KMUTNB
Department of Computer and Information Science

อัลกอริทึมการเรียงลำดับและค้นหาข้อมูล

Sorting & Searching algorithms

อ.ดร.ลือพล พิพานเมฆาภรณ์

luepol.p@sci.kmutnb.ac.th

Content

- Searching algorithms
  - Sequential Search algorithm
  - Binary Search / interpolation Search
- Sorting algorithms
  - Comparison-based sort algorithms
    - Bubble Sort / Selection Sort / Insertion Sort / Shell sort
  - Partition-based sort algorithms
    - Merge Sort / Quick Sort
  - Memory-based sort algorithms
    - Counting Sort / Bucket sort

นิยามของปัญหา Search และ Sort

กำหนดให้รายการเลขจำนวนเต็ม n จำนวน x = {x1, x2,...., xn} และเลขจำนวนเต็ม k

- วัตถุประสงค์ของการค้นหาข้อมูล คือการค้นหาตำแหน่ง i โดยที่ 1<= i <= n ซึ่ง xi มีค่าเท่ากับ k

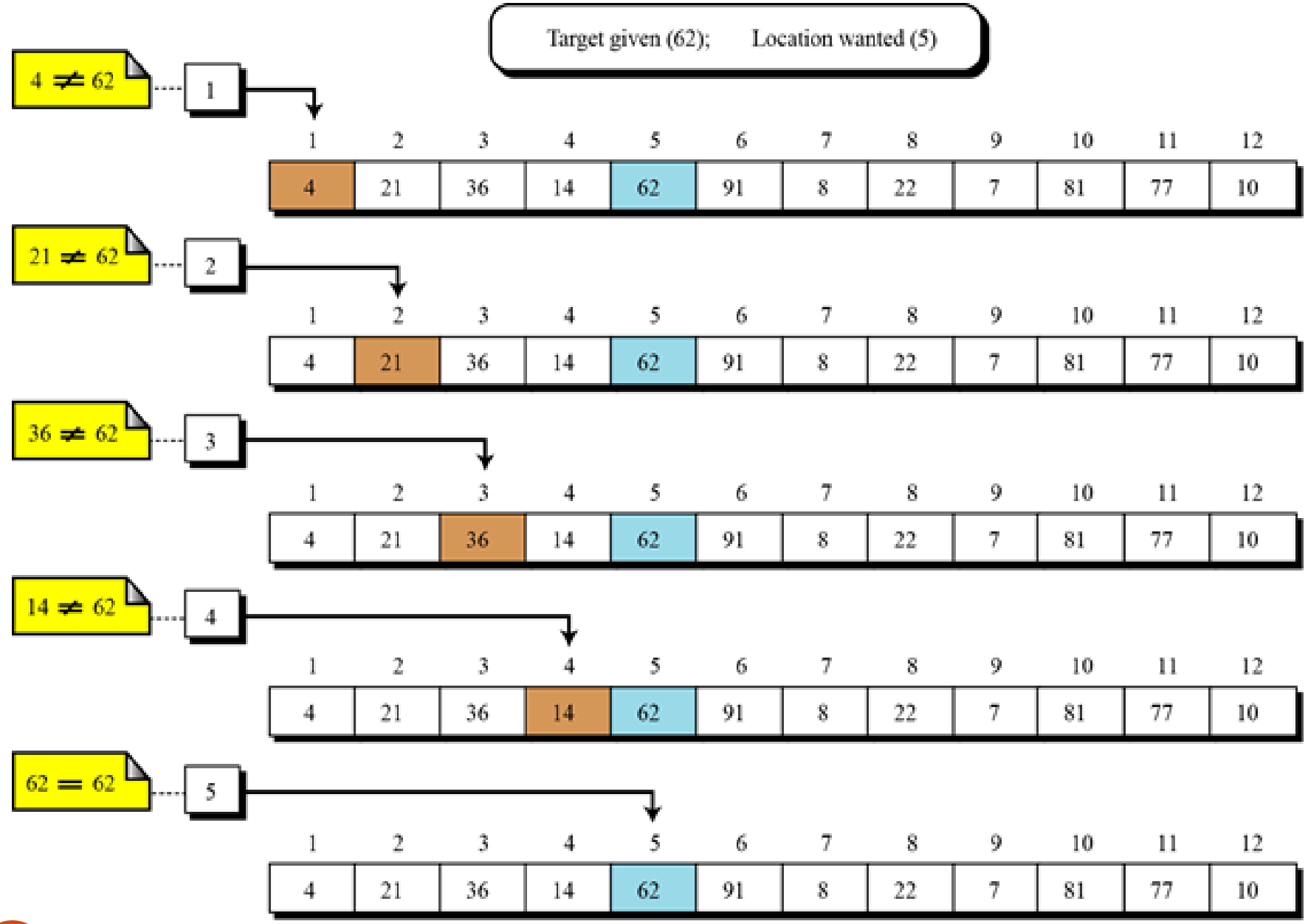
- วัตถุประสงค์ของการจัดเรียงข้อมูล คือ สำหรับทุกๆ ตำแหน่ง i และ j โดยที่ 1 <= i, j <= n
  - xi >= xj กรณีเรียงลำดับจากมากไปน้อย (Descending order)
  - xi <= xj กรณีเรียงลำดับจากน้อยไปมาก (Ascending order)

Sequential Search

- เรียกอีกอย่างว่า Linear search จะเปรียบเทียบคีย์ข้อมูล k กับข้อมูลที่อยู่ในอาร์เรย์ A ทีละตัว เริ่มต้นจากตัวแรกถึงตัวสุดท้าย (ในภาษาซี สมาชิกตัวแรกของอาร์เรย์คือ A[0] ดังนั้น A[n-1] ก็คือสมาชิกตัวสุดท้าย) ซึ่งหากพบว่าคีย์ข้อมูลใดตรงกับ k ก็จะทำการส่งค่าตำแหน่งที่พบ (index) c กลับไป แต่หากไม่พบก็จะทำการส่งค่ากลับคือ -1

```
int sequential_search(int A[], int k ,int n) {
  int i = 0;
  while((A[i] != k)&& (i < n))
     i++;

   if(i < n)
       return i;
   else
       return -1;
}
```


Target given (62); Location wanted (5)
4 ≠ 62
1
21 ≠ 62
2
36 ≠ 62
3
14 ≠ 62
4
62 = 62
5
1 2 3 4 5 6 7 8 9 10 11 12
4 21 36 14 62 91 8 22 7 81 77 10

Binary search algorithm

- ข้อเสียของการค้นหาแบบ sequential search ก็คือมักทำงานช้า โดยเฉพาะอย่างยิ่งเมื่อจำนวนอินพุตมาก
- เพื่อที่จะแก้ไขข้อเสียดังกล่าว binary search ถูกออกแบบมาโดยมีเงื่อนไขเริ่มต้นที่ว่าข้อมูลอินพุตจะต้องถูกจัดเรียงจากน้อยไปมาก (หรือมากไปน้อย) ก่อน
- กำหนดให้ข้อมูลถูกเก็บในอาร์เรย์ **A** และข้อมูลที่ต้องการค้นหา K อัลกอริทึม binary search จะมีขั้นตอนต่อไปนี้

