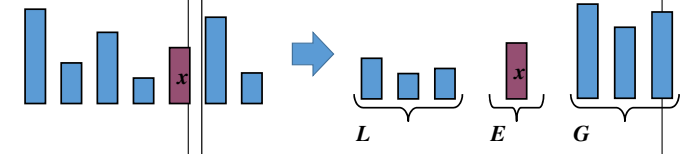


05506006 โครงสร้างข้อมูล

Lecture 3: Selection and Bubble Sort, Quick Sort and Radix Sort
ดร.รุ่งรัตน์ เวียงศรีพนาวัลย์

เค้าโครงการบรรยาย

- อัลกอริทึมการเรียงแบบเลือก
- อัลกอริทึมการเรียงแบบฟอง
- การเรียงข้อมูลแบบ Quick
 - หลักการทำงานของ Quick Sort
 - โค้ดของ QuickSort
 - การวิเคราะห์ QuickSort
- RadixSort
 - การทำงานของ RadixSort
 - การวิเคราะห์ RadixSort
- Bigoh



- การค้นหาข้อมูล
 - การค้นหาแบบเชิงเส้น (Linear Search)
 - การค้นหาแบบไบนารี (Binary Search)

อัลกอริทึมการเรียงแบบเลือก: ขั้นตอนปกติ

- ให้ค่าต่างๆ ถูกเก็บไว้ใน Array $a[]$
- Loop $n-1$ times
 - // Goal. To Find the value of position i i is a number of loop starting from 0
 - Begin Loop
 1. Go through all values from position $i+1$ to $n-1$ => Result: min and minlocation
 2. ถ้า minlocation ไม่ใช่ตำแหน่ง i ให้ ค่าของตำแหน่ง i มีค่าเท่ากับค่าในตำแหน่ง minlocation
 - End Loop

For $i = 0$ to $n-1$

- 1.
2. If minlocation is not equal i
 $a[i] = a[\text{minlocation}]$

อัลกอริทึมการเรียงแบบเลือก (2)

- ให้ค่าต่างๆ ถูกเก็บไว้ใน Array $a[]$
- Goal. To Find the value of position i i is a number of loop starting from 0
- for $i=0$ to $n-1$ (Loop $n-1$ times)
 - Begin Loop
 1. หาค่าที่น้อยที่สุดของค่าจากตำแหน่ง i ถึงตำแหน่งสุดท้าย Go through all values from position $i+1$ to n => Result: min and minlocation
 2. ถ้า minlocation ไม่ใช่ตำแหน่ง i ให้ ค่าของตำแหน่ง i มีค่าเท่ากับค่าในตำแหน่ง minlocation
 - End Loop

ให้ $\text{min} = a[i]$ และ $\text{minlocation} = j$
ให้ j เป็นตำแหน่งเริ่มต้น มีค่าเท่ากับ $i+1$
Loop จนกว่า j จะถึงตำแหน่งสุดท้าย
Begin Loop
ถ้าค่าในตำแหน่ง j น้อยกว่า min
ให้ $\text{min} = \text{ค่าในตำแหน่ง } j$ และ $\text{minlocation} = j$
เลื่อน j ไปตำแหน่งถัดไป
End Loop

อัลกอริทึมการเรียงแบบเลือก: Pseudocode

- ให้ค่าต่างๆ ถูกเก็บไว้ใน Array a[]
- for i=0 to n-1 (Loop n-1 times)

Begin Loop

- 1.1 ให้ min = a[i] และ minlocation = i
- 1.2 ให้ j เป็นตำแหน่งเริ่มต้น มีค่าเท่ากับ i+1
- 1.3 Loop จนกว่า j จะถึงตำแหน่งก่อนสุดท้าย

Begin Loop

- 1.3.1 ถ้าค่าในตำแหน่ง j น้อยกว่า min
ให้ min = ค่าในตำแหน่ง j และ minlocation = j
- 1.3.2 เลื่อน j ไปตำแหน่งถัดไป

End Loop

2. ถ้า minlocation ไม่ใช่ตำแหน่ง i ให้ สลับค่าของตำแหน่ง i กับ
ค่าในตำแหน่ง minlocation

End Loop

- for i=0 to n-1 (Loop n-1 times)

Begin Loop

- 1.1 min = a[i] minlocation = i
- 1.2 j = i+1 เป็นตำแหน่งเริ่มต้น มีค่าเท่ากับ i+1
- 1.3 Loop until j=n-1
 - Begin Loop
 - 1.3.1 if (a[j] < min)
min = a[j] and minlocation = j
 - 1.3.2 j = j+1
 - End Loop

2. if (minlocation is not equal i)
Swap(a[i],a[minlocation])

End Loop

5

อัลกอริทึมการเรียงแบบเลือก: Java Code

for i=0 to n-1 (Loop n-1 times)

Begin Loop

- 1.1 min = a[i] minlocation = i
- 1.2 j = i+1 เป็นตำแหน่งเริ่มต้น มีค่าเท่ากับ i+1
- 1.3 Loop until j=n-1
 - Begin Loop
 - 1.3.1 if (a[j] < min)
min = a[j] and minlocation = j
 - 1.3.2 j = j+1
 - End Loop

2. if (minlocation is not equal i)
Swap(a[i],a[minlocation])

End Loop

```
int i=0; int min,minlocation,temp;
for (i=0,i<=n-1,i++)
{ min =a[i]; minlocation =i;
  for (j=i+1;j<=n-1;j++)
    if (a[j]< min)
      { min = a[j];
        minlocation = j;}
  if (minlocation !=i)
  { temp=a[i];
    a[i]=a[minlocation];
    a[minlocation]=temp;
  }
}
```

