BACS2063 Data Structures and Algorithms

Algorithms for Sorting

Chapter 8b

Chapter 8 Part 2

Algorithms for Sorting

Learning Outcomes

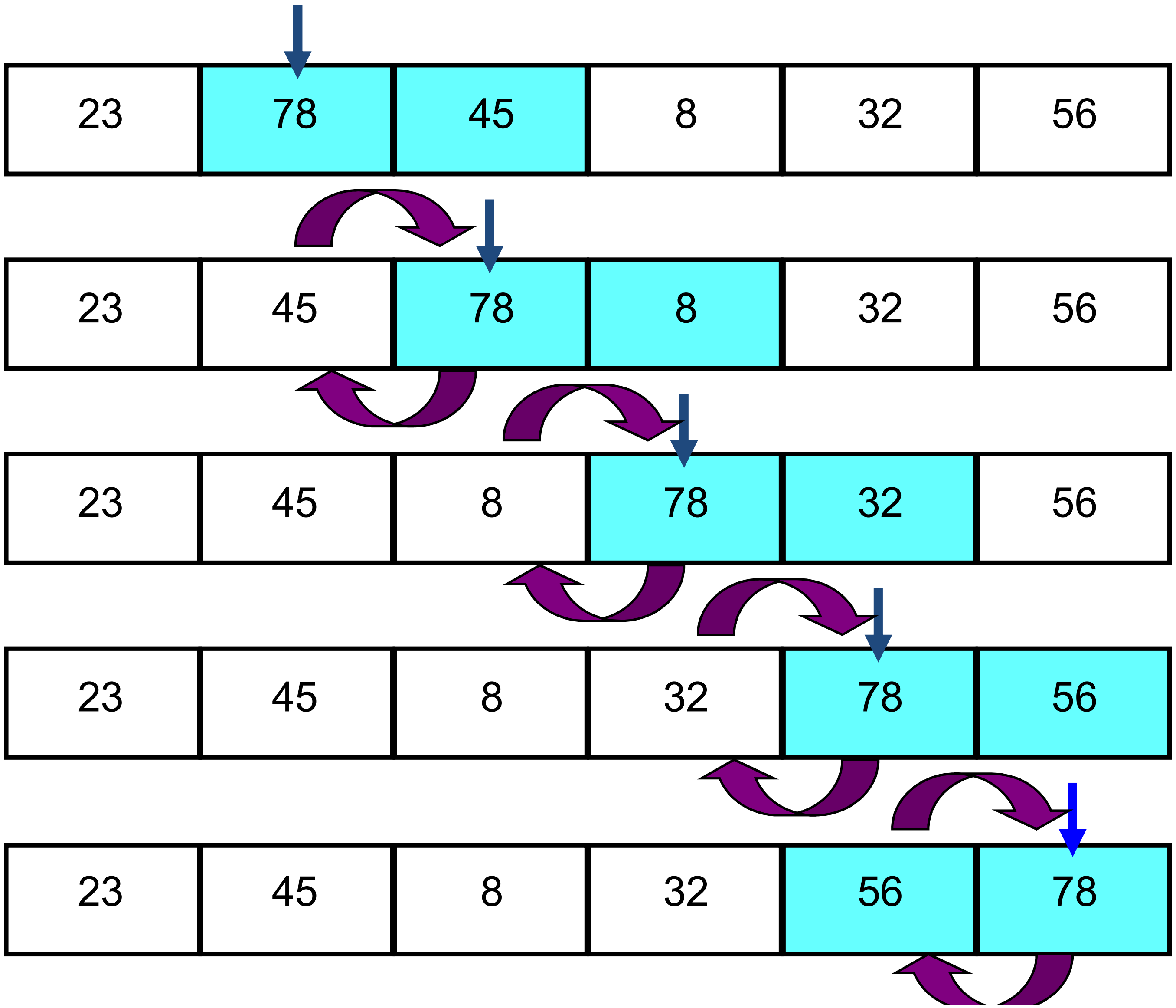
At the end of this chapter, you should be able to

- Sort an array using the sorting methods:
  - Bubble sort, selection sort, insertion sort, shell sort, merge sort, quick sort and radix sort.
- Assess the time efficiencies of various sorting methods, expressed in Big-O notation.

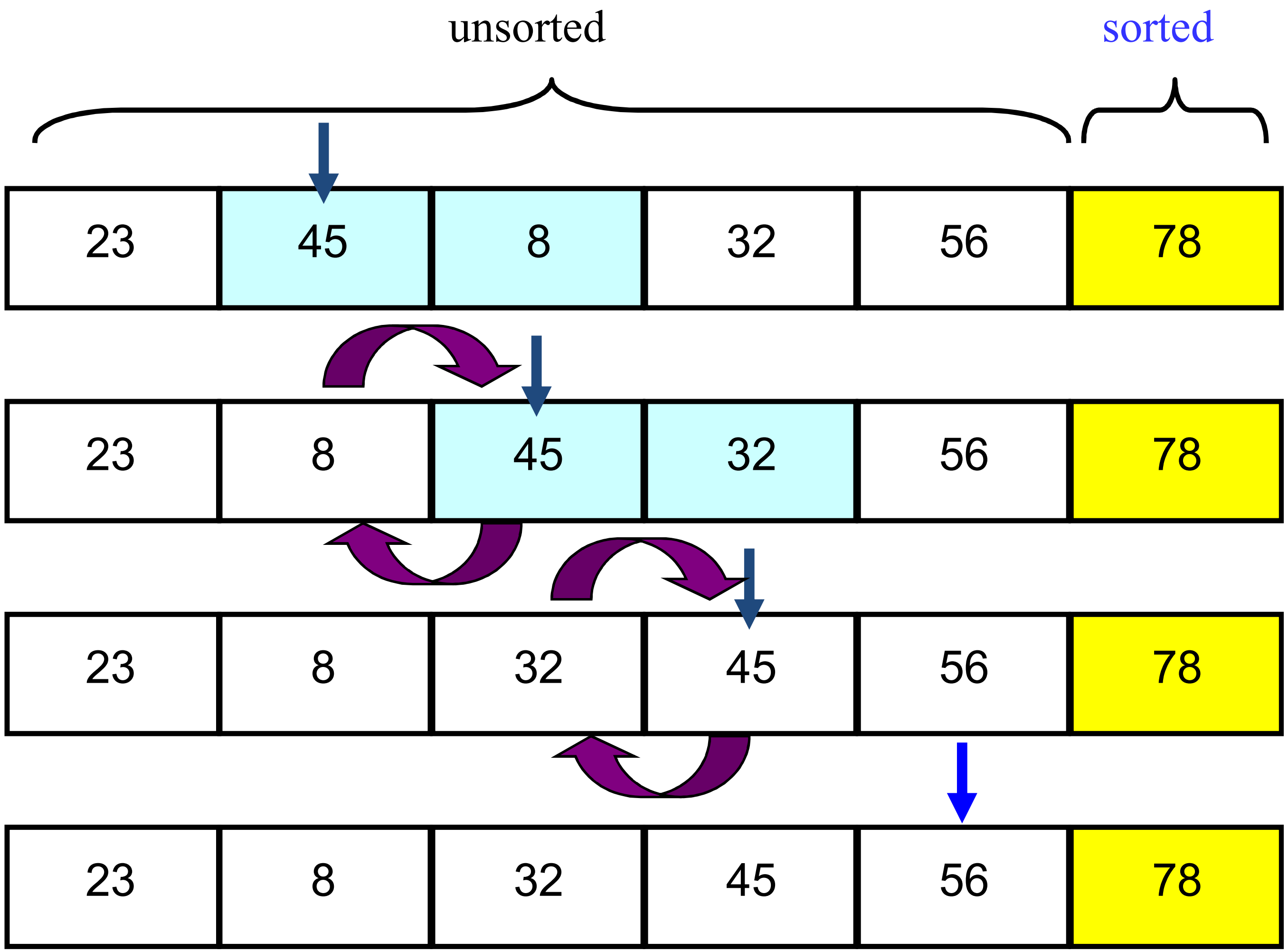
Bubble sort

- The simplest and slowest algorithm as it compares 2 adjacent elements sequentially and swaps them if they are out of order. It uses iteration to move the largest element to the end of the array.
- Called bubble sort because larger values gradually "*bubble*" to the end of the array.
- Disadvantage: slowest algorithm.
- It makes several passes through the array.
  - On each pass, neighboring pairs of elements are compared.
    - If the pair is in increasing order, we leave the values as they are.
    - If the pair is in decreasing order, we swap the values.

