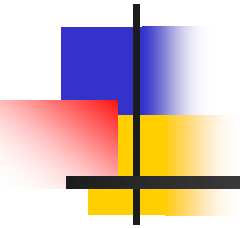


Chapter 02

Advanced Sorting

Sizes of Data

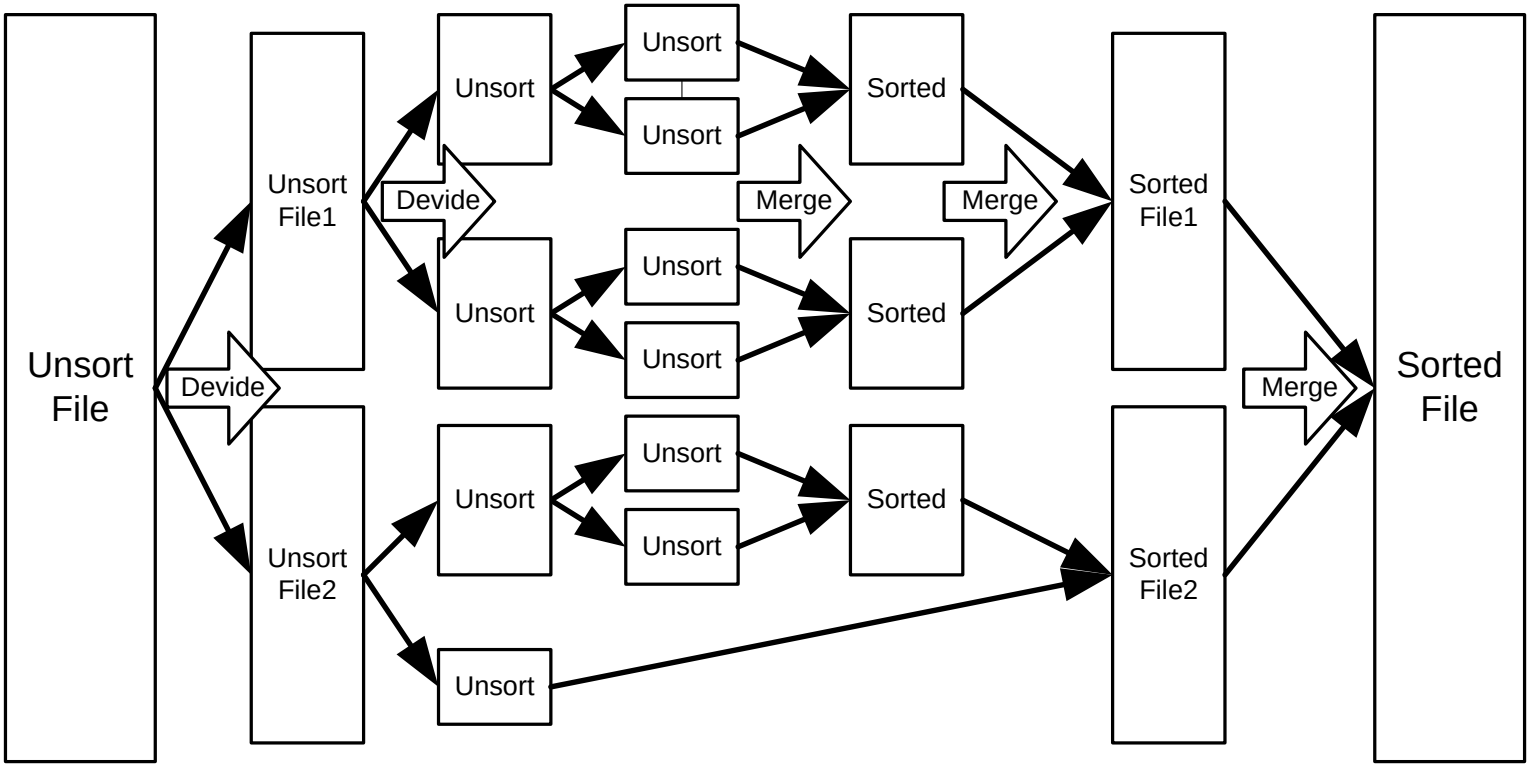
- ประสิทธิภาพ(เวลา)ที่ใช้เรียงลำดับข้อมูลขนาดเล็ก จะเร็วกว่าเมื่อเทียบเป็นสัดส่วนกับการเรียงลำดับข้อมูลขนาดใหญ่ เนื่องจากความซับซ้อนของการเรียงลำดับเป็นแบบสมการกำลังสอง $O(n^2)$ ถ้าขนาดข้อมูลลดลงหรือเพิ่มขึ้น 2 เท่า เวลาที่ใช้เรียงลำดับจะลดลงหรือเพิ่มเป็น 4 เท่า
- Merge Sort O(n log n)
  - แบ่งข้อมูลออกเป็น 2 กลุ่มย่อย ให้มีขนาดลดลงครึ่งหนึ่ง O(log n) เพื่อเรียงลำดับแล้วจึงนำกลุ่มย่อยที่เรียงลำดับแล้วทั้ง 2 กลุ่ม มารวมกันอีกครั้ง O(n) ด้วยวิธีการ Merge
  - ใช้หลักการเรียกตัวเอง Recursion ในการแบ่งข้อมูลให้ย่อยลงได้อีก แล้วจึงนำมารวมกัน จนกว่าจะเสร็จ
  - ต้องมีหน่วยความจำเพิ่มอีก 1 เท่า เพื่อใช้เก็บข้อมูลชั่วคราว
- Quick Sort O(n log n)
  - ใช้หลักการแบ่งกลุ่ม(Partitioning) แบ่งข้อมูลออกเป็น 2 กลุ่ม กลุ่มแรกมีค่าน้อย กลุ่มที่สองมีค่ามากกว่ากลุ่มแรก O(n)
  - ใช้หลักการเรียกตัวเอง Recursion กับข้อมูลทั้ง 2 กลุ่ม เพื่อแบ่งข้อมูลต่อไปเรื่อยๆ จนกว่าจะเรียงลำดับ O(log n)
  - กรณี worst case (เรียงลำดับกลับกัน) จะได้ $O(n^2)$

