COSC 121

Computer Programming II

Sorting

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled "Introduction to Java Programming, 10th Ed.", Pearson Edu. Inc.

Outline

Previously:

- Algorithms Efficiency
- Best, Average, and Worst Cases
- Simple Sort Techniques
  - Selection Sort
  - Bubble Sort
  - Insertion Sort

Today:

- More Advanced Sort Techniques
  - Merge Sort
  - Quick Sort
- Positive Integers Sorting: Bucket and Radix Sort

More Advanced Sort Techniques

- Merge Sort
- Quick Sort

Wikipedia

Merge Sort

John von Neumann, **1945**

Merge Sort

IDEA: Divide the array into two halves and apply a merge sort on each half *recursively*. After the two halves are sorted, merge them.

6 5 3 1 8 7 2 4

Algorithm:

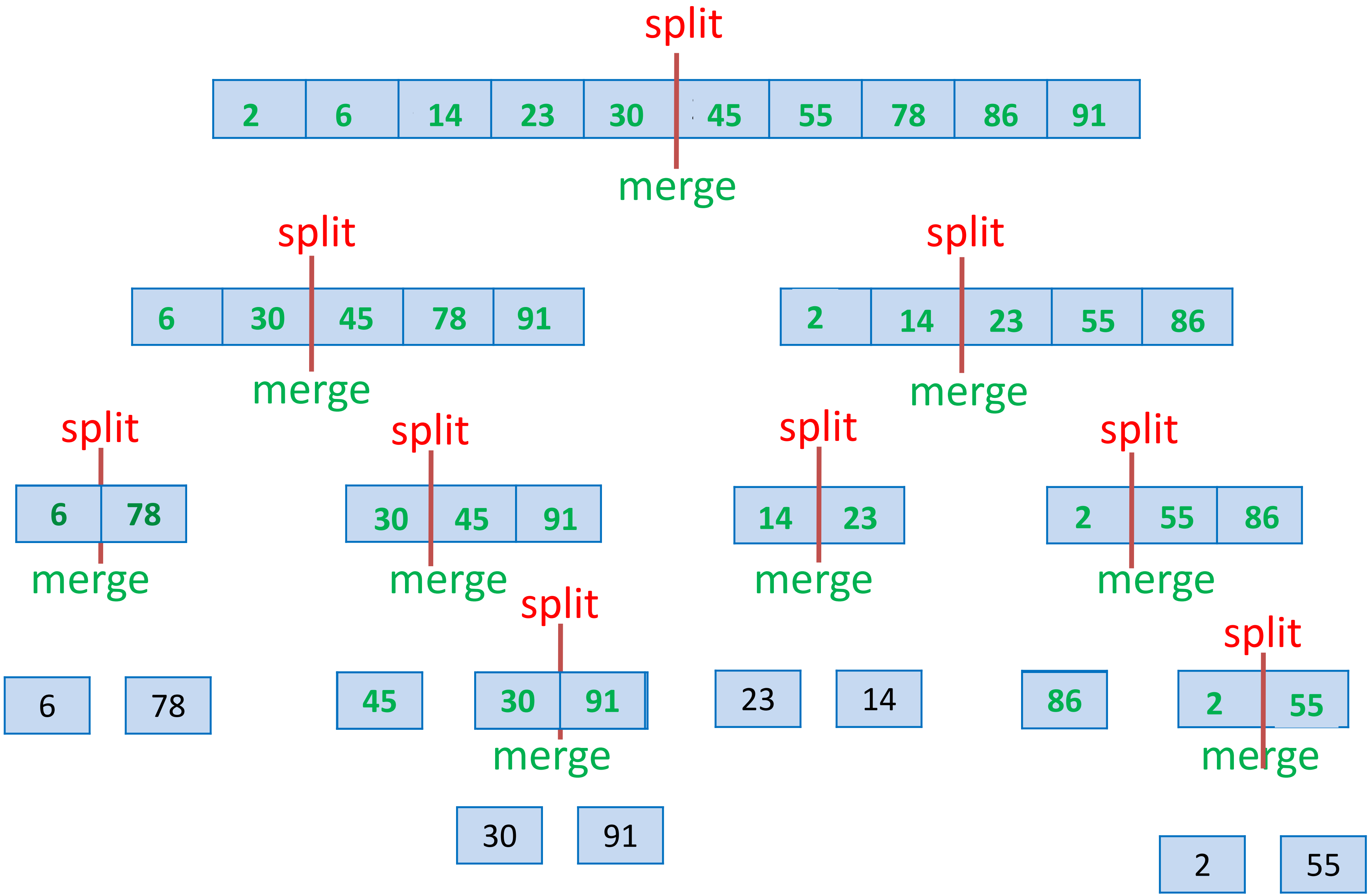
```
void sort(int[] list){
    if(list.length > 1)
        half1 = 1st half of list;    sort(half1);
        half2 = 2nd half of list;    sort(half2);
        merge (half1, half2, list);
}

void merge(half1, half2, list){…}
```