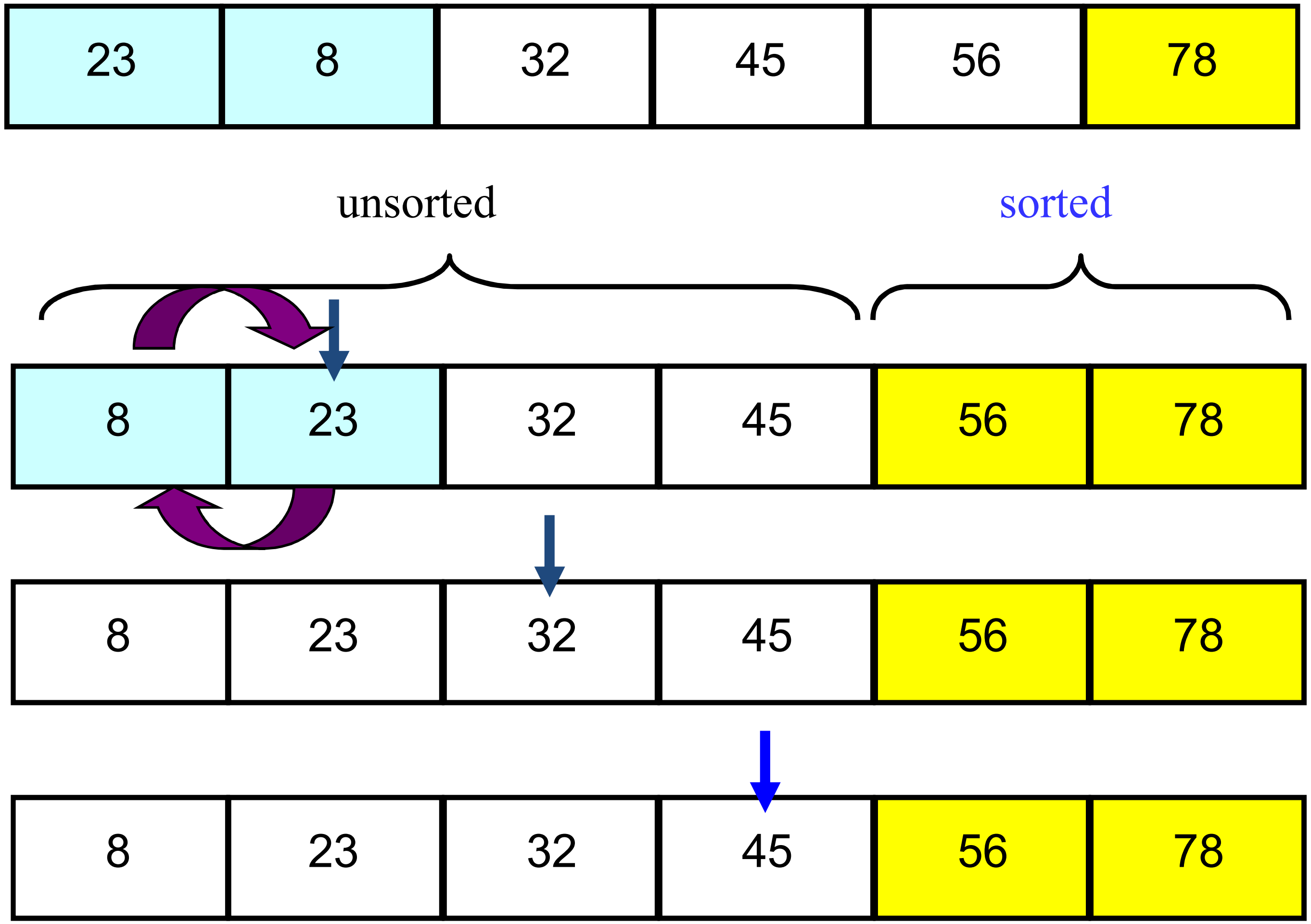
Bubble Sort Example – 1st pass



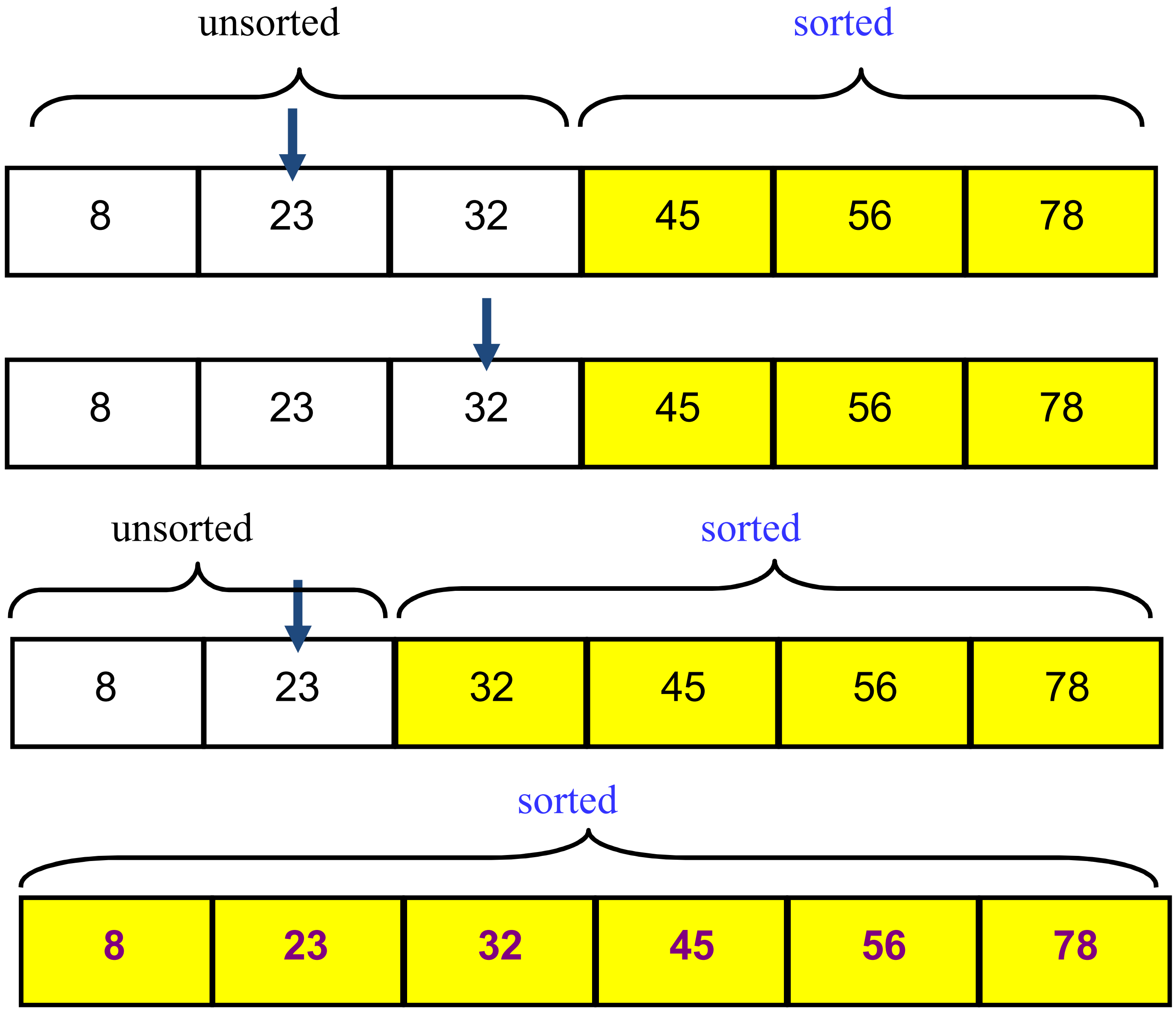
Bubble Sort Example - 2^{nd} pass



Bubble Sort Example – 3rd pass



Bubble Sort Example (Cont.)



4^{th} Pass

Bubble sort – Best & Worst Cases

- Note: *The bubble sort algorithm can detect a sorted array when there is a pass with no swaps*.
- Best case:
  - The bubble sort needs only the first pass to sort the array, no next pass is needed.
  - Since the number of comparisons is n-1 in the first pass, the best case time is O(n).
- Worse case:
  - Inverted array - maximum passes (n-1) is required.
  - Each pass requires maximum comparisons, i.e. the 1st pass takes n-1 comparisons, the 2nd pass takes n-2 comparisons and so on.
  - The worse case time is $O(n^2)$.

Source Code for Bubble Sort

- `Chapter8\sorting\`**`BubbleSort.java`**
  - Demonstrates the bubble sort algorithm on an array of integers

    (Note: the source code can be easily amended to cater for arrays of any primitive data types).
  - However, it does not work for array of objects as objects cannot be compared using the relational operators (e.g. **>** and **<**)

- Points for consideration
  - How do we compare two *objects*?

Bubble Sort for Objects

- `Chapter8\sorting\`**`SortArray.java`**
  - Examine the **`bubbleSort`** method's
    - Method header
    - Use of the **`compareTo`** method for comparing adjacent array elements (objects)
- Points for consideration
  - How do we organize methods that sort arrays in a class?
  - How do we write sorting methods that cater for an object, including objects of its superclass?

Organizing Methods that Sort an Array

- Create a class of static sort methods that perform the various sorting algorithms
- The methods define a generic type **`T`** for the objects in the array
- To sort an array of objects, the objects in that array must be *Comparable*.
  - To ensure this requirement, we write

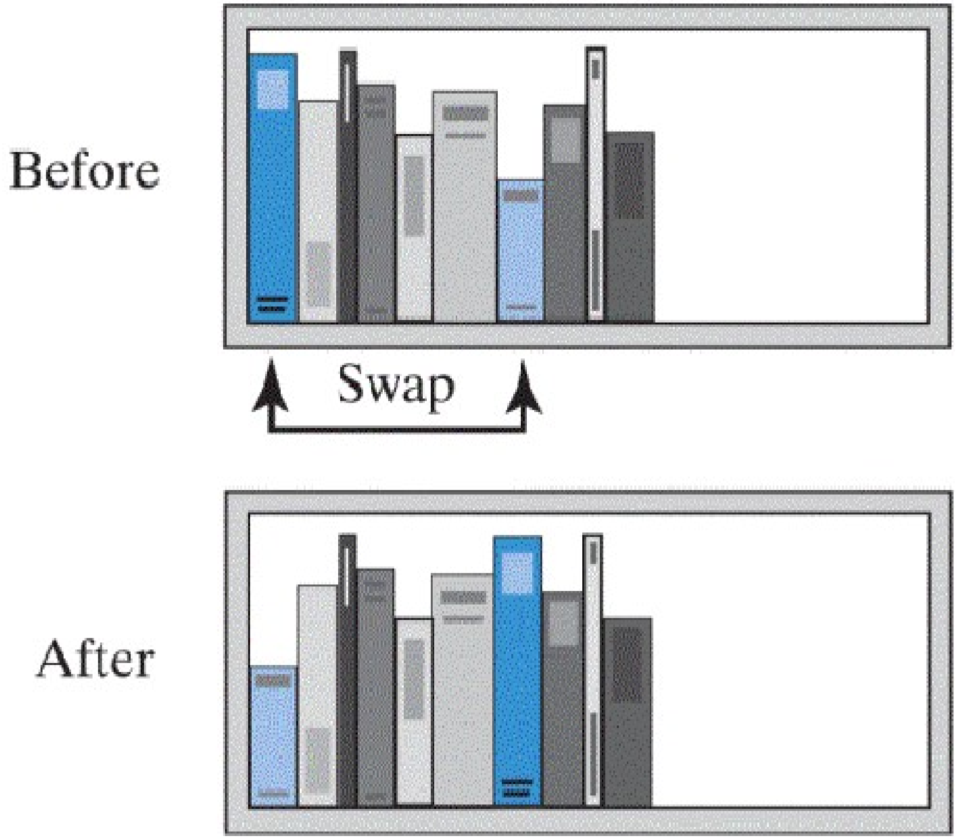
    `<T extends Comparable<T>>`

    before the return type in the header of the sort method.
  - We can then use `T` as the data type of the parameters and local variables within the methods.

