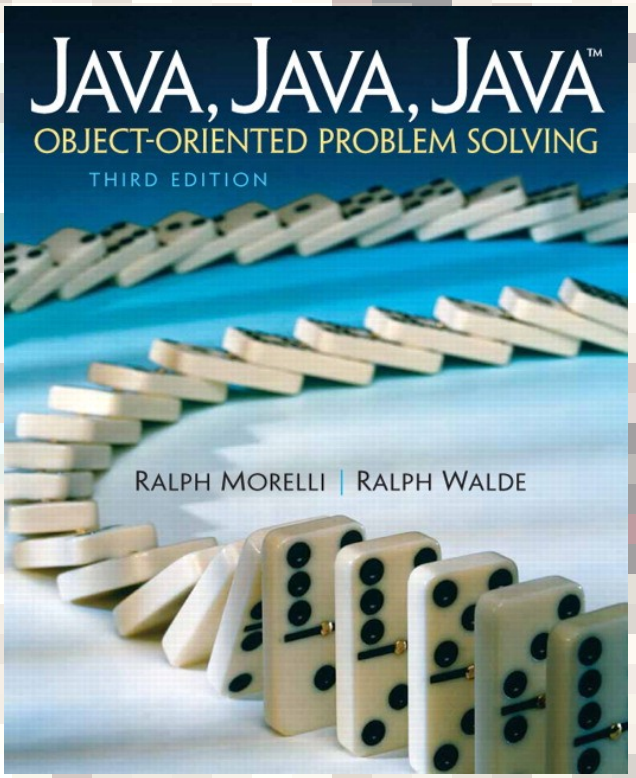


presentation slides for

JAVA, JAVA, JAVA

Object-Oriented Problem Solving

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 10: Arrays Sorting

Objectives

- Know how to sort an array of data.

Outline

- Introduction to Array Sorting Algorithms
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort

Sorting Introduction

- Sorting: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
  - One of the fundamental problems in computer science
  - Can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...
  - comparison-based sorting : determining order by comparing pairs of elements:
    - `<`, `>`, `compareTo`, ...

Sorting methods in Java

- The `Arrays` and `Collections` classes in `java.util` have a static method `sort` that sorts the elements of an array/list

```
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]
List<String> words2 = new ArrayList<String>();
for (String word : words) {
    words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```

Sorting algorithms

Some of the famous algorithms:

- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value

other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...

Selection sort

- Selection sort: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.
  - Pseudocode:
    - Look through the list to find the smallest values
    - Swap it with the element at the index 0
    - Look through the list to find the second-smallest element
    - Swap it with the element at the index 1
    - …
    - Repeat until all values are in the proper values

Selection sort example

- Initial array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

- After 1st, 2nd, 3rd passes:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98

Algorithm Design: Swapping Elements

- A *temporary variable* must be used when swapping the values of two variables in memory.
- Suppose you have the array: 1 4 2 8
- Swapping 4 and 2 the wrong way:

```
arr[pair-1] = arr[pair];
arr[pair] = arr[pair-1];
```

results in: 1 2 2 8

- Swapping 4 and 2 the proper way:

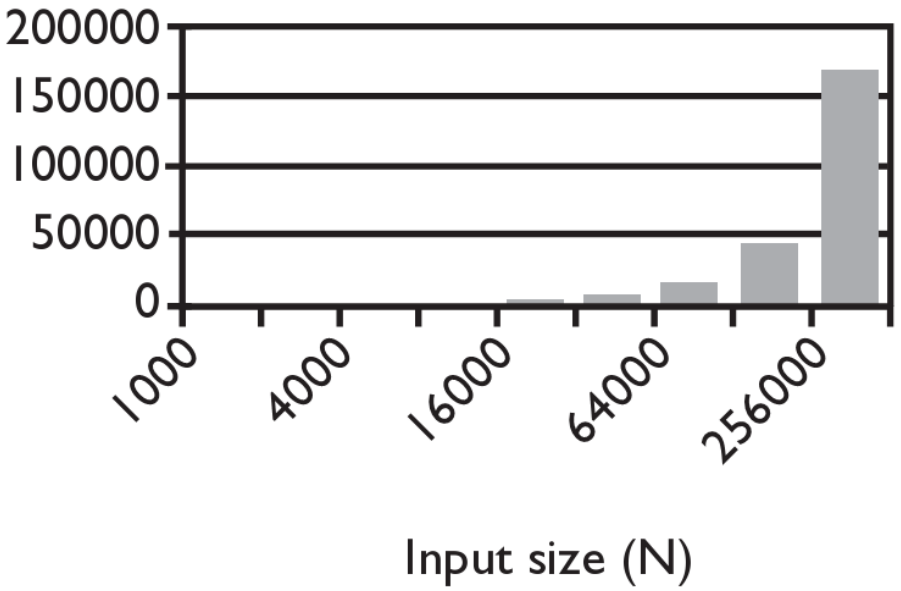
```
temp = arr[pair-1];       // Save first element in temp
arr[pair-1] = arr[pair]; // Overwrite first with second
arr[pair] = temp;         // Overwrite second with temp (i.e.,first)
```

results in: 1 2 4 8

Selection sort runtime

- What is the running time of selection sort?

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Bubble sort

- Bubble sort: a simple **sorting** algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is **sorted**.
  - slower than selection sort (has to do more swaps)

- Original array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

- First loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
18	12	-4	22	27	30	36	7	50	68	56	2	85	42	91	98

22 → 50 → 91 →

Bubble sort example

- After the first loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
18	12	-4	22	27	30	36	7	50	68	56	2	85	42	91	98

- The second loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	-4	**18**	22	27	30	7	**36**	50	56	2	**68**	42	**85**	91	98

18 → **36** → **68** → **85** →

- The third loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	**12**	18	22	27	7	**30**	36	50	2	56	**42**	68	85	91	98

12 → **30** → **56** → **68** →

Bubble sort example (cont.)

- After the third loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	22	27	7	30	36	50	2	56	42	68	85	91	98

- After the four loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	22	7	27	30	36	2	50	42	56	68	85	91	98

27 → 50 → 56 →

- After the fifth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	7	22	27	30	2	36	42	50	56	68	85	91	98

22 → 36 → 50 →

Bubble sort example (cont.)

- After the fifth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	7	22	27	30	2	36	42	50	56	68	85	91	98

- After the sixth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	7	18	22	27	2	30	36	42	50	56	68	85	91	98

18 → 30 →

- After the seventh loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	22	2	27	30	36	42	50	56	68	85	91	98

12 → 27 →

Bubble sort example (cont.)

- After the seventh loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	22	2	27	30	36	42	50	56	68	85	91	98

- After the eighth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	2	22	27	30	36	42	50	56	68	85	91	98

22 →

- After the ninth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	2	18	22	27	30	36	42	50	56	68	85	91	98

18 →

Bubble sort example (cont.)

- After the ninth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	2	18	22	27	30	36	42	50	56	68	85	91	98

- After the tenth loop

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	2	12	18	22	27	30	36	42	50	56	68	85	91	98

12 →

- After the eleventh loop – final - sorted

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	7	12	18	22	27	30	36	42	50	56	68	85	91	98

7 →

Insertion sort

- The *insertion sort algorithm* views the array as divided into two parts, the sorted and unsorted parts. On each iteration, it moves one element from the unsorted to its correct position in the sorted part. Generally, faster than selection sort

Insertion Sort of an array, arr, of N elements into ascending order

1. For k assigned 1 through N-1
2. Remove the element arr[k] and store it in x.
3. For i starting at k-1 and for all preceding elements greater than x
4. Move arr[i] one position to the right in the array.
5. Insert x at its correct location.

Insertion sort example

- Original array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

- First iteration – insert a[1] into a[0] – a[0]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
18	22	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

← 18

- Second iteration – insert a[2] into a[0] – a[1]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	18	22	-4	27	30	36	50	7	68	91	56	2	85	42	98

← 12

Insertion sort example (cont.)

- Third iteration – insert a[3] into a[0] – a[2]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98

- Fourth iteration to seventh iteration do nothing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98

- Eighth iteration – insert a[8] into a[0] – a[7]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	22	27	30	36	50	68	91	56	2	85	42	98

Insertion sort example (cont.)

- Fourth ninth to tenth iteration do nothing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	22	27	30	36	50	68	91	56	2	85	42	98

- Eleventh iteration – insert a[11] into a[0] – a[10]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	7	12	18	22	27	30	36	50	**56**	68	91	2	85	42	98

- Twelfth iteration – insert a[12] into a[0] – a[11]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	**2**	7	12	18	22	27	30	36	50	56	68	91	85	42	98

Insertion sort example (cont.)

- 13th iteration – insert a[13] into a[0] – a[12]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	7	12	18	22	27	30	36	50	56	68	85	91	42	98

← 85

- 14th iteration – insert a[14] into a[0] – a[13]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	7	12	18	22	27	30	36	42	50	56	68	85	91	98

← 42

- 15th iteration – do nothing as already sorted

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-4	2	7	12	18	22	27	30	36	42	50	56	68	85	91	**98**

Method Design: insertionSort()

- Array parameters are references. Changes made to the array in the method will persist.

```
/**
  * Goal: Sort the values in arr into ascending order
  * Pre: arr is not null.
  * Post: The values arr[0]...arr[arr.length-1] will be
  *  arranged in ascending order.
  */
public void insertionSort(int arr[]) {
    int temp;                                  // Temporary variable for insertion
    for (int k = 1; k < arr.length; k++) {  // For each pass
        temp = arr[k];                         // Remove element from array
        int i;
        for (i = k-1; i >= 0 && arr[i] > temp; i--) // Move larger preceding
            arr[i+1] = arr[i];                 //    elements right by one space
        arr[i+1]= temp;
    } // for
} // insertionSort()
```

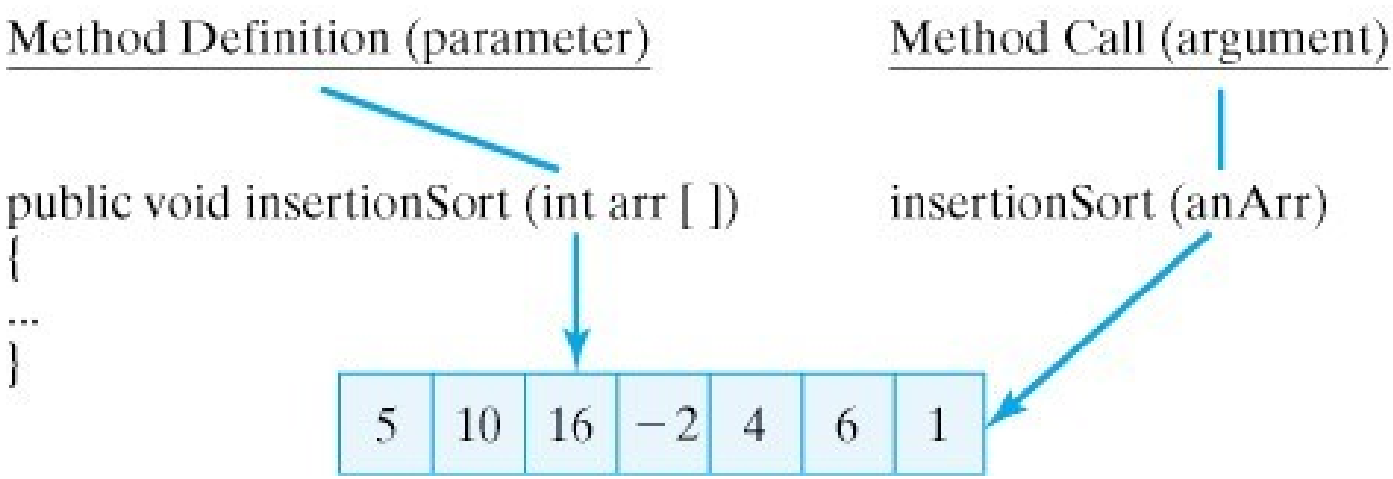
Note how an array parameter is specified.

Note how an array argument is passed.

```
int myArr[] = { 21, 13, 5, 10, 14 };
insertionSort(myArr);
```

Passing an Array Parameter

- When an array is passed to a method, both the array reference (anArr) and the parameter (arr) refer to the same object.



Implementation: The Sort Class

```
public class Sort {
   public void print(int arr[]) {
        for (int k = 0; k < arr.length; k++)        // For each integer
            System.out.print( arr[k] + " \t ");     //  Print it
        System.out.println();
    } // print()


    public static void main(String args[]) {
        int intArr[] = { 21, 20, 27, 24, 19 };
        Sort sorter = new Sort();
        sorter.print(intArr);
        sorter.insertionSort(intArr);
        sorter.print(intArr);
    } // main()
} // Sort
```

```
public void insertionSort(int arr[]) {
    int temp;                                   // Temporary variable for insertion
    for (int k = 1; k < arr.length; k++) {  // For each pass
        temp = arr[k];                          // Remove element from array
        int i;
        for (i = k-1; i >= 0 && arr[i] > temp; i--) // Move larger preceding
            arr[i+1] = arr[i];                  //    elements right by one
        arr[i+1]= temp;
    } // for
} // insertionSort()
```

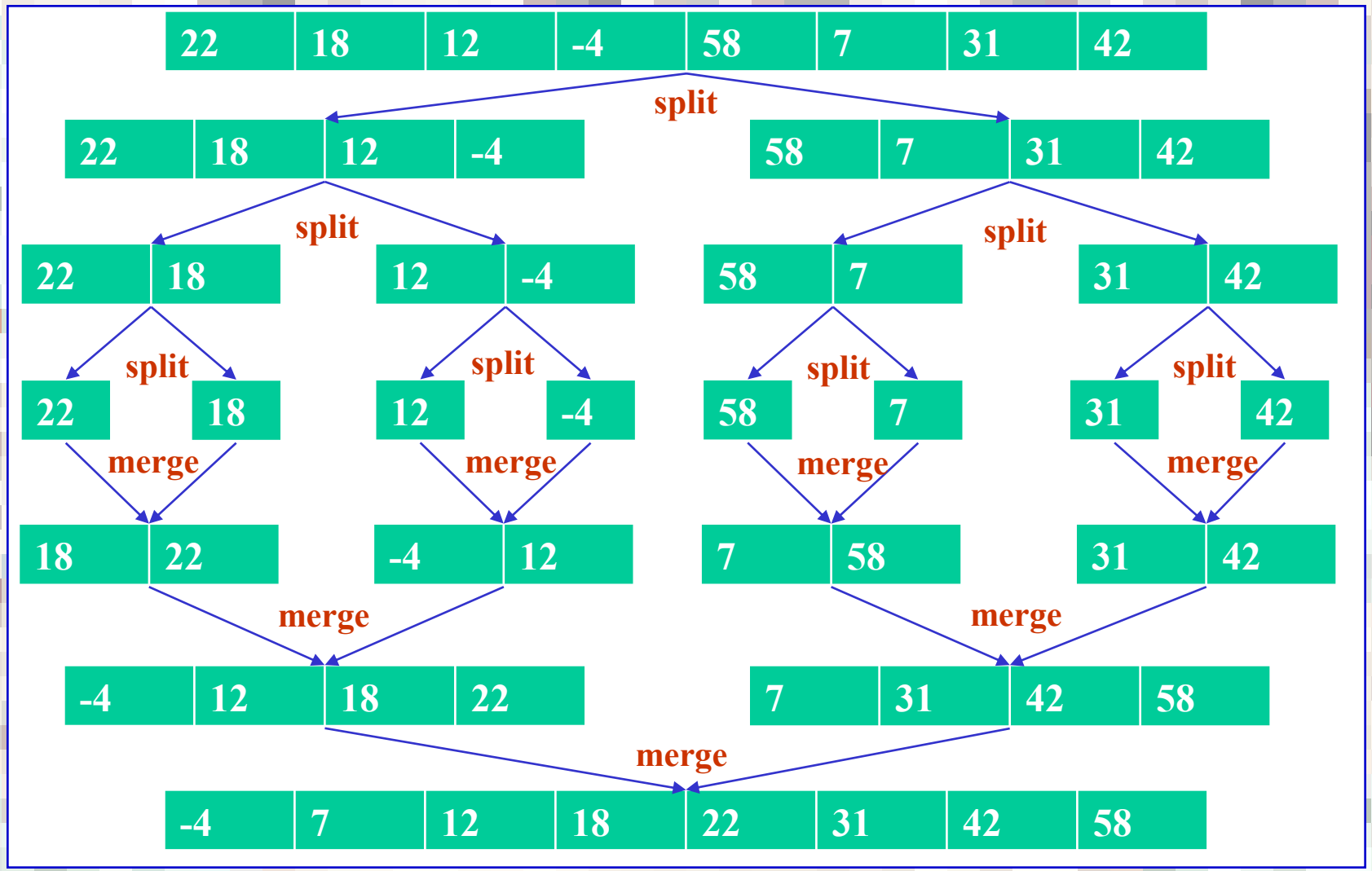
Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

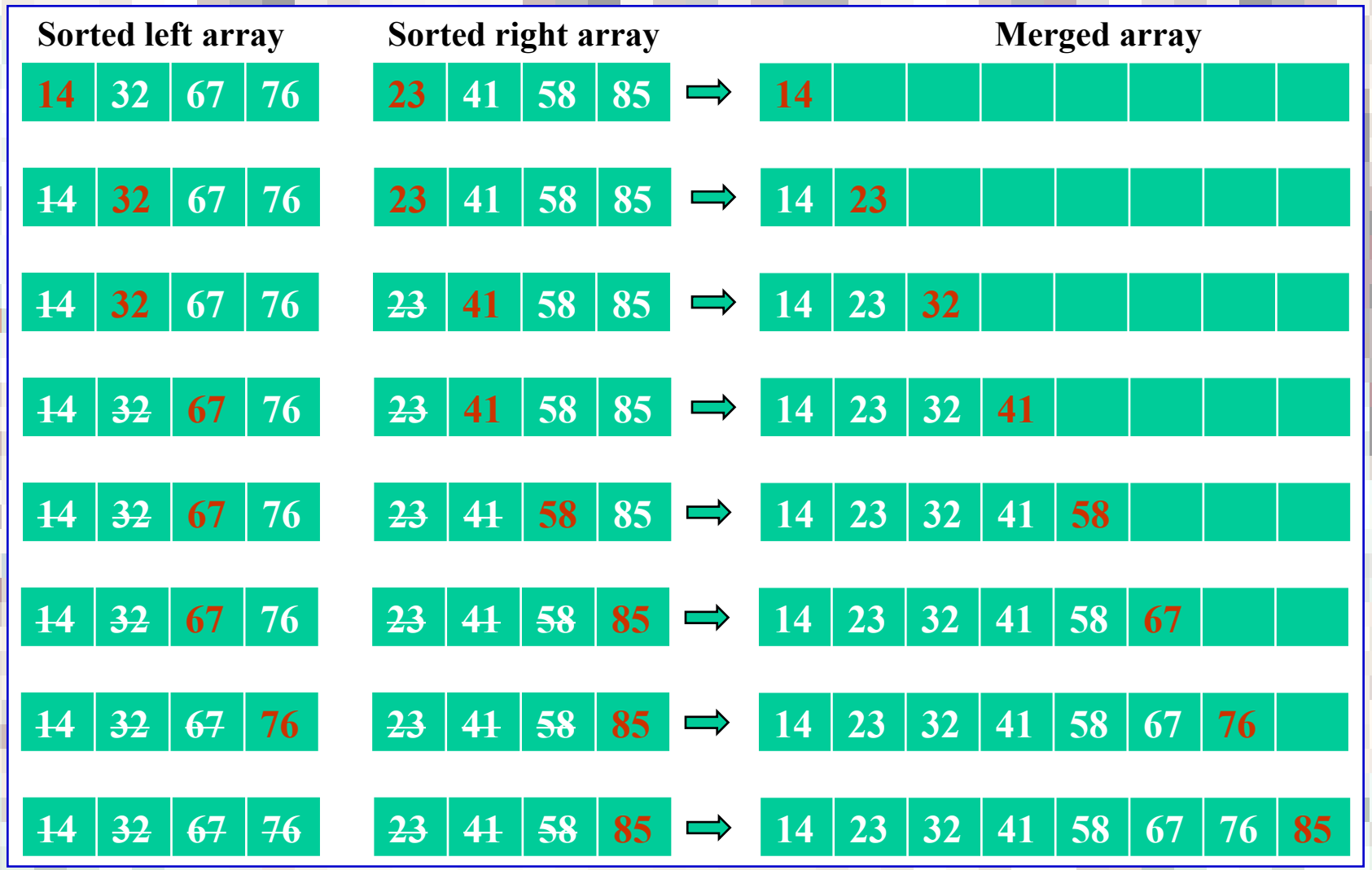
  The algorithm:
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.

  - An example of a "divide and conquer" algorithm.
    Invented by John von Neumann in 1945

Merge sort example

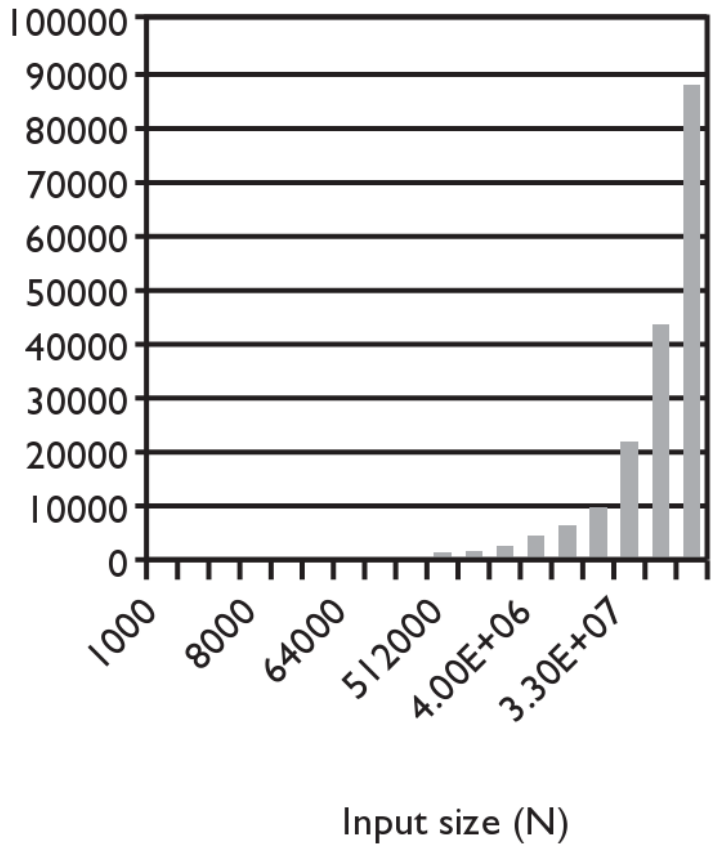


Merging sorted halves example



Merge sort runtime

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



Technical Terms

- array
- array sorting
- bubble sort
- selection sort
- insertion sort
- merge sort

Summary Of Important Points

- *Sorting* is rearranging the values in an array or collection into a specific order.
- *Selection sort* finds the smallest element and swap with the first, and repeat with the second smallest, etc. until the last element.
- *Bubble sort* continuously swap two adjacent elements in the array until it is sorted.
- *Insertion sort* iterates from the second to the last and move them into the correct position on the left.
- *Merge sort* repeatedly divides the array into halves, sorted them, and then merge them.