6

อัลกอริทึมการเรียงแบบฟอง: ขั้นตอนปกติ

Goal. Each round, to Find the value of position n-i-1 i is a number of loop starting from 0

- ให้ค่าต่างๆ ถูกเก็บไว้ใน Array a[]

- Loop n-1 times

Begin Loop

Go through all values from position i to n-1
=> Result: max goes to position n-i-1

End Loop

Loop เริ่มต้นตั้งแต่ตำแหน่งแรก (0) จนถึงตำแหน่งที่ n-i-1

1. Compare a[j] and a[j+1]
if a[j+1]<a[j] Swap(a[j],a[j+1])

7

อัลกอริทึมการเรียงแบบฟอง: PseudoCode และ Java Code

Goal. Each round, find the value of position n-i-1 i is a number of loop starting from 0

- ให้ค่าต่างๆ ถูกเก็บไว้ใน Array a[]

- for i=1 to n-1

Begin Loop

for j=0 to n-i-2

Begin Loop

if (a[j+1]<a[j]) Swap(a[j],a[j+1])

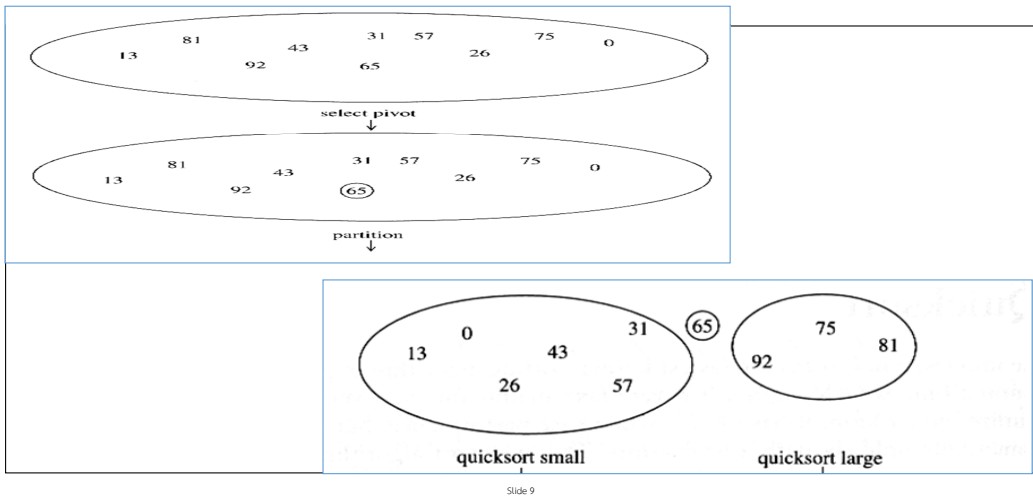
End Loop

End Loop

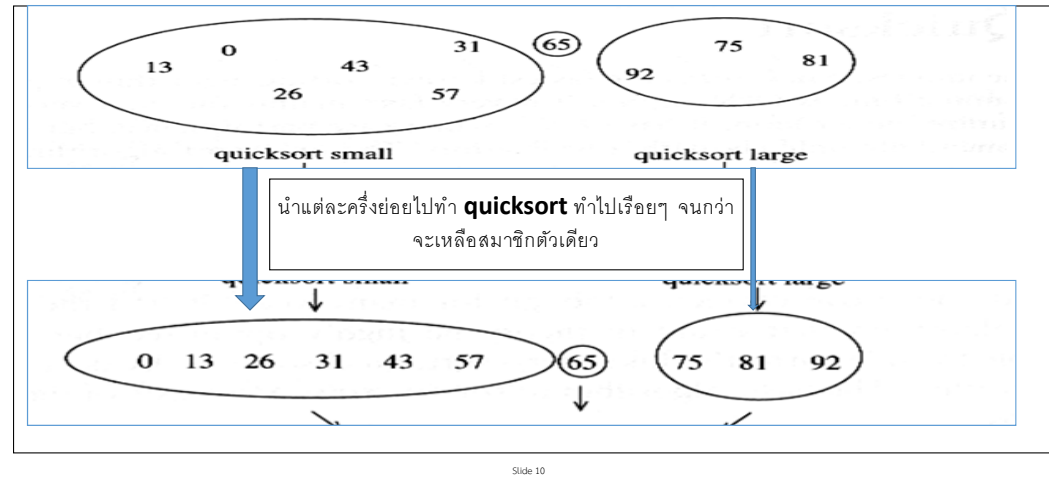
```
int i=0; int j,temp;
for (i=0,i<=n-1,i++)
{ for (j=0;j<=n-i-2;j++)
  { temp=a[j+1];
    a[j+1]=a[j];
    a[j]=temp;
  }
}
```

8

รูปแสดงการทำงานของ quicksort



รูปแสดงการทำงานของ quicksort (ต่อ)



Quick Sort

- เป็น sorting algorithm ที่มีประสิทธิภาพดีกว่า sorting algorithm แบบง่ายสามแบบที่นักศึกษาได้เรียนมา
- ค้นพบโดย C.A.R. Hoare
- เป็นตัวอย่างหนึ่งของ **Divide and Conquer** algorithm
- ประกอบไปด้วย 2 ขั้นตอนหลักใหญ่ๆ คือ
 - ขั้นตอนในการ Partition (Partition Phase)
 - **Divides** แบ่งงานเป็นครึ่งๆ
 - ขั้นตอนในการเรียง (Sort Phase)
 - **Conquers** the halves! ทำการเรียงในครึ่งนั้น