Runtime efficiency

- **Average/ worst case: O(n log n)**
  - (see calculations in textbook)

Merge Sort



split
2 6 14 23 30 45 55 78 86 91
merge
split
6 30 45 78 91
merge
split
2 14 23 55 86
merge
split
6 78
merge
split
30 45 91
merge
split
14 23
merge
split
2 55 86
merge
6
78
45
split
30 91
merge
23
14
86
split
2 55
merge
30
91
2
55

Merge Sort

How to merge?

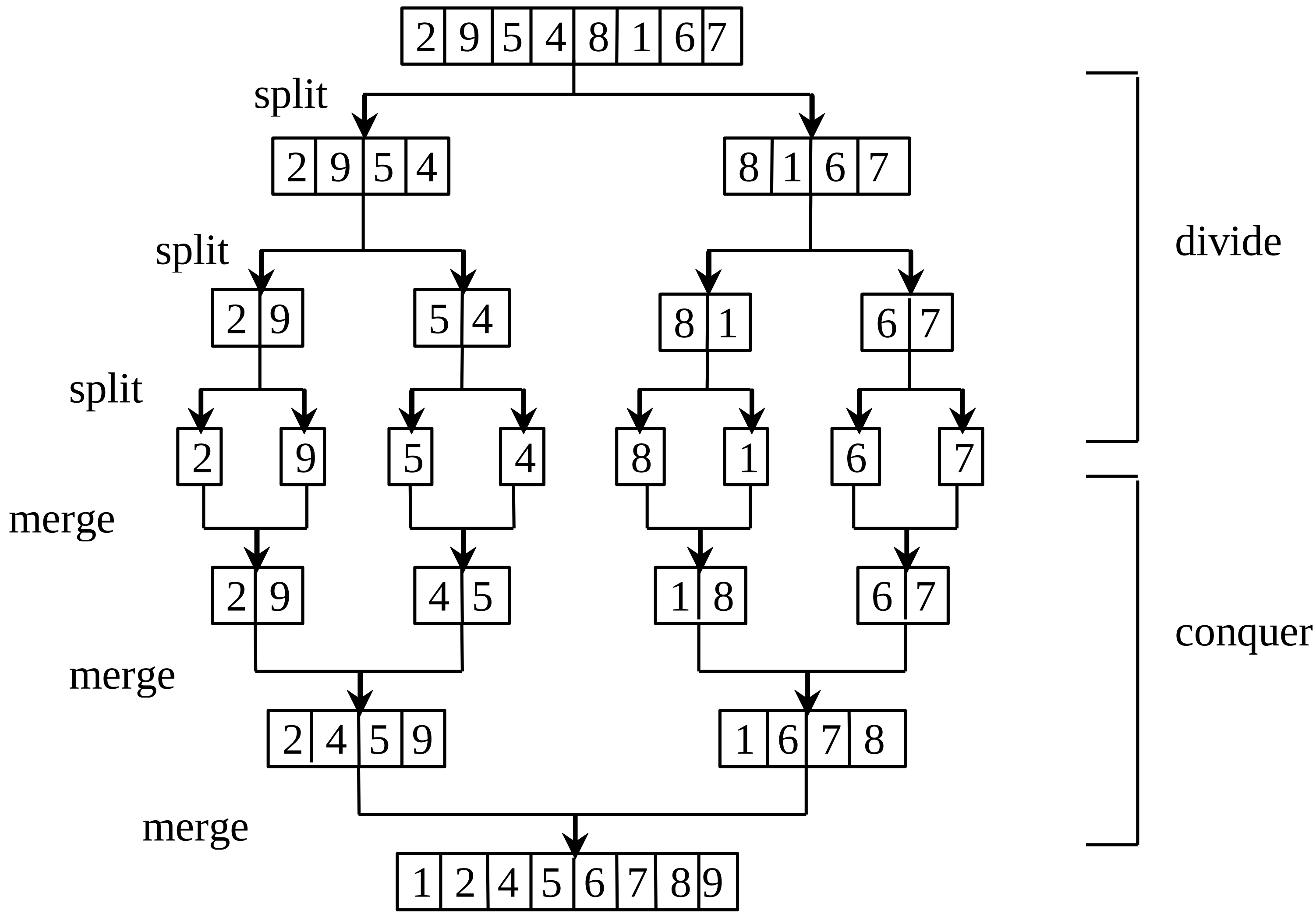
Keep record of index

2	6	14	23	30	45	55	78	86	91

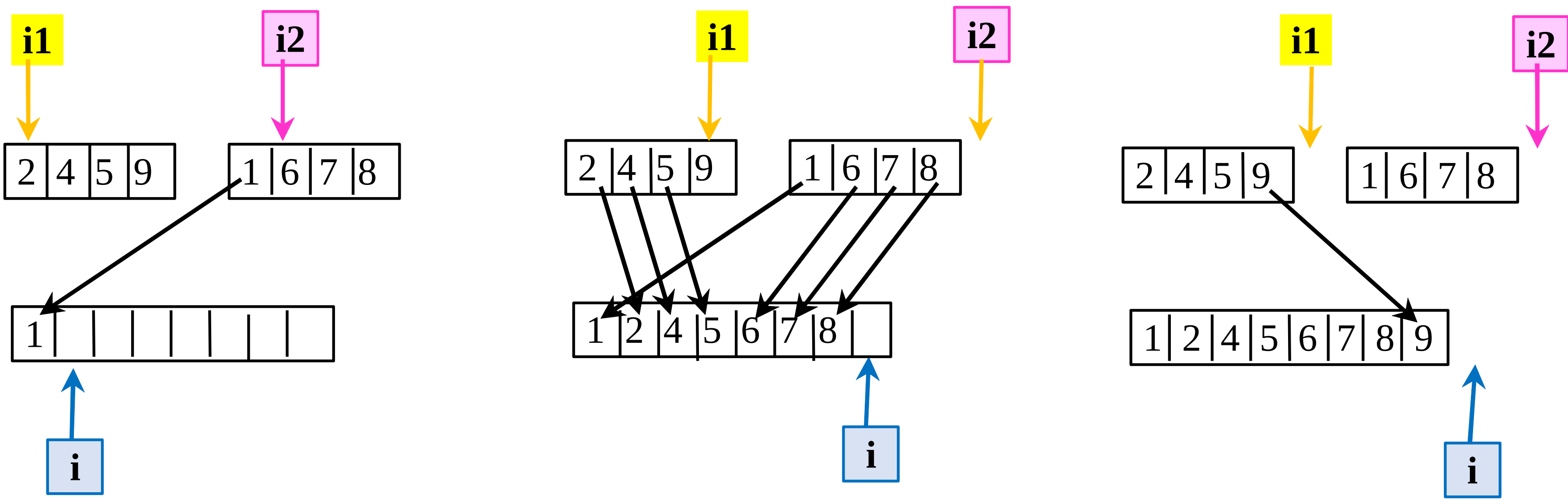
index	0	1	2	3	4
	6	30	45	78	91

index	0	1	2	3	4	5
	2	14	23	55	86	

Merge Sort


2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conquer

Merge Two Sorted Lists



(a) After moving 1 to list

(b) After moving all the elements in list2 to list

(c) After moving 9 to list

Merge Sort Code

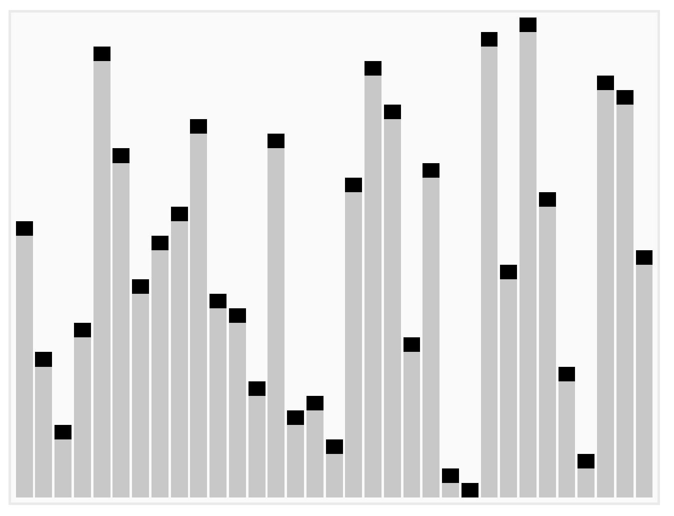