1. คำนวณตำแหน่งกลาง (**mid**) ของข้อมูลในอาร์เรย์ A โดย

$$\textbf{mid} = (\textbf{left} + \textbf{right}) / 2$$

2. เปรียบเทียบคีย์ข้อมูลในตำแหน่งกลางที่ได้กับข้อมูลที่ต้องการค้นหา **K**

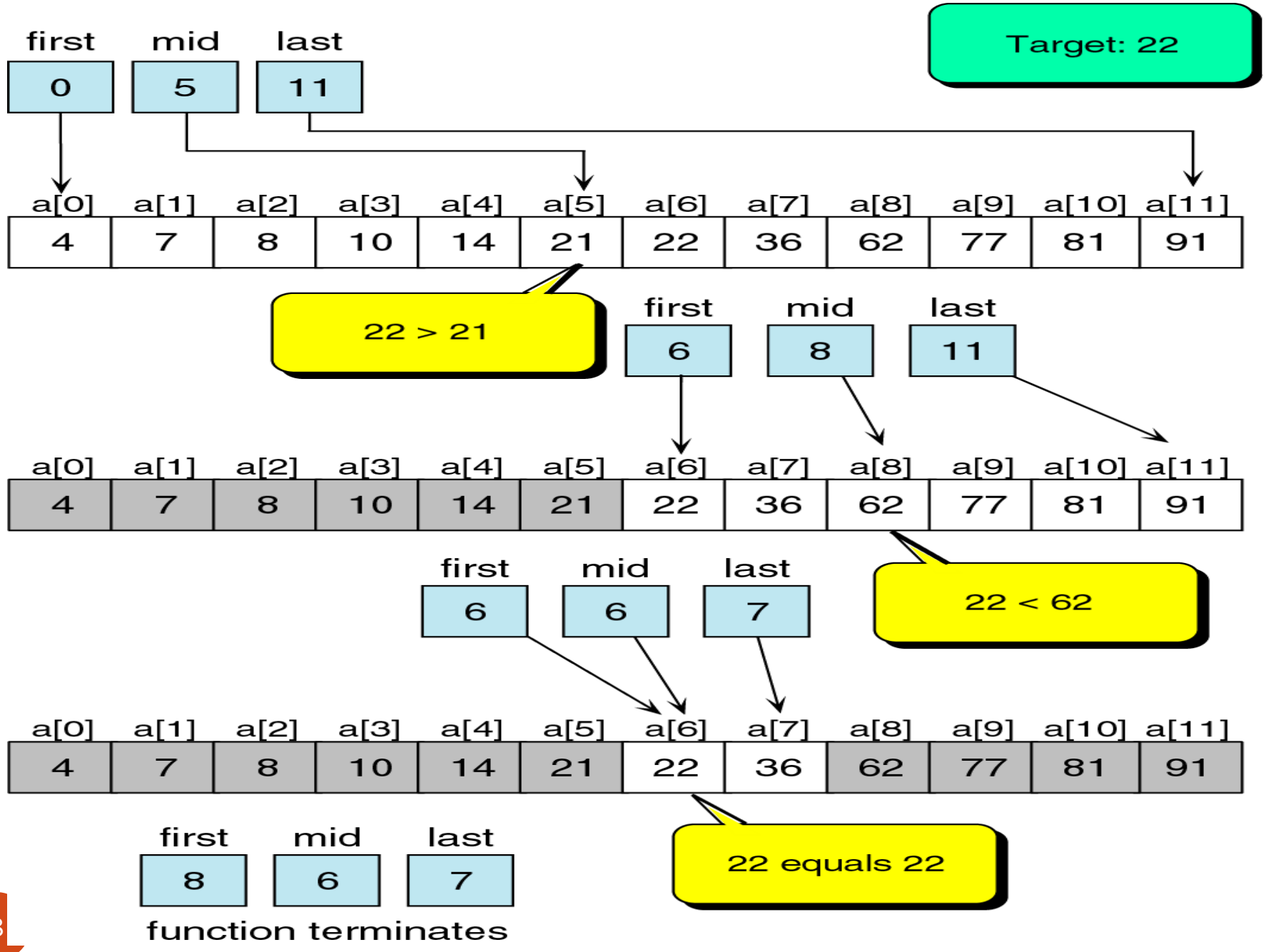
Binary search algorithm

2.1 หาก **k = A(mid)** คืนค่า **mid** แทนตำแหน่งที่พบข้อมูลใน **A**

2.2 หาก **k < A(mid)** ปรับปรุงค่า **right = mid - 1**

2.3 หาก **k > A(mid)** ปรับปรุงค่า **left = mid + 1**

3. ทำซ้ำจนกระทั่ง **left >= right** และคืนค่า -1 แทนหากไม่พบข้อมูลดังกล่าว


Target: 22
first
mid
last
0
5
11
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 > 21
first
mid
last
6
8
11
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 < 62
first
mid
last
6
6
7
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 equals 22
first
mid
last
8
6
7
function terminates

```
int binary_search(int A[], int n, int k)
{ int l = 0, r = (n-1);

  while(l <= r)
    {  int mid = (l + r) /2;
       if (A[mid] == k)
            return mid;
       if (A[mid] < k)
            l = mid + 1;
       else
            r = mid - 1;
    }

    return -1;
}
```

Workshop 1

จงปรับปรุงฟังก์ชั่น binary search ให้ทำงานแบบฟังก์ชั่น recursive

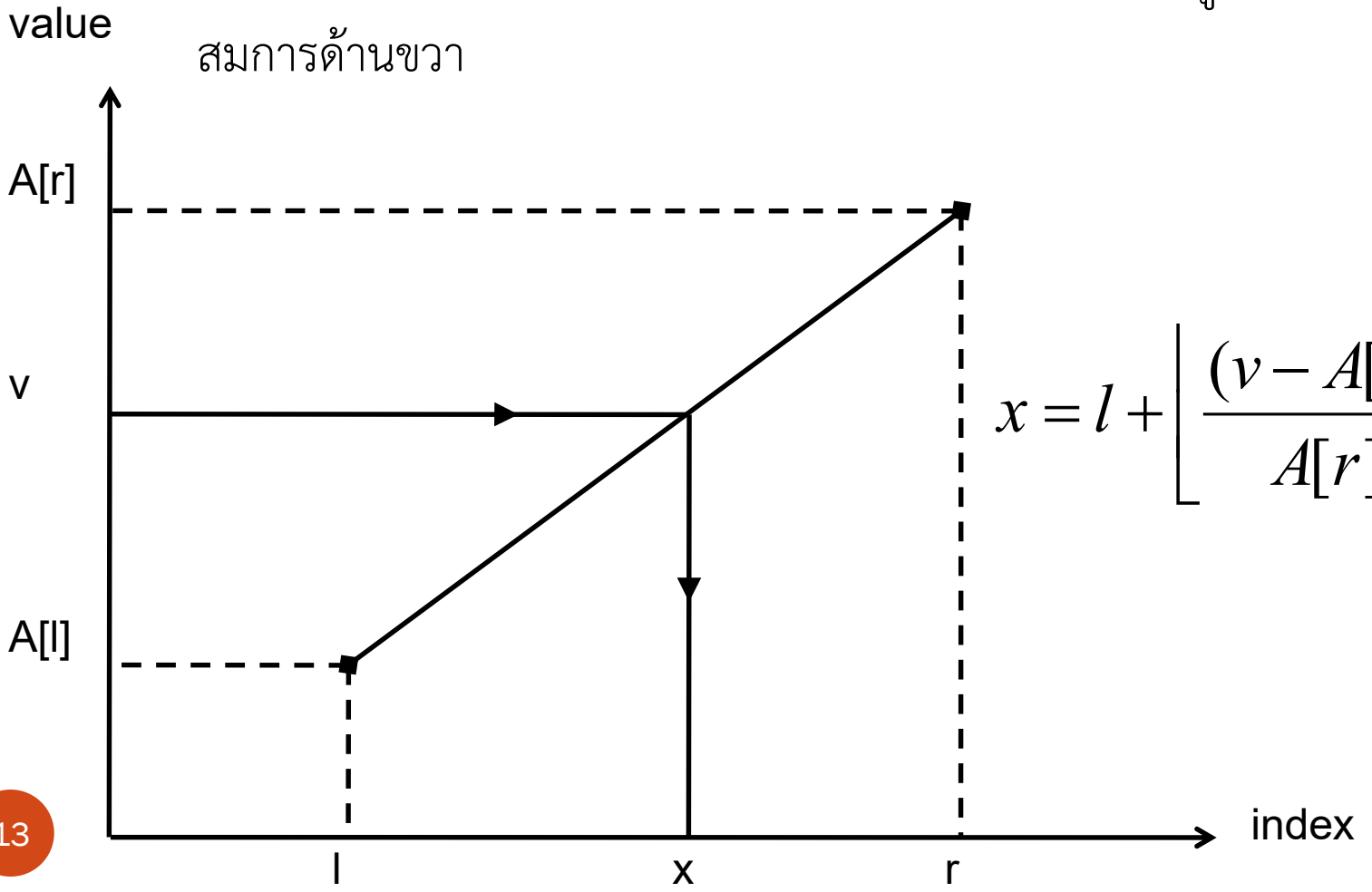
Interpolation Search

- แม้ binary search จะทำงานรวดเร็วกว่า Sequential Search แต่มีข้อจำกัด คือจะเลือกตำแหน่งกึ่งกลางข้อมูล (middle) เสมอ ซึ่งอาจส่งผลให้เข้าถึงข้อมูลได้ช้า