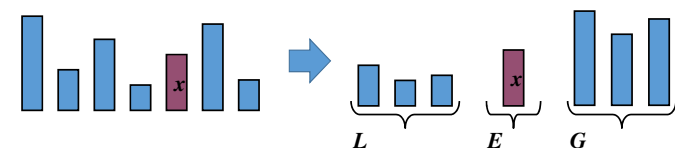
Slide 11

หลักการของ Quick Sort

$\leq \text{pivot}$	pivot	$> \text{pivot}$
---------------------	-------	------------------

ขั้นตอนในการ Partition

- เลือก ค่าหนึ่งใน list ขึ้นมา ให้ชื่อค่านี้ว่า pivot (ในรูป
- ผลลัพธ์ของการทำ Quick Sort ในขั้นตอนนี้คือ
 - ค่าที่เลือกมาเป็น pivot จะอยู่ในตำแหน่งที่ถูกต้องใน list ดังรูป
 - ค่าที่อยู่ทางซ้ายของ pivot จะน้อยกว่า ค่า pivot
 - ค่าที่อยู่ทางขวาของ pivot จะมากกว่า ค่า pivot



Slide 12

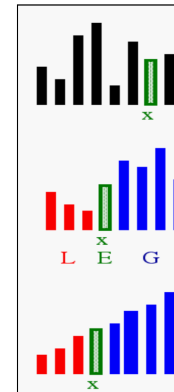
หลักการของ Quick Sort

- ได้ข้อมูลย่อย 2 ชุด ครึ่งซ้ายของ pivot และ ครึ่งขวาของ pivot
- แต่ละชุด นำไปทำ Partition และทำตามขั้นตอนข้างบน
 - นำแต่ละครั้งมาทำการ Partition ไปเรื่อยๆ จนกว่าจะหมด
 - s



Slide 13

สรุปหลักการของ Quick Sort



- 1) Select: เลือก pivot x (การเลือก pivot มีหลายแบบให้นักศึกษาสังเกตว่าในตัวอย่าง เลือกอย่างไร)
- 2) Divide: จัดการเรียงจนครบ 1 รอบ ผลลัพธ์ x จะมาอยู่ตำแหน่งที่ต้องการอยู่ โดยสมาชิกทางซ้ายของ x จะน้อยกว่าหรือเท่ากับ x แต่ สมาชิกทางขวาจะมากกว่า x
- 3) Recurse and Conquer: แต่ละครั้งย่อย (L และ R นำไปทำแบบเดิม (แบ่งครึ่งหาตำแหน่ง pivot ย่อย) ทำไปเรื่อยๆ จนกว่าแต่ละส่วนย่อยจะเหลือสมาชิกตัวเดียวการทำซ้ำๆ แบบนี้ทางคอมพิวเตอร์เรียกว่า Recursive

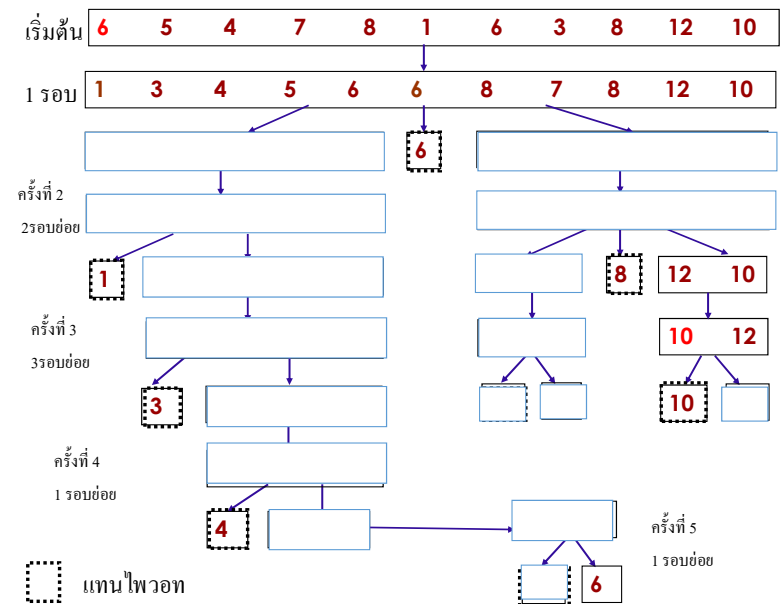
คำถาม แล้วแต่ละครั้งย่อยไปทำอย่างไรต่อ ?????