C: มีฟังก์ชัน qsort อยู่ใน <stdlib.h> ใช้เทคนิคของ Quick Sort

Java : มีเมธอด Arrays.sort ซึ่งจะเลือกใช้เทคนิคของ Quick Sort ถ้าเรียงลำดับข้อมูลที่มีชนิดเป็น primitive และใช้เทคนิคของ Merge Sort ถ้าเรียงลำดับข้อมูลที่มีชนิดเป็น object[]

1. Merge Sort

- แบ่งข้อมูลออกเป็นกลุ่มย่อยที่เล็กลง(แบ่งข้อมูลเป็น 2 ส่วน ไปเรื่อยๆจนกระทั่งแบ่งไม่ได้) แล้วจึงรวม 2 ส่วนเข้าด้วยกันแบบมีลำดับ
- นำกลุ่มข้อมูลย่อยที่เรียงลำดับแล้ว กลับมารวมกันใหม่ให้เป็นข้อมูลกลุ่มใหญ่อีกครั้งแบบมีลำดับ (Merge)
- สามารถใช้ Recursion ในการแบ่งกลุ่ม และรวมกลุ่มได้



devide and Merge

42	23	74	11	65	58	94	36	99	87	Original Arrays
42	23	74	11	65	58	94	36	99	87	แบ่งกลุ่มเป็น 2 กลุ่ม
42	23	74	11	65						แบ่งกลุ่ม 42,23 กับ 74,11,65
42	23									แบ่งกลุ่ม 42 กับ 23
23	42									รวมกลุ่ม 42 กับ 23
		11	74	65						แบ่งกลุ่ม 74,11 กับ 65
		11	74	65						รวมกลุ่ม 74,11
23	42	11	65	74						รวมกลุ่ม 11,74 กับ 65
11	23	42	65	74						รวมกลุ่ม 23,42 กับ 11,74,65

devide and Merge

					58	94	36	99	87	แบ่งกลุ่ม 58,94 กับ 36,99,87
					58	94				แบ่งกลุ่ม 58 กับ 94
					58	94				รวมกลุ่ม 58 กับ 94
							36	99	87	แบ่งกลุ่ม 36,99 กับ 87
							36	99	87	รวมกลุ่ม 36 กับ 99
					58	94	36	87	99	รวมกลุ่ม 36,99,87
11	23	42	65	74	36	58	87	94	99	รวมกลุ่ม 58,94 กับ36,87,99
11	23	36	42	58	65	74	87	94	99	รวมทั้งสองกลุ่มที่เหลือเข้าด้วยกัน

Merge Sort Method

```
void  mergeSort(int data[], int temp[], int first, int last)
{ int mid ;
        if (first < last)
        {   mid = (first+last)/2;
            mergeSort(data,temp,first, mid);
            mergeSort(data,temp,mid+1,last);
            mergeData(data,temp,first,mid,last);
        }
}
```

ต้องมีตัวแปร temp[] ขนาดเท่า data เพื่อช่วยในการจัดเรียง

หาจุดแบ่งครึ่งข้อมูล

แบ่งข้อมูลเป็น 2 กลุ่ม เพื่อเรียงลำดับทีละกลุ่ม

เรียงลำดับเสร็จ ให้นำข้อมูลทั้ง 2 กลุ่มกลับมารวมกัน

Java จะมีเมธอด Arrays.sort ซึ่งจะเลือกใช้เทคนิคของ Merge Sort ถ้าเรียงลำดับข้อมูลที่มีชนิดเป็น object[]