- Interpolation search ถูกพัฒนาขึ้นเพื่อแก้ปัญหา binary search โดยมีการประมาณตำแหน่งที่ใกล้เคียงกับข้อมูล k ส่งผลให้เข้าถึงข้อมูลได้เร็วมากขึ้น
  - ❖เปรียบเสมือนการค้นหาคำศัพท์ในพจนานุกรม หรือ การค้นหาเบอร์โทรศัพท์ในสมุดโทรศัพท์

แนวคิดของ Interpolation search

กำหนดให้ v เป็นคีย์ข้อมูลที่จะค้นหา และ ข้อมูลในอาร์เรย์ถูกเรียงลำดับไว้แล้ว A[l] <= A[r] โดยที่ l < r ค่าตำแหน่ง x จะสามารถถูกประมาณได้โดยสมการด้านขวา



$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$

Workshop 2

จงปรับปรุง binary search จาก workshop 1 ให้เป็น interpolation search

Comparison-based Sort Algorithms

อัลกอริทึมจัดเรียงข้อมูลโดยทั่วไปจะใช้วิธีเปรียบเทียบคีย์ข้อมูลในอาร์เรย์ซึ่งกันและกัน โดยมักใช้เวลาทำงานเฉลี่ย $O(n^2)$ ได้แก่

- Bubble Sort
- Selection Sort
- Insertion Sort
- Shell Sort

Bubble Sort

i=1	5	2	4	6	1	3
i=2	2	4	5	1	3	6
i=3	2	4	1	3	5	6
i=4	2	1	3	4	5	6
i=5	1	2	3	4	5	6
i=6	1	2	3	4	5	6

Bubble Sort

```
void bubble_sort(int *A, int n)
{ int i, j;

   for(i=0;i<n;i++)
     { for(j=0;j<n-1;j++)

         if(A[j] > A[j+1])
           swap(&A[j], &A[j+1]);
     }
}
```

เปรียบเทียบคีย์ข้อมูลที่อยู่ในตำแหน่งติดกัน ไปจนกระทั่งรับประกันได้ว่าข้อมูลถูกจัดเรียงแล้ว

Selection Sort

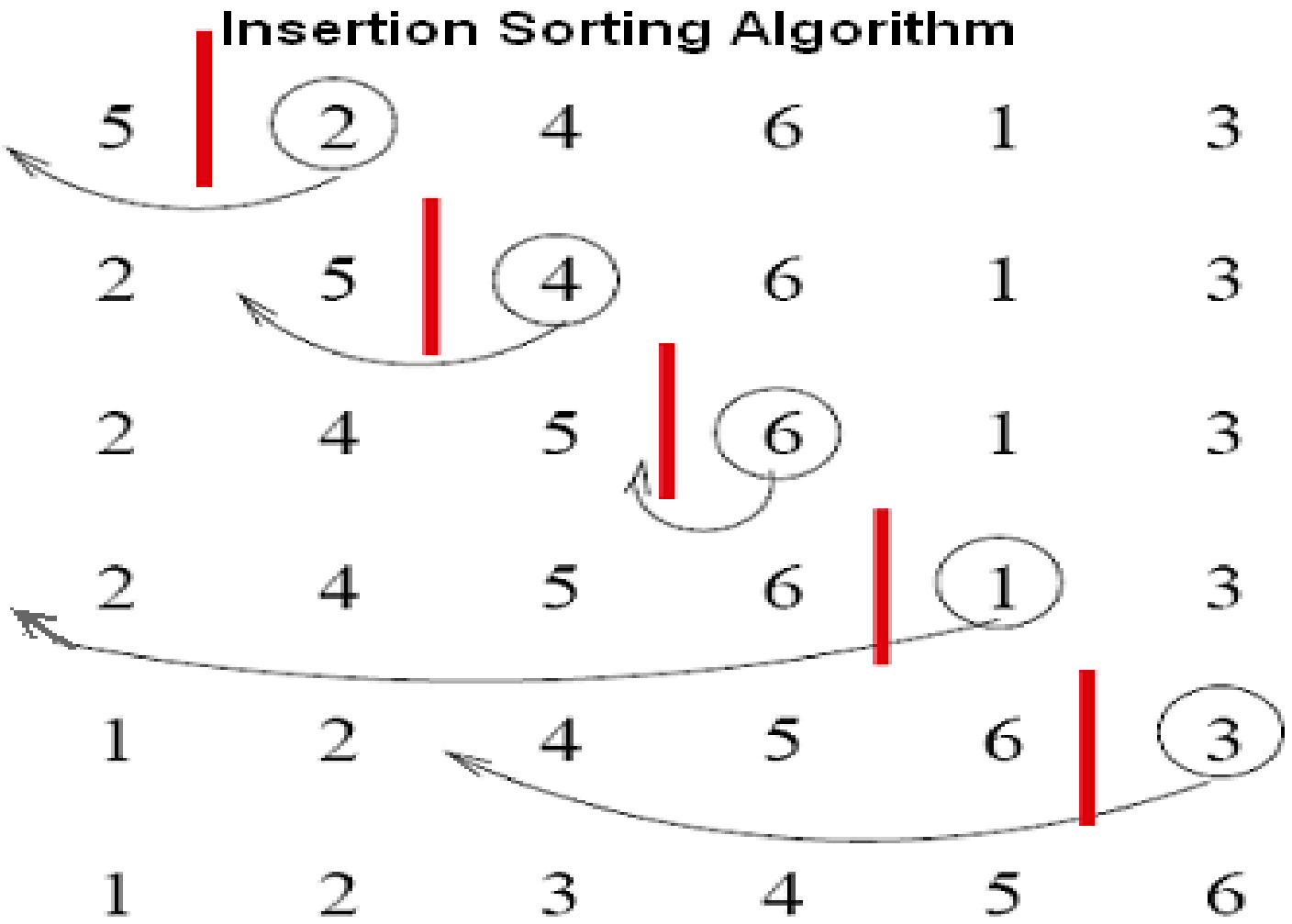
i=1	|5	2	4	6	1	3
i=2	1	| 2	4	6	5	3
i=3	1	2	|4	6	5	3
i=4	1	2	3	|6	5	4
i=5	1	2	3	|4	5	6
i=6	1	2	3	4	|5	6
i=7	1	2	3	4	5	|6

Selection Sort

```
void selection_sort(int *A, int n)
{ int i, j, min, tmp;
      for(i=0; i<n-1; i++)
        { min = i;
          for(j=i+1; j<n; j++)
           { if(A[j] < A[min])
                 min = j;
           }
          tmp = A[i];
          A[i] = A[min];
          A[min] = tmp;
        }
}
```