```
void mergeSort(int[] list) {
  if (list.length>1) {        // sort as long as there're at least 2 elements
    //divide into two halves
    int length1 = list.length / 2;
    int length2 = list.length - length1;
    int[] half1 = new int[length1];
    int[] half2 = new int[length2];
    System.arraycopy(list, 0, half1, 0, length1);
    System.arraycopy(list, length1, half2, 0, length2);

    //recursively sort each half
    mergeSort(half1);
    mergeSort(half2);

    //merge the two halves
    merge(half1, half2, list);
  }
}
```

Merge Sort Code

```
void merge(int[] half1, int[] half2, int[] list) {
    //current indexes in list, half1, and half2
    int i = 0, i1 = 0, i2 = 0;
    //put the smaller value from half1 or 2 into list
    while(i1 < half1.length && i2 < half2.length) {
        if(half1[i1] < half2[i2])
            list[i++] = half1[i1++];
        else
            list[i++] = half2[i2++];
    }
    //put remaining elements when half1.length != half2.length
    while(i1 < half1.length)
        list[i++] = half1[i1++];
    while(i2 < half2.length)
        list[i++] = half2[i2++];
}
```



Wikipedia

Quick Sort

C. A. R. Hoare (**1962**)

Quick Sort

IDEA: Select a pivot that divides the array into:

part1 (elements ≤ pivot) & **part2 (**elements > pivot).

Then recursively repeat for each part.

Algorithm:

```
void quickSort(int[] list) {
  if (list.length > 1) {
     selectPivot();
     partitionList() into list1 & list2 such that elements
          in list1 <= pivot and elements in list2 > pivot;
     quickSort(list1);
     quickSort(list2);
   }
}

int selectPivot(){…}

int partitionList(){…}
```