Merge Data (Sorted) Method

```
void  mergeData(int data[], int temp[], int first, int mid, int last)
{int i, i1, i2 ;
    i1 = first;
    i2 = mid+1;
    for (i=0;i<=last-first;i++)
     { if (i1<=mid && i2<=last)
         { if  (data[i1] < data[i2])
                 temp[i] = data[i1++];
           else  temp[i] = data[i2++]; }
       else  if (i1>mid)
                 temp[i] = data[i2++];
        else  if (i2>last)
                 temp[i] = data[i1++];
     }
     for (i=0;i<=last-first;i++)
           data[first+i] = temp[i];
}
```

ตัวแปร temp[] ใช้สำหรับเก็บข้อมูลที่เรียงชั่วคราว

กรณีข้อมูลยังอยู่ในช่วง

เปรียงเทียบข้อมูลชุดที่ 1 และ 2
copy data[] ในตำแหน่งที่มีค่าน้อยสุดไว้ใน temp[i]

ถ้าข้อมูลชุด 1 หมด ให้ copy ข้อมูลของข้อมูลชุดที่ 2

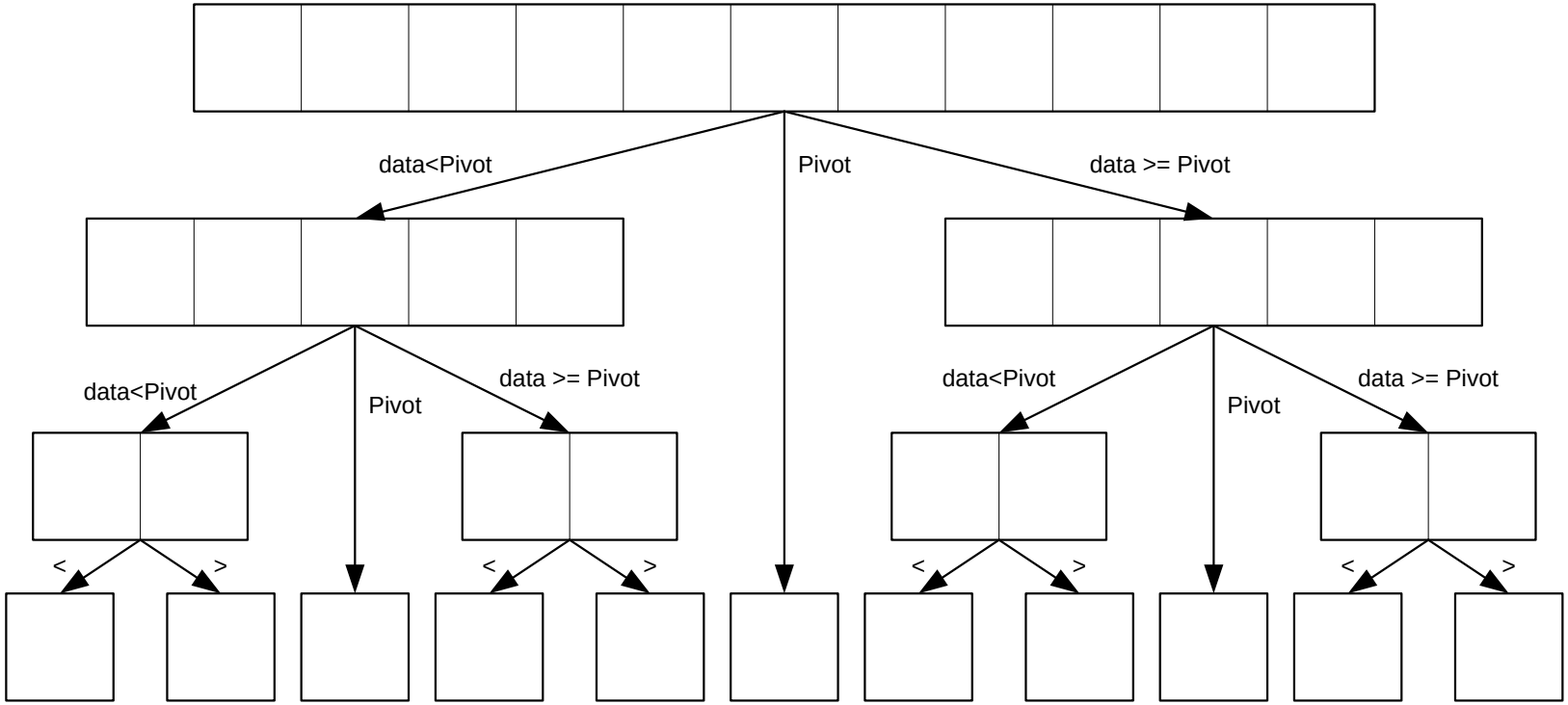
ถ้าข้อมูลชุด 2 หมด ให้ copy ข้อมูลของข้อมูลชุดที่ 1

copy ข้อมูลทั้งหมดจาก temp[] กลับคืนไป data[]

	i1 ->				mid	i2->				last
data	11	23	42	65	74	36	58	87	94	99
temp	11	23	36	42	58	65	74	87	94	99