14

ให้นักศึกษาสังเกตการทำงานของ Quick Sort ในรอบต่อไปนี้
ให้ Pivot คือ 6

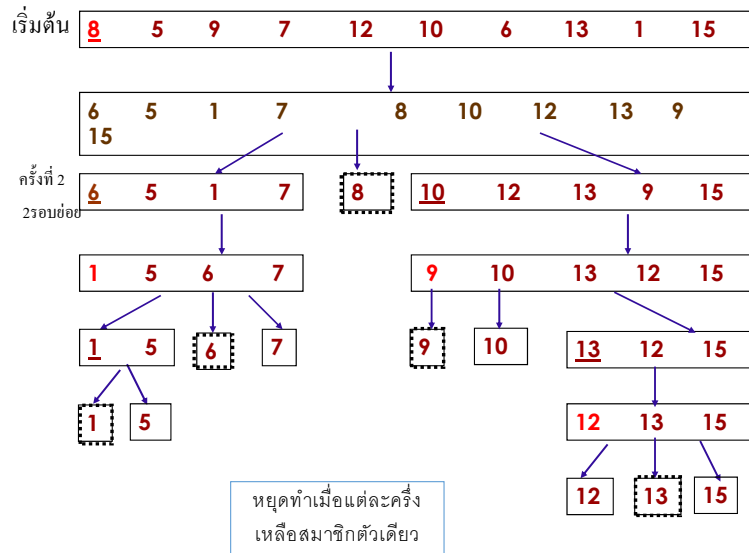


15



แทนไพวอท

ครั้งที่ 5
1 รอบย่อย



17

อัลกอริทึม QuickSort (Partition) กรณีใช้อาร์เรย์

- 1. ถ้า อาร์เรย์มีสมาชิกตัวเดียว **หยุด**
- 2. ถ้าไม่
 - 2.1 สุ่ม Pivot
 - 2.2 ทำการแบ่งครึ่งอาร์เรย์เป็นสองส่วนย่อย
 - ส่วนซ้ายน้อยกว่า หรือเท่ากับ pivot
 - ส่วนขวามากกว่า pivot
 - 2.3 ทำการ Quicksort sub-array ย่อยนั้น

18

Quicksort

Implementation

```
public static void quicksort(int[] a, int low, int high)
```

```
{
    int pivot;
    /* Termination condition */
    if ( high > low )
    {
        pivot = partition( a, low, high );
        quicksort( a, low, pivot-1 );
        quicksort( a, pivot+1, high );
    }
}
```

หยุด - high < = low

Divide

Conquer

19

Quicksort - Partition

```
public static int partition( int[] a, int low, int high )
{
    int left, right, pivot;
    left = pivot = low;
    right = high;
    int pivot_item = a[pivot]; // กำหนดให้ pivot เป็นตำแหน่งแรก
    while( left < right ) {
        /* เคลื่อนไปทางซ้ายขณะที่มีค่าน้อยกว่า หรือเท่ากับ pivot_item */
        while((a[left] <= pivot_item) && (left < high)) left++;
        /* เคลื่อนไปทางขวาขณะที่มีค่ามากกว่า pivot_item */
        while((a[right] > pivot_item) && (right > low)) right--;
        if (left < right) Swap(a, left, right);
    }
    a[pivot] = a[right];
    a[right] = pivot_item;
    return right;
}
```



20

Quicksort - Partition

```
public static int partition( int[] a, int low, int high )
{
    int left, right, pivot;
    left = pivot = low;
    right = high;
    int pivot_item = a[pivot];

    while( left < right ) {
        /* เคลื่อนไปทางซ้ายจนกระทั่งมีค่าน้อยกว่า หรือเท่ากับ pivot_item */
        while((a[left] <= pivot_item) && (left < high)) left++;
        /* เคลื่อนไปทางขวาจนกระทั่งมีค่ามากกว่า pivot_item */
        while((a[right] > pivot_item) && (right > low)) right--;
        if( left < right ) swap(a, left, right);
    }
    a[pivot] = a[right];
    a[right] = pivot_item;
    return right;
}
```

เลือก ตัว **left** และ **right** จนมันสวนทางกัน

low pivot: 23 high

21

Quicksort - Partition

```
public static int partition( int[] a, int low, int high )
{
    int left, right, pivot;
    left = pivot = low;
    right = high;
    int pivot_item = a[pivot];

    while( left < right ) {
        /* เคลื่อนไปทางซ้ายจนกระทั่งมีค่าน้อยกว่า หรือเท่ากับ pivot_item */
        while((a[left] <= pivot_item) && (left < high)) left++;
        /* เคลื่อนไปทางขวาจนกระทั่งมีค่ามากกว่า pivot_item */
        while((a[right] > pivot_item) && (right > low)) right--;
        if( left < right ) Swap(a, left, right);
    }
    a[pivot] = a[right];
    a[right] = pivot_item;
    return right;
}
```

สลับที่ ข้อมูลที่อยู่ตรงตำแหน่งของ **left** และ **right**

low pivot: 23 high

22

```
public static int partition( int[] a, int low, int high )
{
    int left, right, pivot;
    left = pivot = low;
    right = high;
    int pivot_item = a[pivot];

    while( left < right ) {
        /* เคลื่อนไปทางซ้ายจนกระทั่งมีค่าน้อยกว่า หรือเท่ากับ pivot_item */
        while((a[left] <= pivot_item) && (left < high)) left++;
        /* เคลื่อนไปทางขวาจนกระทั่งมีค่ามากกว่า pivot_item */
        while((a[right] > pivot_item) && (right > low)) right--;
        if( left < right ) Swap(a, left, right);
    }
    a[pivot] = a[right];
    a[right] = pivot_item;
    return right;
}
```

left และ **right** สลับที่กันเรียบร้อยแล้ว

low pivot: 23 high

23

Quicksort - Partition

```
public static int partition( int[] a, int low, int high )
{
    int left, right, pivot;
    left = pivot = low;
    right = high;
    int pivot_item = a[pivot];

    while( left < right ) {
        /* เคลื่อนไปทางซ้ายจนกระทั่งมีค่าน้อยกว่า หรือเท่ากับ pivot_item */
        while((a[left] <= pivot_item) && (left < high)) left++;
        /* เคลื่อนไปทางขวาจนกระทั่งมีค่ามากกว่า pivot_item */
        while((a[right] > pivot_item) && (right > low)) right--;
        if( left < right ) Swap(a, left, right);
    }
    a[pivot] = a[right];
    a[right] = pivot_item;
    return right;
}
```

