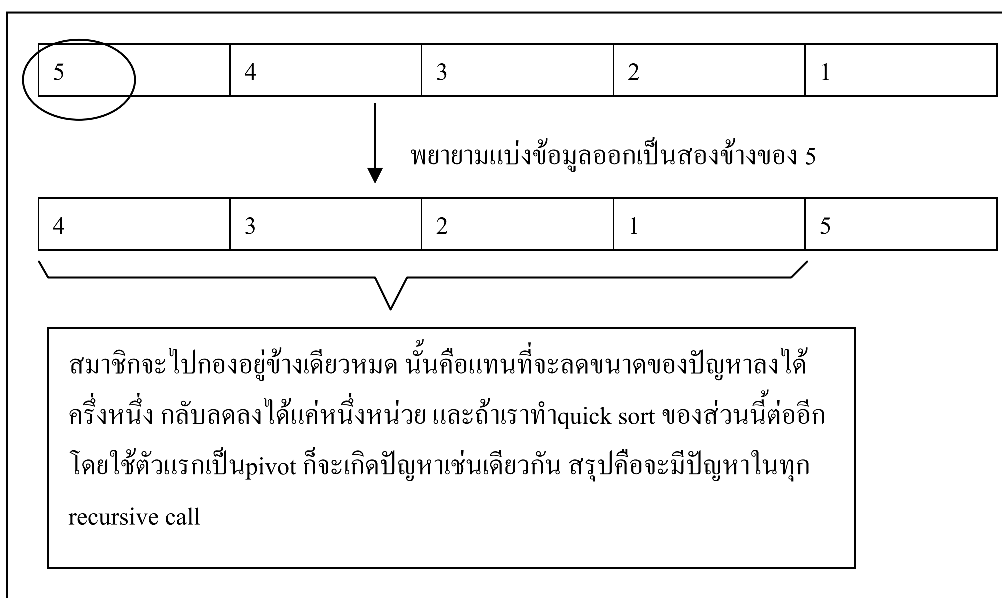
ต่อไปเรามาดูขั้นตอนต่างๆ โดยละเอียดกัน

การทำควิกซอร์ทกับอาร์เรย์เล็กๆ

ถ้าไม่มีสมาชิกในอาร์เรย์หรือมีสมาชิกแค่ตัวเดียวให้ตอบเป็นอาร์เรย์นั้นเลย เพราะถือว่าแบบนี้ข้อมูลจัดเรียงในตัวเองอยู่แล้ว แต่เราจะเห็นว่าจริงๆแล้วถ้าอาร์เรย์มีจำนวนสมาชิกน้อย(น้อยกว่า 20) ใช้ insertion sort จะเร็วกว่าซะอีก ดังนั้นถ้าอาร์เรย์มีขนาดเล็ก เราควรใช้วิธีจัดเรียงแบบอื่นแล้วค่อยเอาคำตอบจากวิธีนั้นมา

การเลือกจุดหลัก (pivot selection)

มีวิธีการหลายวิธีที่เราสามารถเอามาใช้เลือก pivot ได้ ยังไงก็ตาม ห้ามใช้ตัวแรกของอาร์เรย์เป็น pivot เด็ดขาด นี่เพราะว่า ถ้าอาร์เรย์นั้นจัดเรียงอยู่แล้ว(ไม่ว่าจะมากไปน้อยหรือน้อยไปมากก็ตาม) ตอนที่แบ่งข้างโดยใช้จุดหลักนี้จะพบว่าข้างหนึ่งเป็นอาร์เรย์ว่างเสมอ ทำให้ไร้ประสิทธิภาพในการจัด ดังรูปที่ 2.16



รูป2.16 quick sort ที่ใช้ตัวแรกของอาร์เรย์เป็น pivot

วิธีที่จะทำให้ได้ pivot ที่ดี มีที่นิยมใช้กันอยู่สองวิธีด้วยกันคือ

1. เลือก pivot โดยใช้ random number ส่วนใหญ่แล้วจะได้การแบ่ง partition สองข้างที่ดี แต่ว่าวิธีนี้มีข้อเสียที่ว่าการสร้าง random number นั้นเสียเวลาในตัวของมันเอง

2. ใช้ค่ามัธยฐานจากอาร์เรย์ช่องแรกกับช่องกลางและช่องสุดท้าย จริงๆแล้วค่า pivot ที่จะทำให้แบ่งอาร์เรย์ได้เป็นสองข้างเท่ากันนั้นจะต้องเป็นค่ามัธยฐานจากทั้งอาร์เรย์ แต่จะมานั่งคิดค่านี้มันก็จะเสียเวลาเกินไป ดังนั้นจึงได้มีการลองใช้ค่ามัธยฐานจากอาร์เรย์ช่องแรกกับช่องกลางและช่องสุดท้าย พบว่าได้ผลดีเช่นกัน

รูป 2.17 แสดงโค้ดของการเลือกจุด pivot โดยหาจากมัธยฐานของอาร์เรย์ช่องแรกกับช่องกลางและช่องสุดท้าย (ในที่นี้ช่องแรกกับช่องสุดท้ายถูกกำหนดโดย l และ r ตามลำดับ)

```
1:    private static int pivotIndex(int[] a, int l, int r){
2:         int c = (l+r)/2;
3:         if((a[l]<=a[r] && a[l]>=a[c]) ||
4:            (a[l]>=a[r] && a[l]<=a[c]))
5:             return l;
6:         if((a[c]<=a[l] && a[c]>=a[r]) ||
7:            (a[c]>=a[l] && a[c]<=a[r])
8:             return c;
9:         return r;
10:   }
```