2. Quick Sort

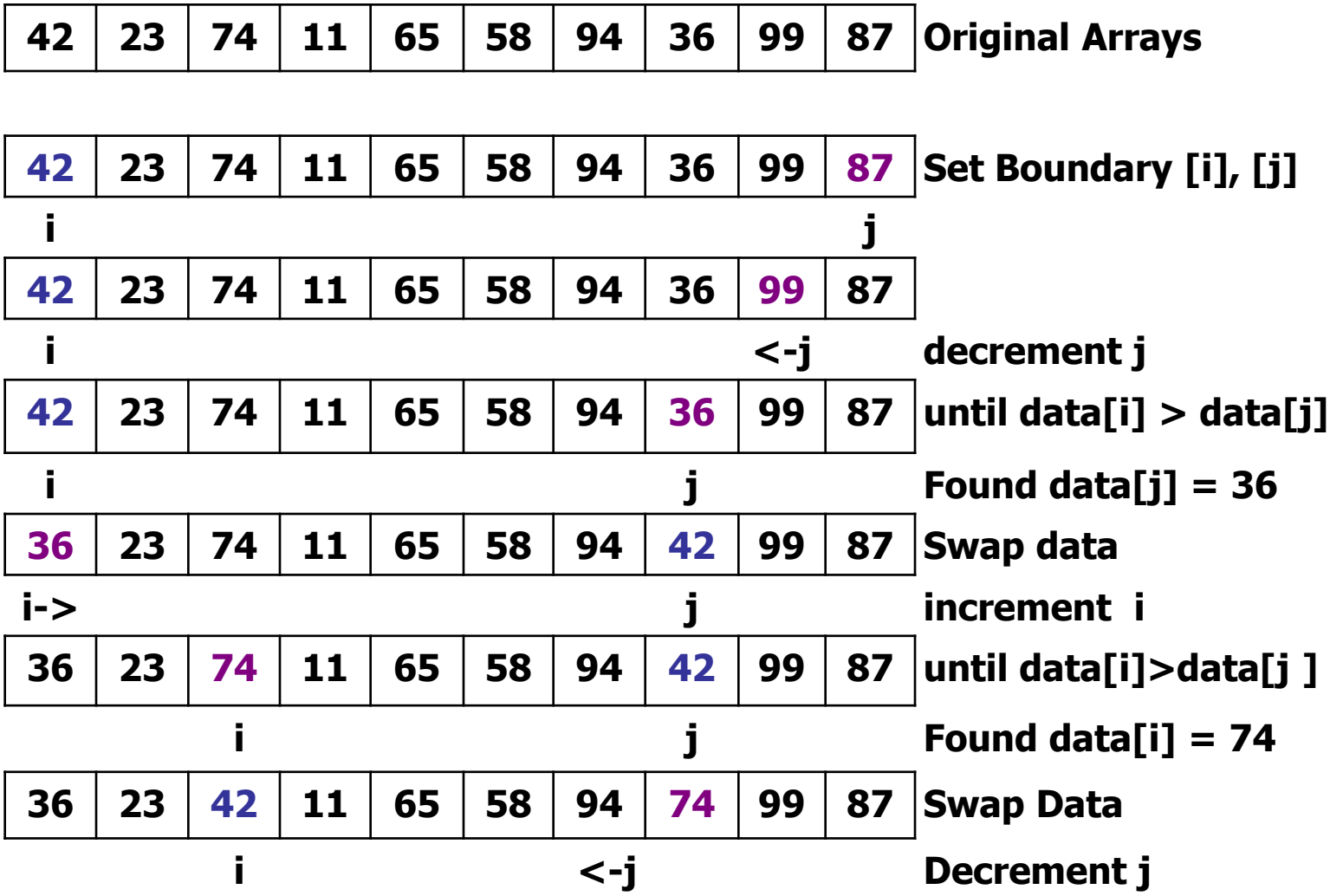
- สลับตำแหน่งเพื่อแบ่งข้อมูลออกเป็น 2 กลุ่ม คือกลุ่มที่มีค่าน้อย และกลุ่มที่มีค่ามาก แล้วทำซ้ำแต่ละกลุ่มให้แบ่งเป็น 2 กลุ่มย่อยอีกโดยการเรียกตัวเองไปเรื่อยๆ จนกว่าจะไม่สามารถแบ่งข้อมูลได้อีก
- การแบ่งข้อมูลเป็น 2 กลุ่ม นิยมใช้วิธี picking pivot และ partitioning
- Picking pivot จะใช้วิธีสมมุติค่าตรงกลางข้อมูล แล้วสลับแบ่งครึ่งข้อมูลเป็น 2 ส่วน



