

# Building Java Programs

Chapter 13  
Sorting

**reading: 13.3, 13.4**



# Collections class

Method name	Description
binarySearch ( <b>list, value</b> )	returns the index of the given value in a sorted list (< 0 if not found)
copy ( <b>listTo, listFrom</b> )	copies <b>listFrom</b> 's elements to <b>listTo</b>
emptyList (), emptyMap (), emptySet ()	returns a read-only collection of the given type that has no elements
fill ( <b>list, value</b> )	sets every element in the list to have the given value
max ( <b>collection</b> ), min ( <b>collection</b> )	returns largest/smallest element
replaceAll ( <b>list, old, new</b> )	replaces an element value with another
reverse ( <b>list</b> )	reverses the order of a list's elements
shuffle ( <b>list</b> )	arranges elements into a random order
sort ( <b>list</b> )	arranges elements into ascending order

# Sorting

- **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

- **bogo sort:** shuffle and pray
- **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.

The algorithm:

- Look through the list to find the smallest value.
  - Swap it so that it is at index 0.
  - Look through the list to find the second-smallest value.
  - Swap it so that it is at index 1.
- ...
- Repeat until all values are in their proper places.

# Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

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

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98	25

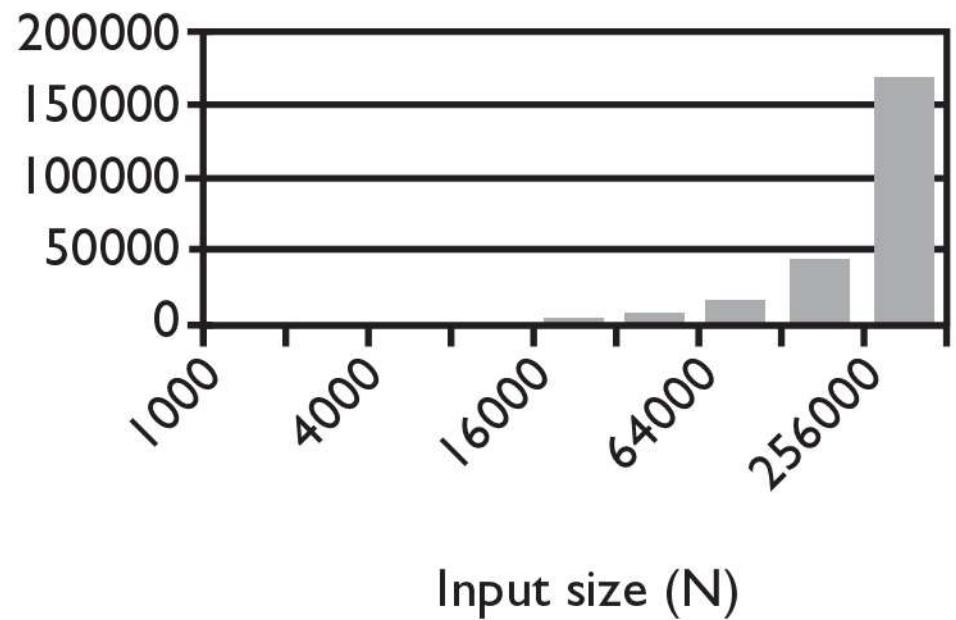
# Selection sort code

```
// Rearranges the elements of a into sorted order using
// the selection sort algorithm.
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // swap smallest value its proper place, a[i]
        swap(a, i, min);
    }
}
```

# Selection sort runtime (Fig. 13.6)

- What is the complexity class (Big-Oh) of selection sort?

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



# Bogo sort

- **bogo sort:** Orders a list of values by repetitively shuffling them and checking if they are sorted.
  - name comes from the word "bogus"

The algorithm:

- Scan the list, seeing if it is sorted. If so, stop.
  - Else, shuffle the values in the list and repeat.
- 
- This sorting algorithm (obviously) has terrible performance!
    - What is its runtime?