รูป 2.17 โค้ดของการหา pivot

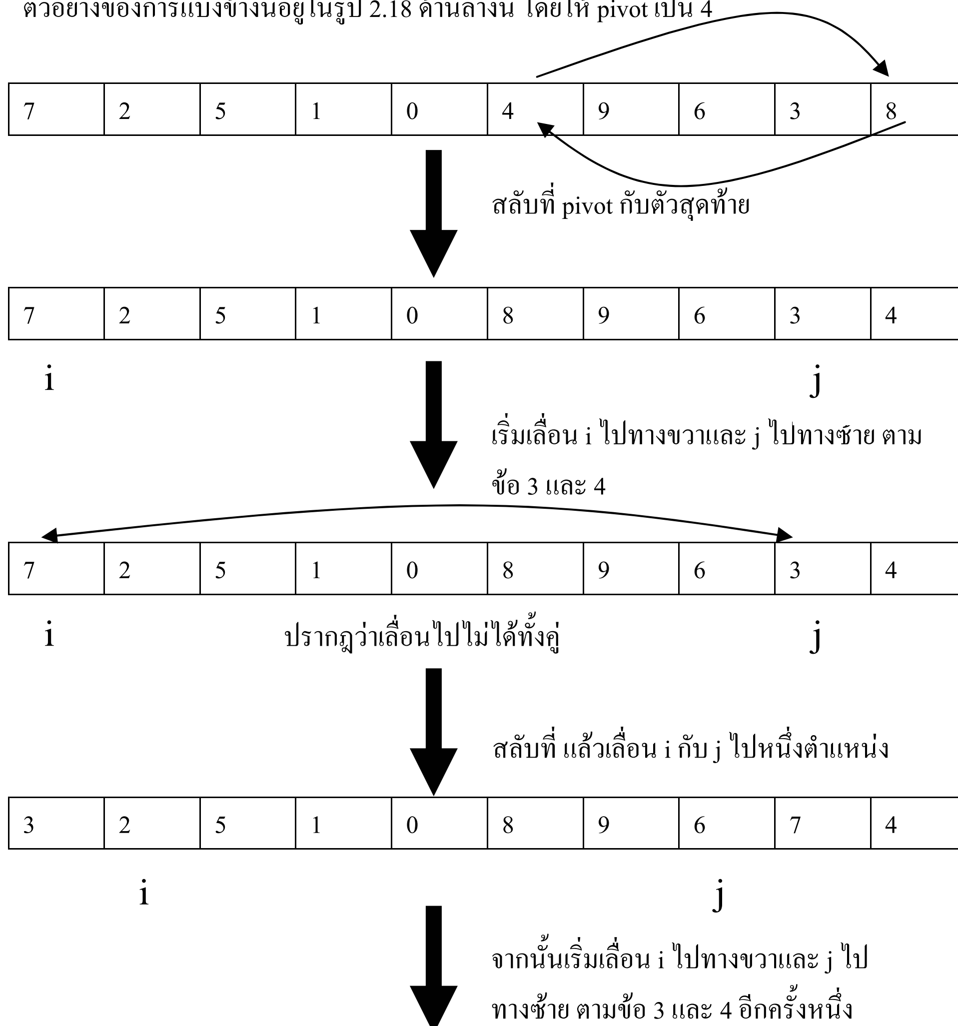
การแบ่งข้าง (partitioning)

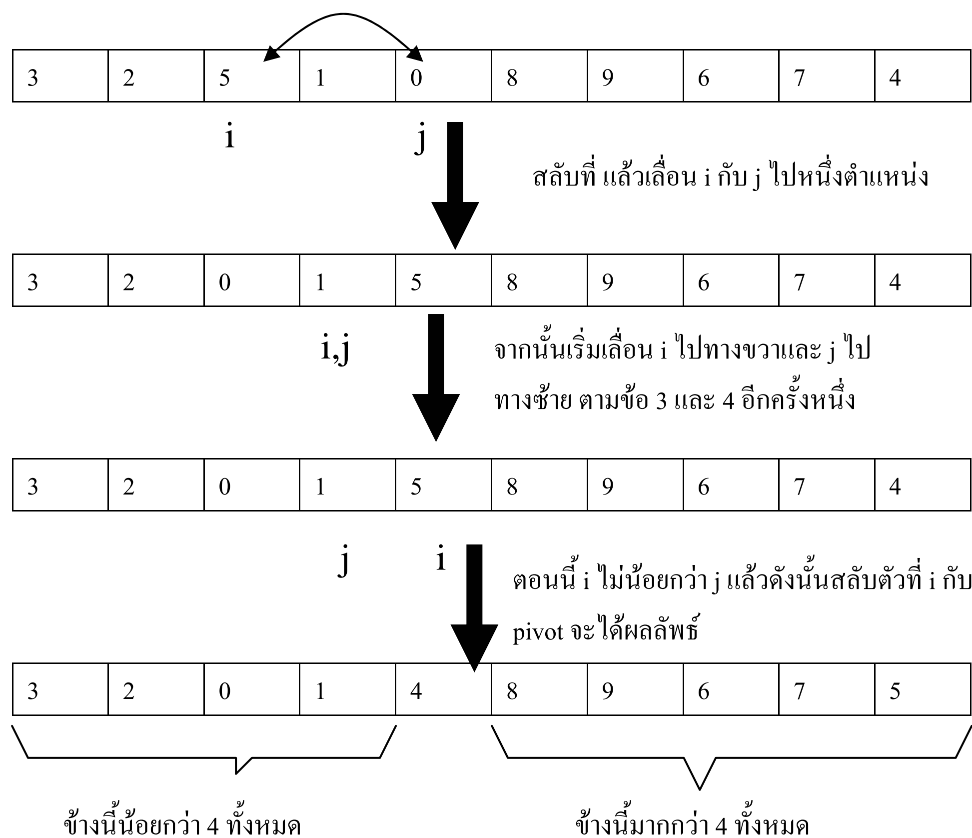
ในที่นี้จะเสนอการแบ่งแบบหนึ่งเท่านั้น แต่การแบ่ง partition นั้นล้วนมีหลักการคล้ายกันหมด วิธีที่จะนำเสนอ (ตอนนี้ให้คิดเพียงว่า สมาชิกทุกตัวในอาร์เรย์มีค่าต่างกันหมดไว้ก่อน) เป็นขั้นตอนดังนี้

1. เอา pivot ที่เลือกได้แล้วออกไปให้พ้นทาง โดยสลับที่กับสมาชิกตัวสุดท้ายของอาร์เรย์
2. ให้ตัวแปร i เป็น index ของตำแหน่งแรกของอาร์เรย์ และตัวแปร j เป็น index ของตำแหน่งรองสุดท้ายของอาร์เรย์ (อย่าลืมว่าตอนนี้เราใช้ตำแหน่งสุดท้ายเก็บ pivot ไปแล้ว)
3. ดูตำแหน่งของ i กับ j ในอาร์เรย์เป็นหลัก ให้เลื่อน i ไปทางขวาเรื่อยๆ ข้ามสมาชิกในอาร์เรย์ที่มีค่าน้อยกว่าค่าของ pivot แต่ต้องหยุดเมื่อเจอสมาชิกในอาร์เรย์ที่มีค่าไม่น้อยกว่าค่าของ pivot
4. ในทำนองเดียวกัน ให้เลื่อน j ไปทางซ้ายเรื่อยๆข้ามสมาชิกที่มีค่ามากกว่าค่าของ pivot และหยุดเมื่อเจอสมาชิกในอาร์เรย์ที่มีค่าไม่มากกว่าค่าของ pivot