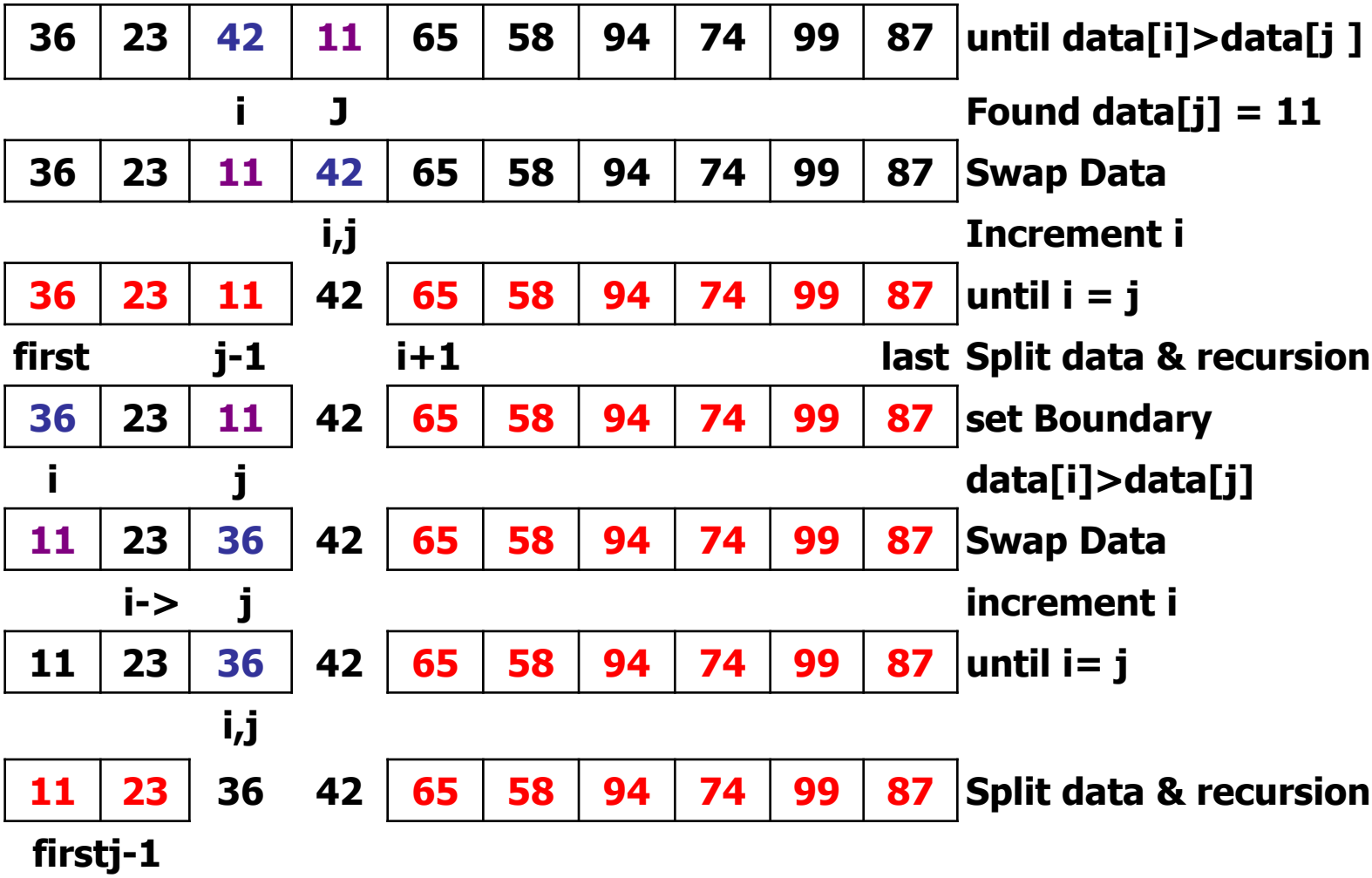
Partitioning

- **Partitioning** จะสลับกันเลื่อนค่าตัวชี้จุดแบ่ง จนกว่าข้อมูลจะแบ่งเป็น 2 ส่วน
  - ใช้ตัวชี้ตัวแรกกำหนดตำแหน่งข้อมูลทางด้านซ้าย(เริ่มที่ตัวแรก) แล้วใช้ตัวชี้ตัวที่สอง ค้นหาข้อมูลจากทางด้านขวา(เริ่มจากตัวสุดท้าย) ย้อนกลับมาจนกว่าจะเจอตำแหน่งข้อมูลที่มีค่าน้อยกว่า แล้วสลับข้อมูลกัน แล้วเลื่อนตำแหน่งตัวชี้ตัวแรกถัดไป
  - กำหนดตำแหน่งข้อมูลด้านขวา(จากตัวชี้ตัวที่สอง) แล้วค้นหาตำแหน่งข้อมูลจากทางด้านซ้ายโดยใช้ตัวชี้ตัวแรก เรื่อยไปจนกว่าจะเจอข้อมูลที่มีค่ามากกว่า แล้วสลับข้อมูลกัน แล้วเลื่อนตำแหน่งตัวชี้ตัวที่สองลงมา
  - ทำซ้ำ จนกว่าตัวชี้ตำแหน่งด้านซ้าย(ตัวแรก) อยู่เลยตำแหน่งตัวชี้ด้านขวา(ตัวที่สอง) ข้อมูลจะถูกแบ่งออกเป็น 2 กลุ่ม
  - ทำซ้ำข้อมูลในแต่ละกลุ่ม โดยแบ่งย่อยไปเรื่อยๆ จนกว่าข้อมูลจะแบ่งไม่ได้ (เหลือตัวเดียว) จึงจะเสร็จสิ้นการเรียงลำดับ