Bounded wildcards

- When using generic types, the wild card **`?`** represents any class. It can bound or limit, the wildcard in 1 of 2 ways:
  - **`<? super Animal>`** means any *superclass* of animal. Animal is the *lower bound* of the wildcard.
  - **`<? extends Animal>`** means any *subclass* of animal. Animal is the *upper bound* of the wildcard.

Selection Sort



- Analogy: Rearrange books on shelf by height
  - Shortest book on the left
- Approach:
  - Look at books, select shortest book
  - Swap with first book
  - Look at remaining books, select shortest
  - Swap with second book, repeat …

Fig. 11-2 Before and after exchanging shortest book and the first book.

Selection Sort

- How does it work?
  - Make ***n-1*** passes through a sequence of ***n*** elements.
  - In the 1st pass, find the smallest element from the subarray **`0..n-1`**. Then, swap the smallest element with the element at location **`0`**.
  - In the 2nd pass, find the smallest element from the subarray **`1..n-1`**. Then, swap the smallest element with the element at location **`1`**.
  - Do the same until the 2nd last array location has been processed.
- Advantage of selection sort
  - It does not depend on the initial arrangement of the data
  - A bit more efficient than bubble sort - only make 1 swap during each pass
- Disadvantage of selection sort
  - It is only appropriate for small size of data.

Selection Sort

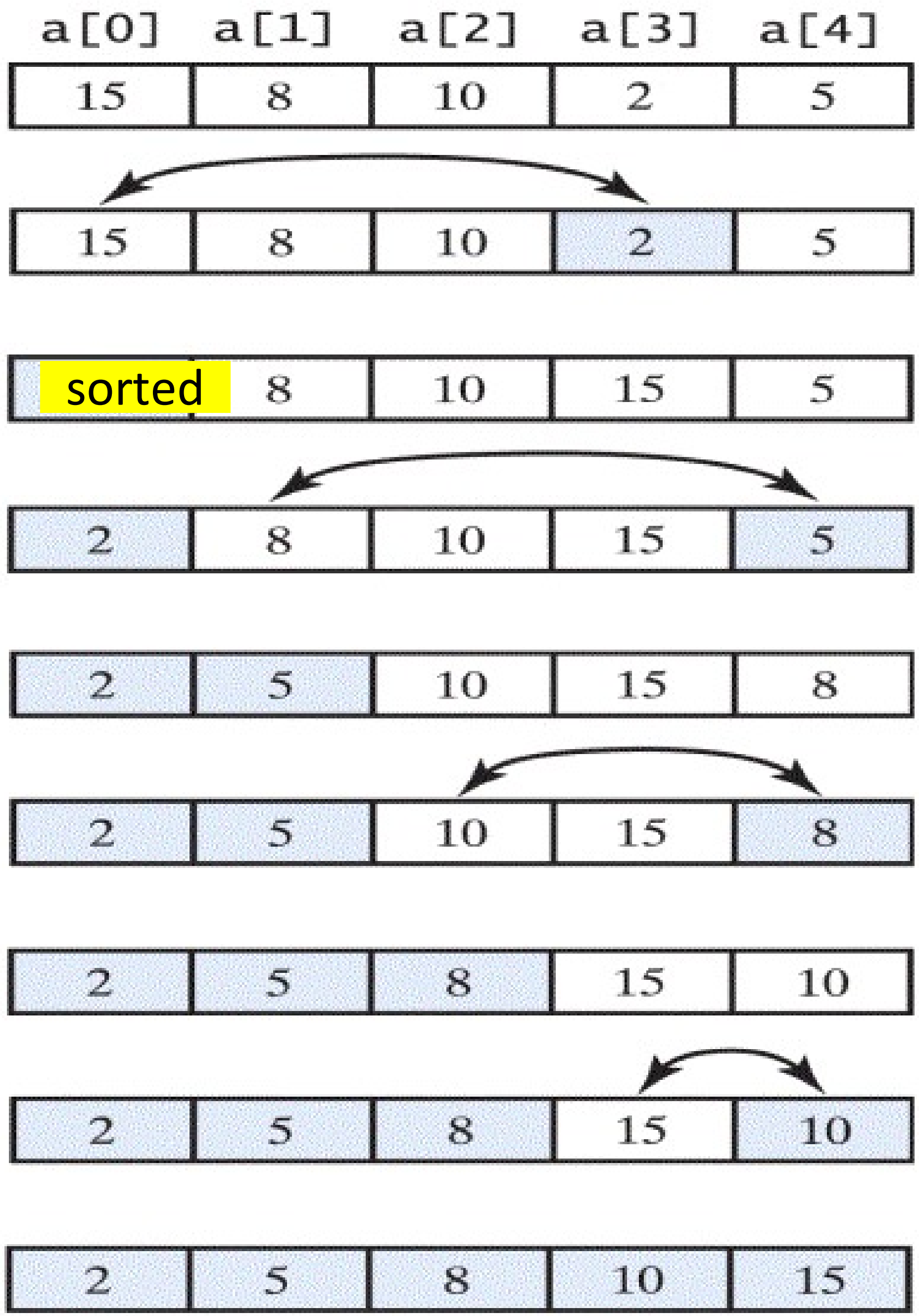


Fig. 11-3 A selection sort of an array of integers into ascending order.

Iterative algorithm for selection sort

Algorithm selectionSort (a, n)

```
for (index = 0; index < n-1 ; index++) {
  indexOfSmallest = the index of the smallest value
                    among a[index], a[index+1], . . ., a[n-1]
  Interchange the values of a[index] and a[indexOfSmallest]
}
```

Recursive algorithm for selection sort

- 3 Parameters: the array, a beginning and ending index:

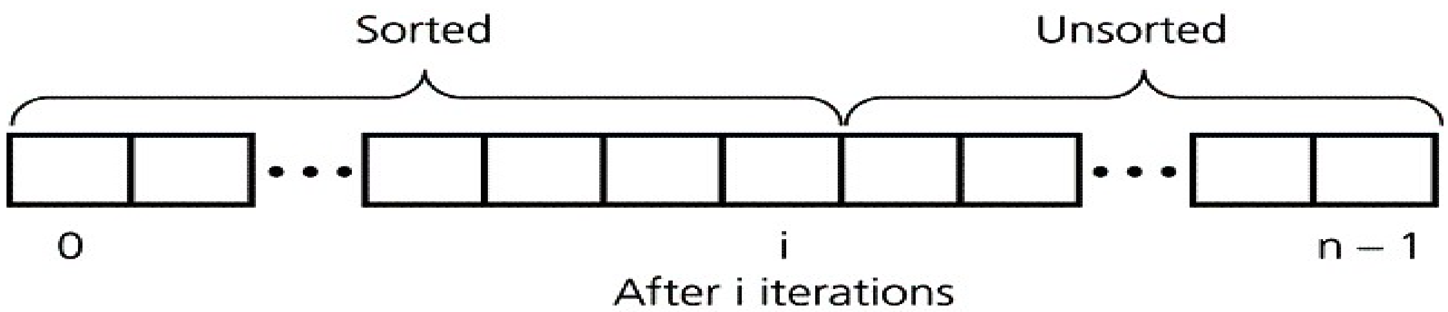
```
Algorithm selectionSort (a, first, last)
 if (first < last)    {
   indexOfSmallest = the index of the smallest value
                among a[index], a[index+1], . . ., a[n-1]
   Interchange the values of a[index] and a[indexOfSmallest]
   selectionSort(a, first + 1, last)
 }
```

The Efficiency of Selection Sort

- Takes $O(n^2)$ time in these 2 cases.
- Iterative method for loop executes $n - 1$ times
  - For each of $n - 1$ calls, inner loop executes $n - 2$ times
  - $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 = O(n^2)$
- Recursive selection sort performs same operations
  - Also $O(n^2)$

Insertion Sort

- Insertion Sort strategy:
  – Partition the array into two regions: sorted and unsorted
  – Take each item from the unsorted region and insert it into its correct order in the sorted region.
- How does it work?
  – Make ***n-1*** passes through a sequence of ***n*** elements
  – Each pass inserts the next element into the sub-array on its left.
- Faster than bubble sort and selection sort - no swapping of elements.



Insertion Sort

- If first two books are out of order
  - Remove second book
  - Slide first book to right
  - Insert removed book into first slot
- Then look at third book, if it is out of order
  - Remove that book
  - Slide 2^{nd} book to right
  - Insert removed book into 2^{nd} slot, recheck first two books again