5. ถ้า i ยังอยู่ด้านซ้ายของ j อยู่ก็สลับที่สมาชิกในอาร์เรย์สองตำแหน่งนั้นเสีย นี่เป็นการเอาค่าน้อย(เมื่อเทียบกับ pivot)ไปไว้ข้างซ้าย และค่ามากไปไว้ด้านขวาของอาร์เรย์ ส่วนกรณีถ้า i ไม่อยู่ทางด้านซ้ายของ j ให้ข้ามไปทำข้อ 8
6. เลื่อน i ไปทางขวาหนึ่งหน่วย และ เลื่อน j ไปทางซ้ายหนึ่งหน่วย เพื่อหลบเลี่ยงช่องที่พึ่งมีการสลับที่สมาชิกไป
7. จากนั้นเริ่มทำจากข้อ 3 อีกครั้งหนึ่ง
8. ถ้ามาถึงขั้นตอนนี้ ให้สลับที่สมาชิกที่อยู่ที่ตำแหน่ง i กับ pivot ก็จะได้อาร์เรย์ที่มี pivot อยู่ในตำแหน่งที่ถูกต้อง ด้านซ้ายจะมีสมาชิกที่มีค่าน้อยกว่า ส่วนด้านขวาจะมีสมาชิกที่มีค่ามากกว่า

ตัวอย่างของการแบ่งข้างนี้อยู่ในรูป 2.18 ด้านล่างนี้ โดยให้ pivot เป็น 4





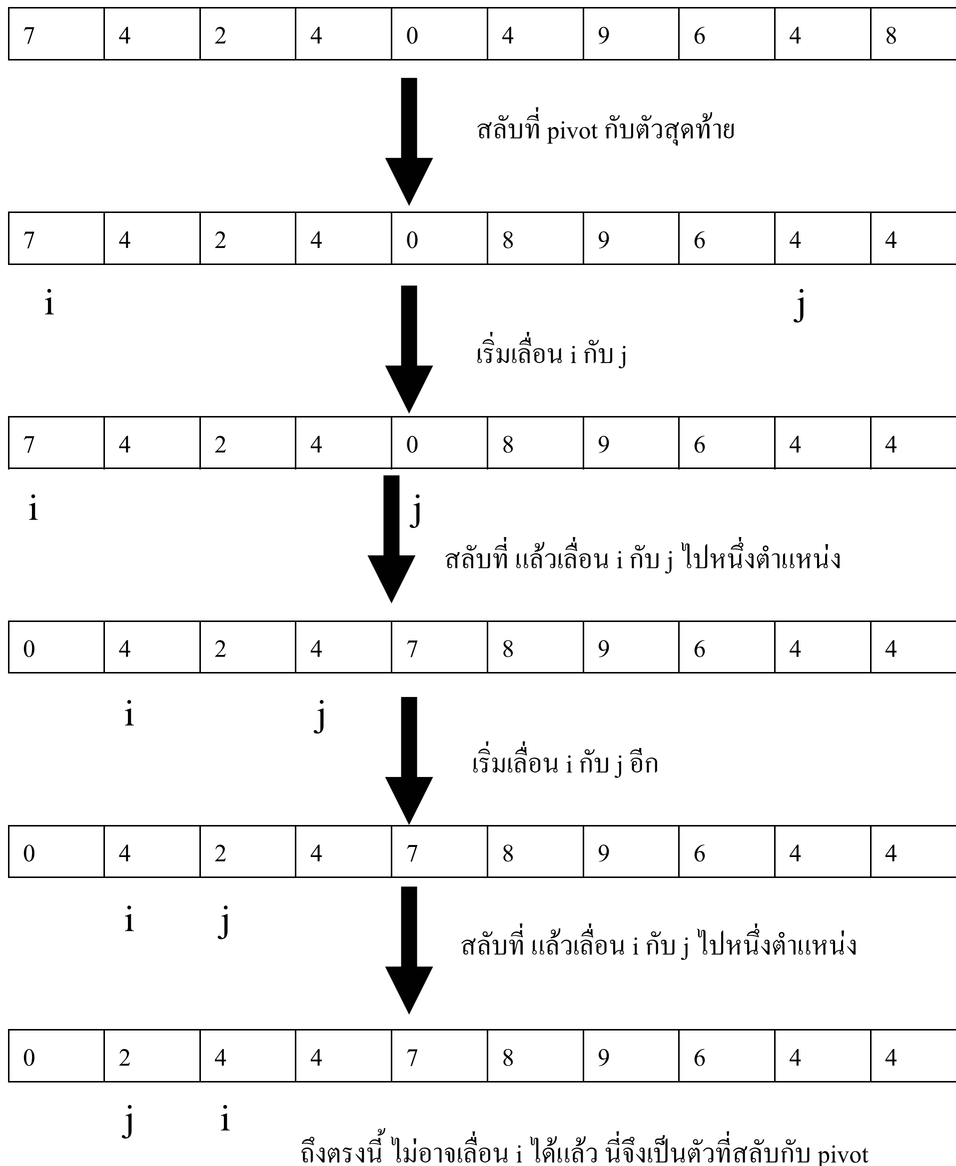
รูป 2.18 ตัวอย่างการแบ่ง partition

แล้วจำนวนที่มีค่าเท่ากับ pivot ล่ะ เราจะทำอย่างไร ควรจะให้ i กับ j หยุดหรือว่าข้ามจำนวนนั้นไปดี เราพิจารณาสามกรณีที่เป็นไปได้ดังนี้

กรณีที่หนึ่งคือ เราทำให้ i หยุด แต่ j ข้ามจำนวนนั้น วิธีนี้ไม่นับว่าเป็นวิธีที่ดี เพราะว่า จำนวนที่เท่ากับ pivot ทั้งหมดจะไปกองอยู่ในอาร์เรย์ข้างเดียวทำให้เสียประสิทธิภาพการจัดเรียง ดังแสดงในรูปที่ 2.19 (ให้เลข 4 ตัวที่สามจากซ้ายเป็น pivot)