สุดท้ายแล้วให้สลับที่ **item** ที่อยู่ที่ตำแหน่ง **pivot** กับ **item** ที่อยู่ในตำแหน่ง **right**

low pivot: 23 high

24

Quicksort - Partition

```
public static int partition( int[] a, int low, int high )
```

```
{ int left, right, pivot;
```

```
left = pivot = low;
```

```
right = high;
```

```
int pivot_item = a[pivot];
```

```
while( left < right ) {
```

```
/* เคลื่อนย้ายองค์ประกอบที่มีค่าน้อยกว่าหรือเท่ากับ pivot_item ไปด้านซ้ายของ pivot_item และองค์ประกอบที่มีค่ามากกว่าหรือเท่ากับ pivot_item ไปด้านขวาของ pivot_item */
```

```
while( a[right] > pivot_item ) right--;
```

```
if( left < right ) Swap(a, left, right);
```

```
left++;
```

```
while( a[left] < pivot_item ) left++;
```

```
if( left < right ) Swap(a, left, right);
```

```
right--;
```

```
Swap(a, left, right);
```

```
return right;
```

```
}
```

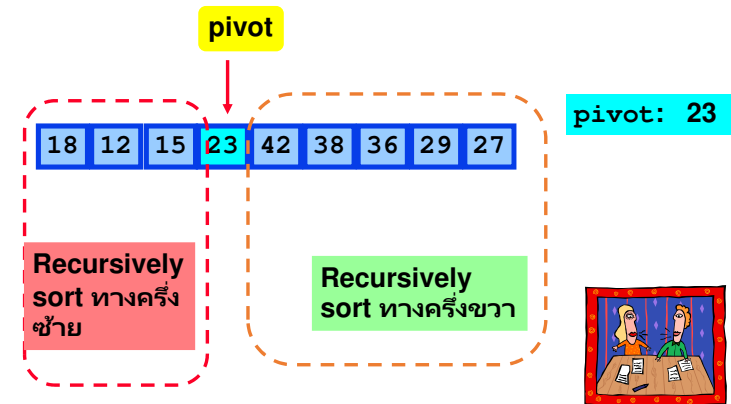


pivot: 23

Return right เป็นตำแหน่งของ pivot

25

Quicksort - Conquer



26

Best Case Analysis

- ในกรณีที่ดีที่สุด คือ การที่ไพวอตอยู่ ณ ตำแหน่งกึ่งกลางของอาร์เรย์เสมอ
- ดังนั้นถ้าอาร์เรย์ A มีขนาดเป็น n
 - เมื่อผ่านไป 1 รอบ จะถูกแบ่งครึ่งเป็น 2 ส่วน
 - เมื่อผ่านไป 2 รอบ จะถูกแบ่งเป็น 4 อาร์เรย์ย่อย
 - ทำไปเรื่อยๆ จนไม่สามารถแบ่งย่อยได้อีก
- ให้ m คือ จำนวนครั้งของการแบ่งครึ่ง
 - $m = \log_2 n$ ดังนั้นจำนวนครั้งของการเปรียบเทียบทั้งหมด

- ดังนั้นจำนวนครั้งของการเปรียบเทียบทั้งหมด

$$n + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{4}\right) + \dots + n\left(\frac{n}{n}\right)$$

หรือ

$$= n + n + n + \dots + n \quad (\text{ทั้งหมด } m \text{ ครั้ง})$$

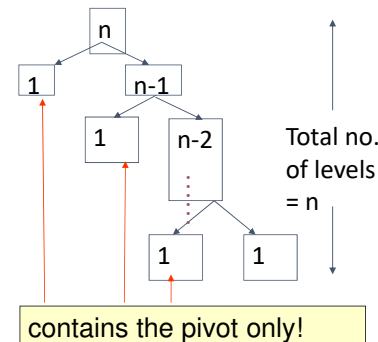
$$= nm = n \log_2 n = O(n \log n)$$

แต่ละ ครั้ง (level) จะเปรียบเทียบประมาณ n หรือ น้อยกว่า

Best Case = $O(n \log n)$

Slide 27

Worst Case Analysis



ในทางปฏิบัติโอกาสที่จะเกิด worst case มีน้อย และโดยเฉลี่ย (average case) มีทั้งแบ่งดีและแบ่งแย

Average time is $O(n \log n)$

- ในกรณีที่แย่ที่สุด คือ ค่าไพวอตที่ได้ทุกครั้งเป็นค่าที่น้อยที่สุดหรือเป็นค่าที่มากที่สุด
- ดังนั้นเมื่อผ่านการทำไปหนึ่งรอบ อาร์เรย์จะไม่ถูกแบ่งเป็น 2 ส่วน แต่จะลดขนาดลงเป็น n-1
- และเมื่อผ่านการทำไป 2 รอบ อาร์เรย์จะมีขนาดลดลงเป็น n-2 ดังนั้นจึงทำเป็นจำนวน n รอบ
- ดังนั้นจำนวนครั้งของการเปรียบเทียบทั้งหมด

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

Slide 28

สิ่งที่น่าสนใจสำหรับ Quicksort

- สำหรับอาร์เรย์ที่มีขนาดเล็ก ($N \leq 20$), quicksort ทำงานได้ไม่เลวนัก
- ควรใช้ insertion sort มากกว่า
- Sort ที่ให้ Order $n \log n$ ในกรณี Best Case หรือ Average Case มีอีกวิธีหนึ่งเรียกว่า Mergesort
- สิ่งที่น่าสนใจ คือ
 - Merge sort
 - จำนวนครั้งในการเปรียบเทียบน้อยกว่า QuickSort
 - แต่มีจำนวนครั้งในการสลับที่มากกว่ามาก
- ภาษาจาวาเป็นภาษาที่ใช้เวลาในการเปรียบเทียบต่อหนึ่งครั้งนาน แต่ใช้เวลาในการก๊อปปี้ค่าไปใส่ (ซึ่งการสลับที่ต้องใช้) ไม่ค่อยนาน ดังนั้นโดยปกติ ไบเบรารีภาษาจาวาในการเรียงข้อมูลจะใช้ Mergesort
- ภาษา C++ เป็นภาษาที่การก๊อปปี้ใช้เวลาไม่นาน แต่การเปรียบเทียบใช้เวลาไม่บ่อยนัก ดังนั้นไลบรารีภาษา C++ มักจะใช้ Quicksort