# Bogo sort code

```
// Places the elements of a into sorted order.  
public static void bogoSort(int[] a) {  
    while (!isSorted(a)) {  
        shuffle(a);  
    }  
}  
  
// Returns true if a's elements are in sorted order.  
public static boolean isSorted(int[] a) {  
    for (int i = 0; i < a.length - 1; i++) {  
        if (a[i] > a[i + 1]) {  
            return false;  
        }  
    }  
    return true;  
}
```

# Bogo sort code, cont'd.

```
// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick a random index in [i+1, a.length-1]
        int range = a.length - 1 - (i + 1) + 1;
        int j = (int) (Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}

// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
    if (i != j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

# Similar algorithms

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- **bubble sort:** Make repeated passes, swapping adjacent values
  - slower than selection sort (has to do more swaps)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	18	12	-4	22	27	30	36	7	50	68	56	2	85	42	91	25	98

22 →                    50 →                    91 →                    98 →

- **insertion sort:** Shift each element into a sorted sub-array
  - faster than selection sort (examines fewer values)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98	25

sorted sub-array (indexes 0-7)

7

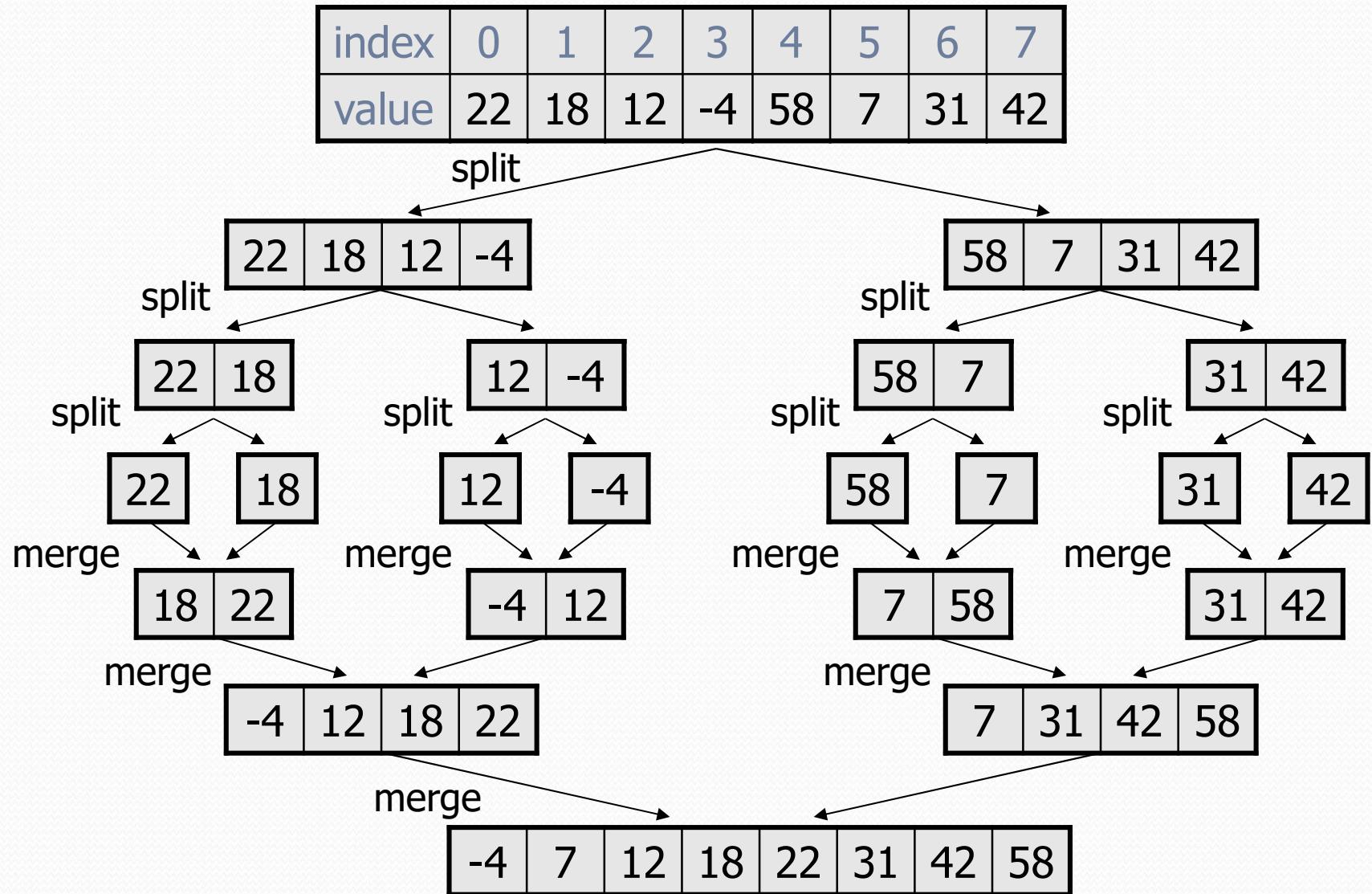
# Merge sort

- **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

Subarrays	Next include	Merged array																																																
<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td> <td>0</td><td>1</td><td>2</td><td>3</td> </tr> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	0	1	2	3	0	1	2	3	14	32	67	76	23	41	58	85	i1				i2				14 from left	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td> </tr> <tr> <td>14</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	2	3	4	5	6	7	14								i							
0	1	2	3	0	1	2	3																																											
14	32	67	76	23	41	58	85																																											
i1				i2																																														
0	1	2	3	4	5	6	7																																											
14																																																		
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				23 from right	<table border="1"> <tr> <td>14</td><td>23</td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23							i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23																																																	
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				32 from left	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32						i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23	32																																																
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				41 from right	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td>41</td><td></td><td></td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32	41					i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23	32	41																																															