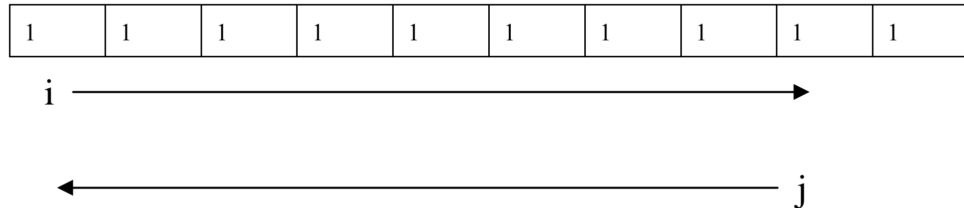
จากรูปจะเห็นว่า หลังจากแบ่งข้าง partition เรียบร้อยแล้ว ค่าที่เท่ากับ pivot จะไปกองอยู่ข้างเดียวของอาร์เรย์ ผลก็จะเป็นเหมือนกันถ้าเราทำให้ j หยุด แต่ i ข้ามค่าของ pivot ดังนั้นถ้าจะให้

ค่าที่เท่ากับ pivot กระจายอยู่สองข้างของอาร์เรย์เท่าๆกัน เราจะต้อง หยุดทั้ง i และ j เมื่อเจอค่าที่เท่ากับ pivot หรือไม่ก็ให้ i และ j เลยไปทั้งคู่ ซึ่งจะดูได้จากกรณีที่สองและสาม



รูป 2.19 การแบ่งข้างโดยให้ i หยุดเมื่อเจอค่าที่เท่า pivot แต่ j ไม่หยุด

กรณีที่สอง คือ ให้ทั้ง i และ j ข้ามค่าที่เท่ากับ pivot ไปให้หมด กรณีนี้ขจัดปัญหาของรูปที่ 2.19 ไป แต่ก็ยังมีปัญหาในตัวเอง ถ้าเราให้จำนวนในอาร์เรย์มีค่าเท่ากันหมดเช่นในรูปที่ 2.20



รูป 2.20 การเลื่อนของ i กับ j เมื่อทั้ง i และ j ข้ามค่าที่เท่ากับ pivot

จากรูปที่ 2.20 จะเห็นว่า i จะเลื่อนไปได้จนสุดท้ายอยู่ที่ตำแหน่งรองสุดท้ายของอาร์เรย์ (ทั้ง i และ j จะเลื่อนได้จนสุดอาร์เรย์ ดังนั้นต้องมีส่วนของโค้ดที่กันการเลื่อนเลยอาร์เรย์ไปด้วย) ดังนั้น เมื่อแบ่งเสร็จ จะกลายเป็นว่า pivot จะอยู่เกือบปลายสุดของอาร์เรย์เลย ดังนั้นวิธีนี้จึงเป็นวิธีที่ไม่ดีนักถ้าเราจัดเรียงอาร์เรย์ใหญ่ๆ(สมมติมีล้านสมาชิก)ที่มีส่วนย่อยเป็นจำนวนเท่ากันเรียงกัน(อาจมีหมื่นสมาชิกที่เท่ากันหมดและเรียงกันเป็นอาร์เรย์ย่อย)

กรณีที่สามคือ ให้ i และ j หยุดลงทั้งคู่เมื่อเจอค่าที่เท่ากับ pivot ในกรณีนี้ จะเกิดการสลับที่ของจำนวนที่เท่ากันโดยเปล่าประโยชน์ ถ้าเราดูอาร์เรย์ในรูป 2.20 จะเห็นว่ามีการสลับที่ทุกขั้นตอน แต่จะมีข้อดีคือ อาร์เรย์จะถูกแบ่งเป็นสองข้างเท่าๆกัน ถ้าเทียบกับกรณีที่สอง แม้วิธีนี้จะมีการสลับที่โดยไร้ประโยชน์แต่ก็ยังเร็วกว่าเพราะแบ่งอาร์เรย์เป็นสองซีกเท่าๆกัน

เมื่อเสร็จสิ้นการแบ่ง partition แล้ว ก็ถือว่าไม่มีอะไรยากสำหรับ quick sort อีกแล้ว รูป 2.21 แสดงโค้ดของ quick sort โดยมี CUTOFF กำหนดขนาดความเล็กของอาร์เรย์ที่เราจะเริ่มใช้การจัดเรียงแบบอื่น ส่วน insertion sort ได้กำหนดขอบเขตซ้ายขวาไว้ด้วย

```
1:    private static void quicksort(int[] a,int l, int r){
2:        if(l+CUTOFF<=r)
3:    {
4:        //ก่อนอื่นต้องหา pivot
5:        int pIndex = pivotIndex(a,l,r);
6:
7:        //เอา pivot ออกไปให้พ้นทาง
8:        swap(a,pIndex,r);
9:        int pivot = a[r];
10:
11:       //เริ่มการแบ่งข้าง
12:       int i=l, j=r-1;
13:       for( ; ; )
14:       {
15:           while(i<r-1 && a[i]<pivot)i++;
16:           while(j>l && a[j]>pivot)j--;
17:           if(i<j){
18:               swap(a,i,j);
19:               i++;
20:               j--;
21:           }else
22:               break;
23:       }
24:       //สลับที่ pivot  ไปที่ที่ถูกต้อง
25:       swap(a,i,r);
26:
27:       //เรียก  quick sort ของอาร์เรย์ย่อยสองข้าง
28:       quicksort(a,l,i-1);
29:       quicksort(a,i+1,r);
30:   }
31:   else
32:       insertionSort(a,l,r);
33:   }
```

รูป 2.21 โค้ดของ quick sort

แบบฝึกหัด

1. จงเขียนโค้ดของ insertion sort ในรูป 2.21

2. เราสามารถทำให้ quick sort เร็วกว่าในรูป 2.21 ได้ค่อนข้างมากสำหรับข้อมูลบางชนิดโดยตอนที่หา pivot ให้จัดเรียงตัวเลขจากในช่องแรก ช่องกลาง และช่องสุดท้ายของอาร์เรย์ในตอนนั้นเลย แล้วตอนที่เอา pivot ออกไปให้พ้นทางก็เอาสลับที่กับตัวรองสุดท้ายของ