Radix Sort

- การเรียงลำดับแบบเรดิก จะทำการแยกค่าที่ต้องการเรียงลำดับออกเป็นตัวเลข หรือ ตัวอักษร และการจัดเรียงข้อมูลใหม่ตามค่าของตัวเลขหรือตัวอักษรที่แยกออกมา
- เช่น ถ้าเป็น ตัวเลข จะมี 0-9 ตัวอักษรจะมี a-z
- ทำการจัดเรียงจากตัวสุดท้ายก่อน เช่น ถ้าเป็น ตัวเลข หลักหน่วย หลักสิบ หลักร้อย
- เช่น 36 9 0 25 1 49 64 16 81 4

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	61			4		16			49

0	1	2	3	4	5	6	7	8	9
0	1								
0	1							81	

Slide 31

Median of Three

P	18	5	9	7	2	15	6	13	1	10	L
	F				M						

- ประสิทธิภาพของการเรียงลำดับแบบควิกสามารถปรับปรุง ได้โดยการเลือกค่าไพวอทที่ดี
- วิธีหนึ่งที่ย่และสามารถหลีกเลี่ยงการได้ค่าไพวอทที่เป็นค่าที่น้อยที่สุดและค่าที่มากที่สุดและเป็นที่ยอมรับ คือ วิธี Median of Three ซึ่งมีวิธีการดังนี้
 1. เลือกค่า 3 ค่าจากข้อมูลที่ต้องการทำพาร์ทิชัน
 - ซึ่งได้แก่ค่าที่อยู่ ณ ตำแหน่งแรก ตำแหน่งกึ่งกลาง และตำแหน่งสุดท้าย
 2. เรียงลำดับค่าทั้ง 3 เลือกค่าไพวอทเป็นค่ากลางของค่าทั้ง 3

F แทนดัชนีของข้อมูลตำแหน่งแรก

L แทนดัชนีของข้อมูลตำแหน่งสุดท้าย

M แทนดัชนีของข้อมูลตำแหน่งกึ่งกลาง $= (F+L)/2$

P แทนดัชนีของค่าไพวอท

$2 < 10 < 18$ ดังนั้น จะได้ 10 เป็นค่าไพวอท

Slide 30

Radix Sort:

ให้เรียงลำดับเลขต่อไปนี้ 10 1234 9 7234 67 9181 733 197 7 3

- Phase 1 10 1234 9 7234 67 9181 733 197 7 3

0	1	2	3	4	5	6	7	8	9
10	9181		733	1234			67		9
			3	7234			197		
							7		

- Phase 2 10 9181 733 3 1234 7234 67 197 7 9

0	1	2	3	4	5	6	7	8	9
3	10		733			67		9181	197
7			1234						
9			7234						



32

- Phase 3 3 7 9 10 733 1234 7234 67 9181 197

0	1	2	3	4	5	6	7	8	9
3	9181	1234					733		
7	197	7234							
9									
10									
67									

- Phase 4 3 7 9 10 67 9181 197 1234 7234 733

0	1	2	3	4	5	6	7	8	9
3	1234						7234		9181
7									
9									
10									
67									
197									
733									

ผลลัพธ์ 3 7 9 10 67 197 733 7234 9181



33

การวิเคราะห์ Radix Sort ของตัวอย่าง

1.Phase 1

- Create m bins $O(m)$
- Allocate n items $O(n)$

2.Phase 2

- Create m bins $O(m)$
- Allocate n items $O(n)$

3.Final

- Link m bins $O(m)$

รวมทุก Step

- $O(m) + O(n) + O(m) + O(n) + O(m)$

Total

- $O(3m+2n) = O(m+n) = O(n)$ for $m \ll n$

Slide 34

ปัญหาของ Radix sort

- แม้ว่า Radix Sort จะมี Big Oh เป็น linear order
- แต่จะต้องมีการใช้เนื้อที่เพิ่มขึ้นมาก (จำนวน phase * จำนวนสมาชิกที่ต้องเรียง)
- ถ้ามีเนื้อที่จำกัดทำไม่ได้

Slide 35

สรุป Big-Oh ของการเรียงข้อมูล

	Worst Case	Best Case	ต้องการเนื้อที่ เพิ่มในการ จัดเรียง
Selection Sort	$O(n^2)$	$O(n^2)$	No
Insertion Sort	$O(n^2)$	$O(n)$	No
Bubble Sort	$O(n^2)$	$O(n^2)$	No
Bubble Sort 2 (improved with flag)	$O(n^2)$	$O(n)$	No
Quick Sort	$O(n^2)$	$O(n \log n)$	No
Radix Sort	$O(n)$	$O(n)$	Yes

Notes: 1. $O(n)$ for Radix Sort is due to non-comparison based sorting.