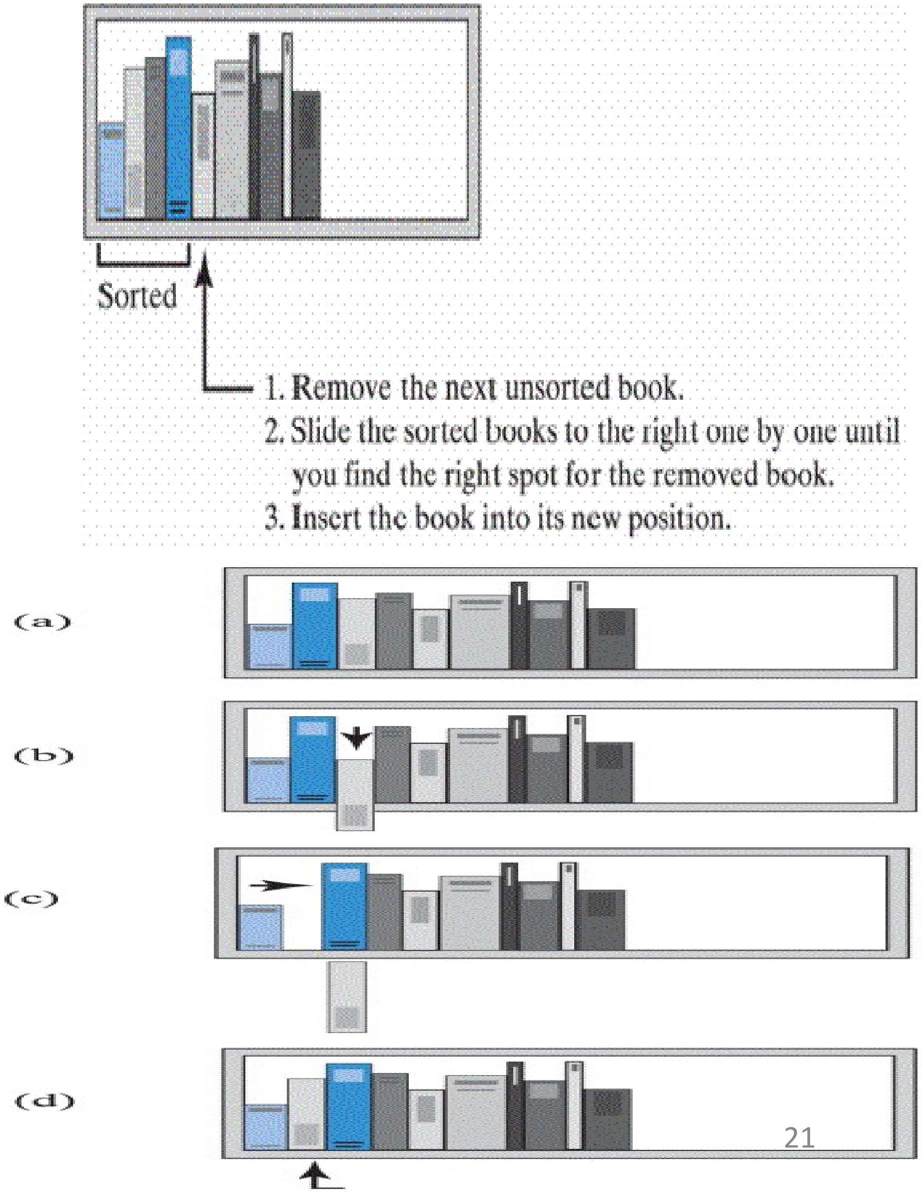


Fig. 11-4 The placement of the third book during an insertion sort.

Iterative Insertion Sort

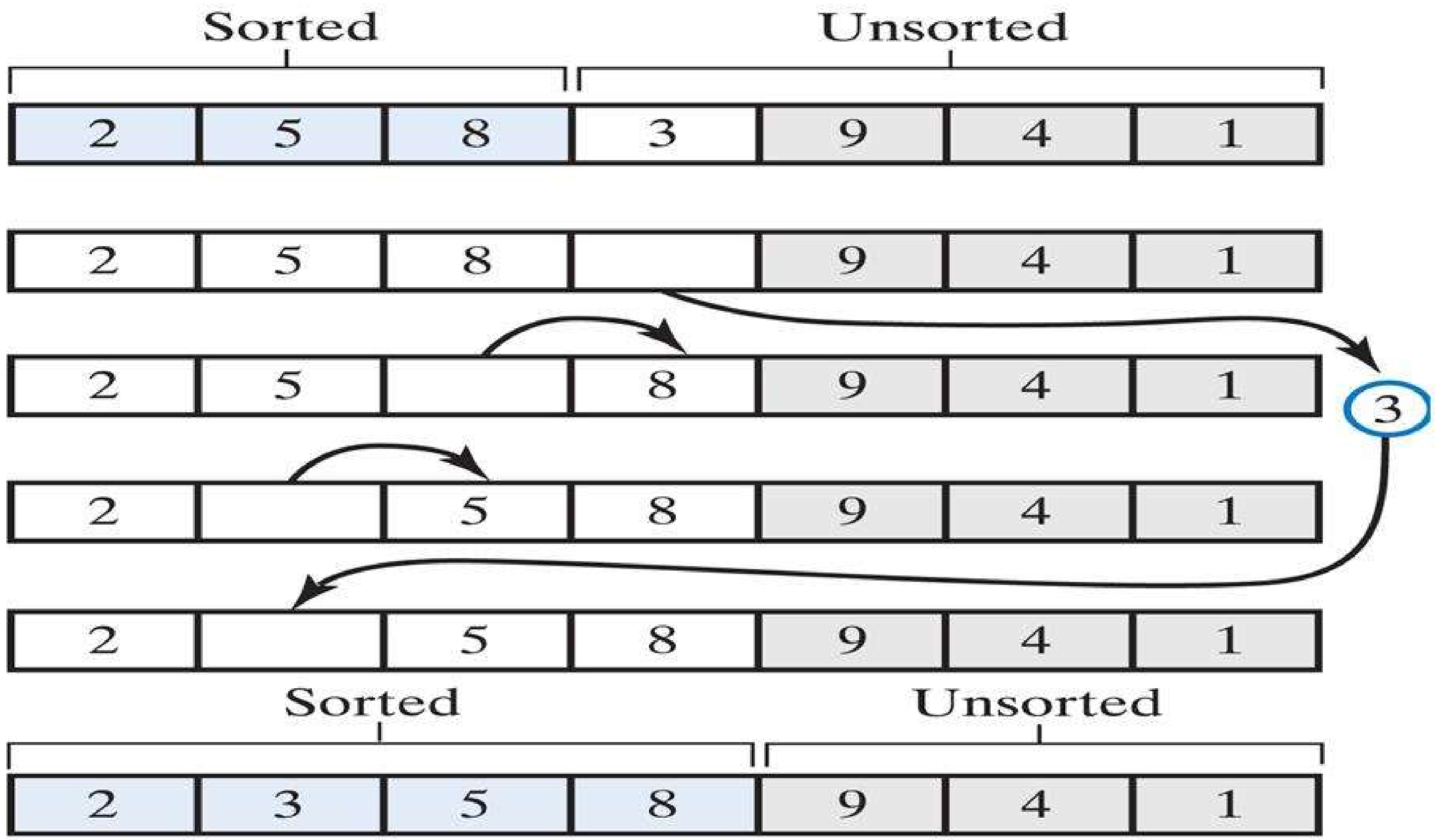


Fig. 11-6 An insertion sort inserts the next unsorted element into its proper location within the sorted portion of an array

Iterative Insertion Sort (cont'd)

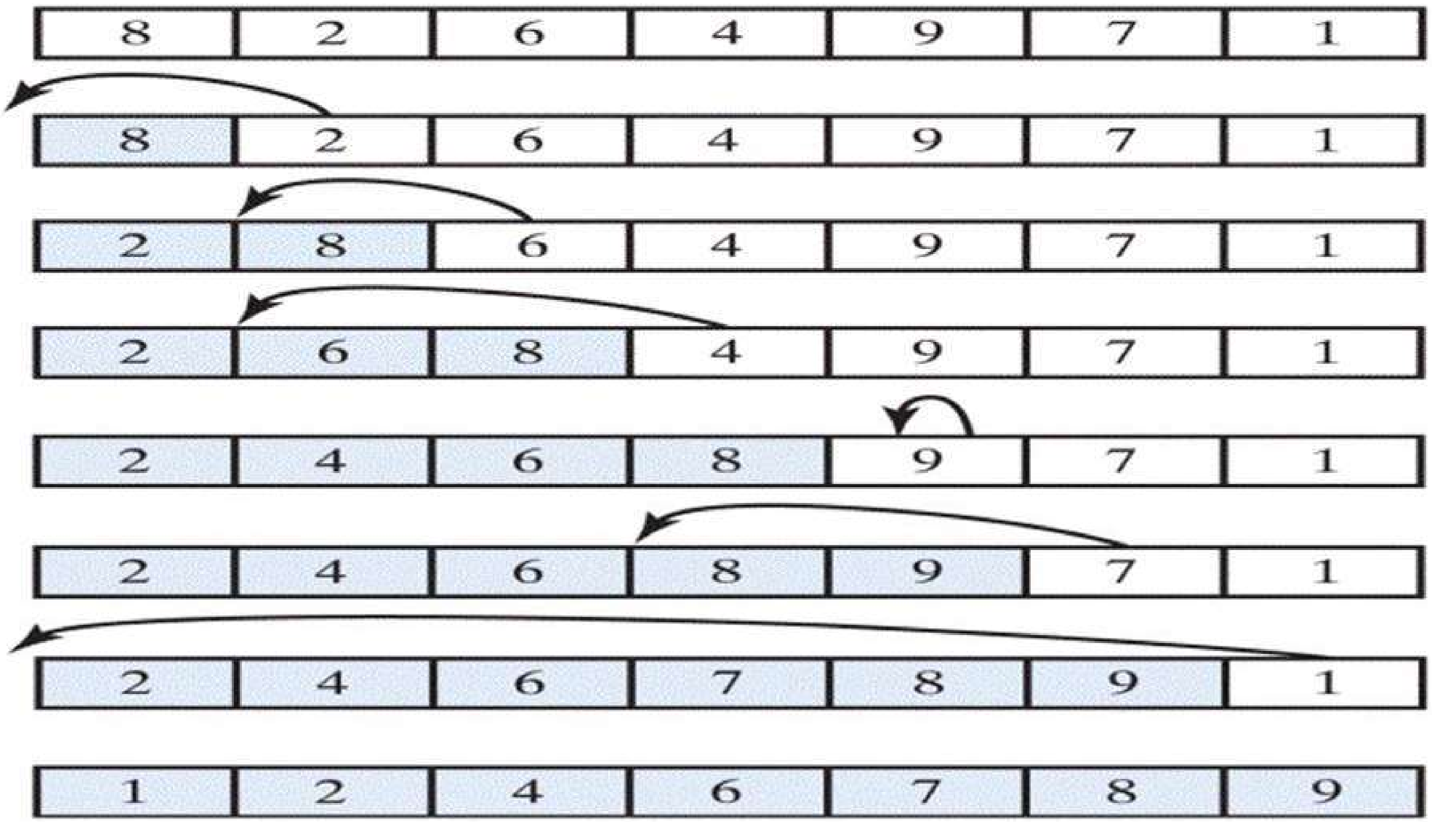


Fig. 11-7 An insertion sort of an array of integers into ascending order

Iterative algorithm for Insertion Sort

```
Algorithm insertionSort(a, n)
    for (unsorted = 1 through n-1) {
        firstUnsorted = a [unsorted]
        insertInOrder (firstUnsorted, a, unsorted - 1)
    }

Algorithm insertInOrder(element, a, end)
// Inserts element into the sorted array elements a[0] through a[end].
    index = end
    while ((index >= 0) and (element < a [index])) {
        a [index + 1] = a [index]  // make room
         index - -;
    }
    a [index + 1] = element       // insert
```