อาร์เรย์(แทนที่จะเป็นตัวสุดท้าย) ถามว่า ทำไมไม่ต้องสลับกับตัวสุดท้ายแล้ว และจงเขียนโค้ดของการหา pivot และการทำ quick sort ด้วยวิธีนี้

3. จงเปรียบเทียบเวลาของ quicksort ในรูป 2.21 กับ quick sort ในแบบฝึกหัดข้อที่แล้ว เมื่อข้อมูลที่ต้องการจัดเรียงเป็น 2,3,4,…,n-1,n,1 และเมื่อข้อมูลเรียงจากมากไปน้อย

คราวนี้เรามาหาเวลาในการทำงานของ quick sort กัน เพื่อความง่ายในการคำนวณ เราจะถือว่าใช้ random pivot และไม่ใช้ insertion sort เมื่ออาร์เรย์มีขนาดเล็ก ให้ T(n) = เวลาในการทำงานของโปรแกรมเมื่อมีอาร์เรย์ขนาด n เราให้ T(0)=1, T(1)=1

สำหรับในกรณีอื่น เวลาของโปรแกรมจะเกิดจากผลรวมของ

1. เวลาในการหา pivot ซึ่งเป็นเวลาคงที่ไม่ว่าขนาดอาร์เรย์จะเป็นเท่าไร ดังนั้นเราตัดตัวนี้ทิ้งได้
2. เวลาในการแบ่ง partition ซึ่งแปรผันตรงกับขนาดของอาร์เรย์ เราให้เวลานี้เท่ากับ c*n
3. เวลาในการทำ quick sort ของสองข้าง ให้ข้างซ้ายมีขนาดอาร์เรย์เป็น i ก็แล้วกัน

สรุปได้ว่า $T(n) = T(i) + T(n-i-1) + cn$ (a)

เวลาในการทำงานของควิกซอร์ทกรณีแย่สุด

กรณี worst case จะเกิดเมื่อ pivot เป็นจำนวนที่น้อยที่สุดเสมอ ดังนั้นการแบ่ง partition จะทำให้เกิดอาร์เรย์ที่ขนาดลดลงไปเพียงหนึ่งในแต่ละครั้ง สมการของเราจะได้

$$T(n) = T(n-1) + T(0) + cn$$

T(0) นั้นมีค่าคงที่เป็น 1 ตลอด ดังนั้นเราจึงสามารถตัดทิ้งได้ และจากการแทนค่าตัวแปรต่างๆกัน เราจะได้ชุดของสมการต่อไปนี้

$$T(n) = T(n-1) + cn$$
$$T(n-1) = T(n-2) + c(n-1)$$
$$T(n-2) = T(n-3) + c(n-2)$$
$$...$$
$$T(2) = T(1) + c * 2$$

เมื่อเอาสมการในชุดเหล่านี้ไปบวกกันหมด เราจะได้

$$T(n) = T(1) + c(2 + 3 + 4 + ... + (n-1) + n)$$
$$T(n) = T(1) + c\sum_{i=2}^{n} i = \mathrm{O}(n^2)$$

เวลาในการทำงานของควิกซอร์ทกรณีดีสุด

กรณีนี้เกิดขึ้นเมื่อแบ่งข้างอาร์เรย์ได้สองข้างเท่าๆกัน สำหรับการคำนวณที่จะให้เห็นนี้ผมสมมติให้ แบ่งอาร์เรย์ได้สองข้างเท่ากันเสมอ วิธีทำจะเหมือนกับกรณีที่เราหาเวลาในการทำงานของ merge sort โดยเราจะได้สมการตอนแรกคือ

$$T(n) = T(n/2) + T(n/2) + cn$$

เราเอามาแปลงนิดหน่อยก็จะได้ชุดของสมการ

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$
$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$
$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$
$$...$$
$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

เมื่อเอาชุดสมการนี้มาบวกกันหมด จะได้

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c\log n$$
$$T(n) = n + cn\log n = O(n\log n)$$

เวลาในการทำงานของควิกซอร์ทกรณีเฉลี่ย

นั่นคือ ขนาดของอาร์เรย์ย่อยข้างหนึ่งอาจเป็น 0 หรือ 1 หรือ 2 หรือ 3 เป็นไปได้จนถึง n-1 เลยทีเดียว(อาร์เรย์ย่อยจะมีขนาด n ไม่ได้ อย่าลืมว่าเราไม่นับตัว pivot) ฉะนั้นถ้าให้แต่ละขนาดมีความเป็นไปได้ที่จะเกิดขึ้นเท่าๆกัน จะได้ว่าแต่ละขนาดมีความเป็นไปได้ = 1/n

ดังนั้น เราสามารถเขียนสมการ (a) ได้เป็น

$$T(n) = 2*\frac{1}{n}\sum_{j=0}^{n-1} T(j) + cn$$

เอาสมการนี้มาคูณ n เข้าไปจะได้

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + cn^2 \qquad \text{(avg1)}$$

ซึ่งเรานำสมการ (avg1) นี้มาดัดแปลงได้เป็น

$$(n-1)T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \qquad \text{(avg2)}$$

เอา (avg1)-(avg2) จะได้

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

ซึ่งถ้าเราตัดค่า c ซึ่งเป็นตัวไม่สำคัญออก เราจะได้

$$nT(n) = (n+1)*T(n-1) + 2cn \qquad \text{(avg3)}$$

จัด (avg3) เสียใหม่โดยหารด้วย n(n+1) เราจะได้

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \qquad \text{(avg4)}$$

จาก (avg4) นี้เราสามารถสร้างชุดของสมการดังนี้

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

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

...