เปรียบเทียบคีย์ข้อมูลในตำแหน่งปัจจุบันกับข้อมูลที่ยังไม่ได้ถูกจัดเรียง

Insertion Sort



```
void insertion_sort(int A[], int n)
{   int i, j, v;
    for(i=1; I <= n-1; i++)
     { v = A[i];
       j = i-1;
       while(j >= 0 && A[j] > v)
        { A[j+1] = A[j];
                j = j-1;
        }
         A[j+1] = v;
     }
}
```

เปรียบเทียบคีย์ข้อมูลในตำแหน่งปัจจุบันกับข้อมูลที่อยู่ในตำแหน่งติดกันข้างหน้า

ให้เวลาทำงานระหว่าง O(n) และ $O(n^2)$

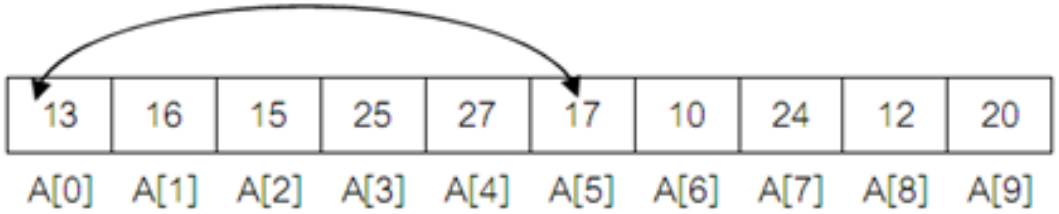
Shell sort

- ปรับปรุงการทำงานของ Insertion sort โดยแต่ละรอบจะมีการแบ่งข้อมูลเป็นกลุ่มย่อยๆ (sub groups) ซึ่งจะประกอบไปด้วยสมาชิกที่มีระยะห่างเท่ากับ k
- จากนั้นจะเรียงข้อมูลแต่ละกลุ่มด้วยวิธี insertion sort เมื่อเรียงเสร็จจะลดค่า k ลงตามลำดับ จนกระทั่งเหลือเพียงกลุ่มข้อมูลเดียว (k=1) จึงทำการเรียงด้วย insertion sort เป็นอันเสร็จสิ้น

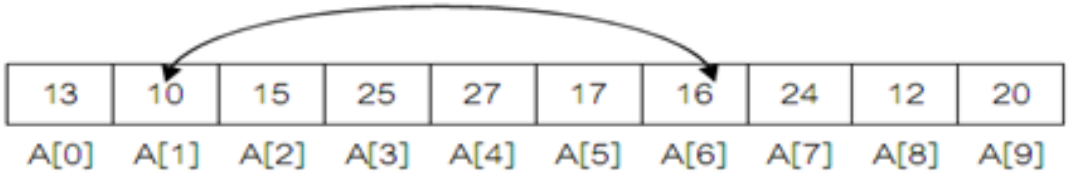
17	16	15	25	27	13	10	24	12	20
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

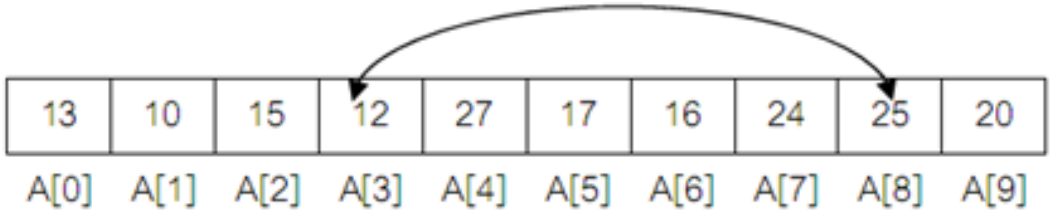
รอบที่ 1 k = n/2

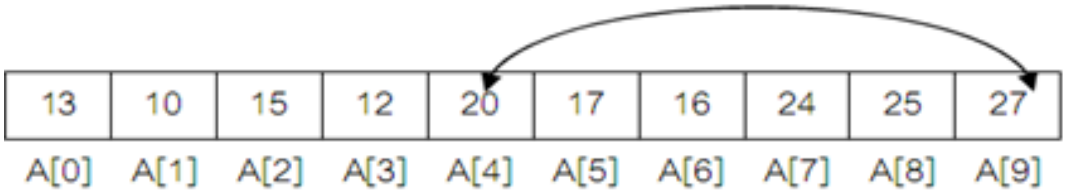
K = 10/2 = 5



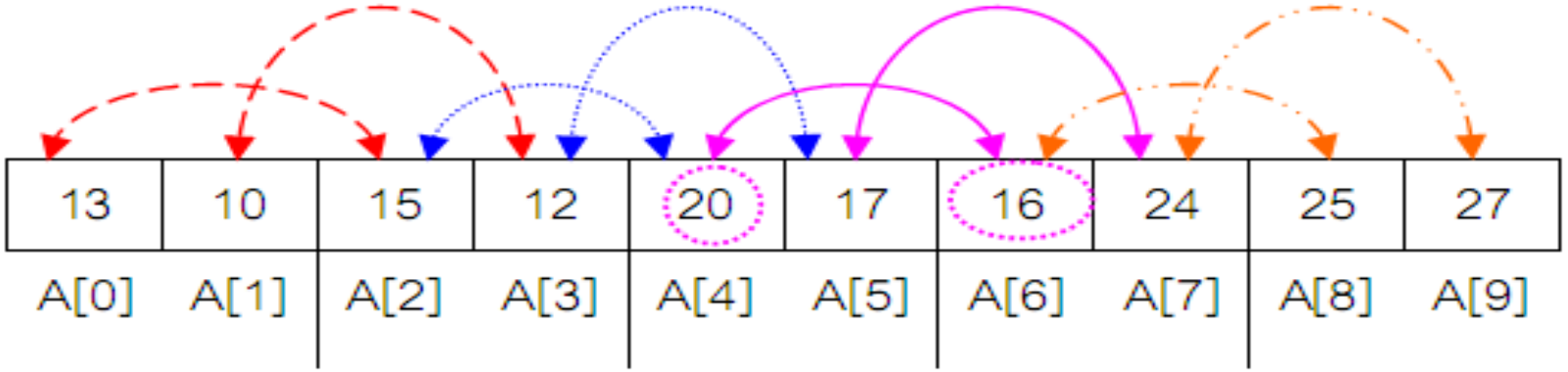
แบ่งข้อมูลย่อยทั้งหมด 4 กลุ่ม แต่ละกลุ่มจะมีสมาชิกที่มีระยะห่างเท่ากับ 5 (k = 5)