Recursive algorithm for Insertion sort

Algorithm insertionSort (a, first, last)

if (the array contains more than one element) {

Sort the array elements a [first] through a [last - 1]

Insert the last element a [last] into its correct sorted position within the rest of the array

}

Recursive Insertion Sort

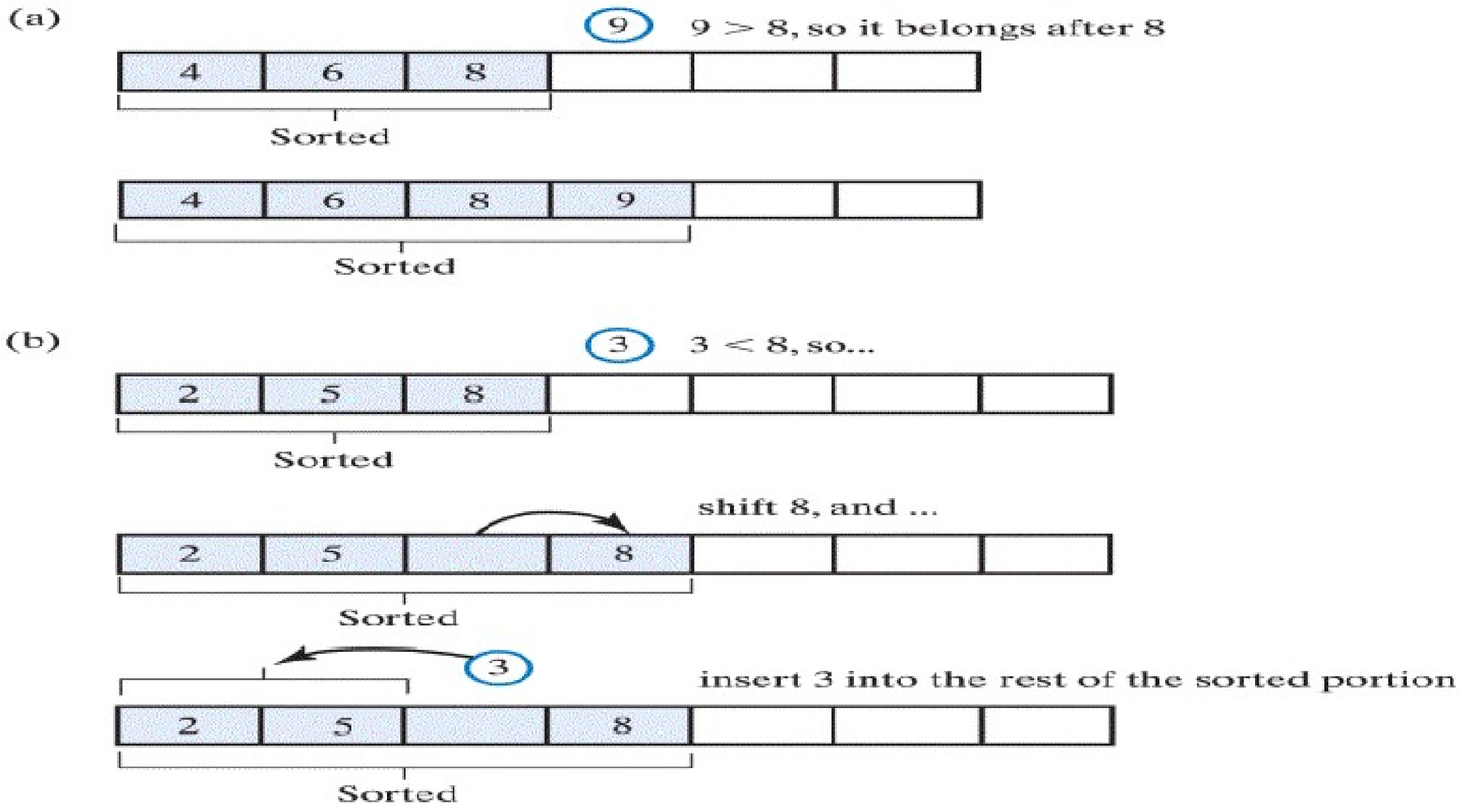


Fig. 11-8 Inserting the first unsorted element into the sorted portion of the array. (a) The element is ≥ last sorted element; (b) the element is < than last sorted element

Efficiency of Insertion Sort

- Best time efficiency is $O(n)$
- Average and Worst time efficiency is $O(n^2)$
- If array is closer to sorted order
  - Less work the insertion sort does
  - More efficient the sort is
- Insertion sort is acceptable for small array sizes

Exercise

Sort the following array using bubble sort, selection sort & insertion sort.
Remember to show the contents of the array *after every pass*.

77	44	22	88	99	55	33	66

Bubble Sort		
Original	77	44
After Pass 1	44	22
After Pass 2	22	44
After Pass 3	22	44
After Pass 4	22	44
After Pass 5	22	33
After Pass 6	22	33

Selection Sort	
Original	77
After Pass 1	22
After Pass 2	22
After Pass 3	22
After Pass 4	22
After Pass 5	22
After Pass 6	22
After Pass 7	22

Insertion Sort								
Original	77	44	22	88	99	55	33	66
After Pass 1	44	77	22	88	99	55	33	66
After Pass 2	22	44	77	88	99	55	33	66
After Pass 3	22	44	77	88	99	55	33	66
After Pass 4	22	44	77	88	99	55	33	66
After Pass 5	22	44	55	77	88	99	33	66
After Pass 6	22	33	44	55	77	88	99	66
After Pass 7	22	33	44	55	66	77	88	99

Quick Sort

- A recursive divide-and-conquer strategy for sorting an array.
  - Divide the array into 2 segments separated by a single element **PIVOT.**
  - Recursively sort each of the 2 segments (left & right).
- Quick sort use pivot element and rearranges the array elements:
  - The pivot is in its final position in sorted array
  - Elements in positions before pivot are less than the pivot
  - Elements after the pivot are greater than the pivot.
  - This arrangement is called a partition of the array.
- Analysis
  - quicksort is usually extremely fast in practice
  - Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

Quick Sort

- View Quick Sort algorithm

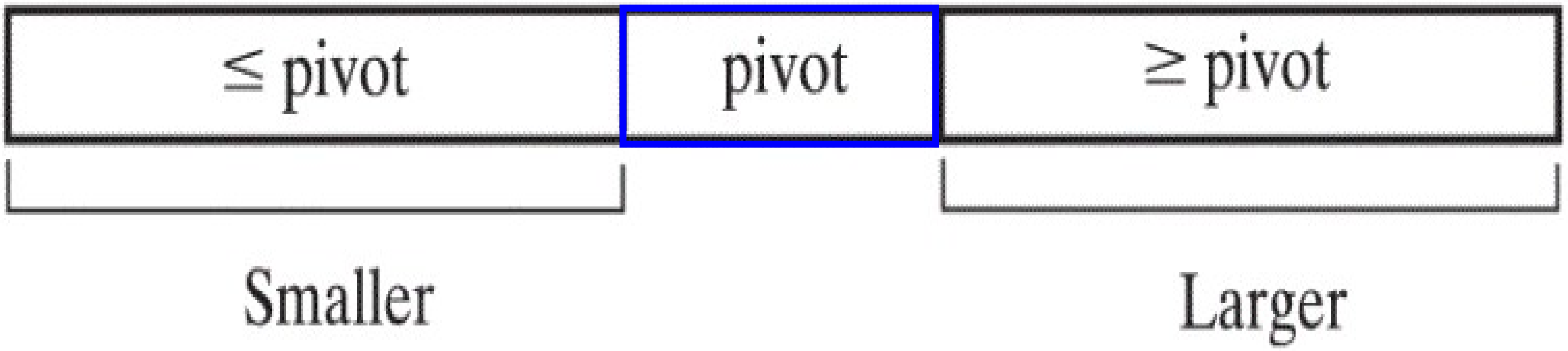


Fig. 12-5 A partition of an array during a quick sort.

Quick sort algorithm

```
Algorithm quickSort (a, first, last)
// Sorts the array elements a[first] through a[last] recursively.
    if (first < last) {
        Choose a pivot
        Partition the array about the pivot
        pivot Index = index of pivot
        quickSort (a, first, pivotIndex - 1) // sort Smaller
        quickSort (a, pivotIndex + 1, last) // sort Larger
    }
```

Efficiency of Quick Sort

- The choice of pivots affects quick sort's efficiency.
- Quick sort is $O(n \log n)$ in the average case
- $O(n^2)$ in the worst case.
- Worst case can be avoided by careful choice of the pivot. Some pivot selection schemes can lead to worst-case behavior if the array is already sorted or nearly sorted.
- Quick sort is faster than merge sort in practice and does not require the additional memory that merge sort needs for merging.