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

ซึ่งเมื่อนำทั้งหมด รวมทั้ง (avg4) มาบวกกัน จะได้

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i} \qquad \text{(avg5)}$$

ซึ่งตัวผลบวกนั้นเป็น harmonic number ซึ่งมีสูตรดังนี้

$$\sum_{i=1}^{n}\frac{1}{i} = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon \qquad \text{(harmo)}$$

โดย $n \geq 1, 0 < \varepsilon < \frac{1}{256n^6}, \gamma \approx 0.5772$

แทนค่า (harmo) ลงใน (avg5) เราจะได้

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c(\ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon - 1 - \frac{1}{2})$$

ซึ่งจะเห็นได้ชัดว่าข้างขวาของสมการนั้นมี ln n เป็นหลัก ดังนั้นจึงเป็น O(log n) นั่นเอง เพราะฉะนั้นพอเราย้าย n+1 มาคูณข้างซ้าย ก็จะเห็นว่า

$$T(n) = \mathrm{O}(n \log n)$$

แบบฝึกหัด

1. จงนำ quick sort ไปประยุกต์ใช้หาจำนวนที่น้อยที่สุดเป็นอันดับที่ k ในอาร์เรย์ จงเขียนวิธีทำเป็นขั้นตอน รวมทั้งเขียนโค้ดด้วย
2. จงพิสูจน์ว่า วิธีการหาจำนวนที่น้อยที่สุดเป็นอันดับที่ k ในข้อที่แล้วมี worst case เป็น $O(n^2)$ และมีเวลาเฉลี่ยเป็น O(n)

การจัดเรียงแบบใช้ที่ฝากข้อมูล(Bucket Sort)

วิธีแบบนี้ใช้ในการจัดเรียงข้อมูลที่เรารู้ว่าจะอยู่ตำแหน่งไหนตั้งแต่แรก ดังนั้นไม่ต้องมีการเปรียบเทียบสมาชิกกันเลย ตัวอย่างเช่นการเรียงไพ่ทั้ง 52 ใบ เรารู้อยู่แล้วว่าไพ่ใดจะอยู่ตำแหน่งไหน ดังนั้นก็แค่ทำที่ไว้สำหรับไพ่แต่ละใบ ในตอนที่เลือกไพ่ใบหนึ่ง ก็แค่เอาไพ่ใบนั้นใส่ในที่ที่จองไว้ให้มัน อย่างนี้พอหยิบแต่ละใบ ก็ถือว่าใบนั้นเรียงเข้าถูกที่แล้ว ดังนั้นเวลาในการจัดเรียงก็ย่อมเป็น O(n) แน่ๆ ที่ของไพ่แต่ละใบก็จะถือเป็นหนึ่ง bucket (ตะกร้า) ซึ่งในกรณีแบบนี้ หนึ่ง bucket จะจุของได้หนึ่งอย่าง

ถ้าเรามีตัวเลขตั้งแต่ 1 ถึง m โดยมี n จำนวน ($n<m$) เราสามารถจัดเรียงจำนวนเหล่านี้ได้โดยทำอาร์เรย์ขนาด m ขึ้นมา ให้ตอนแรกทุกสมาชิกของอาร์เรย์นี้มีค่าเป็น 0 พอเราอ่านตัวเลขที่ยังไม่เรียงทีละตัว เจอตัวเลข k ก็ไปเพิ่มค่าใน a[k] ไปหนึ่ง ดังนั้นพอเราอ่านจบก็จะได้อาร์เรย์ที่บอกถึงการจัดเรียง ต่อจากนั้นเราก็อ่านอาร์เรย์นี้แล้วพิมพ์ตัวเลขออกมา เวลาที่ใช้ก็จะเป็น O(n) สำหรับการอ่านข้อมูลตอนแรก และ O(m) สำหรับการพิมพ์คำตอบ ดังนั้นวิธีนี้จะมีค่าเวลาโดยรวมเป็น O(n+m)

หรือเราอาจให้หนึ่ง bucket จุของได้หลายชิ้นที่ไม่เหมือนกันก็ได้ แล้วค่อยหาทางอื่นจัดเรียงของใน bucket อีกทีหนึ่ง เช่น การจัดเรียงข้อสอบของนักเรียน 50 คน ในตอนที่เก็บข้อสอบ ครูก็สามารถแบ่งนักเรียนออกเป็นห้ากลุ่มได้ (กลุ่มคือ bucket นั่นเอง) โดยดูจากหลักสิบของเลขที่สอบเป็นหลัก(คนสุดท้ายให้อยู่กลุ่มเดียวกับพวกที่มีเลขหลักสิบเป็น 4) ภายในกลุ่มเราสามารถ

ใช้ insertion sort จัดได้ หลังจากนั้นก็เอาแต่ละกลุ่มที่จัดแล้วมาเรียงกัน ก็จะได้คำตอบเลย ดังนั้นเวลาที่ใช้จะขึ้นอยู่กับวิธีที่ใช้จัดเรียงของภายในแต่ละ bucket

การจัดเรียงโดยอาศัยฐานเลข(Radix Sort)

ถือเป็น bucket sort แบบหนึ่ง นี่คือการทำ bucket sort หลายที โดยแต่ละทีเราจะใช้ส่วนหนึ่งของข้อมูลเป็นตัวกำหนด bucket ตัวอย่างเช่นการจัดเรียงตัวเลข(ต้องให้ตัวเลขมีจำนวนหลักเท่ากัน ดังนั้นจึงต้องเติมศูนย์ลงไปสำหรับตัวเลขที่มีจำนวนหลักน้อย)

143	002	013	328	165

หลักการคือ เราแบ่งเป็น bucket โดยใช้หลักหน่วยจัดเรียงก่อน(จะได้ bucket ของหลักหน่วยที่เป็น 0 ไปจนถึงหลักหน่วยที่เป็น 9) อ่านอาร์เรย์ข้างบนจากซ้ายไปขวา จะได้ bucket ต่างๆดังนี้ เรียงจากน้อยไปมาก(เฉพาะหลักหน่วย)

- 002
- 143,013
- 165
- 328

จะมีบาง bucket ที่ไม่มีสมาชิก ซึ่งเราไม่ต้องไปสนใจ จากนั้นรวมทุก bucket เป็นอาร์เรย์อีกทีหนึ่ง จะได้

002	143	013	165	328

จากนั้นก็แบ่งกลุ่มอีกที คราวนี้ใช้หลักสิบ จะได้

- 002
- 013
- 328
- 143
- 165

เอามารวมกันอีกเป็น

002	013	328	143	165

จากนั้นจัดกลุ่มตามหลักร้อย (นี่เป็นการจัดครั้งสุดท้าย เพราะจำนวนหลักมีแค่นี้)

- 002,013
- 143,165
- 328

เมื่อเอากลุ่มนี้มารวมเป็นอาร์เรย์ก็จะได้อาร์เรย์ที่มีการจัดเรียงจากน้อยไปมาก

สำหรับการโค้ดนั้น เราทำได้โดย ก่อนอื่นทำฟังก์ชั่นที่หาค่าของหลักที่ d ของตัวเลข n ก่อน ดังรูปที่ 2.22 (เรากำหนดให้d และ n มีค่าตั้งแต่ศูนย์ขึ้นไปเท่านั้น)

```
1:    public static int digitTh(int n, int d){
2:        if (d == 0)
3:            return n%10;
4:        else
5:            return digitTh(n/10,d-1);
6:    }
```