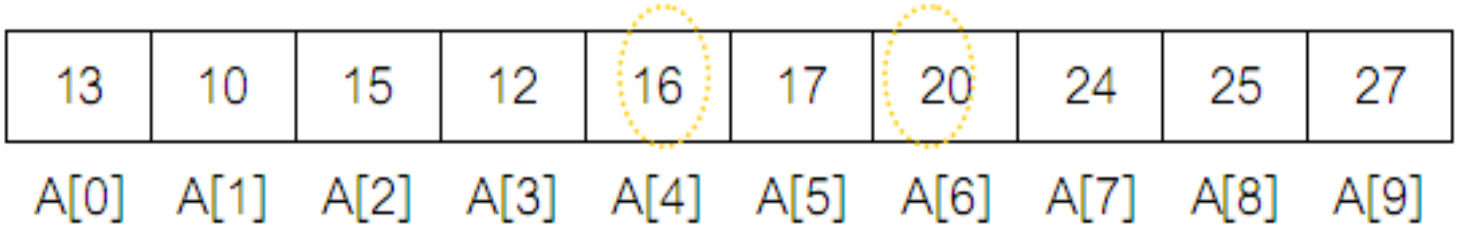


รอบที่ 2 K = 5/2 = 2



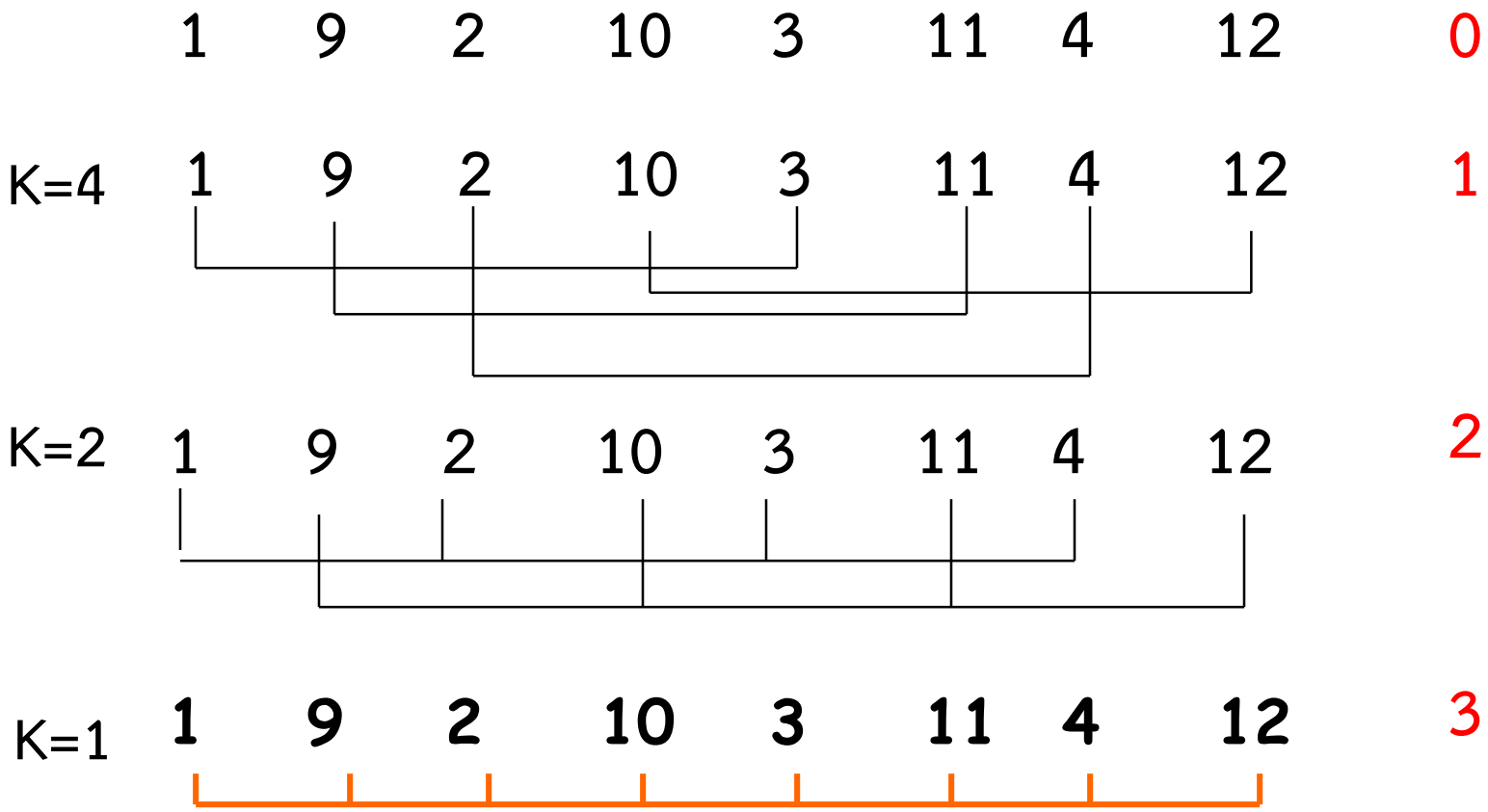
แบ่งข้อมูลย่อยทั้งหมด 2 กลุ่ม แต่ละกลุ่มประกอบไปด้วยสมาชิกที่มีระยะห่างเท่ากับ 2 (k = 2)

รอบที่ 3 K = 1



เมื่อ k = 1 จะทำงานเหมือน insertion sort จะสังเกตุได้ว่าในรอบนี้ข้อมูลส่วนใหญ่จะถูกเรียงไว้แล้ว ทำให้การจัดเรียงแบบ insertion sort ในรอบสุดท้ายจะมีโอกาสเข้าใกล้ O(n) ซึ่งเป็น Best case ผลลัพธ์ก็คือการเรียงลำดับแบบ shell sort โดยเฉลี่ยจะใช้เวลา $O(n^2)$ นั่นเอง

การเลือกค่า K ไม่เหมาะสมใน shell sort



วิธีการกำหนดค่า k ใน Shell Sort

โดยทั่วไปค่า k จะถูก generate ไว้ล่วงหน้า โดยอาจใช้วิธีการ ดังต่อไปนี้

- **Sedgewick's sequence** ลำดับของเลข prime เช่น {1, 3, 5, 7, 11, 13, 17, ...}
- Knuth's sequence $k = \left\lfloor \frac{N}{2^i} \right\rfloor$ เช่น N = 10 จะได้ k={5, 2, 1}
- Hibbard's sequence $k = 2^i - 1$ เช่น k = {1, 3, 7, ... }

โดยที่ i หมายถึง iteration และ N คือขนาดของอาร์เรย์ที่จะเรียง

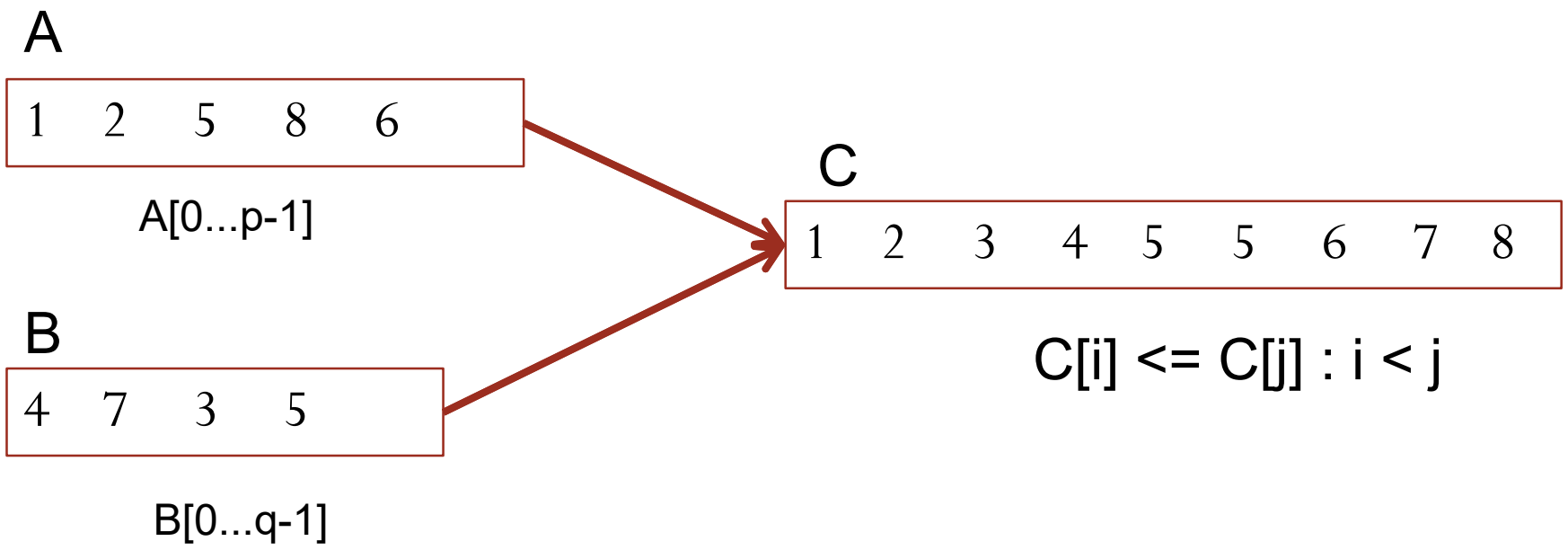
Partition-based sorting algorithms

ใช้วิธีแบ่งข้อมูล (partition) เพื่อลดจำนวนครั้งในการเปรียบเทียบคีย์ข้อมูล อัลกอริทึมลักษณะนี้มักใช้เวลาในการจัดเรียง O(n.log n)