2. $O(n \log n)$ is the best possible for comparison based

ปัญหาการค้นหาข้อมูล

- นอกจากการเรียงข้อมูลแล้ว อีกปัญหาที่ โครงสร้างข้อมูลและอัลกอริทึมที่ดีจะช่วยให้การทำงานมีประสิทธิภาพมากขึ้นได้แก่ ปัญหาเรื่องการค้นหาข้อมูล
- ในที่นี้จะยกตัวอย่างการค้นหา 2 วิธีได้แก่
 - 1. การค้นหาแบบเชิงเส้น (Linear Search)
 - 2. การค้นหาแบบไบนารี (Binary Search)
- โดยใช้โครงสร้างข้อมูลแบบเดียวกันในการเก็บข้อมูลคือ อาร์เรย์
- แต่มีวิธีการเข้าถึงข้อมูลต่างกัน
- จะทำการวัดประสิทธิภาพโดยใช้ Big Oh
- นักศึกษาต้องสามารถตอบได้ว่า ในกรณีใด ควรใช้ Binary Search และ ในกรณีใดควรใช้ Linear Search

37

ปัญหา: การค้นหาข้อมูลแบบเชิงเส้น (Linear Search)

- เป็นวิธีการค้นหาแบบง่าย
- และสามารถใช้ได้กับข้อมูลที่ไม่มีการเรียงลำดับ
- สามารถใช้ในการค้นหาข้อมูลในอาร์เรย์ ลิสต์ และแฟ้มข้อมูล
- ก่อนค้นหา
 - ข้อมูลถูกเก็บไว้ในที่เก็บข้อมูลแบบไม่เรียงลำดับ
- วิธีการค้นหาโดย
 - ทำการเปรียบเทียบค่าที่ต้องการกับข้อมูลในตำแหน่งแรก
 - และเปรียบเทียบในตำแหน่งถัดไปเรื่อย ๆ จนพบข้อมูลที่ต้องการ
 - หรือเปรียบเทียบไปจนถึงข้อมูลในตำแหน่งสุดท้าย แต่ไม่พบข้อมูลที่ต้องการ

การค้นหาแบบเชิงเส้น (Linear Search)

	ข้อมูล
0	67
1	82
2	32
3	57
4	62
5	44

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

	ข้อมูล
0	67
1	82
2	32
3	57
4	62
5	44
6	345
7	3
8	75
9	26

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

	ข้อมูล
0	67
1	82
2	32
3	57
4	62
5	44
6
96	23
97	752
98	276
99	564

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

ข้อมูลไม่เรียง

การค้นหาแบบเชิงเส้น (Linear Search)

	ข้อมูล
0	10
1	20
2	32
3	57
4	62
5	40
6	50

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

	ข้อมูล
0	10
1	20
2	32
3	40
4	62
5	78
6	90
7	103
8	154
9	226

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

	ข้อมูล
0	10
1	20
2	30
3	40
4	62
5	78
6
96	523
97	552
98	576
99	664

- กรณีดีที่สุด :
- กรณีแย่มากที่สุด :
- กรณีเฉลี่ย :

ข้อมูลไม่เรียง

สรุปการค้นหาแบบเชิงเส้น

- ข้อมูลจะเรียงหรือไม่ แต่ละกรณี

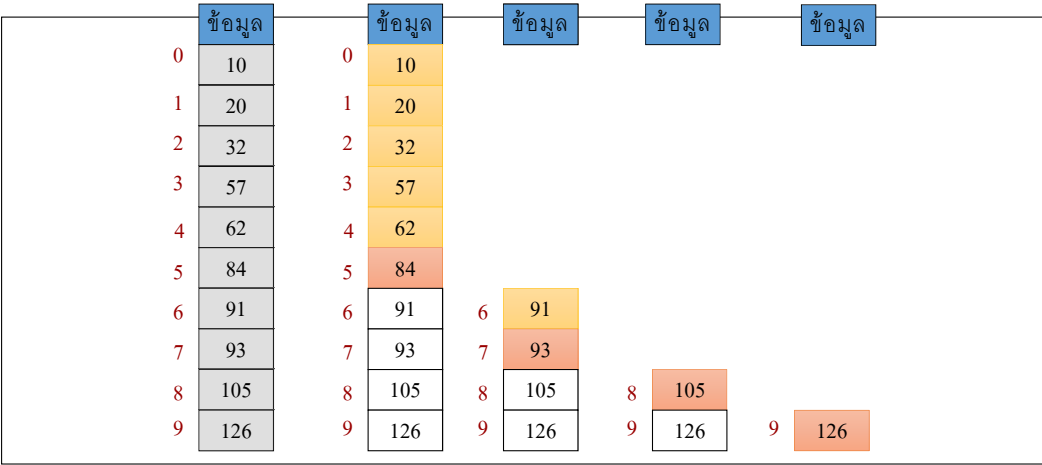
```
public int LinearSearch(int array[], int key )
{
    for ( int n = 0; n < array.length; n++ )
        if ( array[ n ] == key )
            return n; // found
    return -1; // not found
}
```

	เปรียบเทียบ (ครั้ง)	Big Oh
กรณีที่ดีที่สุด		O(...)
กรณีแย่ที่สุด		O(...)
กรณีเฉลี่ย		O(...)