รูป 2.22 โค้ดของการหาค่าของหลักที่ d ของตัวเลข n

จะเห็นได้ว่าโค้ดในรูป 2.22 มี big O เป็น O(d)

รูปที่ 2.23 เป็นโค้ดของการแบ่งอาร์เรย์เป็น 10 bucket โดยใช้หลักที่ d เป็นตัวกำหนด จะเห็นว่าการแบ่งอาร์เรย์เป็นสิบ bucket นี้ ใช้เวลา O(n*d) โดย n เป็นขนาดของอาร์เรย์ นอกจากนี้จะเห็นว่ามีการใช้ Vector ซึ่งเป็นโครงสร้างข้อมูลใน java.util ซึ่งตัวของมันคืออาร์เรย์ของ object ซึ่งขยายขนาดได้เองโดยอัตโนมัติ โค้ดนี้สามารถถูกนำไปใช้ใน radix sort ได้ดังในรูป 2.24 ซึ่งในที่นี้ให้ size คือจำนวนของหลักเลข

จากรูป 2.24 เราจะได้ big O อย่างคร่าวๆเป็น O(n*0)+O(n)+O(2n)+…+ O(n*(size-1)) ซึ่งตามนิยามเราเลือกเอาค่ามากที่สุดมา คือ O(n*(size-1)) แต่ว่า size คือจำนวนของหลักเลข ซึ่งมักเป็นค่าน้อย ดังนั้นเราสามารถสรุปได้ว่า radix sort มีค่าเวลาเท่ากับ O(n)

```
1:    public static void bucketing(int data[], int d){
2:      int i,j,value;
3:
4:      //เราจะให้มี 10 bucket โดยแต่ละ bucket เป็น vector (อาร์เรย์ที่โตได้)
5:      Vector bucket[] = new Vector[10];
6:      for(j=0;j<10;j++)
7:        bucket[j] = new Vector();
8:
9:      //จับของใส่ bucket ตามหลัก
10:     int n =data.length;
11:     for(i=0;i<n;i++){
12:       value = data[i];
13:       j = digitTh(value,d);
14:       bucket[j].add(new Integer(value));
15:     }
16:
17:     //เอาใส่อาร์เรย์อีกโดยใส่ย้อน
18:     i=n;
19:     for(j=9;j>=0;j--){
20:       while(!bucket[j].isEmpty()){
21:         i--;
22:         value=
23:           ((Integer)bucket[j].remove()).intValue();
24:         data[i]=value;
25:       }
26:     }
27:   }
```

รูปที่ 2.23 โค้ดการแบ่งอาร์เรย์เป็น 10 bucket โดยใช้หลักที่ d เป็นตัวกำหนด

```
1:    public static void radixSort(int data[], int size){
2:      for(int j=0;j<size; j++)
3:        bucketing(data,j);
4:    }
```

รูป 2.24 radix sort

การจัดเรียงข้อมูลเชิงวัตถุในภาษาจาวา

หลักการก็เหมือนกับการเรียงตัวเลข จะต่างกันก็ตรงการเปรียบเทียบว่าค่าใดมากกว่าน้อยกว่า ในจาวานั้นมีสามวิธีในการเปรียบเทียบวัตถุ

ใช้เมธอด equals

คลาส Object นั้นมีเมธอด

```
public boolean equals(Object obj)
```

ถ้าเรามี x.equals(y); โค้ดนี้จะ return true เมื่อ x กับ y เป็นวัตถุเดียวกันเท่านั้น มิฉะนั้นจะ return false จะเห็นได้ว่าเมธอดนี้นำไปใช้งานได้ไม่ง่ายนัก เพราะเปรียบเทียบสิ่งของที่เป็นสิ่งเดียวกันเท่านั้น และยังบอกว่าน้อยกว่ามากกว่าไม่ได้เลย อย่างไรก็ตามยังมีหลายคลาสที่ overwrite เมธอดนี้เพื่อให้เทียบกับวัตถุอื่นที่มีค่าภายในเท่ากันได้ด้วย ยกตัวอย่างเช่น คลาส String

ใช้อินเตอร์เฟสคอมแพเรเบิ้ล (Comparable Interface)

Comparable Interface เป็นอินเตอร์เฟสที่ใช้กำหนดว่าวัตถุจะมีค่าเทียบกันอย่างไร โดยคลาสที่จะอิมพลีเม้นท์ตัวอินเตอร์เฟสนี้จะต้องมีเมธอด

```
public int compareTo(Object o)
```

เปรียบเทียบ this object กับวัตถุที่เป็น argument ว่ามีค่ามากน้อยกว่ากันเท่าไร เมธอดนี้ return ค่าลบถ้า this object มีค่าน้อยกว่าตัวที่เป็น argument นอกจากนี้ยัง return ค่าบวกถ้า this object มีค่ามากกว่าตัว argument และยังให้ค่า 0 ถ้า วัตถุที่เราเปรียบเทียบมีค่าเท่ากัน

โครงสร้างข้อมูล ลิสต์หรืออาร์เรย์ที่อิมพลีเม้นท์อินเตอร์เฟสนี้จะใช้เมธอด Collections.sort (และ Arrays.sort) ในการจัดเรียงข้อมูลได้โดยอัตโนมัติ

ใช้อินเตอร์เฟส คอมแพเรเตอร์(Comparator Interface)

โดยคลาสที่จะอิมพลีเม้นท์ตัวอินเตอร์เฟสนี้จะต้องมีเมธอดสองเมธอด เมธอดแรกคือ

```
public int compare(Object o1, Object o2)
```

เปรียบเทียบ o1 กับ o2 ว่ามีค่ามากน้อยกว่ากันเท่าไร เมธอดนี้ return ค่าลบถ้า o1 มีค่าน้อยกว่าo2 นอกจากนี้ยัง return ค่าบวกถ้า o1 มีค่ามากกว่า o2 และยังให้ค่า 0 ถ้า o1 กับ o2 มีค่าเท่ากัน

เมธอดที่สองคือ

```
public boolean equals(Object obj)
```

ซึ่งจะใช้ในการเปรียบเทียบ Comparator ตัวนี้(this) กับอีกตัวหนึ่ง เมธอดนี้ return true ก็ต่อเมื่อ obj เป็น Comparator ซึ่งมีการจัดเรียงวัตถุเหมือนกับ this