- Merge Sort : แบ่งข้อมูลออกเป็น 2 ส่วนขนาดเท่ากัน จนกระทั่งทั้ง 2 ส่วนมีขนาดเล็กจนไม่จำเป็นต้องจัดเรียง จากนั้นจะนำข้อมูลทั้ง 2 ส่วนที่เรียงกันแล้วมาทำการ merge
- Quick Sort : สุ่มข้อมูลที่เรียกว่า pivot เพื่อแบ่งข้อมูลออกเป็น 2 ส่วน (อาจไม่เท่ากัน) จนกระทั่งข้อมูลทั้ง 2 ส่วนมีขนาดเล็กที่สุดจนไม่จำเป็นต้องจัดเรียง

Workshop 3 การ merge ข้อมูล

จงเขียนฟังก์ชั่นรับอินพุตจากอาร์เรย์ A และ B ที่เรียงลำดับข้อมูลแล้ว (อาจมีจำนวนไม่เท่ากัน) เพื่อนำข้อมูลจากอาร์เรย์ดังกล่าวไปเก็บไว้ในอาร์เรย์ C โดยข้อมูลในอาร์เรย์ C ก็ถูกเรียงลำดับด้วยเช่นกัน



```
Merge (A[0..p-1], B[0..q-1]) {
  C = [0.. p+q-1];
  aa = 0 bb = 0, cc=0;

    while aa < A.size and bb < B.size {
        if  A[aa] < B[bb]
             C[cc++] = A[aa++];
       else if A[aa] > B[bb]
             C[cc++] = B[bb++];
       else  { aa += 1; bb += 1; }
    }

   while (aa < A.size) C[cc++] = A[aa++];
   while (bb < B.size) C[cc++] = B[bb++];
  return C;
}
```

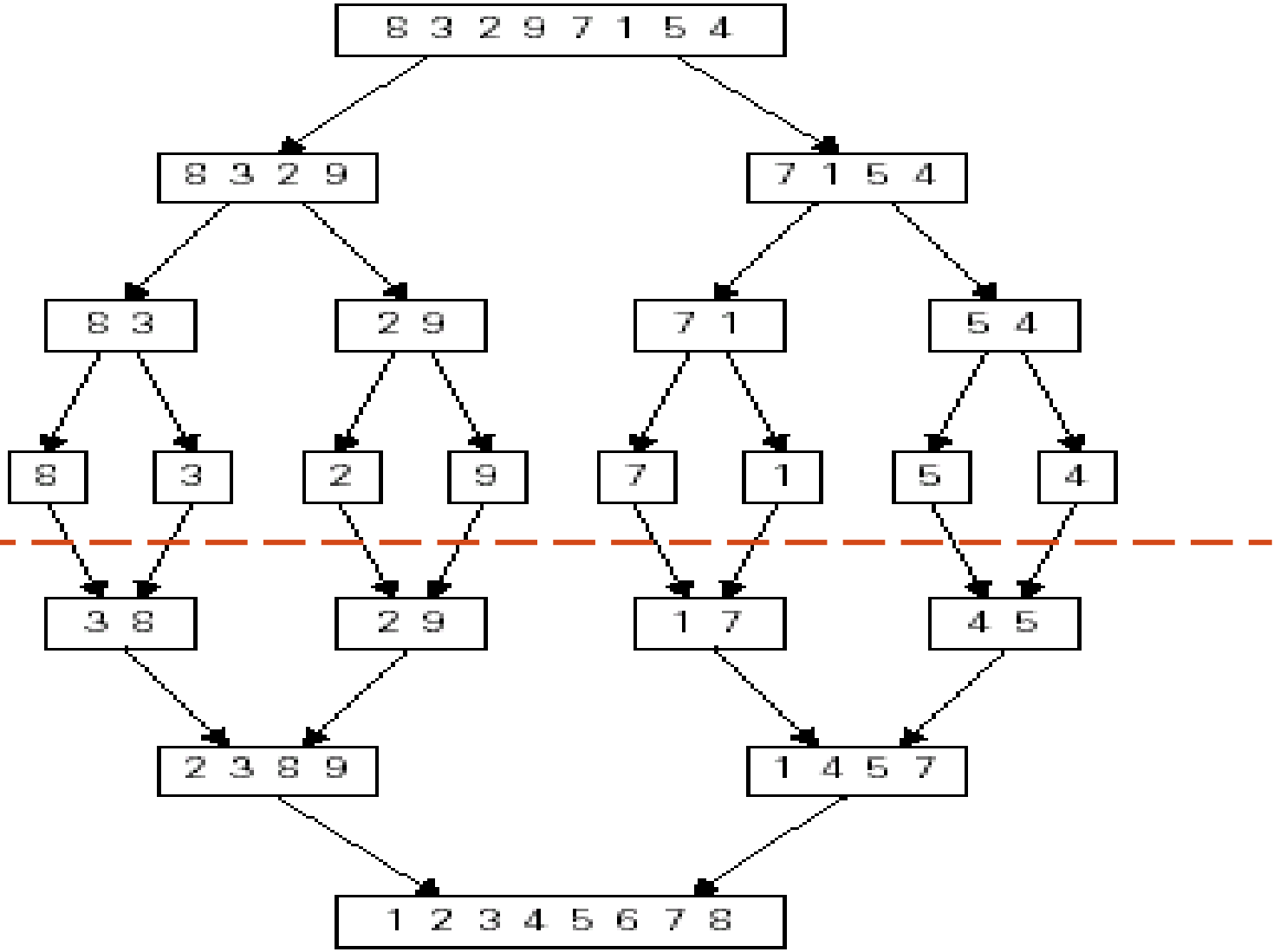
Merge Sort

- แนวคิดของ merge sort จะใช้วิธีการแบ่งข้อมูล (partition) ออกเป็น 2 ส่วนเท่าๆ กัน จากนั้นนำข้อมูลทั้งสองส่วนมาทำการเปรียบเทียบเพื่อหาลำดับที่ถูกต้องสำหรับการรวมข้อมูล
- ปัญหาคือหากข้อมูลมีจำนวนมากจะต้องเสียเวลาในการเรียงลำดับข้อมูลย่อยในแต่ละส่วนก่อนที่จะนำมา merge ได้ ส่งผลกระทบต่อเวลารวม
- ทำอย่างไรจึงจะแก้ปัญหาดังกล่าวได้ ?

Merge Sort

partition

merge



Merge Sort (1)

```
void mergeSort(int arr[], int l, int r)
{
  if (l < r)
    {  int m = l + (r-l) / 2;
       mergeSort(arr, l, m);
       mergeSort(arr, m+1, r);
       merge(arr, l, m, r);
    }

}
```

Merge Sort (2)

```
void merge(int arr[], int l, int m, int r)
{   int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];

     for (i = 0; i < n1; i++)
            L[i] = arr[l + i];
     for (j = 0; j < n2; j++)
            R[j] = arr[m + 1+ j];
```

Merge Sort (3)

```
while (i < n1 && j < n2) // compare L and R
{  if (L[i] <= R[j])
      {   arr[k] = L[i];
           i++;
      }
     else
       {  arr[k] = R[j];
           j++;
       }
       k++;
}
```