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				58 from right	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td>41</td><td>58</td><td></td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32	41	58				i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23	32	41	58																																														
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				67 from left	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td>41</td><td>58</td><td>67</td><td></td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32	41	58	67			i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23	32	41	58	67																																													
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td>i1</td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85	i1				i2				76 from left	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td>41</td><td>58</td><td>67</td><td>76</td><td></td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32	41	58	67	76		i																							
14	32	67	76	23	41	58	85																																											
i1				i2																																														
14	23	32	41	58	67	76																																												
i																																																		
<table border="1"> <tr> <td>14</td><td>32</td><td>67</td><td>76</td> <td>23</td><td>41</td><td>58</td><td>85</td> </tr> <tr> <td></td><td></td><td></td><td></td> <td>i2</td><td></td><td></td><td></td> </tr> </table>	14	32	67	76	23	41	58	85					i2				85 from right	<table border="1"> <tr> <td>14</td><td>23</td><td>32</td><td>41</td><td>58</td><td>67</td><td>76</td><td>85</td> </tr> <tr> <td>i</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	14	23	32	41	58	67	76	85	i																							
14	32	67	76	23	41	58	85																																											
				i2																																														
14	23	32	41	58	67	76	85																																											
i																																																		

# 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 halves code

```
// Merges the left/right elements into a sorted result.  
// Precondition: left/right are sorted  
public static void merge(int[] result, int[] left,  
                         int[] right) {  
    int i1 = 0;      // index into left array  
    int i2 = 0;      // index into right array  
  
    for (int i = 0; i < result.length; i++) {  
        if (i2 >= right.length ||  
            (i1 < left.length && left[i1] <= right[i2])) {  
            result[i] = left[i1];      // take from left  
            i1++;  
        } else {  
            result[i] = right[i2];    // take from right  
            i2++;  
        }  
    }  
}
```

# Merge sort code

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    // split array into two halves
    int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2,
a.length);

    // sort the two halves
    ...

    // merge the sorted halves into a sorted whole
    merge(a, left, right);
}
```

# Merge sort code 2

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left   = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right  = Arrays.copyOfRange(a, a.length/2,
a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

# Merge sort runtime

- What is the complexity class (Big-Oh) of merge sort?

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