อินเตอร์เฟสนี้สามารถใช้ได้แบบเดียวกับ Comparator เลย แต่มักนิยมใช้เป็น argument ของคลาสต่างๆเพื่อที่จะสามารถปรับเปลี่ยนวิธีการเปรียบเทียบได้โดยง่าย อย่างเช่นเราสามารถสร้างโปรแกรมจัดเรียงคำที่เก็บในอาร์เรย์ขึ้นมาให้เรียกใช้ Comparator ได้สองแบบ แบบแรกเป็นการเปรียบเทียบจากน้อยไปมาก คือมี compare(“a”,”b”) ให้ค่าเป็น –1 ส่วนแบบที่สองเป็นการเปรียบเทียบจากมากไปน้อย คือมี compare(“a”,”b”) ให้ค่าเป็น 1 ดังนั้นจะเห็นว่าเราสามารถเลือกใช้ตัว Comparator เพื่อให้ได้ผลการจัดเรียงที่ต่างกัน

แบบฝึกหัด

1. จงหา best case และ average case ของ selection sort
2. ถ้าตัวเลขในอาร์เรย์ที่ต้องจัดเรียงมีค่าเท่ากันหมด selection sort จะกินเวลาเท่าใด
3. อะไรจะเกิดขึ้นถ้าใน radix sort เราจัดข้อมูลโดยใช้หลักร้อยก่อน แล้วค่อยมาหลักสิบและหลักหน่วย
4. เราสามารถเพิ่มความเร็วของ radix sort ที่ยกเป็นตัวอย่างได้อีกหรือไม่ อย่างไร จงเสนอแนวคิด
5. ถ้าการจัดเรียงข้อมูลไม่มีการสลับที่ของข้อมูลที่มีค่าเท่ากัน เราเรียกวิธีการจัดเรียงแบบนี้ว่าเป็นการจัดเรียงแบบ stable ถามว่ามีการจัดเรียงชนิดใดบ้างที่ถือว่า stable

6. จงเขียน insertion sort ขึ้นมาใหม่โดยใช้ recursion
7. จงแสดงการ merge sort 4,78,3,34,1,45,7,8
8. จงหาเวลาในการทำงานของ merge sort เมื่อ ข้อมูลจัดเรียงกันอยู่แล้ว เมื่อข้อมูลเรียงแบบสุ่ม และเมื่อข้อมูลเรียงจากมากไปน้อย
9. จงแสดงการ quick sort 4,78,3,34,1,45,7,8,10,20,15,24
10. ในการ partition ของ quick sort นั้น ถ้าเราใช้ตัวแรกของอาร์เรย์เป็นหลัก แล้วนับหาจำนวนที่น้อยกว่าเพื่อให้ได้ index ที่ตัวแรกของอาร์เรย์นั้นควรจะอยู่ จากนั้นสลับที่ตัวแรกในอาร์เรย์กับสิ่งที่อยู่ที่ index นั้น เพื่อให้ตัวแรกไปอยู่ตำแหน่งที่ถูกต้องของมัน ถามว่า วิธีนี้มีข้อผิดพลาดหรือไม่ อย่างไร
11. จงหาเวลาในการทำงานของ quick sort เมื่อ ข้อมูลจัดเรียงกันอยู่แล้ว เมื่อข้อมูลเรียงแบบสุ่ม และเมื่อข้อมูลเรียงจากมากไปน้อย
12. ถ้าเราเลือก pivot เป็นตัวแรกของอาร์เรย์ เวลาในข้อ 11 จะเป็นเท่าใด
13. ถ้าเราใช้ pivot ที่เป็นค่าสุ่ม เวลาในข้อ 11 จะเป็นเท่าใด
14. ถ้าเราเลือก pivot เป็นตัวกลางของอาร์เรย์ อาร์เรย์แบบไหนจะทำให้เวลาในการทำงานเป็น $O(n^2)$
15. ถ้าเราเลือก pivot เป็นตัวกลางของอาร์เรย์ ถามว่าอาร์เรย์ที่สมาชิกเรียงกันเรียบร้อยแล้วจะมีเวลาในการทำ quick sort นี้เป็นเท่าใด
16. ถ้าเราใช้วิธี partition ของ quick sort แบบที่ให้ i หยุดแต่ j ไม่หยุดเมื่อเจอค่าที่เท่ากับ pivot เราต้องแก้โค้ดในรูป 2.21 อย่างไรบ้าง และเวลาในการทำงานของ quick sort นี้จะเป็นเท่าใดเมื่อข้อมูลมีค่าเท่ากันหมด
17. ถ้าเราใช้วิธี partition ของ quick sort แบบที่ให้ i และ j ไม่หยุดเมื่อเจอค่าที่เท่ากับ pivot เราต้องแก้โค้ดในรูป 2.21 อย่างไรบ้าง และเวลาในการทำงานของ quick sort นี้จะเป็นเท่าใดเมื่อข้อมูลมีค่าเท่ากันหมด
18. จงเปลี่ยน โค้ดของ quick sort ในรูป 2.21 ให้ใช้ลูปแทน การเรียก recursive call ของ partition ข้างขวา

19. ถ้าในการทำ quick sort มีจำนวนซ้ำเยอะๆ เราอาจใช้วิธีแบ่งอาร์เรย์เป็นสามส่วน คือส่วนที่น้อยกว่า เท่ากัน และมากกว่า pivot จงแก้โค้ดของ quick sort ให้ทำตามนี้ และหาเวลาในการทำงานด้วย

20. จงเขียนโปรแกรมวิธีการจัดเรียงเลขเศษส่วน(ของจำนวนเต็ม) โดยให้มีเวลาเป็น O(n) เมื่อ n เป็นจำนวนเศษส่วน

21. ถ้าในอาร์เรย์มีแต่ค่า 0 กับ 1 เรียงกันมั่วๆ จงเรียงให้ 0 มาอยู่หน้า 1 ทั้งหมด อนุญาตให้ใช้พื้นที่ขนาดคงที่เพิ่มเติมเท่านั้น

22. สมมติในห้องเรียนมีนักเรียน n คนและมีการเลือกหัวหน้าห้อง จงออกแบบวิธีการหาคนที่ได้รับเลือก โดยใช้เวลาเป็น O(n log n)