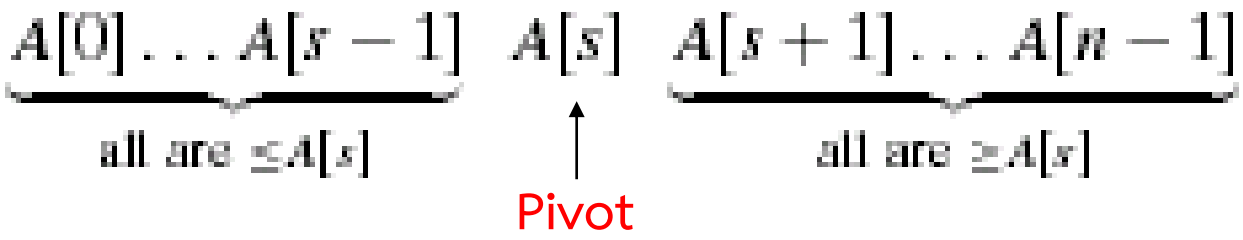
Merge Sort (4)

```
while(i < n1)                  // copy left sub-array
   {   arr[k] = L[i];
       i++;
       k++;
   }

while (j < n2)                 // copy right sub-array
     { arr[k] = R[j];
       j++;
       k++;
     }
}
```

Quick Sort

- Quick sort จะแบ่งข้อมูลอินพุตออกเป็นสองส่วนสัมพันธ์กับค่า pivot



- โดยที่ข้อมูลที่มีค่าน้อยกว่าหรือเท่ากับ pivot จะอยู่ด้านหน้า แต่ถ้าหากข้อมูลมีค่ามากกว่า pivot จะถูกนำไปไว้ด้านหลัง

Median of three

1. คำนวณตำแหน่ง centre ในอาร์เรย์อินพุต

$$center = \frac{(left + right)}{2}$$

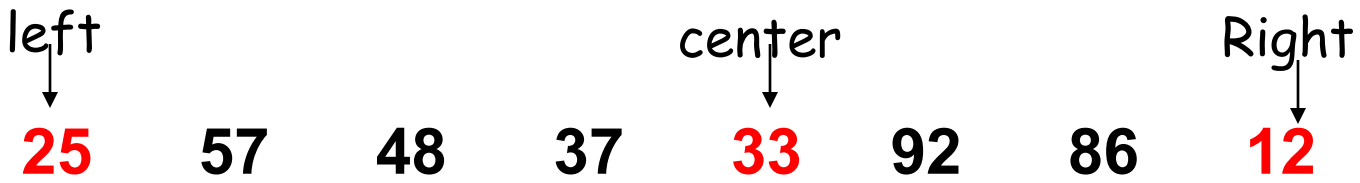
2. เรียงลำดับ 3 ค่า ในอาร์เรย์อินพุต

$$A[left] \leq A[center] \leq A[right]$$

** 3. สลับตำแหน่ง ระหว่าง centre กับ right

$$A[center] \leftrightarrow A[right] \qquad A[right]$$

Median of three

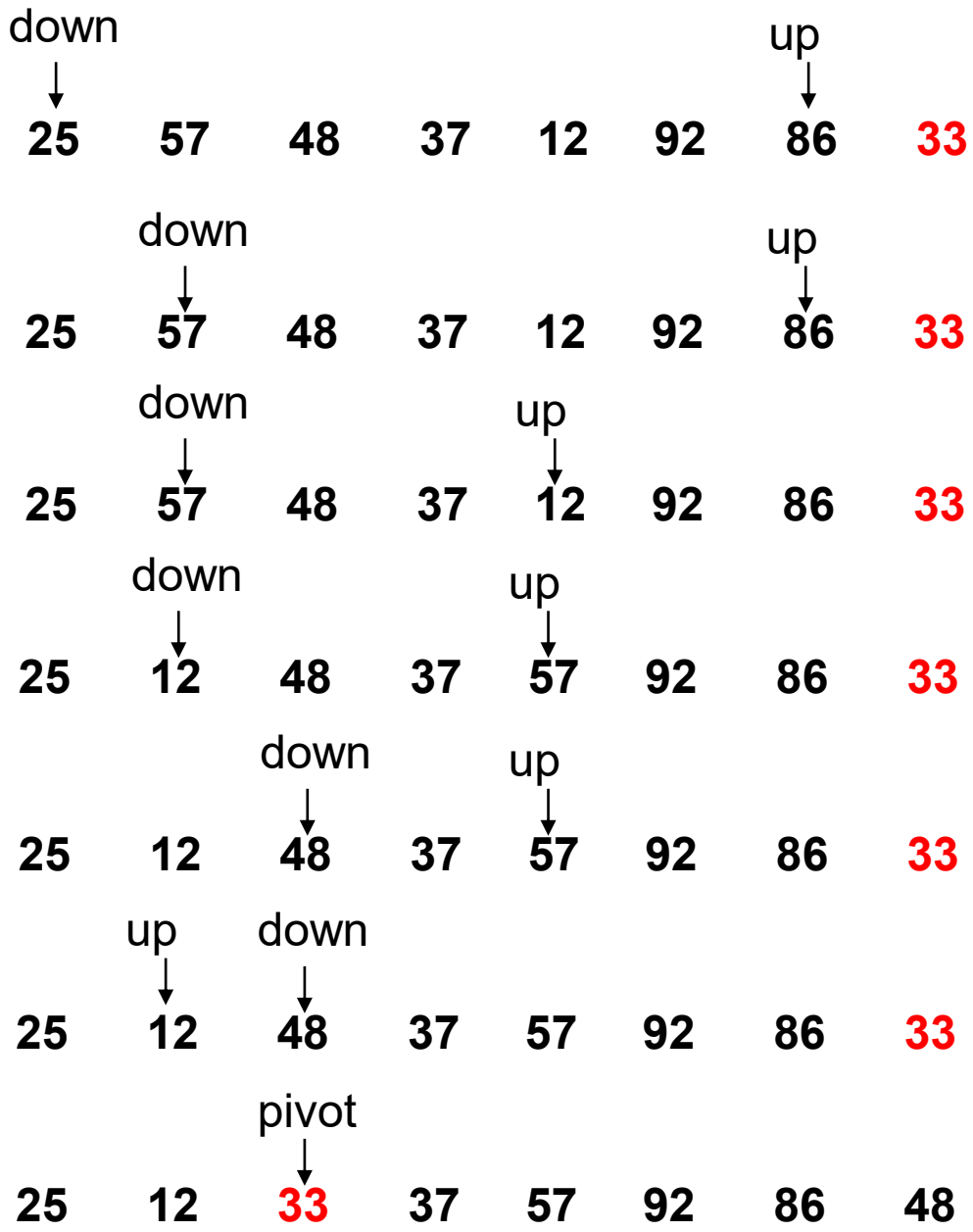


$$center = \frac{(left + right)}{2}$$

$$center = \frac{(0 + 8)}{2} = 4$$

การ partition ข้อมูล

1. เริ่มต้นจากตัวชี้ up และ down ถูกกำหนดให้ไปชี้ไว้ในตำแหน่งขวาสุด (up) และซ้ายสุด (down) ของอาร์เรย์อินพุต A ตามลำดับ
2. เลื่อนตัวชี้ down ไปทางด้านขวาของอาร์เรย์ A ถ้าพบว่าค่า A[down] มากกว่า ค่า pivot
3. เลื่อนตัวชี้ up ไปทางซ้ายของอาร์เรย์ A ถ้าพบว่า A[up] มีค่าน้อยกว่าค่า pivot
4. ถ้า up มีค่ามากกว่า down ให้ทำการสลับข้อมูลระหว่าง A[down] และ A[up]
5. ทำซ้ำตั้งแต่ขั้นตอนที่ 2 จนกระทั่ง up มีค่าน้อยกว่าเท่ากับ down
6. สลับข้อมูลระหว่าง A[down] กับ pivot


down
up
25 57 48 37 12 92 86 33
down
up
25 57 48 37 12 92 86 33
down
up
25 57 48 37 12 92 86 33
down
up
25 12 48 37 57 92 86 33
down
up
25 12 48 37 57 92 86 33
up
down
25 12 48 37 57 92 86 33
pivot
25 12 33 37 57 92 86 48

```
int partition(int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];       // first item always as the pivot
   i = l;
   j = r + 1;
   while(1)
     {   do { ++i;
            }while(a[i] <= pivot);
         do { --j;
            }while(a[j] > pivot);
        if(i >= j)
              break;
          swap(&a[i], &a[j]);
     }
     swap(&a[l], &a[j]);
    return j;
```

```
void quickSort( int s[], int l, int r)
{   int p;
   if((r-l)>0) {
      p = partition( s, l, r);
      quickSort( s, l, p-1);
      quickSort( s, p+1, r);
  }
}
```