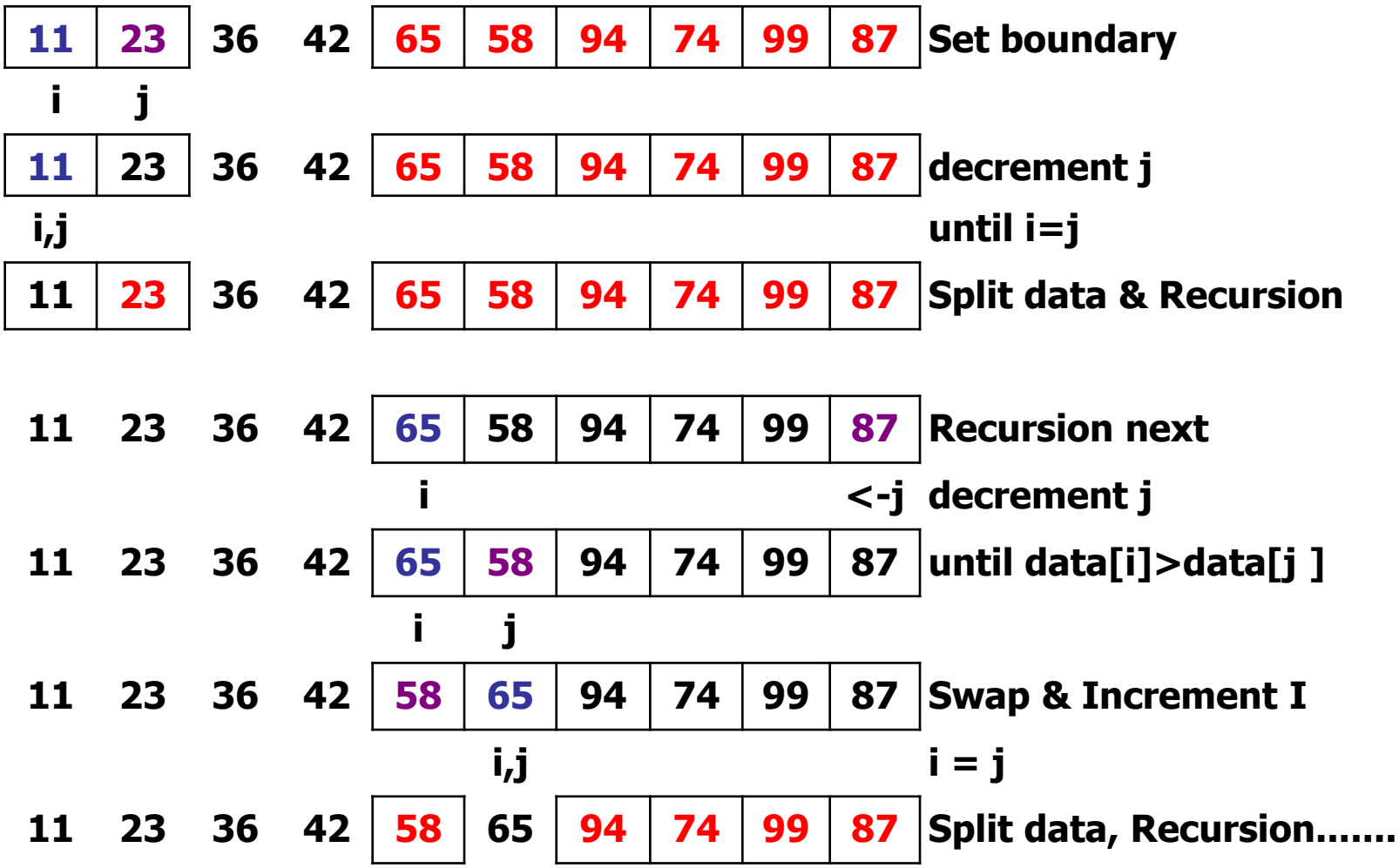
Partitioning



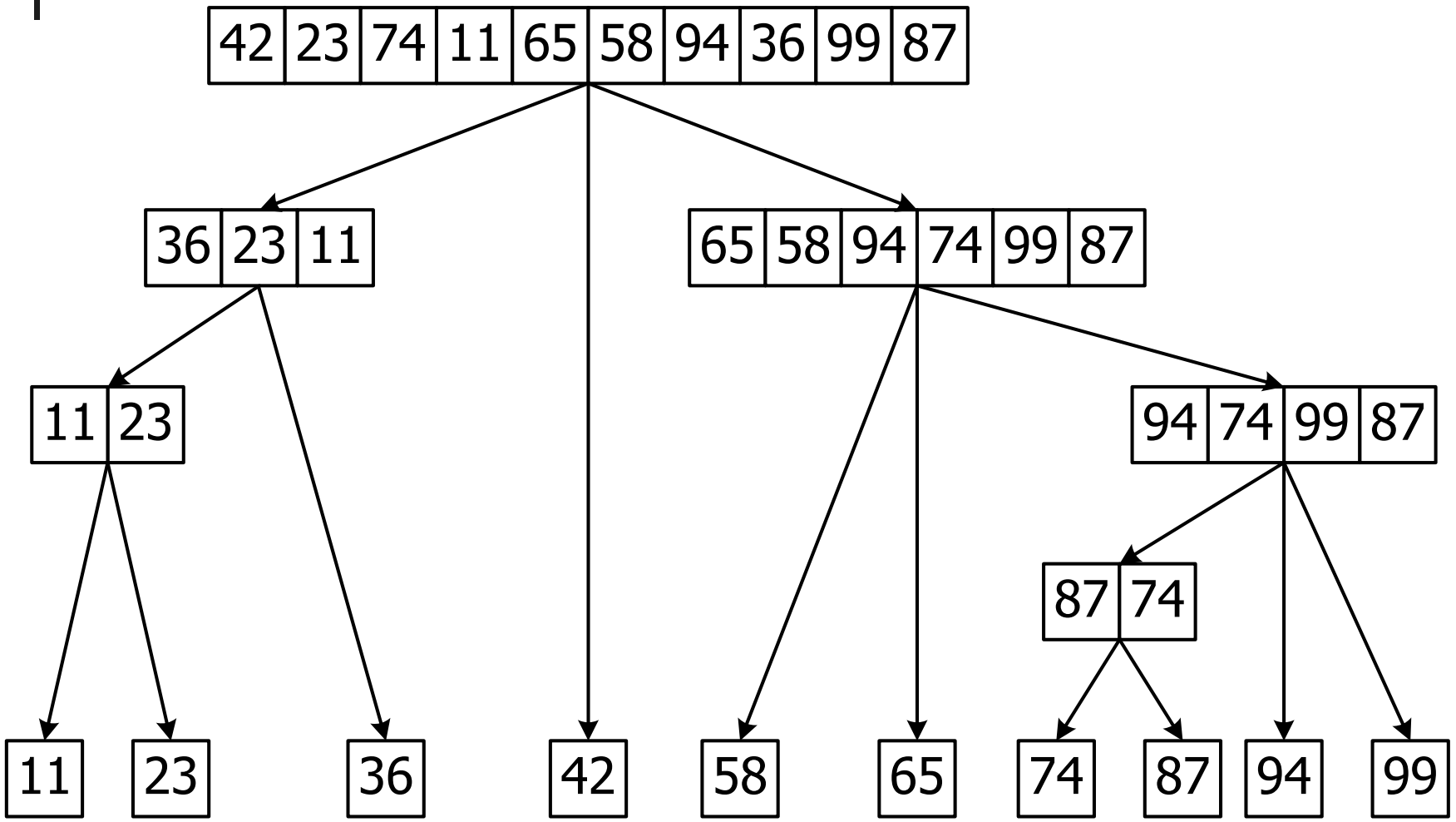
Partitioning



Partitioning



Tree Diagram



Quick Sort Method

```
void  quickSort(int data[], int first, int last)
{ int i=first, j=last ;
  if (first < last)
  { do    /* partitioning */
    { while ((data[i] <= data[j]) && (i < j)) { j--;}
          if (data[i] > data[j]) {swap(&data[i],&data[j]); i++;}
      while ((data[i] <= data[j]) && (i < j)) { i++;}
          if (data[i] > data[j]) {swap(&data[i],&data[j]); j--;}
    } while(i<j);
    if (first < j-1) quickSort(data, first, j-1);
    if (i+1 < last) quickSort(data, i+1, last);
  }
}
```