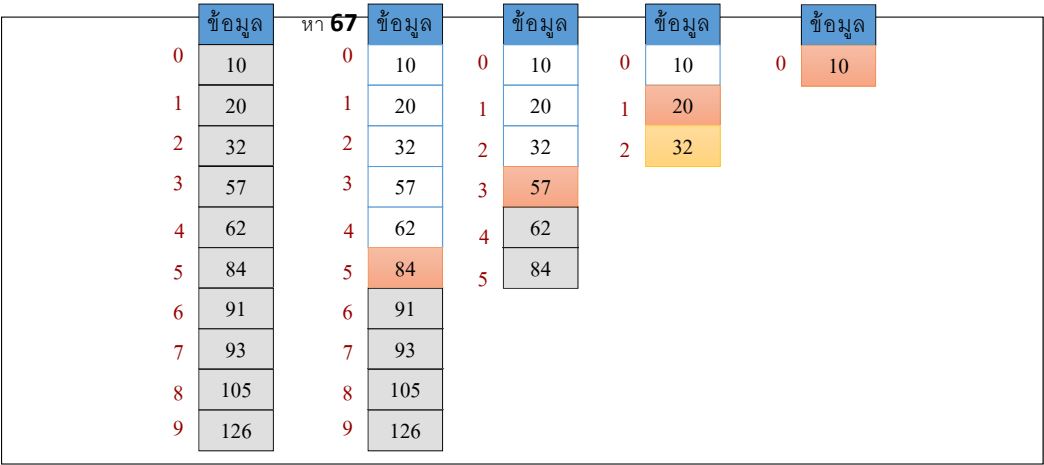
การค้นหาแบบไบนารี (Binary Search)

- ใช้ได้กับข้อมูลที่มีการเรียงลำดับเท่านั้น
- โดยเริ่มจาก
 - การเปรียบเทียบข้อมูลที่ต้องการกับข้อมูลในตำแหน่งกึ่งกลาง
 - หากค่าที่ต้องการค้นหามีค่ามากกว่าให้ตัดข้อมูลในครึ่งแรกออกจากการพิจารณา
 - หากมีค่าน้อยกว่าให้ตัดข้อมูลในครึ่งหลังออกจากการพิจารณา
 - ทำอย่างนี้จนกว่าจะเจอ หรือจนกว่าจะไม่เหลือจำนวนให้ตัด (ไม่เจอ)

การค้นหาแบบเชิงเส้น (Linear Search) หา126

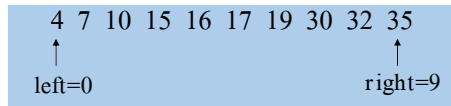


การค้นหาแบบไบนารี(Binary Search)หา 10



Binary Search


1. ตั้งค่าพอยน์เตอร์ 2 ค่า คือ ค่าหนึ่งสำหรับหัวแถว อีกค่าสำหรับปลายแถว



2. คำนวณค่ากึ่งกลาง mid

$$\text{mid} = (\text{left} + \text{right}) / 2 = (0 + 9) / 2 = 9 / 2 \Rightarrow 4$$

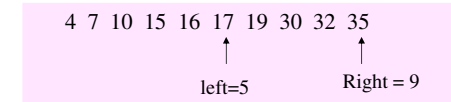
เราเลือกค่า mid ให้เท่ากับ 4 ดังนั้น ค่ากึ่งกลางได้แก่



Example: Binary Search

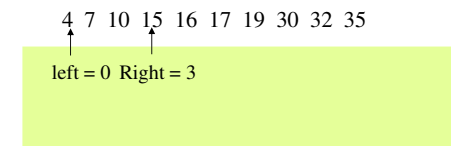
3. ถ้าค่า $k > k_{\text{mid}}$ แสดงว่าค่าที่ต้องการหาอยู่ครึ่งหลัง

ทำการเลื่อน left



- ถ้าค่า $k < k_{\text{mid}}$ แสดงว่าค่าที่ต้องการหาอยู่ครึ่งแรก

ทำการเลื่อน right



Example: Binary Search

4. สรุปกรณีทั่วไป

If $k > k_{\text{mid}}$ then left = mid+1;
else right = mid-1;

4 7 10 15 16 17 19 30 32 35

5. เมื่อ left > right แสดงว่าข้อมูลไม่อยู่ในตาราง จะหยุดการค้นหา

สมมติหาค่า $k = 17$

left = 0, right = 9 ได้ mid = 4 รู้ว่า k อยู่ครึ่งหลัง
left = 5, right = 9 ได้ mid = 7 รู้ว่า k อยู่ครึ่งแรก
left = 5, right = 6 ได้ mid = 5 รู้ว่า $k = k_{\text{mid}}$



การค้นหาแบบไบนารี (Binary Search)

```
public int binarySearch( int array[], int key )
{
    int low = 0; // low subscript
    int high = array.length - 1; // high subscript
    int middle; // middle subscript

    while ( low <= high ) {
        middle = ( low + high ) / 2;
        if ( key == array[ middle ] ) // match
            return middle;
        else if ( key < array[ middle ] )
            high = middle - 1; // search low end of array
        else
            low = middle + 1; // search high end of array
    }
    return -1; // searchKey not found
}
```

สรุป การค้นหาแบบไบนารี

- ข้อมูลต้องทำการเรียงก่อน จึงจะทำได้

	เปรียบเทียบ(ครั้ง)	Big Oh
กรณีดีที่สุด		$O(n)$
กรณีแย่ที่สุด*		$O(n^2)$
กรณีเฉลี่ย		$O(n^2)$

*กรณีแย่ที่สุด เกิดในกรณี เจอบที่ตัวสุดท้ายหรือ ไม่เจอ

แบบฝึกหัด

จงแสดงขั้นตอนในการเรียงลำดับข้อมูลจากน้อยไปมากโดยใช้ Quick sort และ Radix sort จากข้อมูลต่อไปนี้

1. 45, 18, 71, 54, 81, 49, 31, 61, 21, 95, 37, 96

Reference:

- สไลด์ประกอบการสอนวิชา SE 311 Algorithms Design and Analysis โดย ผศ.ดร. สมศรี บัณฑิตวิไล