Array								iteration
25	57	48	37	12	92	86	33	0
25	12	33	37	57	92	86	48	1
12	25	33	37	57	92	86	48	2
12	25	33	37	48	92	86	57	3
12	25	33	37	48	92	86	57	4

Memory-based sorting algorithms

ใช้หน่วยความจำเพิ่มขึ้นเพื่อจัดเตรียมข้อมูลก่อนทำการเรียงลำดับ ซึ่งจะช่วยลดจำนวนรอบในการทำงานได้ อัลกอริทึมลักษณะนี้จะใช้เวลา O(n)

- Counting Sort
- Bucket Sort

Counting Sort (Bucket Sort)

- เรียงข้อมูลโดยนับความถี่ของข้อมูลที่ปรากฏในอินพุต โดยการสร้าง frequency count distribution
- จากนั้นคำนวณหาตำแหน่งที่เหมาะสมของข้อมูลที่ถูกจัดเรียงในเอาท์พุต โดยใช้ข้อมูลจาก distribution ที่สร้างขึ้น
- ข้อดี counting sort ไม่มีการเปรียบเทียบระหว่างข้อมูลในอาร์เรย์อินพุต
  - แต่ต้องการหน่วยความจำเพิ่มเพื่อความถี่และผลลัพธ์

Counting sort

Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

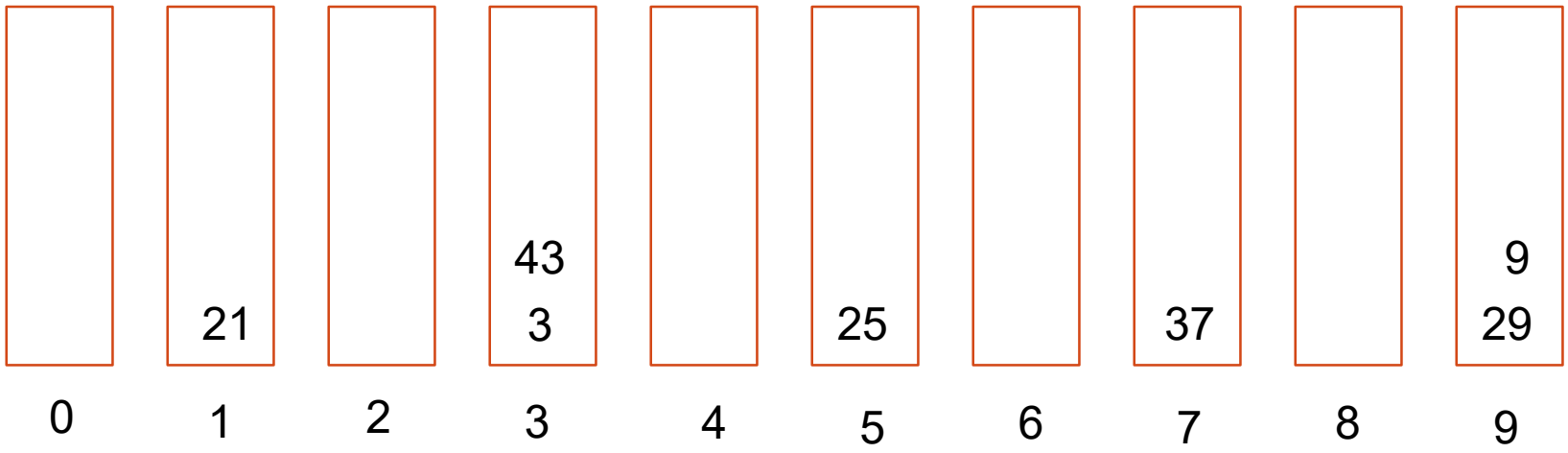
Workshop 4: Counting Sort

จงเขียนฟังก์ชั่นเพื่อรับอินพุตจากอาร์เรย์ A จำนวน n ตัว จากนั้นเรียงลำดับข้อมูลด้วยวิธี Counting Sort พร้อมส่งผลลัพธ์กลับ

Bucket Sort

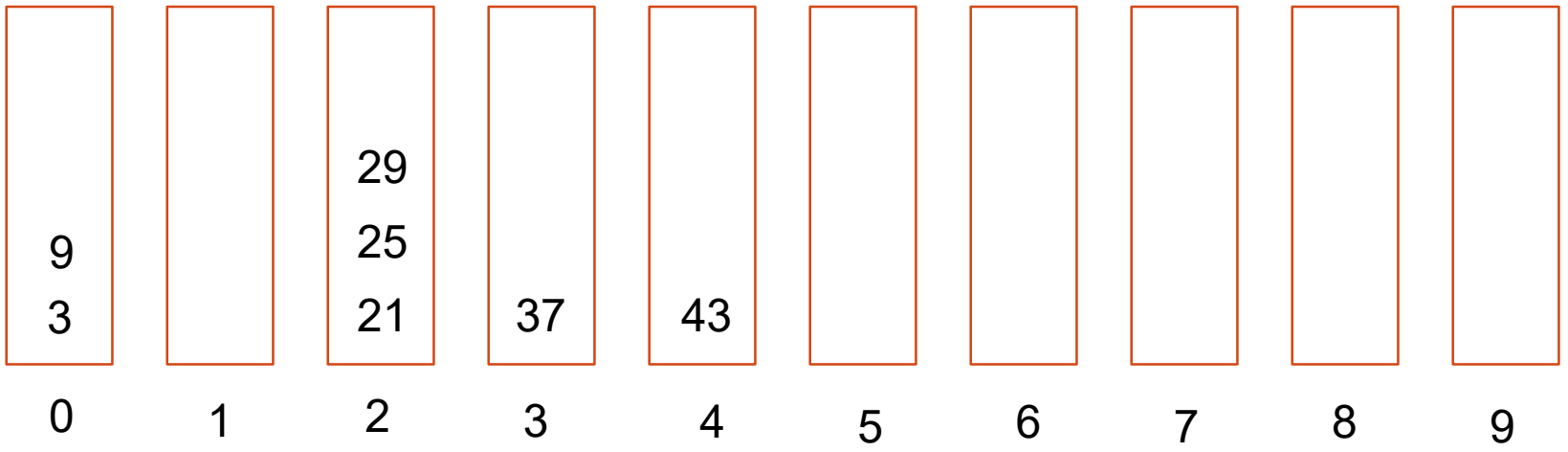
- เรียงข้อมูลโดยใช้ถัง (bucket) จำนวน 10 ถึง แต่ละถังจะถูกกำหนดให้ใส่ข้อมูลตามหลัก (digit) ของข้อมูล
- เริ่มต้นจากหลักหน่วยของข้อมูลแต่ละตัว จะถูกนำไปใส่ bucket ที่มีเลขตรงกับหลักหน่วย จากนั้นดึงผลลัพธ์ขึ้นมาแบบลำดับ
- เลื่อนตำแหน่งไปเป็นหลักสิบ และนำลงไปใส่ bucket ที่มีเลขตรงกับหลักสิบ และทำการดึงผลลัพธ์ขึ้นมาเป็นลำดับ
- ทำซ้ำจนกระทั่งข้อมูลทุกหลักถูกใส่ bucket

29 25 3 49 9 37 21 43



21 3 43 25 37 29 9

21 03 43 25 37 29 09



3 9 21 25 29 37 43