หาข้อมูลตำแหน่งทางขวา(j) ที่มีค่าน้อยกว่าข้อมูลทางตำแหน่งทางซ้าย(i) แล้วสลับค่า

หาข้อมูลตำแหน่งทางซ้าย(i) ที่มีค่ามากกว่าข้อมูลทางตำแหน่งทางขวา(j) แล้วสลับค่า

หลุดออกจากวนรอบที่ i = j

ตำแหน่ง i=j ข้อมูลถูกต้องแล้ว เหลืออีก 2 กลุ่มย่อยที่ต้องเรียงใหม่

ตรวจสอบว่ามีข้อมูลต้องเรียงใหม่หรือไม่

```
void  swap(int *a, int *b)
{ int c;
    c = *a; *a=*b, *b=c;
}
```

การกำหนดวิธีการเรียงลำดับให้กับภาษาซี

```
typedef struct st_info {
            long long id;
            char name[40];
            double mid,final,atten,total,gpoint;
            char grade[3];} stnode;
int  cmptotal(const void *a, const void *b)
{ double i,j;
  i=(*(stnode *)a).total;
  j=((stnode *)b)->total;
  if ((i-j)>0) return 1;
  else if ((i-j)<0) return -1;
     else return 0;
}
int cmpname(const void *a, const void *b)
{ char x[40],y[40];
  int i;
  strcpy(x,((stnode *)a)->name);
  strcpy(y,(*(stnode *)b).name);
  i = strcmp(x,y);
  return i;
}
```

เขียนแบบ pointer

เขียนแบบ structure

สร้างฟังก์ชัน comparator ให้กับ qsort หรือ bsearch
เลือก element ที่ต้องการ 2 ตัว มาเปรียบเทียบกัน
return ค่าเป็น -1(น้อยกว่า) หรือ 0(เท่ากัน) หรือ 1(มากกว่า)

```
//  quick sort by total score (double)
void Sort_by_score (stnode st[], int stcount) {
      qsort(st,stcount,sizeof(stnode),cmptotal);
}
```

```
// binary search by total score (double)
int score_search(stnode st[], int stcount) {
    bsearch(st,stcount,sizeof(stnode),cmptotal);
}
```

```
//  quick sort by name(string)
void Sort_by_name (stnode st[], int stcount) {
     qsort(st,stcount,sizeof(stnode),cmpname);
}
```

```
// binary search by name (string)
int score_search(stnode st[], int stcount) {
    bsearch(st,stcount,sizeof(stnode),cmpname);
}
```

การกำหนดวิธีเรียงลำดับในภาษาจาวา

- สร้างคลาสเพื่อใช้จัดการข้อมูล 1 ตัว ที่ implements Comparable และ เมธอดชื่อ compareTo เพื่อให้สามารถนำไปใช้กับ Arrays.sort() และ Arrays.binarySearch() ที่มีอยู่

```
class Stnode implements Comparable < Stnode > {
        long id;
        string name;
        double mid,final,atten,total,gpoint;
        string  grade;
  public int compareTo(Stnode x) {//เปรียบเทียบสตริงในส่วนของ word
        return (int) this.name.compareToIgnoreCase(x.name);
  }
```

สร้างเมธอดชื่อ compareTo ที่ return ตัวเลข <0, 0 ,>0 เพื่อเปรียบเทียบค่าคีย์

- จองตัวแปรสำหรับใช้งาน

```
Stnode [] st = new Stnode[1000]; // จองขนาดตัวแปร เพื่อนำไปใส่ข้อมูล
```

```
Arrays.sort (st );
i =Arrays.binarySearch (st, key);
```

สร้างเมธอดชื่อ compare ที่ return ตัวเลข <0, 0 ,>0 เพื่อเปรียบเทียบค่าคีย์

- ถ้าต้องสร้างให้มีการเรียงลำดับมากกว่า 1 วิธี (เช่น เรียงตาม รหัส เรียงตามชื่อ เรียงตาม ...) ให้สร้างเป็นคลาสที่ implements Comparator และ เมธอดชื่อ compare เพิ่มเติมได้

```
class StCmpId implements Comparator <Stnode> {
public int compare (Stnode x, Stnode y) {
        return (int) x.id - y.id ;  }
}
```

```
Collections.sort (data, new StCmpId() );
i =Collections.binarySearch (data, key, new StCmpId() );
```