

---

# CS2040 Data Structures and Algorithms

## Lecture Note #4

---

# Sorting

# Objectives

1

- To learn some classic sorting algorithms

2

- To analyse the running time of these algorithms

3

- To learn concepts such as in-place sorts and stable sorts

4

- Using Java methods to perform sorting

# Programs used in this lecture

- SelectionSort.java
- BubbleSort.java, BubbleSortImproved.java
- InsertionSort.java
- MergeSort.java
- QuickSort.java
- Sort.java, Sort2.java
- Person.java, AgeComparator.java, NameComparator.java, TestComparator.java

# Why Study Sorting?

- When an input is sorted by some **sort key**, many problems become easy (eg. searching, min, max,  $k^{\text{th}}$  smallest, etc.)

Q: What is a sort key?

- Sorting has a variety of interesting algorithmic solutions, which embody many ideas:
  - **Internal** sort vs **external** sort
  - **Iterative** vs **recursive**
  - **Comparison** vs **non-comparison** based
  - **Divide-and-conquer**
  - **Best/worst/average** case time complexity bounds

# Sorting applications

- Uniqueness testing
- Deleting duplicates
- Frequency counting
- Efficient searching
- Dictionary
- Telephone/street directory
- Index of book
- Author index of conference proceedings
- etc.

# Outline

- *Comparison based and Iterative algorithms*
  1. Selection Sort
  2. Bubble Sort
  3. Insertion Sort
- *Comparison based and Recursive algorithms*
  4. Merge Sort
  5. Quick Sort
- *Non-comparison based*
  6. Radix Sort
- 7. Comparison of Sort Algorithms
  - In-place sort
  - Stable sort
- 8. Use of Java Sort Methods

Note: In the lecture, we consider only sorting in **ascending order** of data.

---

# **1 Selection Sort**

---

# 1 Idea of Selection Sort

- Given an array of  $n$  items
  1. Find the **largest** item.
  2. **Swap** it with the item at the **end** of the array.
  3. Go to step 1 by excluding the largest item from the array.



# 1 Selection Sort of 5 integers

|    |    |    |    |    |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

**37** is the largest, swap it with the last element, i.e. **13**.

**Q: How** to find the largest?

|    |    |    |    |    |
|----|----|----|----|----|
| 29 | 10 | 14 | 13 | 37 |
|----|----|----|----|----|

|    |    |    |    |    |
|----|----|----|----|----|
| 13 | 10 | 14 | 29 | 37 |
|----|----|----|----|----|

|    |    |    |    |    |
|----|----|----|----|----|
| 13 | 10 | 14 | 29 | 37 |
|----|----|----|----|----|

|    |    |    |    |    |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

Sorted!

# 1 Code of Selection Sort

```
public static void selectionSort(int[] a) {  
    for (int i = a.length-1; i >= 1; i--) {  
        int index = i; // i is the last item position and  
                       // index is the largest element position  
        // loop to get the largest element  
        for (int j = 0; j < i; j++) {  
            if (a[j] > a[index])  
                index = j; // j is the current largest item  
        }  
        // swap the largest item a[index] with the last item a[i]  
        int temp = a[index];  
        a[index] = a[i];  
        a[i] = temp;  
    }  
}
```

SelectionSort.java

# 1 Analysis of Selection Sort

```
public static void selectionSort(int[] a)
{
    for (int i=a.length-1; i>=1; i--) {
        int index = i;
        for (int j=0; j<i; j++) {
            if (a[j] > a[index])
                index = j;
        }
        SWAP( ... )
    }
}
```

$t_1$  and  $t_2$  = costs of statements in outer and inner blocks.

Number of times the statement is executed:

- $n-1$
- $n-1$
- $(n-1)+(n-2)+\dots+1$   
 $= n \times (n-1)/2$

- $n-1$

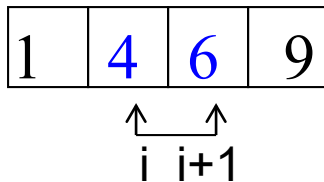
$$\begin{aligned}\text{Total} &= t_1 \times (n-1) \\ &\quad + t_2 \times n \times (n-1)/2 \\ &= O(n^2)\end{aligned}$$

## **2 Bubble Sort**

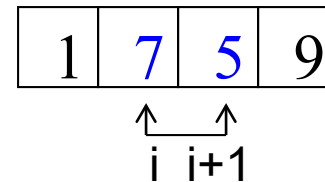
---

## 2 Idea of Bubble Sort

- “Bubble” down the largest item to the end of the array in each iteration by examining the **i-th** and **(i+1)-th** items
- If their values are not in the correct order, i.e.  $a[i] > a[i+1]$ , **swap** them.



// no need to swap



// not in order, need to swap

## 2 Example of Bubble Sort

- The first two passes of Bubble Sort for an array of 5 integers

(a) Pass 1

|    |    |    |    |    |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
| 10 | 29 | 14 | 37 | 13 |
| 10 | 14 | 29 | 37 | 13 |
| 10 | 14 | 29 | 37 | 13 |
| 10 | 14 | 29 | 13 | 37 |

At the end of **pass 1**, the largest item **37** is at the last position.

(b) Pass 2

|    |    |    |    |    |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
| 10 | 14 | 29 | 13 | 37 |
| 10 | 14 | 29 | 13 | 37 |
| 10 | 14 | 13 | 29 | 37 |

At the end of **pass 2**, the second largest item **29** is at the second last position.

## 2 Code of Bubble Sort

```
public static void bubbleSort(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        for (int j = 0; j < a.length - i; j++) {  
            if (a[j] > a[j+1]) { // the larger item bubbles down (swap)  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```

BubbleSort.java

## 2 Analysis of Bubble Sort

- 1 iteration of the inner loop (test and swap) requires time bounded by a constant **c**
- Doubly nested loops:
  - **Outer loop:** exactly  $n-1$  iterations
  - **Inner loop:**
    - When  $i=1$ ,  $(n-1)$  iterations
    - When  $i=2$ ,  $(n-2)$  iterations
    - ...
    - When  $i=(n-1)$ , 1 iteration
- Total number of iterations =  $(n-1) + (n-2) + \dots + 1$   
 $= n \times (n-1) / 2$
- Total time = **c**  $\times n \times (n-1) / 2 = O(n^2)$

```
public static void bubbleSort(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        for (int j = 0; j < a.length - i; j++) {  
            if (a[j] > a[j+1]) { // (swap)  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```



## 2 Bubble Sort can be improved

- Given a sorted input, Bubble Sort still requires  $O(n^2)$  to sort.
- It does not make an effort to check whether the input has been sorted.
- Thus it can be improved by using a **flag**, **isSorted**, as follows (next slide):

## 2 Code of Bubble Sort (Improved version)

```
public static void bubbleSort2(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        → boolean isSorted = true; // isSorted = true if a[] is sorted  
        for (int j = 0; j < a.length-i; j++) {  
            if (a[j] > a[j+1]) { // the larger item bubbles up  
                int temp = a[j]; // and isSorted is set to false,  
                a[j] = a[j+1]; // i.e. the data was not sorted  
                a[j+1] = temp;  
                → isSorted = false;  
            }  
        }  
        → if (isSorted) return; // why?  
    }  
}
```

BubbleSortImproved.java

## 2 Analysis of Bubble Sort (Improved version)

### ■ Worst case

- ❑ Input in **descending order** (any other situation?)
- ❑ How many iterations in the outer loop are needed?  
Answer:  **$n-1$**  iterations
- ❑ Running time remains the same:  **$O(n^2)$**

### ■ Best case

- ❑ Input is already in **ascending order**
- ❑ The algorithm returns after a **single iteration** in the outer loop. (Why?)
- ❑ Running time:  **$O(n)$**

---

## **3 Insertion Sort**

---

# 3 Idea of Insertion Sort

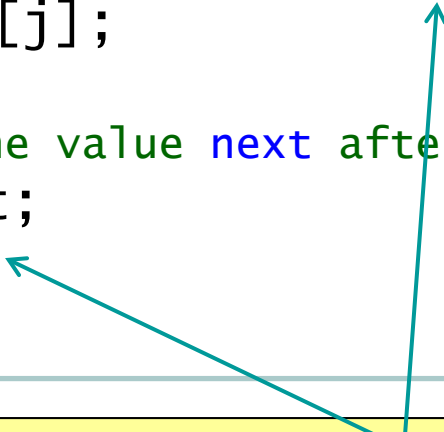
- Arranging a hand of poker cards
  - Start with one card in your hand
  - Pick the next card and **insert** it into its **proper sorted order**
  - Repeat previous step for all the rest of the cards

### 3 Example of Insertion Sort

- $n = 4$
  - Given a seq:
  - $i=1$
  - $i=2$
  - $i=3$
- |  | S1 |    | S2    |
|--|----|----|-------|
|  | 40 | 13 | 20 8  |
|  | 13 | 40 | 20 8  |
|  | 13 | 20 | 40 8  |
|  | 8  | 13 | 20 40 |
- $n$  = no of items to be sorted
  - S1 = sub-array sorted so far
  - S2 = elements yet to be processed
  - In each iteration, how to insert the next element into S1 efficiently?

### 3 Code of Insertion Sort

```
public static void insertionSort(int[] a) {  
    for (int i=1;i<a.length;i++) { //Q: why i starts from 1?  
        // a[i] is the next data to insert  
        int next = a[i];  
        // scan backwards to find a place. Q: why not scan forwards?  
        int j; // Q: why is j declared here?  
        // Q: what if a[j] <= next?  
        for (j=i-1; j>=0 && a[j]>next; j--)  
            a[j+1] = a[j];  
  
        // Now insert the value next after index j at the end of loop  
        a[j+1] = next;  
    }  
}
```



InsertionSort.java

**Q:** Can we replace these two “next” with a[i]?

**Ans:** No! (Why?)

### 3 Analysis of Insertion Sort

- Outer loop executes exactly  $n-1$  times
- Number of times inner loop executes depends on the inputs:
  - **Best case:** array already sorted, hence  $(a[j] > \text{next})$  is always false
    - No shifting of data is necessary; Inner loop not executed at all.
  - **Worst case:** array reversely sorted, hence  $(a[j] > \text{next})$  is always true
    - Need  $i$  shifts for  $i = 1$  to  $n-1$ .
    - Insertion always occurs at the front.
- Therefore, the **best case** running time is  $O(n)$ . (Why?)
- The **worst case** running time is  $O(n^2)$ . (Why?)

```
... insertionSort(int[] a) {  
    for (int i=1; i<a.length; i++) {  
        int next = a[i];  
        int j;  
        for (j=i-1; j>=0 && a[j]>next; j--)  
            a[j+1] = a[j];  
  
        a[j+1] = next;  
    }  
}
```



# **4 Merge Sort**

---

## 4 Idea of Merge Sort (1/3)

- Suppose we **only know how to merge** two sorted lists of elements into one combined list
- Given an unsorted list of  $n$  elements
- Since each element is a sorted list, we can repeatedly...
  - **Merge** each pair of lists, each list containing one element, into a sorted list of 2 elements.
  - **Merge** each pair of sorted lists of 2 elements into a sorted list of 4 elements.
  - ...
  - The final step **merges** 2 sorted lists of  $n/2$  elements to obtain a sorted list of  $n$  elements.

## 4 Idea of Merge Sort (2/3)

- **Divide-and-conquer** method solves problem by three steps:
  - **Divide Step:** divide the larger problem into smaller problems.
  - **(Recursively)** solve the smaller problems.
  - **Conquer Step:** combine the results of the smaller problems to produce the result of the larger problem.

## 4 Idea of Merge Sort (3/3)

- **Merge Sort** is a divide-and-conquer sorting algorithm
  - **Divide Step:** Divide the array into two (equal) halves.
  - **(Recursively)** Merge sort the two halves.
  - **Conquer Step:** Merge the two sorted halves to form a sorted array.
- Q: What are the base cases?

## 4 Example of Merge Sort

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 7 | 2 | 6 | 3 | 8 | 4 | 5 |
|---|---|---|---|---|---|---|

Divide into  
two halves

|   |   |   |   |
|---|---|---|---|
| 7 | 2 | 6 | 3 |
|---|---|---|---|

|   |   |   |
|---|---|---|
| 8 | 4 | 5 |
|---|---|---|

Recursively  
sort the halves

|   |   |   |   |
|---|---|---|---|
| 2 | 3 | 6 | 7 |
|---|---|---|---|

|   |   |   |
|---|---|---|
| 4 | 5 | 8 |
|---|---|---|

Merge the halves

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|

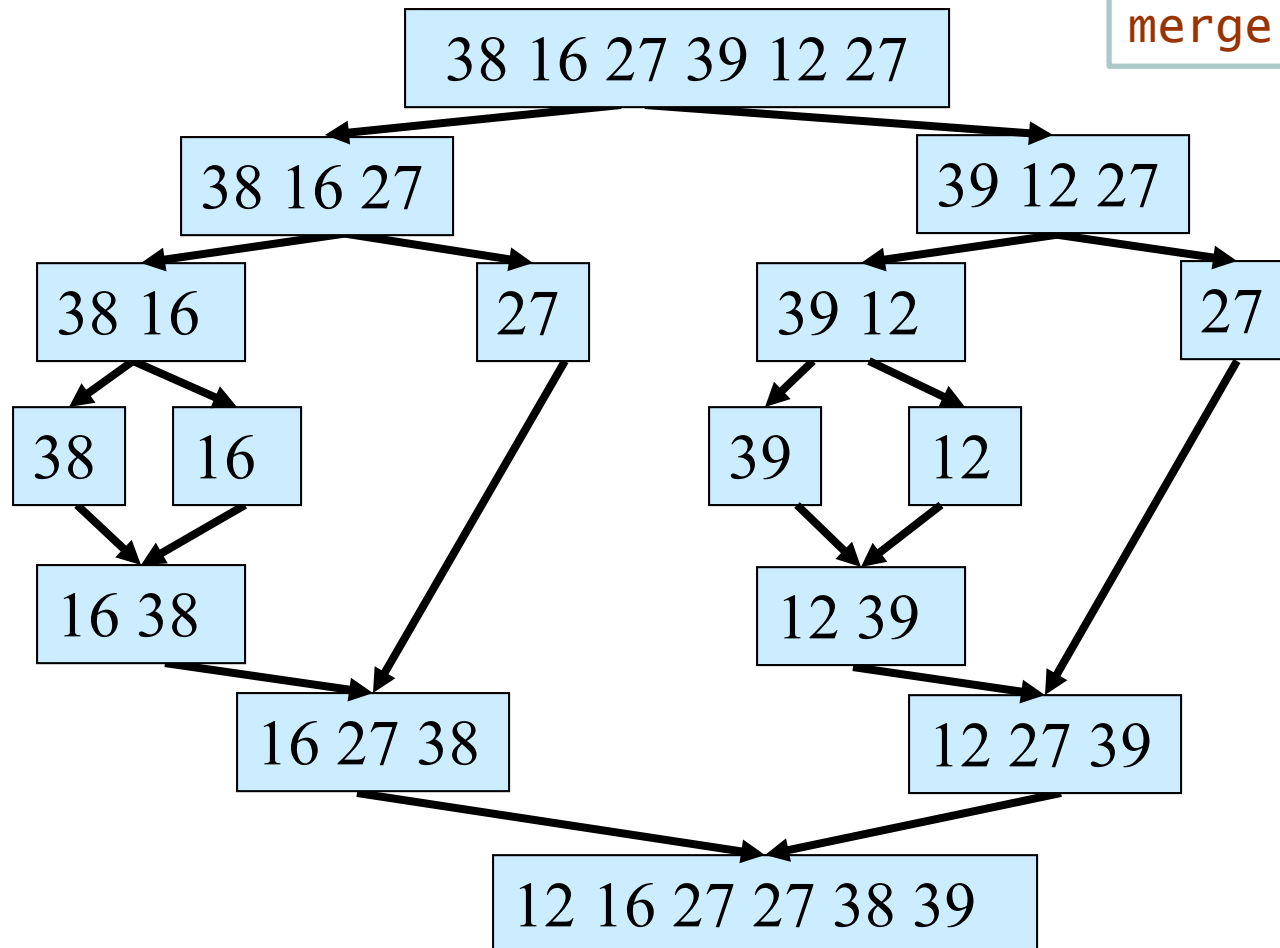
## 4 Code of Merge Sort

```
... mergeSort(int[] a, int i, int j) {  
    // to sort data from a[i] to a[j], where i<j  
    if (i < j) { // Q: what if i >= j?  
        int mid = (i+j)/2; // divide  
        mergeSort(a, i, mid); // recursion  
        mergeSort(a, mid+1, j);  
        merge(a,i,mid,j); //conquer: merge a[i..mid] and  
                           //a[mid+1..j] back into a[i..j]  
    }  
}
```

MergeSort.java

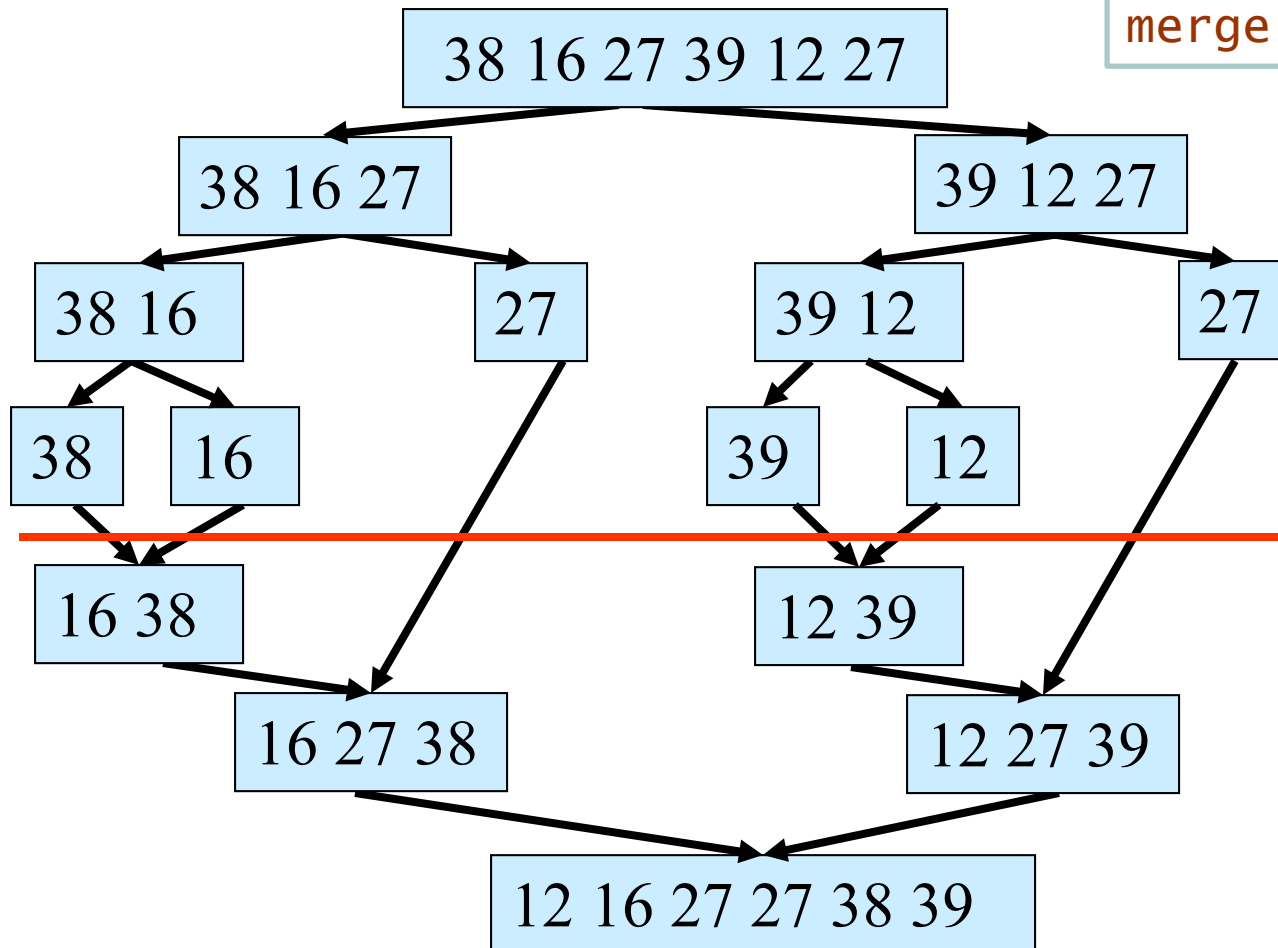
## 4 Merge Sort of a 6-element Array (1/2)

```
mergeSort(a,i,mid);  
mergeSort(a,mid+1,j);  
merge(a,i,mid,j);
```



## 4 Merge Sort of a 6-element Array (2/2)

```
mergeSort(a,i,mid);  
mergeSort(a,mid+1,j);  
merge(a,i,mid,j);
```

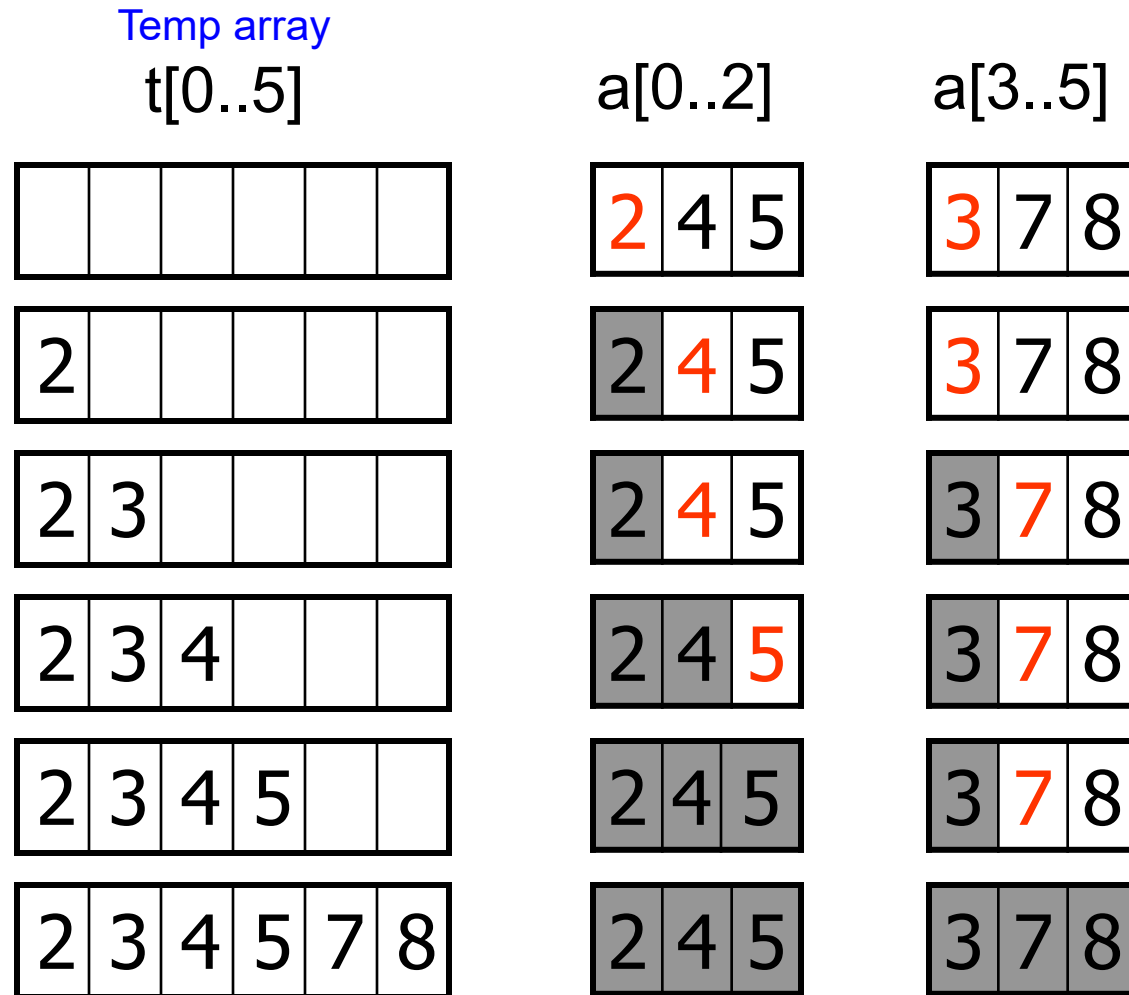


**Divide** phase:  
**Recursive** call to  
mergeSort

**Conquer** phase:  
**Merge** steps  
The sorting is done  
here



## 4 How to Merge 2 Sorted Subarrays?



## 4 Merge Algorithm (1/2)

```
... merge(int[] a, int i, int mid, int j) {  
    // Merges the 2 sorted sub-arrays a[i..mid] and  
    // a[mid+1..j] into one sorted sub-array a[i..j]  
  
    int[] temp = new int[j-i+1]; // temp storage  
    int left = i, right = mid+1, it = 0;  
    // it = next index to store merged item in temp[]  
    // Q: what are left and right?  
  
    while (left<=mid && right<=j) { // output the smaller  
        if (a[left] <= a[right])  
            temp[it++] = a[left++];  
        else  
            temp[it++] = a[right++];  
    }  
}
```

## 4 Merge Algorithm (2/2)

```
// Copy the remaining elements into temp. Q: why?
while (left<=mid) temp[it++] = a[left++];
while (right<=j)  temp[it++] = a[right++];
// Q: will both the above while statements be executed?

// Copy the result in temp back into
// the original array a
for (int k = 0; k < temp.length; k++)
    a[i+k] = temp[k];
}
```

## 4 Analysis of Merge Sort (1/3)

- In Merge Sort, the bulk of work is done in the Merge step  
`merge(a, i, mid, j)`
- Total number of items =  $k = j - i + 1$ 
  - Number of comparisons  $\leq k - 1$  (Q: Why not =  $k - 1$ ?)
  - Number of moves from original array to temp array =  $k$
  - Number of moves from temp array to original array =  $k$
- In total, number of operations  $\leq 3k - 1 = O(k)$
- How many times is `merge()` called?

```
... mergeSort(int[] a, int i, int j) {  
    if (i < j) {  
        int mid = (i+j)/2;  
        mergeSort(a, i, mid);  
        mergeSort(a, mid+1, j);  
        merge(a, i, mid, j);  
    }  
}
```

# 4 Analysis of Merge Sort (2/3)

Level 0:  
Mergesort  $n$  items

Level 1:  
2 calls to Mergesort  $n/2$  items

Level 2:  
4 calls to Mergesort  $n/2^2$  items

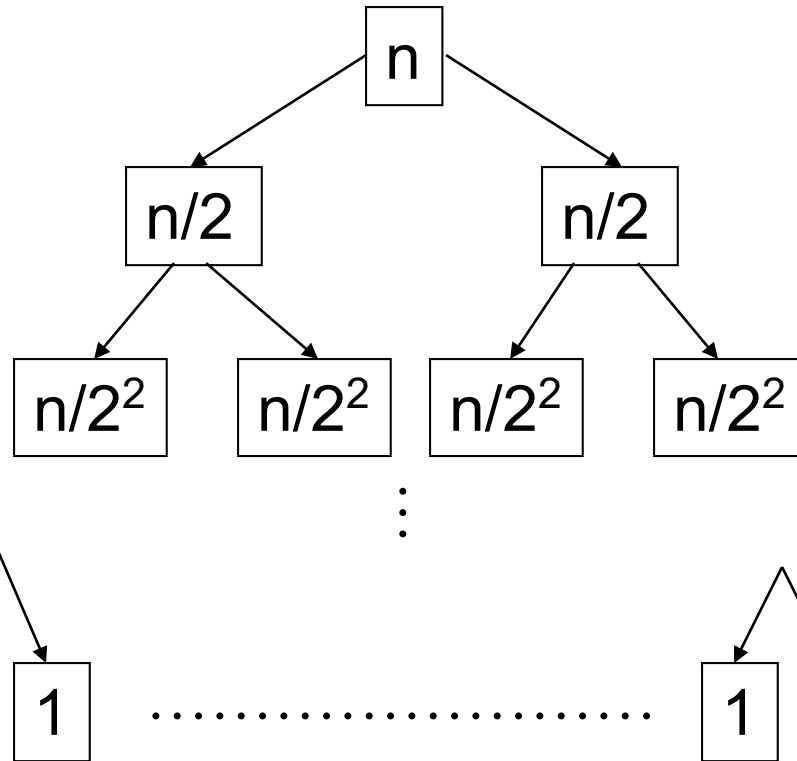
Level ( $\log n$ ):  
 $n$  calls to Mergesort 1 item

Level 0:  
0 call to Merge

Level 1:  
1 calls to Merge

Level 2:  
2 calls to Merge

Level ( $\log n$ ):  
 $2^{(\log n) - 1} (= n/2)$   
calls to Merge



Let  $h$  be the maximum level, ie. Mergesort 1 item.  
 $n/(2^h) = 1 \quad \rightarrow \quad n = 2^h \quad \rightarrow \quad h = \log n$

## 4 Analysis of Merge Sort (3/3)

- Level 0: 0 call to Merge
- Level 1: 1 call to Merge with  $n/2$  items each,  
 $O(1 \times 2 \times n/2) = O(n)$  time
- Level 2: 2 calls to Merge with  $n/2^2$  items each,  
 $O(2 \times 2 \times n/2^2) = O(n)$  time
- Level 3:  $2^2$  calls to Merge with  $n/2^3$  items each,  
 $O(2^2 \times 2 \times n/2^3) = O(n)$  time
- ...
- Level ( $\log n$ ):  $2^{(\log n)-1} (= n/2)$  calls to Merge with  $n/2^{\log n}$   
(= 1) item each,  
 $O(n/2 \times 2 \times 1) = O(n)$  time
- In total, running time =  $(\log n) \times O(n) = O(n \log n)$

## 4 Drawbacks of Merge Sort

- Implementation of merge() is not straightforward
- Requires **additional temporary arrays** and to copy the merged sets stored in the temporary arrays to the original array
- Hence, **additional** space complexity =  $O(n)$

# **5 Quick Sort**

---



## 5 Idea of Quick Sort

- Quick Sort is a **divide-and-conquer** algorithm
- **Divide Step:** Choose a **pivot** item **p** and partition the items of  $a[i..j]$  into **2 parts** so that
  - Items in the first part are  $< p$ , and
  - Items in the second part are  $\geq p$ .
- **Recursively** sort the 2 parts
- **Conquer Step:** Do nothing! No merging is needed.
- What are the base cases?

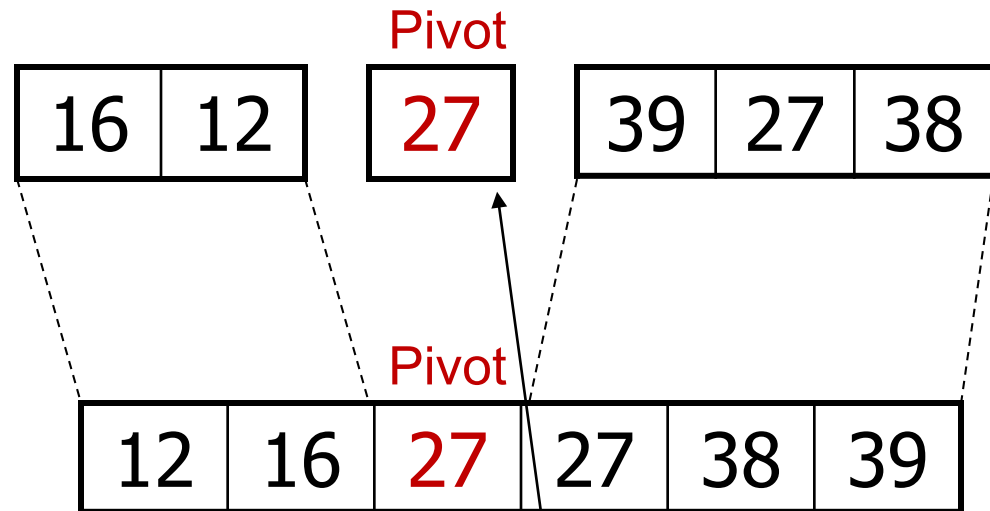
## 5 Example of Quick Sort

Choose the **1<sup>st</sup>** item as **pivot**

Pivot

|           |    |    |    |    |    |
|-----------|----|----|----|----|----|
| <b>27</b> | 38 | 12 | 39 | 27 | 16 |
|-----------|----|----|----|----|----|

Partition  $a[]$  about  
the pivot 27



Recursively sort  
the two parts

Note that after the partition,  
the pivot is moved to its **final position**!  
**No** merge phase is needed.

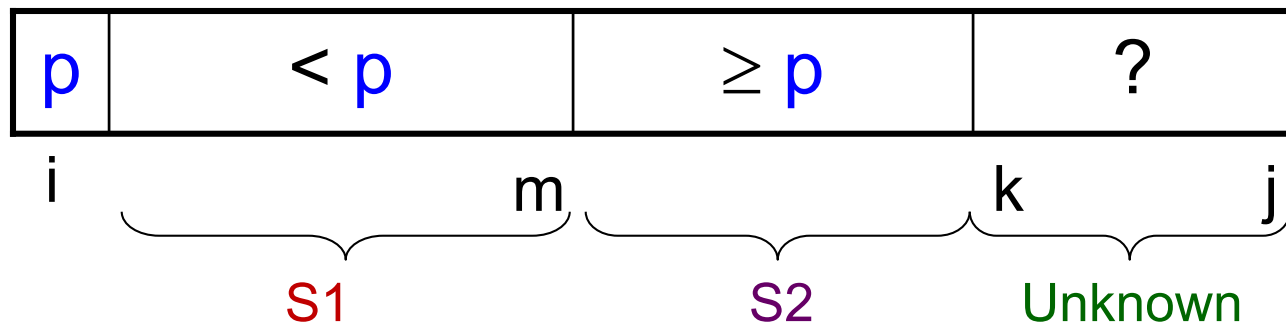
## 5 Code of Quick Sort

```
... quickSort(int[] a, int i, int j) {  
    if (i < j) { // Q: what if i >= j?  
        int pivotIdx = partition(a, i, j);  
        quickSort(a, i, pivotIdx-1);  
        quickSort(a, pivotIdx+1, j);  
        // No conquer part!  
    }  
}
```

QuickSort.java

## 5 Partition algorithm idea (1/4)

- To partition  $a[i..j]$ , we choose  $a[i]$  as the **pivot**  $p$ .
  - Why choose  $a[i]$ ? Are there other choices?
- The remaining items (i.e.  $a[i+1..j]$ ) are divided into 3 regions:
  - **S1** =  $a[i+1..m]$  where items  $< p$
  - **S2** =  $a[m+1..k-1]$  where item  $\geq p$
  - **Unknown** (unprocessed) =  $a[k..j]$ , where items are yet to be assigned to S1 or S2.



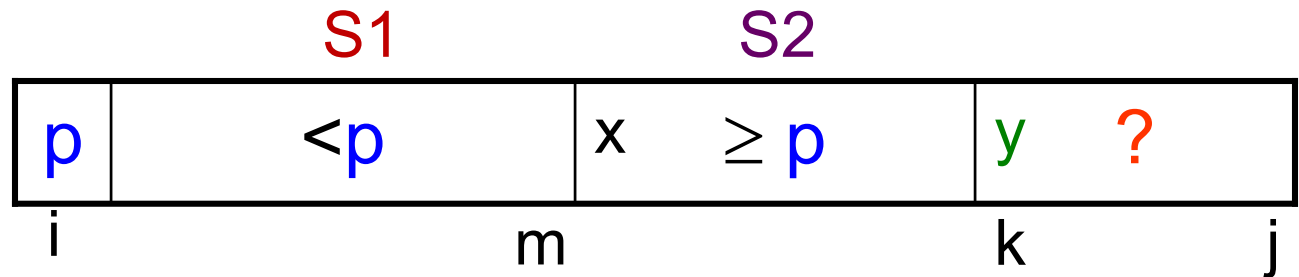
## 5 Partition algorithm idea (2/4)

- Initially, regions **S1** and **S2** are empty. All items excluding **p** are in the **unknown** region.
- Then, for each item  $a[k]$  (for  $k=i+1$  to  $j$ ) in the **unknown** region, compare  $a[k]$  with **p**:
  - If  $a[k] \geq p$ , put  $a[k]$  into **S2**.
  - Otherwise, put  $a[k]$  into **S1**.
- Q: How about if we change  $\geq$  to  $>$  in the condition part?

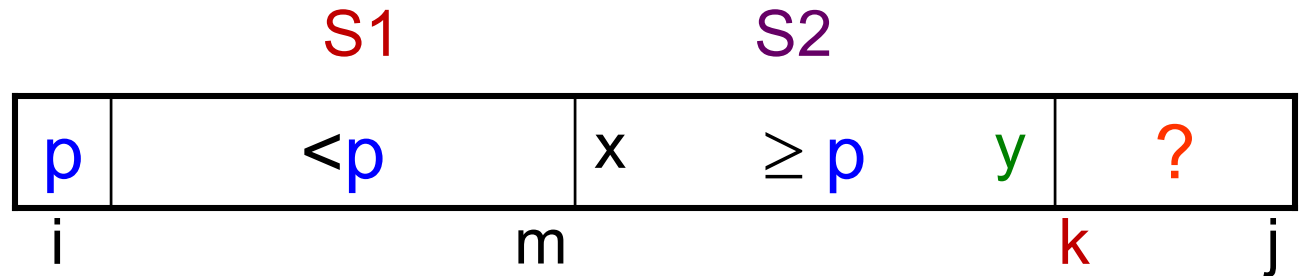
## 5 Partition algorithm idea (3/4)

### ■ Case 1:

If  $a[k] = y \geq p$ ,



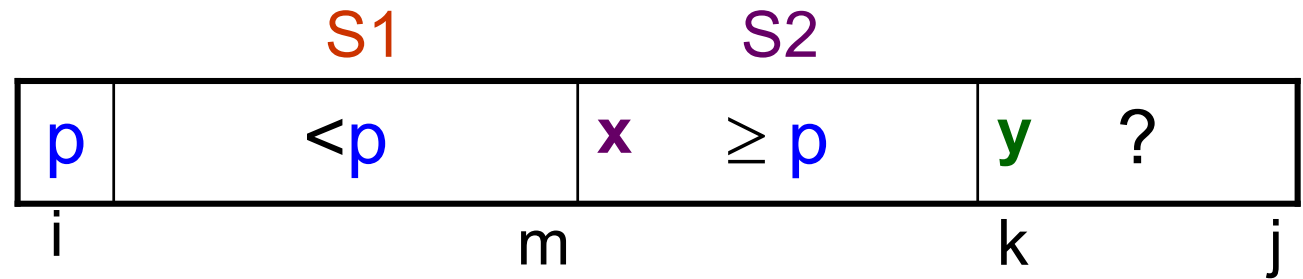
Increment  $k$



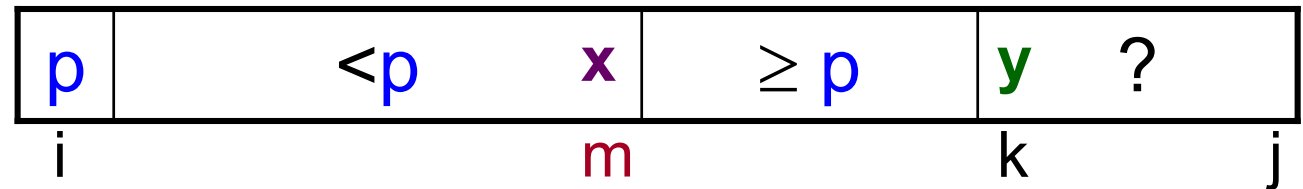
## 5 Partition algorithm idea (4/4)

### ■ Case 2:

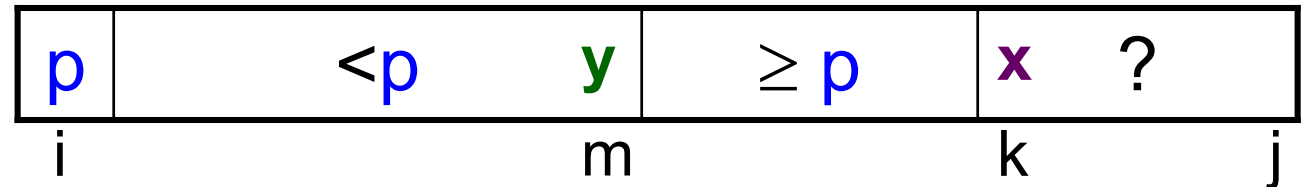
If  $a[k]=y < p$



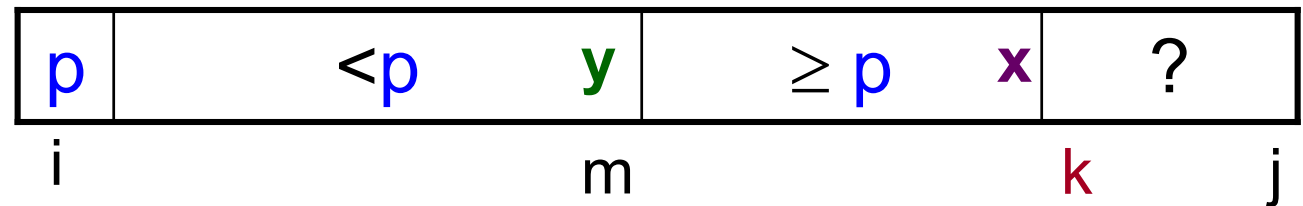
Increment *m*



Swap x and y



Increment *k*



## 5 Code of Partition Algorithm

```
... partition(int[] a, int i, int j) {  
    // partition data items in a[i..j]  
    int p = a[i]; // p is the pivot, the ith item  
    int m = i;    // Initially S1 and S2 are empty  
    for (int k=i+1; k<=j; k++) { //process unknown region  
        if (a[k] < p) { // case 2: put a[k] to S1  
            m++;  
            swap(a,k,m);  
        } else { // case 1: put a[k] to S2. Do nothing!  
        } // else part should be removed since it is empty  
    }  
    swap(a,i,m); // put the pivot at the right place  
    return m;    // m is the pivot's final position  
}
```

- As there is only one 'for' loop and the size of the array is  $n = j - i + 1$ , so the complexity for partition() is  $O(n)$



## 5 Partition Algorithm: Example

| Pivot | Unknown |    |    |    |    |
|-------|---------|----|----|----|----|
| 27    | 38      | 12 | 39 | 27 | 16 |

| Pivot | $S_2$ | Unknown |    |    |    |
|-------|-------|---------|----|----|----|
| 27    | 38    | 12      | 39 | 27 | 16 |



| Pivot | $S_1$ | $S_2$ | Unknown |    |    |
|-------|-------|-------|---------|----|----|
| 27    | 12    | 38    | 39      | 27 | 16 |

| Pivot | $S_1$ | $S_2$ | Unknown |    |    |
|-------|-------|-------|---------|----|----|
| 27    | 12    | 38    | 39      | 27 | 16 |

| Pivot | $S_1$ | $S_2$ | Unknown |    |    |
|-------|-------|-------|---------|----|----|
| 27    | 12    | 38    | 39      | 27 | 16 |



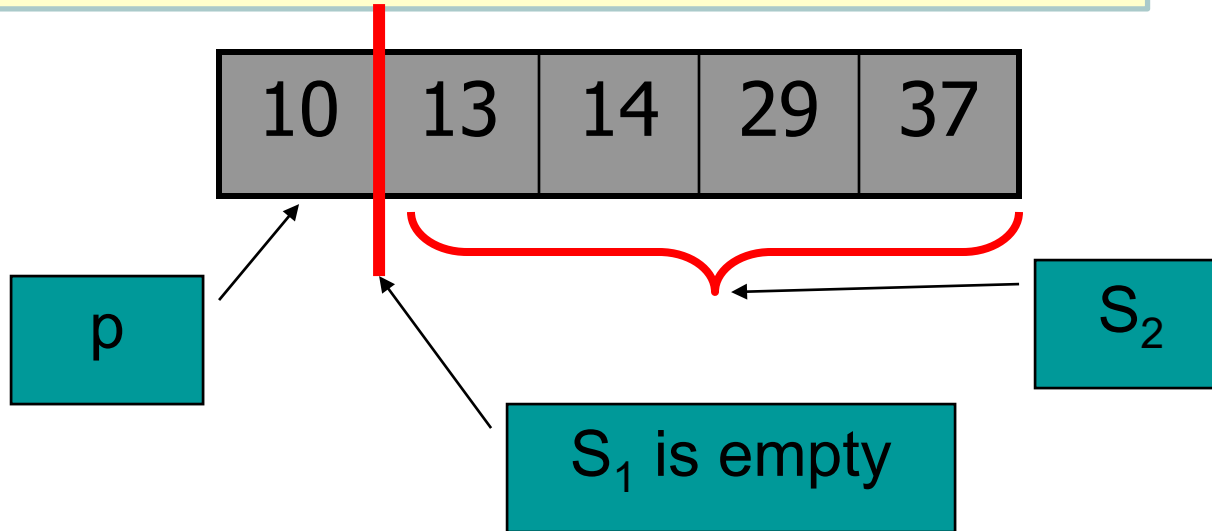
| Pivot | $S_1$ | $S_2$ | Unknown |    |    |
|-------|-------|-------|---------|----|----|
| 27    | 12    | 16    | 39      | 27 | 38 |

| $S_1$ | Pivot | $S_2$ | Unknown |    |    |
|-------|-------|-------|---------|----|----|
| 16    | 12    | 27    | 39      | 27 | 38 |

Same value, no need to swap them.

## 5 Analysis of Quick Sort: Worst Case (1/2)

When  $a[0..n-1]$  is in increasing order:



What is the index returned by `partition()`?

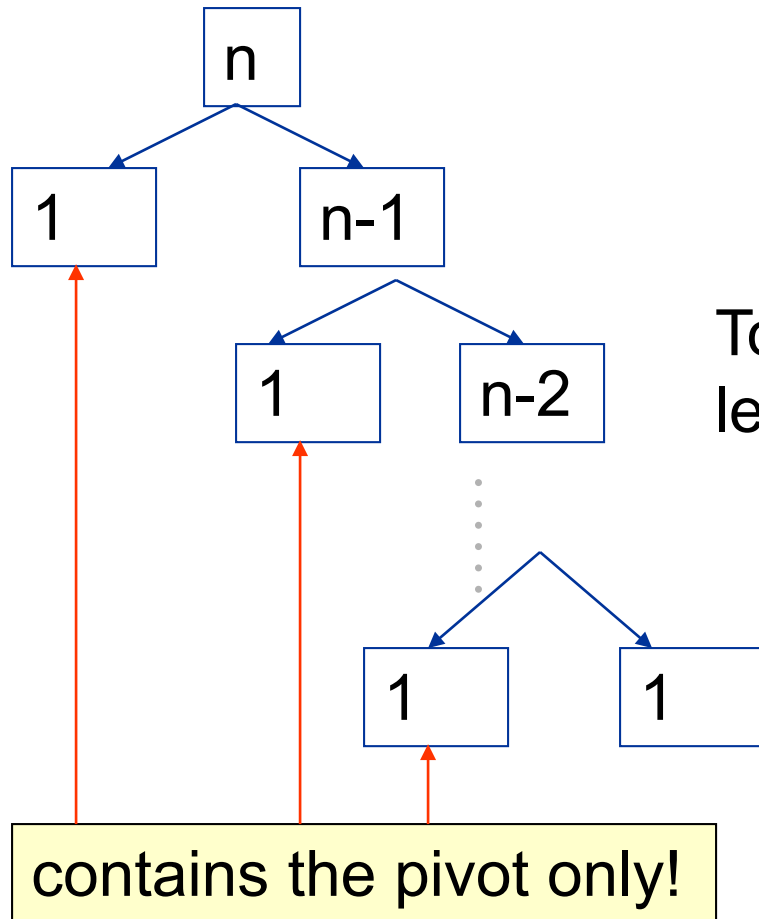
`swap(a,i,m)` will swap the pivot with itself!

The left partition ( $S_1$ ) is **empty** and

The right partition ( $S_2$ ) is the rest excluding the pivot.

What if the array is in decreasing order?

## 5 Analysis of Quick Sort: Worst Case (2/2)



Total no. of  
levels =  $n$

As each partition takes linear time, the algorithm in its worst case has  $n$  levels and hence it takes time  $n + (n-1) + \dots + 1 = O(n^2)$

## 5 Analysis of Quick Sort: Best/Average case

- **Best case** occurs when partition always splits the array into **2 equal halves**
  - Depth of recursion is  **$\log n$** .
  - Each level takes  **$n$**  or fewer comparisons, so the time complexity is  **$O(n \log n)$**
- In practice, worst case is rare, and on the average, we get some good splits and some bad ones
- **Average time** is  **$O(n \log n)$**  → \*especially true if using randomized pivoting (randomly choose the pivot rather than keep using the 1<sup>st</sup> element). Can get even better theoretical bounds if use median as pivot (outside scope of this course).

# **6 Radix Sort**

---

## 6 Idea of Radix Sort

- Treats each data to be sorted as a **character string**.
- It is not using comparison, i.e., **no comparison** among the data is needed.
- Hence it is a **non-comparison based sort** (the preceding sorting algorithms are called comparison based sorts)
- In each iteration, organize the data into groups according to the **next** character in each data.

## 6 Radix Sort of Eight Integers

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150

Original integers

(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)

Grouped by fourth digit

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004

Combined

(000**4**) (022**2**, 012**3**) (215**0**, 215**4**) (156**0**, 106**1**) (028**3**)

Grouped by third digit

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

Combined

(000**4**, 106**1**) (012**3**, 215**0**, 215**4**) (022**2**, 028**3**) (156**0**)

Grouped by second digit

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

Combined

(000**4**, 012**3**, 022**2**, 028**3**) (106**1**, 156**0**) (215**0**, 215**4**)

Grouped by first digit

0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

Combined (sorted)

## 6 Pseudocode and Analysis of Radix Sort

```
radixSort(int[] array, int n, int d) {  
    // Sorts n d-digit numeric strings in the array.  
    for (j = d down to 1) { // for digits in last position to 1st position  
        initialize 10 groups (queues) to empty // Q: why 10 groups?  
  
        for (i=0 through n-1) {  
            k = jth digit of array[i]  
            place array[i] at the end of group k  
        }  
        Replace array with all items in group 0, followed by all items  
        in group 1, and so on.  
    }  
}
```

Complexity is  $O(d \times n)$  where  $d$  is the maximum number of digits of the  $n$  numeric strings in the array. Since  $d$  is fixed or bounded, so the complexity is  $O(n)$ .



---

# **7 Comparison of Sorting Algorithms**

---

## 7 In-place Sort

- A sorting algorithm is said to be an **in-place** sort if it requires only a **constant amount**, i.e.  $O(1)$ , of **extra space** during the sorting process.
- Merge Sort is not in-place. (Why?)
- How about Quick Sort?

## 7 Stable Sort

- A sorting algorithm is **stable** if the **relative order of elements with the same key value** is preserved by the algorithm.
- Example 1 – An application of stable sort:
  - Assume that names have been sorted in alphabetical order.
  - Now, if this list is sorted again by tutorial group number, a **stable sort** algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names.
- Quick Sort and Selection Sort are not stable. (Why?)

## 7 Non-Stable Sort

- Example 2 – Quick Sort and Selection Sort are not stable:

### Quick sort:

**1285** 150 5\* 4746 602 5<sup>+</sup> 8356 // pivot in bold

**1285** (150 5\* 602 5<sup>+</sup>) (4746 8356)

5<sup>+</sup> 150 5\* 602 **1285** 4746 8356 //pivot swapped with the last one in S1  
// the **2 5's** are in different order of the initial list

**Selection sort:** select the largest element and swap with the last one

4746 1285 5\* 602 **8356** 5<sup>+</sup> 277

**4746** 1285 5\* 602 277 5<sup>+</sup> (8356)

5<sup>+</sup> 1285 5\* 602 277 (4746 8356)

// the **2 5's** are in different order of the initial list

# 7 Summary of Sorting Algorithms

|                                       | Worst Case           | Best Case     | In-place? | Stable? |
|---------------------------------------|----------------------|---------------|-----------|---------|
| Selection Sort                        | $O(n^2)$             | $O(n^2)$      | Yes       | No      |
| Insertion Sort                        | $O(n^2)$             | $O(n)$        | Yes       | Yes     |
| Bubble Sort                           | $O(n^2)$             | $O(n^2)$      | Yes       | Yes     |
| Bubble Sort 2<br>(improved with flag) | $O(n^2)$             | $O(n)$        | Yes       | Yes     |
| Merge Sort                            | $O(n \log n)$        | $O(n \log n)$ | No        | Yes     |
| Radix Sort (non-comparison based)     | $O(n)$ (see notes 1) | $O(n)$        | No        | Yes     |
| Quick Sort                            | $O(n^2)$             | $O(n \log n)$ | Yes       | No      |

**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 sorting.

---

## **8 Use of Java Sort Methods**

---

## 8 Java Sort Methods (in Arrays class)

```
static void sort(byte[] a)
static void sort(byte[] a, int fromIndex, int toIndex)
static void sort(char[] a)
static void sort(char[] a, int fromIndex, int toIndex)
static void sort(double[] a)
static void sort(double[] a, int fromIndex, int toIndex)
static void sort(float[] a)
static void sort(float[] a, int fromIndex, int toIndex)
static void sort(int[] a)
static void sort(int[] a, int fromIndex, int toIndex)
static void sort(long[] a)
static void sort(long[] a, int fromIndex, int toIndex)
static void sort(Object[] a)
static void sort(Object[] a, int fromIndex, int toIndex)
static void sort(short[] a)
static void sort(short[] a, int fromIndex, int toIndex)
static <T> void sort(T[] a, Comparator<? super T> c)
static <T> void sort(T[] a, int fromIndex, int toIndex,
                    Comparator<? super T> c)
```

## 8 To use `sort( )` in Arrays

- The entities to be sorted must be stored in an **array** first.
- If they are stored in a **list**, then we have to use **`Collections.sort()`**
- If the data to be sorted are not primitive, then **`Comparator`** must be defined and used

**Note:** **`Collections`** is a Java public class and **`Comparator`** is a public interface. Comparators can be passed to a sort method (such as `Collections.sort()`) to allow precise control over the sort order.



## 8 Simple program using Collections.sort()

```
import java.util.*;
public class Sort {
    public static void main(String args[]) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Sort.java

- Run the program:  
`java Sort We walk the line`
- The following output is produced:  
`[We, line, the, walk]`

**Note:** `Arrays` is a Java public class and `asList()` is a method of `Arrays` which returns a fixed-size list backed by the specified array.

## 8 Another solution using **Arrays.sort()**

```
import java.util.*;
public class Sort2 {
    public static void main(String args[]) {
        Arrays.sort(args);
        System.out.println(Arrays.toString(args));
    }
}
```

Sort2.java

- Run the program:  
`java Sort2 We walk the line`
- The following output is produced:  
`[We, line, the, walk]`

## 8 Example: class Person

```
class Person {  
    public String name;  
    public int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() { return name; }  
    public int getAge() { return age; }  
    public String toString() {  
        return name + " - " + age;  
    }  
}
```

Person.java

# 8 Comparator interface

java.util

## Interface Comparator<T>

### Type Parameters:

T - the type of objects that may be compared by this comparator

### All Known Implementing Classes:

Collator, RuleBasedCollator

### Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

### Method Summary

| All Methods       | Static Methods | Instance Methods   | Abstract Methods | Default Methods |
|-------------------|----------------|--|------------------|-----------------|
| Modifier and Type |                | Method and Description   |                  |                 |
| int               |                | <b>compare</b> (T o1, T o2)<br>Compares its two arguments for order.                             |                  |                 |
| boolean           |                | <b>equals</b> (Object obj)<br>Indicates whether some other object is "equal to" this comparator. |                  |                 |

## 8 Comparator: AgeComparator

```
import java.util.Comparator;
class AgeComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        // Returns the difference:
        // if positive, age of p1 is greater than p2
        // if zero, the ages are equal
        // if negative, age of p1 is less than p2
        return p1.getAge() - p2.getAge();
    }

    public boolean equals(Object obj) {
        // Simply checks to see if we have the same comparator object
        return this == obj;
    }
} // end AgeComparator
```

AgeComparator.java

**Note:** `compare()` and `equals()` are two methods of the interface `Comparator`. Need to implement them.

## 8 Comparator: NameComparator

```
import java.util.Comparator;
class NameComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // Compares its two arguments for order by name
        return p1.getName().compareTo(p2.getName());
    }

    public boolean equals(Object obj) {
        // simply checks to see if we have the same object
        return this == obj;
    }
} // end NameComparator
```

NameComparator.java

## 8 TestComparator (1/3)

```
import java.util.*;

public class TestComparator {

    public static void main(String args[]) {
        NameComparator nameComp = new NameComparator();
        AgeComparator ageComp = new AgeComparator();
        Person[] p = new Person[5];

        p[0] = new Person("Michael", 15);
        p[1] = new Person("Mimi", 9);
        p[2] = new Person("Sarah", 12);
        p[3] = new Person("Andrew", 15);
        p[4] = new Person("Mark", 12);
        List<Person> list = Arrays.asList(p);
    }
}
```

TestComparator.java

## 8 TestComparator (2/3)

```
System.out.println("Sorting by age:");
Collections.sort(list, ageComp);
System.out.println(list + "\n");

List<Person> list2 = Arrays.asList(p);
System.out.println("Sorting by name:");
Collections.sort(list2, nameComp);
System.out.println(list2 + "\n");

System.out.println("Now sort by name, then sort by age:");
Collections.sort(list2, ageComp); // list2 is already
                                   // sorted by name

System.out.println(list2);
} // end main

} // end TestComparator
```

TestComparator.java



## 8 TestComparator (3/3)

```
java TestComparator
```

Sorting by age:

[Mimi - 9, Sarah - 12, Mark - 12, Michael - 15, Andrew - 15]

Sorting by name:

[Andrew - 15, Mark - 12, Michael - 15, Mimi - 9, Sarah - 12]

Now sort by name, then sort by age:

[Mimi - 9, Mark - 12, Sarah - 12, Andrew - 15, Michael - 15]

## 8 Another solution using **Arrays.sort( )**

We can replace the statements

```
List<Person> list = Arrays.asList(p);  
System.out.println("Sorting by age:");  
Collections.sort(list, ageComp);  
System.out.println(list + "\n");
```

with

```
System.out.println("Sorting by age using Arrays.sort():");  
Arrays.sort(p, ageComp);  
System.out.println(Arrays.toString(p) + "\n");
```

# Summary

- We have introduced and analysed some classic sorting algorithms.
- Merge Sort and Quick Sort are in general faster than Selection Sort, Bubble Sort and Insertion Sort.
- The sorting algorithms discussed here are comparison based sorts, except for Radix Sort which is non-comparison based.
- $O(n \log n)$  is the best possible worst-case running time for comparison based sorting algorithms.
- There exist Java methods to perform sorting.

End of file