Quick Sort – partition strategy

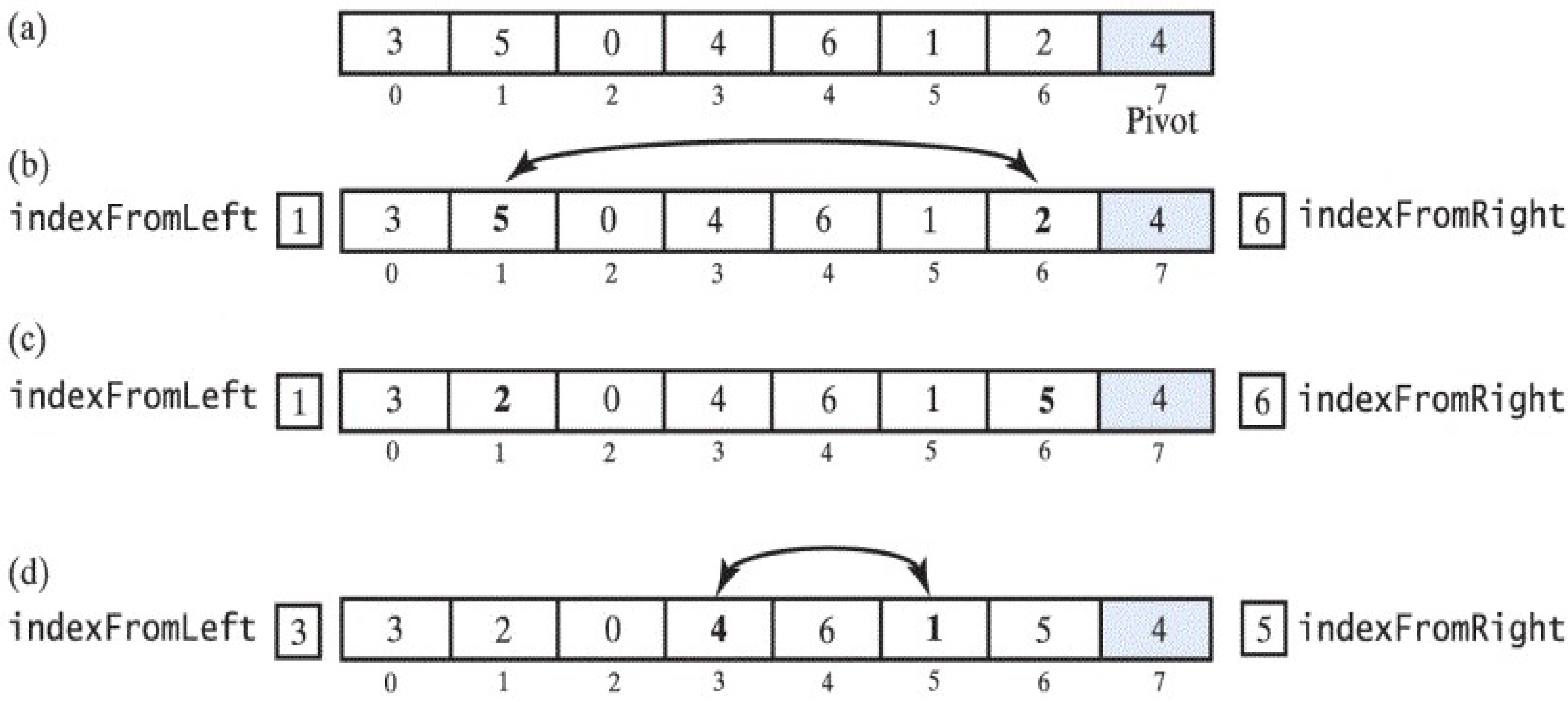


Fig. 12-6 A partition strategy for quick sort … continued→

Quick Sort - partition strategy

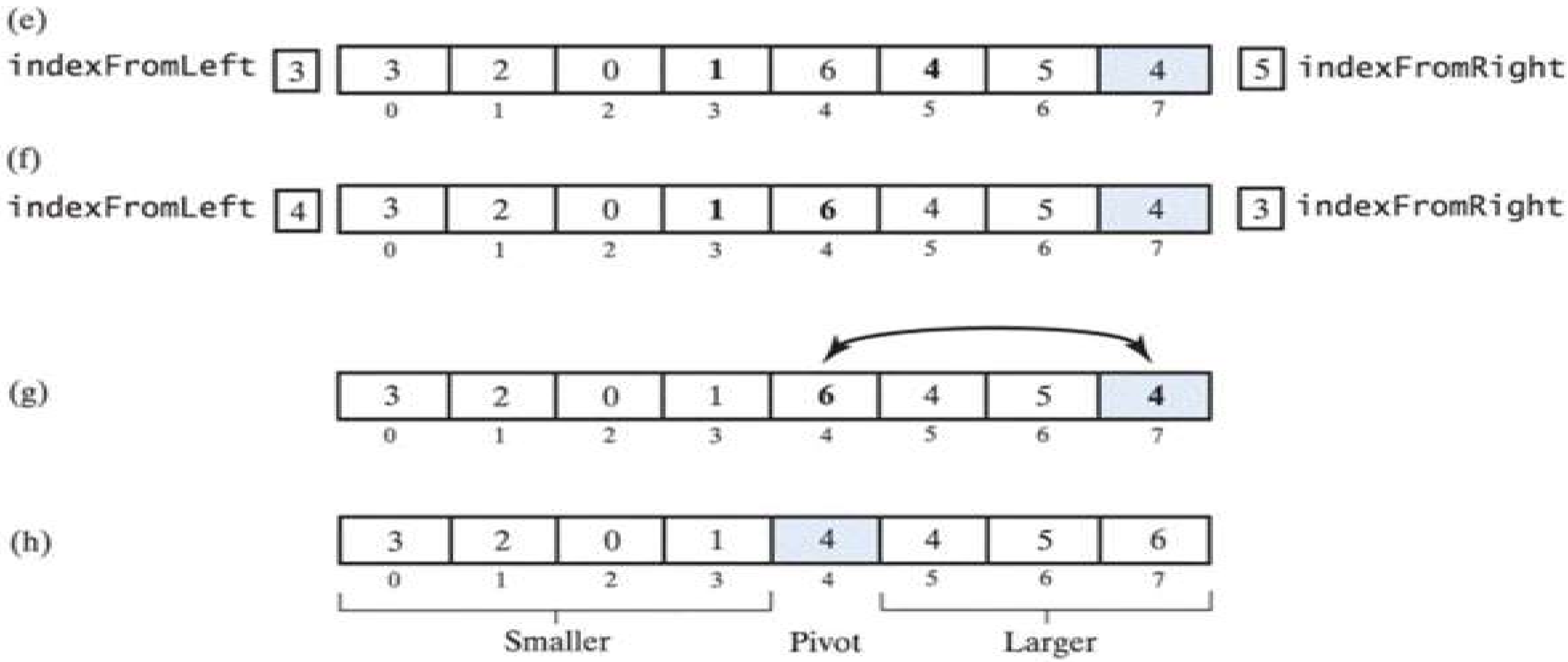


Fig. 12-6 (ctd.) A partition strategy for quick sort.

Quick Sort

- Quick sort rearranges the elements in an array during partitioning process
- After each step in the process
  - One element (the pivot) is placed in its correct sorted position
- The elements in each of the two sub arrays
  - Remain in their respective subarrays
- Note sort methods in **`java.util`** for **`Arrays`** class

Quick Sort: Sample Code

- `Chapter8\sorting\`**`SortArray.java`** methods
  - **`quickSort(T[] a, int n)`**
  - **`quickSort(T[] a, int first, int last)`**
  - **`partition(T[] a, int first, int last)`**
- See Appendix 8.1 for trace of quickSort and partition methods

Exercise

(b) Using Quicksort ,arrange the data.

55	99	44	77	88	33	22	66

Suppose an array contains the following initial values:

- Assume that the *pivot value is the first array element*.
- Show the array *after each swap* and *briefly explain the steps* that were carried out.

	0	1	2	3	4	5	6	7
a	55	99	44	77	88	33	22	66

f = 0, l = 7, pivIndex = 0, pivot = 55

	0	1	2	3	4	5	6	7
a	55	99	44	77	88	33	22	66
		iFL					iFR	

Swap 99 with 22, when iFL & iRL stop

	0	1	2	3	4	5	6	7
	55	22	44	77	88	33	99	66
				iFL		iFR		

Swap 77 with 33

	0	1	2	3	4	5	6	7
	55	22	44	33	88	77	99	66
				iFR	iFL			

	0	1	2	3	4	5	6	7
	55	22	44	33	88	77	99	66
				iFR*	iFL	*dividing point		

Swap pivot 55 with 33

	0	1	2	3	4	5	6	7
	33	22	44	55	88	77	99	66
				iFR*	iFL			

left					right			
quickSort(a, 0, 3)					quickSort(a, 4, 7)			
f = 0, l = 3, pivIndex = 0, pivot = 33							Pivot =88	
0	1	2	3		4	5	6	7
33	22	44	55		88	77	99	66
	iFR	iFL			swap 66 & 99		iFL	iRL
	1	2	3				Pivot =88	
33	22	44	55		4	5	6	7
					88	77	66	99
	iFR*	iFL						
*dividing point							iRL	iFL
0	1	2	3		4	5	6	7
22	33	44	55		66	77	88	99
				Merge				
0	1	2	3	4	5	6	7	
22	33	44	55	66	77	88	99	

Quick Sort – Pivot Selection

- The pivot should be the median value in the array, so that the subarrays Smaller and Larger each have the same – nearly the same – number of elements.
- One way to find the median value is to find the median of 3 elements in the array: *first*, *middle* and the *last* element.
  - Median-of-three-pivot selection avoid worst-case performance by quick sort when the given array is already sorted / nearly sorted.