Quick Sort

Runtime efficiency

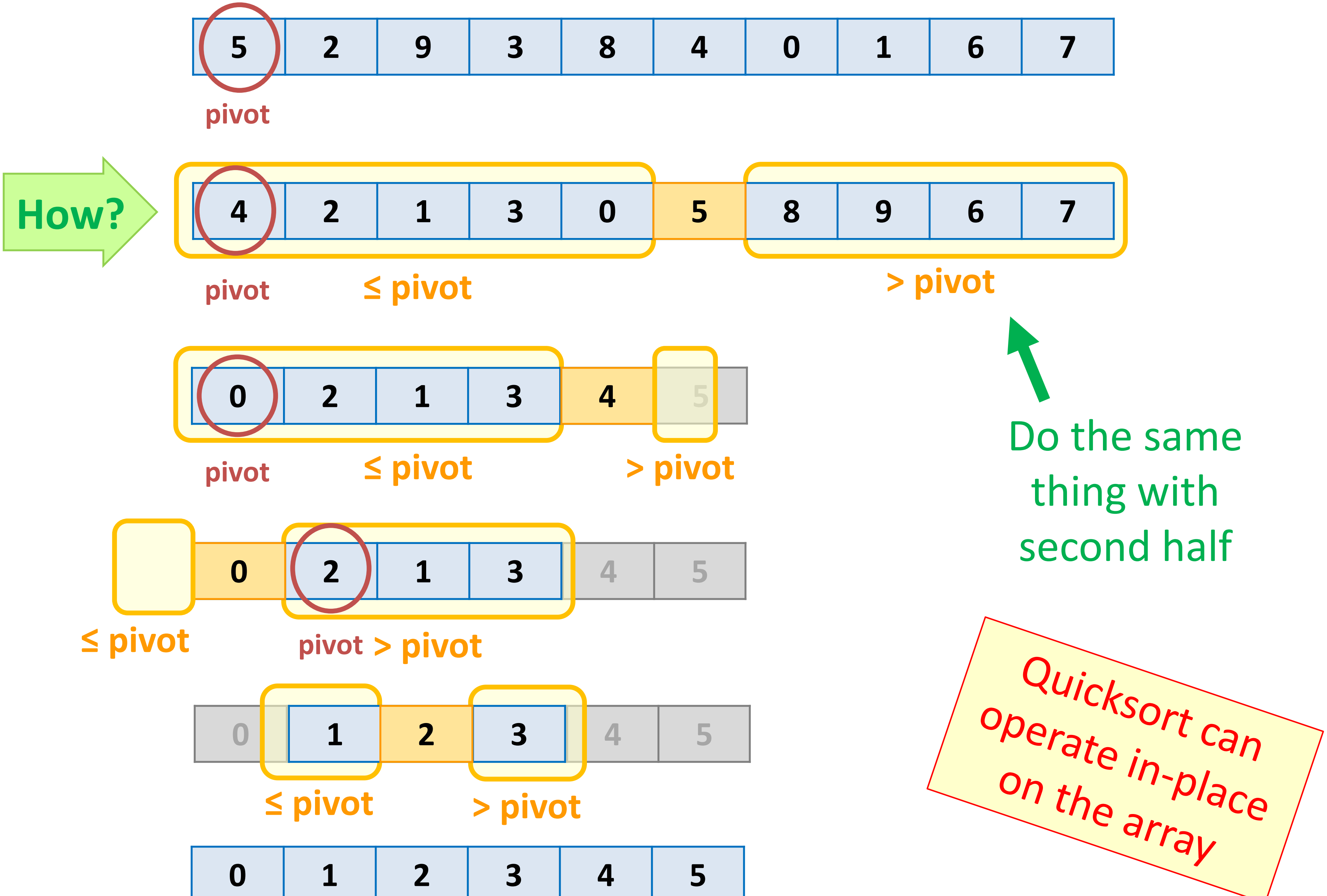
- **Best/Average case: $O(n \log n)$**
- **Worst case $O(n^2)$ in rare situations**

When?

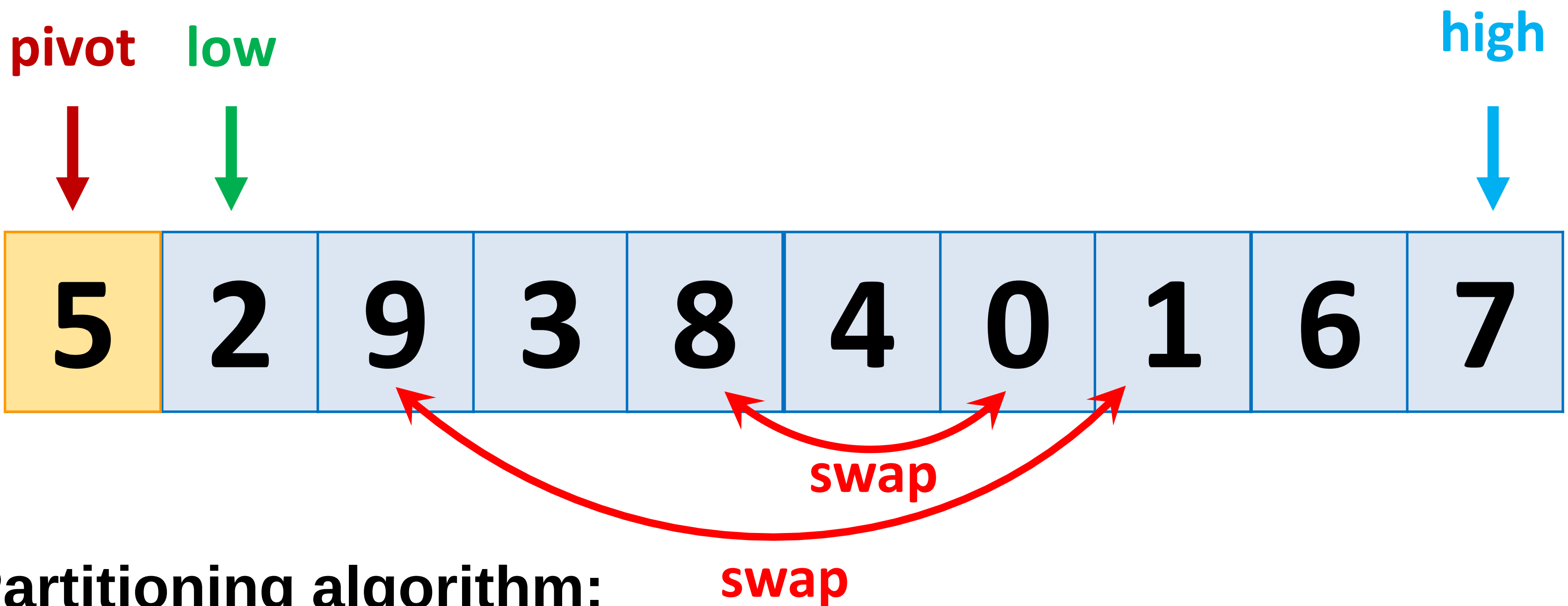
- In the **best case**, the pivot divides the array each time **into two parts of about the same size**.

- In the **worst case**,
  - pivot divides the array each time into one big subarray with the other array empty. So the algorithm requires $(n-1) + (n-2) + \ldots + 2 + 1$ times $= O(n^2)$
  - This happens **when the smallest or largest element** is always chosen as the **pivot**.

Quick sort: Basic Idea



Basic Partitioning



Partitioning algorithm:

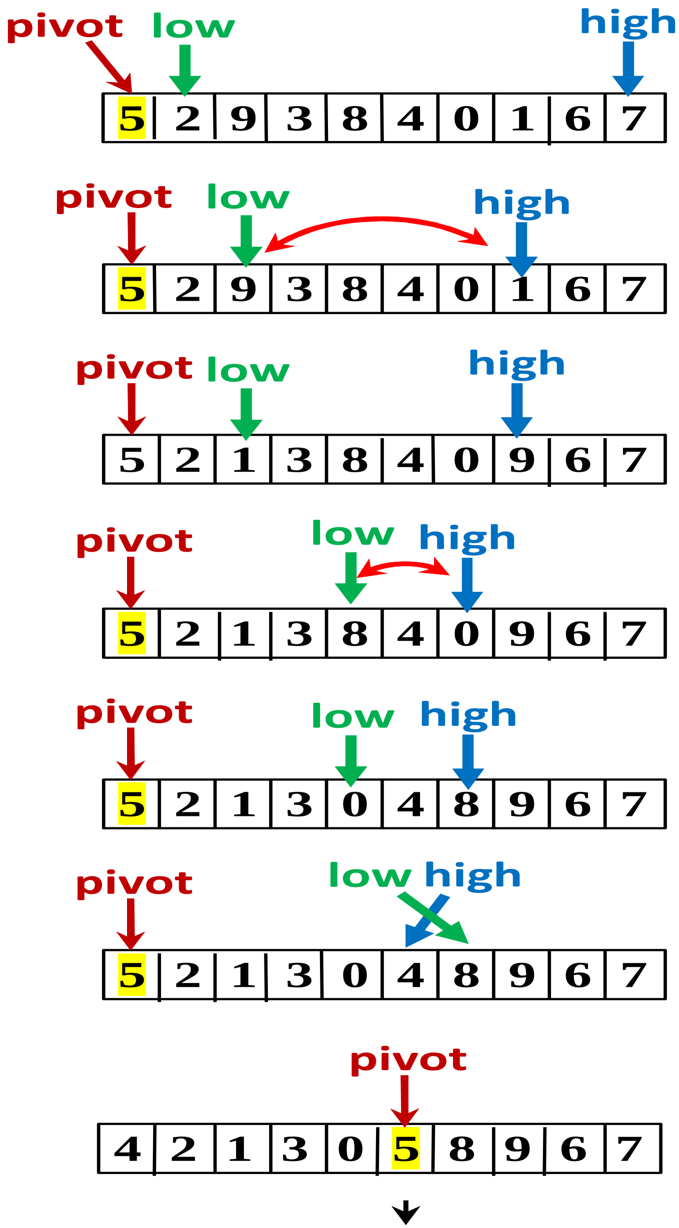
1- initialize pivot, low, high

2- while(low < high)

- Search from both sides:
  - from left (low) for first element > pivot.
  - from right(high) for first element ≤ pivot
- swap the elements at low <> high

3- Move pivot element at the correct location and return its index

How to Partition?



(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

The index of the pivot is returned

Practice Questions

Describe how an Quick Sort works. What is the time complexity?

Apply a Quick sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}

Improving the Quick Sort

Better choice of the pivot

- Always choosing the smallest or the largest element as the pivot leads to the worst case scenario $O(N^2)$
  - Can you think of a case which this could happen?
    - Early versions of quicksort suggested choosing the leftmost element as the pivot. What happens if the array is already sorted (or reverse-sorted)?
- How to avoid the worst case scenario?
  - Choose a random index for the pivot
  - Choose the middle element as the pivot
  - ** Choose the median between the first, middle, and last elements
    - Aim: to find a pivot that divides the array into 2 parts of almost the same size.
    - Called "median of three" partitioning → good estimate of the optimal pivot.
    - Median-of-three works even with sorted or reverse-sorted array.

Other improvements were suggested

- Outside the scope of this course.

Java uses *Dual-Pivot Quicksort* technique (by V. Yaroslavskiy, J. Bentley, and J. Bloch)

Positive Integers Sorting

Bucket Sort and Radix Sort

Previous sort algorithms are **general sorting algorithms**

- work for any types of keys (e.g., integers, strings, and any comparable objects).
- The best average-case performance is O(N log N).

If the keys are small integers, you can use **Bucket or Radix sort** without having to compare the keys.

Radix Sort

If K is too large, using the bucket sort is not desirable.

radix sort is based on bucket sort, but uses only **ten buckets**.

IDEA:

- divide the keys into subgroups based on their radix (base) positions.
- apply a bucket sort repeatedly for the key values on radix positions, starting from the **least-significant position**.

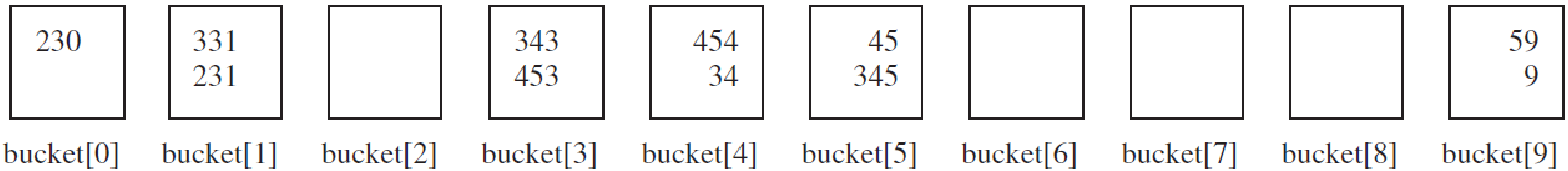
Performance: O(dN)

- where d is maximum number of radix positions

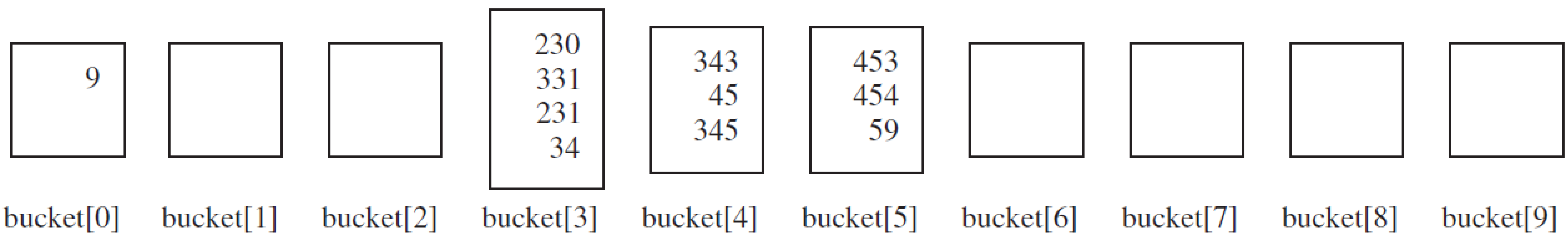
http://www.cs.armstrong.edu/liang/animation/web/RadixSort.html

Radix Sort

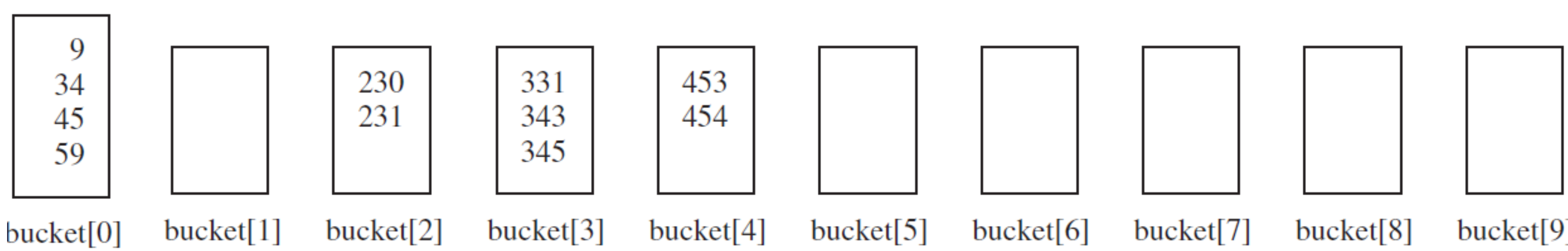
SORT: 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9



After removed from buckets: 230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9



09, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59



009, 034, 045, 059, 230, 231, 331, 343, 345, 453, 454

Practice Questions

Describe how Bucket and Radix Sort work. What is the time complexity?

Apply a Bucket sort on {8, 11, 5, 9, 6, 2, 4, 7, 1}.

Apply a Radix sort on {81, 161, 25, 990, 16, 562, 24, 127, 11}.

Solution for Radix Sort

0	1	2	3	4	5	6	7	8	9
990	81 161 11	562		24	25	16	127		
	11 16	24 25 127				161 562		81	990
11 16 24 25 81	127 161				562				990

Next …

In Next Lab Assignment, you will need to write code for selection, insertion, and bubble sorting algorithms

- I am not going to ask you to memorize the code of the advanced algorithms for the exam. However, You need to be able to
  - explain all sorting algorithms and know how to apply them (manually)
  - Recognize which algorithm is more efficient than which.