Quick Sort - Median-of-three pivot selection:

(a)	**5**	8	6	4	**9**	3	7	1	**2**
(b)	**2**	8	6	4	**5**	3	7	1	**9**
					Pivot				

Fig. 12-7 Median-of-three pivot selection:
(a) the original array;

(b) the array with its first, middle, & last elements sorted

Quick Sort - Median-of-three pivot selection:

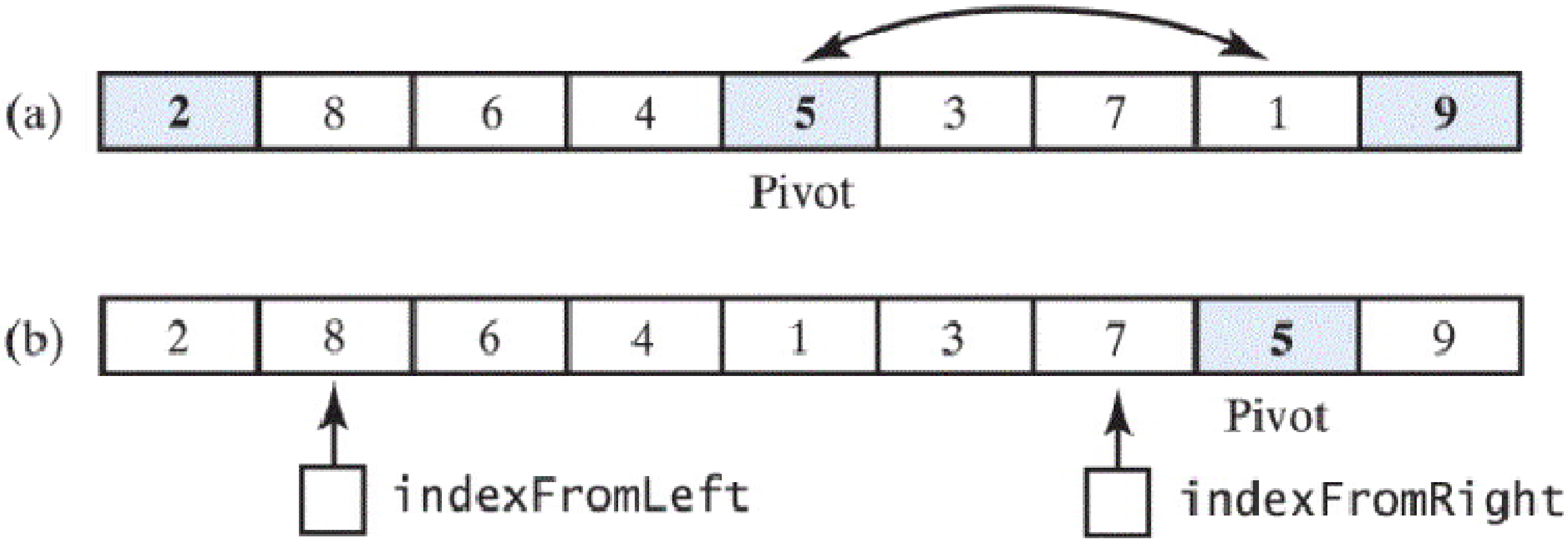


Fig. 12-8 (a) The array with its first,middle & last elements sorted; (b) The array after positioning the pivot & just before partitioning.

Comparing the Sorting Algorithms

Technique	Best case	Average Case	Worst Case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix sort	$O(n)$	$O(n)$	$O(n)$
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$ or $O(n^2)$

Selection for a suitable sorting methods

Factors:

- Speed
- Consistency of performance
- Memory requirements
- Stability .
- Versatility of handling various data types
- Complexity of coding and others programming language used
- Nature of the datasets

The `Comparator` Interface

- For comparing objects according to an order other than their natural ordering.
- A comparator is an object that is external to the class of the keys it compares.
- Provides multiple/alternative sorting sequence, i.e. you can sort the elements based on any data member.
- Method: **`compare(a, b)`**

`int compare(Object o1, Object o2)`

- Compares the first object with second object
- Returns an integer with similar meaning to the `compareTo` method of the `Comparable` interface.

`void sort(List list, Comparator c)`

- The **`Collections`** class provides static methods for sorting the elements of collections.
- The **`sort`** method is used to sort the elements of `List` by the given comparator

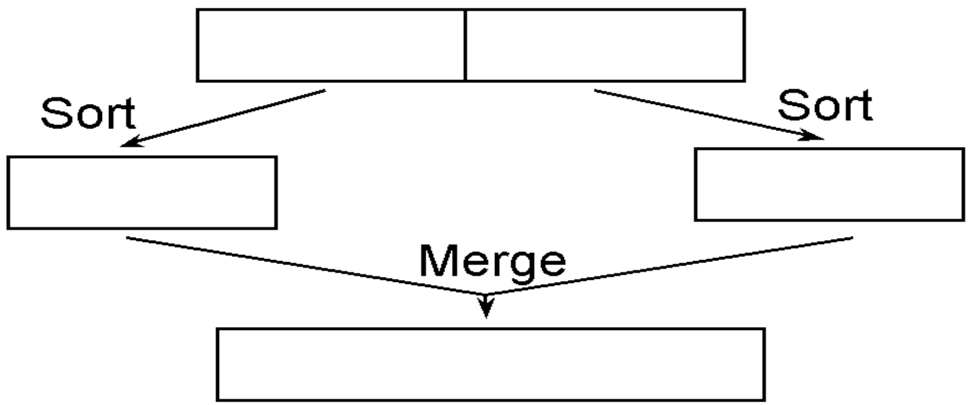
Sample Code

In **`Chapter8\sorting\`**

- **`Course.java`**
- **`CreditHoursComparison.java`**
- **`TestComparators.java`**
- **`TitleComparator.java`**

Merge Sort

- A recursive standard sorting algorithm
- Gives the same performance, regardless of the initial order of the array items.
- Strategy
  - Divide an array into 2 halves
  - Sort each half
  - Merge the sorted halves into one sorted array
- Referred to as a divide and conquer algorithm.



Merge Sort

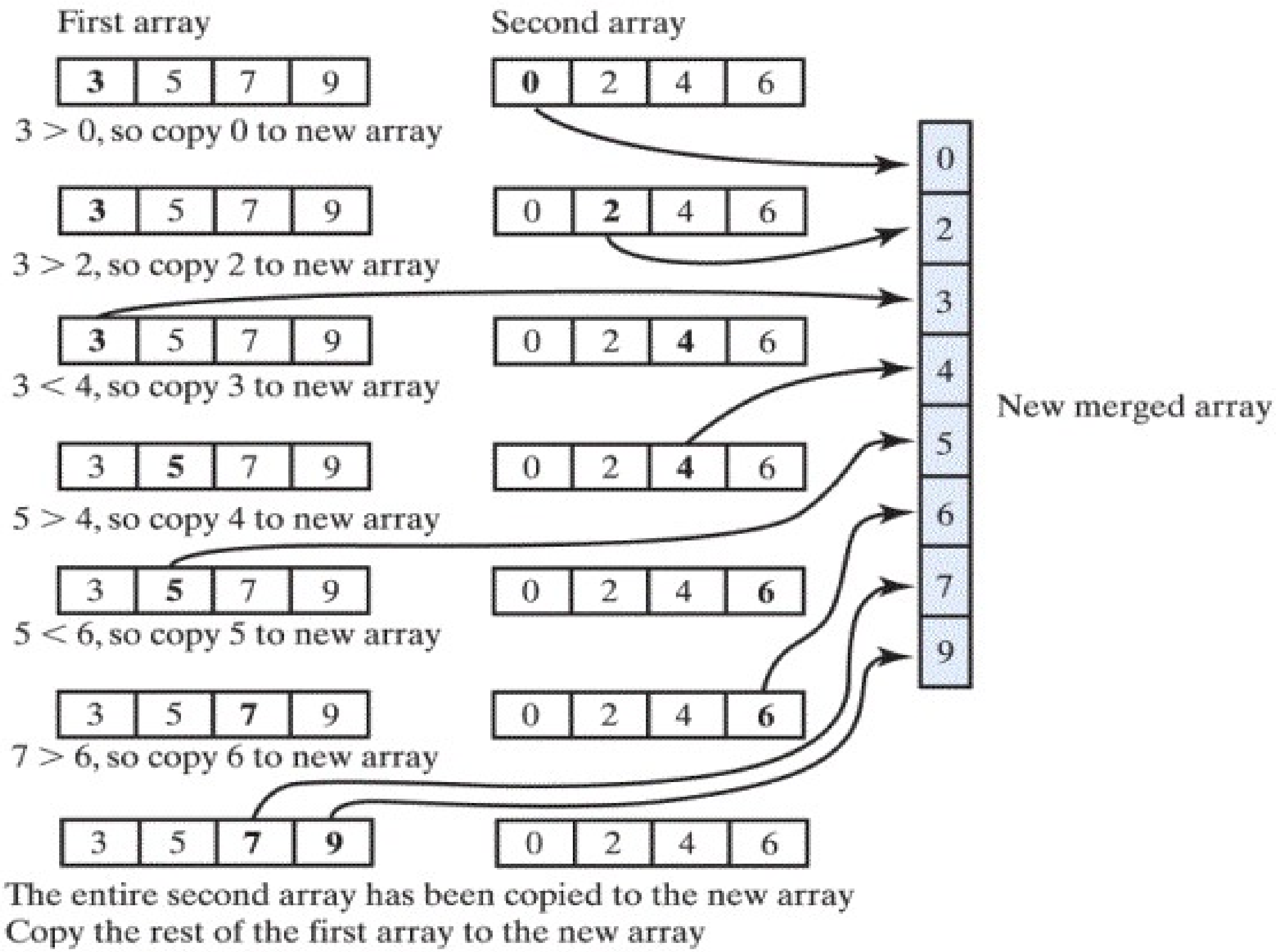


Fig. 12-1 Merging two sorted arrays into one sorted array.

Merge Sort

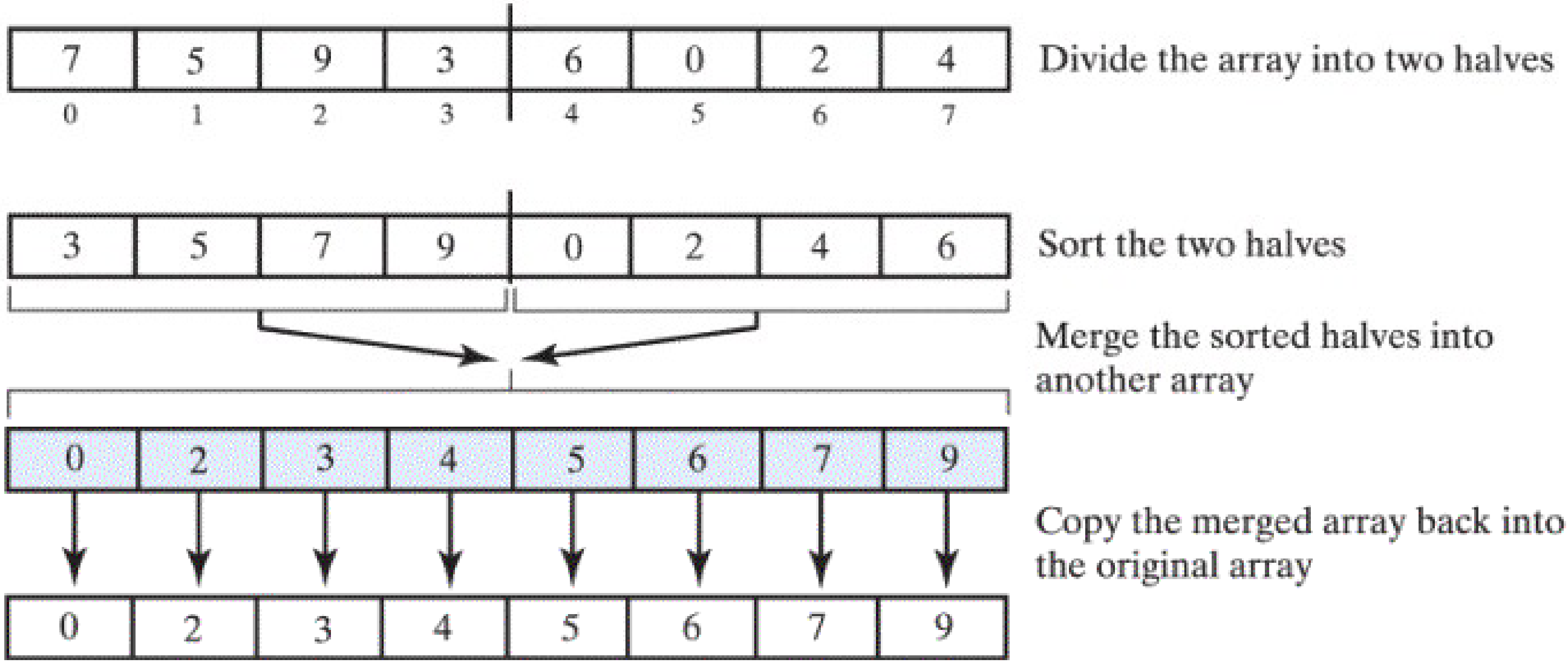


Fig. 12-2 The major steps in a merge sort.

Merge Sort

- A disadvantage of merge sort is the need for the temporary array during merge step.
- Advantage
  - It is an extremely efficient algorithm with respect to time

Algorithm for merge Sort

```
Algorithm mergeSort(a, first, last)
// Sorts the array elements a[first] through a[last] recursively.
   if (first < last)  {
        mid = (first + last)/2
        mergeSort(a, first, mid)
        mergeSort(a, mid+1, last)
        Merge the sorted halves a[first..mid] and a[mid+1..last]
   }
```

Algorithm for merge

```
Algorithm merge (a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].
    beginHalf1 = first              endHalf1 = mid
    beginHalf2 = mid + 1            endHalf2 = last
    /* While both subarrays are not empty, compare an element in 1 subarray
    with an element in the other; then copy the smaller item into the temporary
    array  */
    index = 0 // next available location in tempArray
    while ((beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2))  {
        if (a [beginHalf1] < a [beginHalf2])  {
          tempArray [index] = a [beginHalf1]
          beginHalf1++ }
        else {     tempArray [index] = a [beginHalf2]
                   beginHalf2++                    }
        index++
    } // Assertion: 1 subarray has been completely copied to tempArray
```

Merge Sort

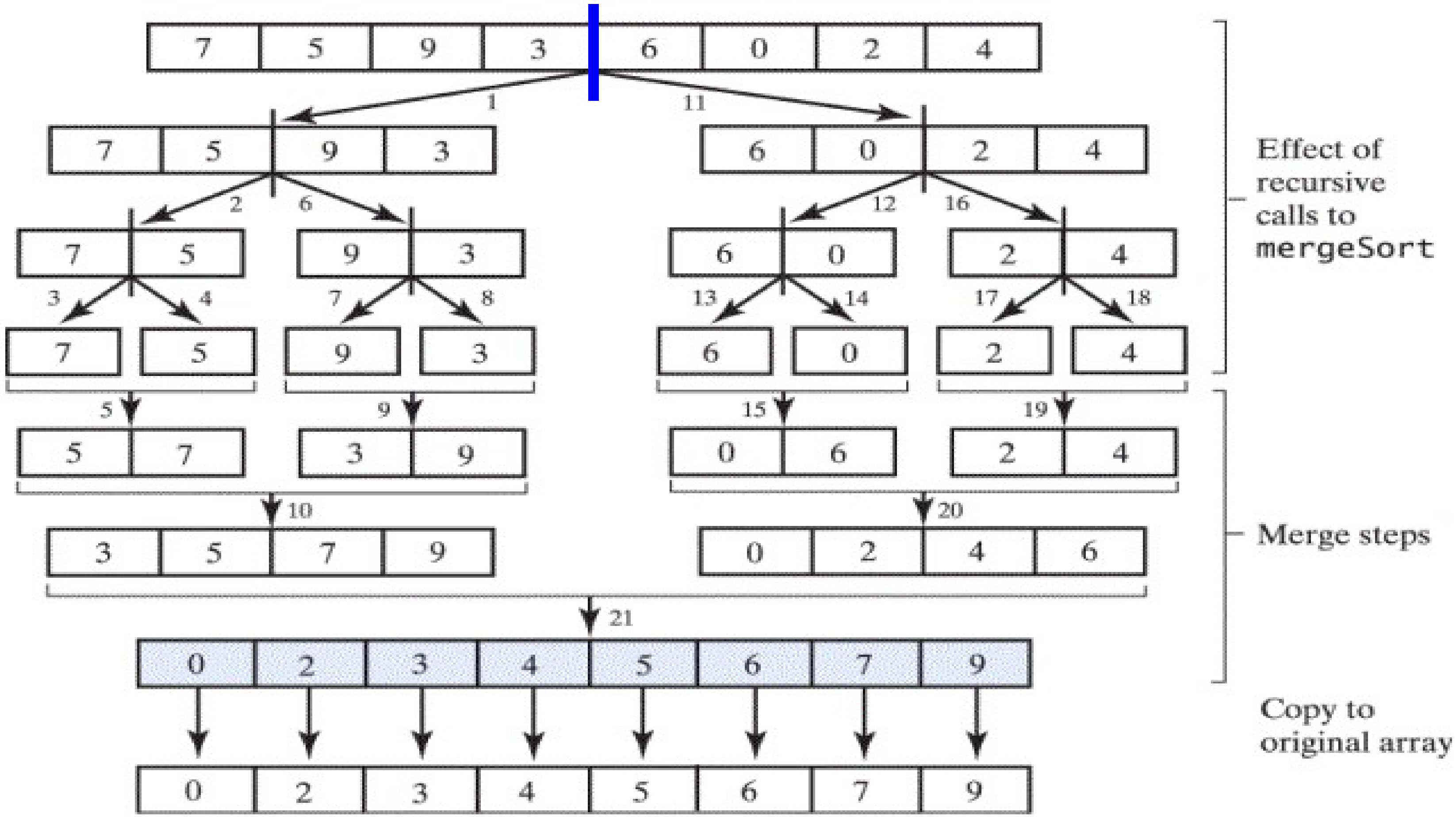


Fig. 12-3 The effect of the recursive calls and the merges during a merge sort.

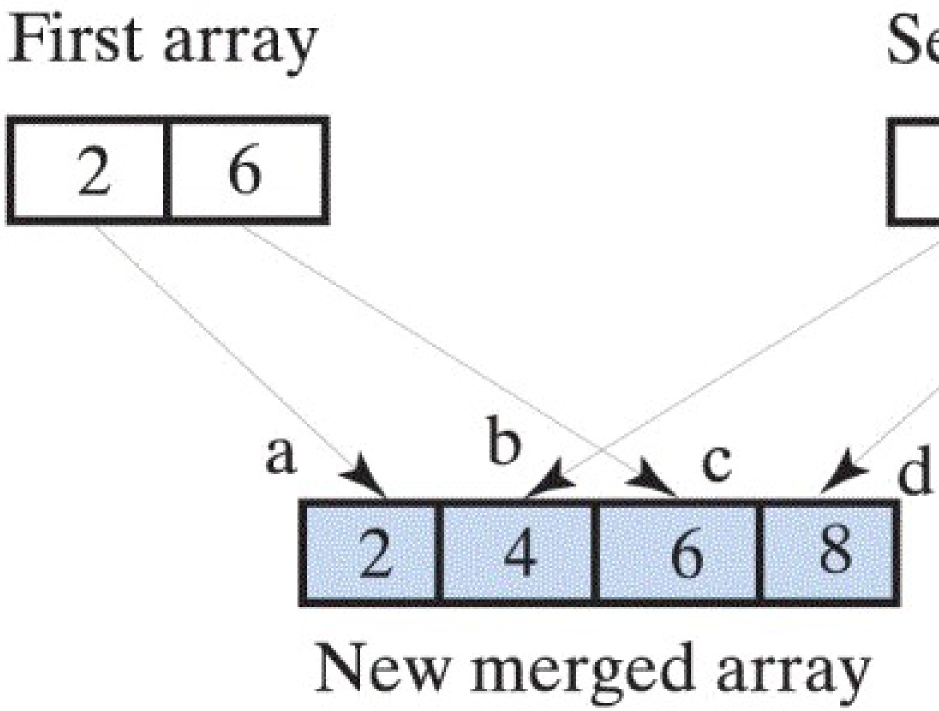
Merge Sort Efficiency

- Merge sort is O(n log n) in all cases
  - It's need for a temporary array is a disadvantage
- Merge sort in the Java Class Library
  - The class **`Arrays`** has **`sort`** routines that uses the merge sort for arrays of objects

```
public static void sort(Object[] a);

public static void sort
        (Object[] a, int first, int last);
```

Merge Sort Efficiency



a. 2 < 4, so copy 2 to new array

b. 6 > 4, so copy 4 to new array

c. 6 < 8, so copy 6 to new array

d. Copy 8 to new array

Fig. 12-4 A worst-case merge of two sorted arrays

Review of learning outcomes

You should now be able to:

- Sort an array using the sorting methods:
  - Selection sort, insertion sort, bubble sort, shell sort, merge sort, quick sort and radix sort.
- Assess the time efficiencies of various sorting methods, expressed in Big O notation.

To Do

- Review the slides and source code for this chapter.
- Read up the relevant portions of the recommended text.

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson