

4.15 Sorting Algorithms

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Explain what it means for an array or ArrayList to be sorted, and check whether one is.
- Carry out selection sort by hand and show the list after every pass.
- Trace the Java code for selection sort, including the inner loop that finds the smallest element and the swap.
- Carry out insertion sort by hand and show how elements shift to make room for each inserted value.
- Trace the Java code for insertion sort, including the while loop that shifts elements.
- Follow both sorts on an ArrayList, including a list of String values.
- Tell selection sort and insertion sort apart from the in-between states they produce, and compare how much work each does.

What to remember from earlier

- Traverse an array with a for loop whose index runs from 0 to arr.length - 1. ([Topic 4.4: Array Traversals](#), [4.4.1 Visiting Every Element with a for Loop](#))
- Swap two elements with a temporary variable. ([Topic 4.5: Implementing Array Algorithms](#), [4.5.8 Reversing an Array](#))
- Finding the smallest value uses a best-so-far variable. ([Topic 4.5: Implementing Array Algorithms](#), [4.5.2 Minimum and Maximum](#))
- a.compareTo(b) is negative when a comes first, zero when equal, and positive when a comes after b. ([Topic 1.15: String Manipulation](#), [1.15.9 Comparing Strings with equals and compareTo](#))
- An inner loop runs completely for every pass of the outer loop. ([Topic 2.11: Nested Iteration](#), [2.11.2 The Inner Loop Finishes First](#))

4.15.1 What It Means to Be Sorted

A music app shows your songs from most played to least played. A teacher's gradebook lists students alphabetically. A race posts finishing times from fastest to slowest. In each case, someone had a jumbled collection and put it in order. Programs do this constantly, because ordered data is far easier for people to read and for other algorithms to use.

A collection is **sorted** when its elements are arranged in order according to some rule. In **ascending order**, each element is less than or equal to the one after it (smallest first): 3 7 7 12 20. In **descending order**, each element is greater than or equal to the one after it (largest first): 20 12 7 7 3. Notice "or equal to": repeated values are allowed, and they simply sit next to each other. Unless a question says otherwise, "sorted" in this chapter means ascending.

That definition only talks about **neighbors**. To check whether an array is in ascending order, you do not have to compare every element with every other element. You only need to check that no element is bigger than the one right after it. If every neighboring pair is in order, the whole array is in order.

KEY IDEA

An array is in ascending order when every element is less than or equal to the element right after it. Checking each neighboring pair is enough to check the whole array.

WORKED EXAMPLE Checking whether an array is sorted

This code uses the consecutive-pairs traversal from Topic 4.5 to look at each neighboring pair. Watch the loop condition: the index stops at `scores.length - 2`, because `scores[i + 1]` must still be a real element.

```
1 int[] scores = {12, 18, 18, 25, 31};
2 boolean sorted = true;
3 for (int i = 0; i < scores.length - 1; i++)
4 {
5     if (scores[i] > scores[i + 1])
6     {
7         sorted = false;
8     }
9 }
10 System.out.println("scores sorted? " + sorted);
11
12 int[] times = {40, 35, 52, 61};
13 boolean timesSorted = true;
14 for (int i = 0; i < times.length - 1; i++)
15 {
16     if (times[i] > times[i + 1])
17     {
18         timesSorted = false;
19     }
20 }
21 System.out.println("times sorted? " + timesSorted);
```

```
scores sorted? true
times sorted? false
```

For `scores`, the pairs are (12, 18), (18, 18), (18, 25), and (25, 31). None of them has a bigger value first, so `sorted` stays `true`. The pair (18, 18) is fine because equal values are allowed. For `times`, the very first pair (40, 35) is out of order, so `timesSorted` becomes `false`. The loop keeps checking the rest, but nothing can change it back to `true`.

A **sorting algorithm** is a step-by-step process that rearranges the elements of a collection into sorted order. This chapter teaches two of them: **selection sort** and **insertion sort**. Both are **iterative**: they repeat their steps with loops. Both work on an array or an `ArrayList`.

Both algorithms share one big idea. At any moment during the sort, the list is split into two parts. The **sorted portion** at the front is already in order. The **unsorted portion** after it has not been dealt with yet. Each pass of the algorithm moves one more element into the sorted portion, so the sorted portion grows by one and the unsorted portion shrinks by one, until the unsorted portion is empty. The two algorithms differ in **how** they choose the next element and where they put it.

sorted portion		unsorted portion
7 12		45 29 18

WATCH OUT

Letting the index run to the last element. It seems natural to loop over every index, but this check compares `scores[i]` with `scores[i + 1]`:

```
1 int[] scores = {12, 18, 18, 25, 31};
2 boolean sorted = true;
3 for (int i = 0; i < scores.length; i++)
4 {
5     if (scores[i] > scores[i + 1])
6     {
7         sorted = false;
8     }
9 }
10 System.out.println("scores sorted? " + sorted);
```

When `i` reaches 4, the last index, the code asks for `scores[5]`, which does not exist. The program crashes with a message like this (the wording can vary between Java versions):

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of
bounds for length 5
```

An array of 5 elements has only 4 neighboring pairs. Students make this mistake because most traversals do visit every index. When the body also looks at the **next** element, stop one early with `i < scores.length - 1`, as in the worked example.

PAUSE AND TRY

Question 1 Multiple choice

Which array is in ascending order?

- A. {2, 4, 4, 8}
- B. {8, 6, 4, 2}
- C. {1, 3, 2, 4}
- D. {5, 7, 9, 8}

Question 2 Tracing

What does this code print?

```

1 int[] heights = {4, 9, 6, 8, 2};
2 int outOfOrder = 0;
3 for (int i = 0; i < heights.length - 1; i++)
4 {
5     if (heights[i] > heights[i + 1])
6     {
7         outOfOrder++;
8     }
9 }
10 System.out.println(outOfOrder);

```

4.15.2 Selection Sort by Hand

Imagine five numbered cards face up on a table, in a jumbled row. One natural way to put them in order: find the smallest card and move it to the far left. Then find the smallest of the cards that are left and put it second. Keep going. That is selection sort. Before writing any code, you need to be able to do it by hand, because that is exactly what AP questions ask: "What does the array look like after two passes?"

In Topic 4.5 you found the smallest value in an array with a traversal. Selection sort runs that same search over and over, each time on a smaller part of the array.

Here is the algorithm in plain steps. Each repetition is called a **pass**.

1. Look at the unsorted portion and find its smallest element.
2. Swap that element with the **first** element of the unsorted portion.
3. The sorted portion now has one more element. Repeat on what is left.

Selection sort gets its name from step 1: each pass **selects** the smallest remaining element. The element it places is in its **final position**: since it is the smallest of everything left, nothing that comes later can belong in front of it. It will never move again.

Selection sort uses a **swap**, not a shift. Only two elements change places in a pass. The element that was sitting in the first unsorted spot jumps to wherever the smallest element came from.

KEY IDEA

Each pass of selection sort finds the smallest element of the unsorted portion and swaps it into the first unsorted position. That element is now in its correct and final position.

WORKED EXAMPLE Selection sort on five scores

Start with 29 12 45 7 18. The bar | marks where the sorted portion ends.

Pass	Smallest in unsorted portion	Swap	Array after the pass
1	7 (index 3)	29 and 7	7 12 45 29 18
2	12 (index 1)	12 with itself (no change)	7 12 45 29 18
3	18 (index 4)	45 and 18	7 12 18 29 45

4	29 (index 3)	29 with itself (no change)	7 12 18 29 45
---	--------------	----------------------------	-----------------

Step by step:

- **Pass 1:** the whole array is unsorted. Scanning 29, 12, 45, 7, 18, the smallest is 7 at index 3. Swap it with index 0. The 29 that was at index 0 moves to index 3.
- **Pass 2:** the unsorted portion is 12, 45, 29, 18. The smallest is 12, which is already first. Swapping it with itself changes nothing, but the pass still happens: the algorithm had to scan to find out.
- **Pass 3:** the unsorted portion is 45, 29, 18. The smallest is 18 at index 4. Swap it with 45 at index 2.
- **Pass 4:** the unsorted portion is 29, 45. The smallest is 29, already first.

After pass 4, only one element, 45, is left in the unsorted portion. A single element is always in order with itself, and every smaller value has already been placed in front of it, so it must be the largest. No fifth pass is needed. **An array of n elements takes $n - 1$ passes.**

Selection sort works just as well by selecting the **largest** element instead. Choosing the largest each pass and swapping it to the front sorts into descending order. Here it is by hand on

14 30 9 30 22:

Pass	Largest in unsorted portion	Array after the pass
1	30 (index 1, the first 30 found)	30 14 9 30 22
2	30 (index 3)	30 30 9 14 22
3	22 (index 4)	30 30 22 14 9
4	14 (index 3, already first)	30 30 22 14 9

Two things changed from the first example: the rule (largest instead of smallest) and the data (it has a repeated value). The structure is identical: find, swap, grow the sorted portion by one. In pass 1, two elements tie for largest. Taking the first one found works fine; the second 30 is placed on the very next pass. (You could also select the largest each pass and swap it to the **end** of the array; that builds an ascending list from the back.)

WATCH OUT

Shifting instead of swapping. After finding 7 in 29 12 45 7 18, many students slide everything over to make room at the front, writing 7 29 12 45 18. That is not what selection sort does. Selection sort swaps exactly two elements: 7 and the 29 it trades places with. The correct state after pass 1 is 7 12 45 29 18. The other elements, 12, 45, and 18, do not move at all. Students mix these up because "move the smallest to the front" sounds like inserting. Shifting is the idea behind insertion sort, which you will meet in 4.15.4. When tracing selection sort, ask: which **two** positions trade?

PAUSE AND TRY**Question 3** Multiple choice

Selection sort (smallest first) is applied to 50 20 40 10 30. What does the array look like after **two** passes?

- A. 10 20 40 50 30
- B. 10 20 30 40 50
- C. 10 50 20 40 30
- D. 20 50 40 10 30

Question 4 Short answer

Selection sort (smallest first) is partway through sorting an array of 8 different numbers. It has finished 3 passes. What can you say for certain about the elements at indices 0, 1, and 2? Can any of them move during the remaining passes?

4.15.3 Selection Sort in Java

Doing selection sort by hand, you **saw** the smallest element right away. Java cannot glance at an array; it has to scan element by element, exactly like the find-the-minimum traversal from Topic 4.5. So the code is a loop (one pass per repetition) with a find-the-minimum loop nested inside it, which is the nested loop shape from Topic 2.11.

Here is selection sort as a static method (Topic 3.5), with a `println` at the end of each pass so you can watch it work. The `printArray` helper just prints the elements on one line, separated by spaces.

WORKED EXAMPLE Selection sort, pass by pass

Match each part of the code to a step from 4.15.2: lines 7–14 find the smallest element, and lines 15–17 swap it into place.

SelectionSort.java

```
1 public class SelectionSort
2 {
3     public static void selectionSort(int[] arr)
4     {
5         for (int i = 0; i < arr.length - 1; i++)
6         {
7             int minIndex = i;
8             for (int j = i + 1; j < arr.length; j++)
9             {
10                if (arr[j] < arr[minIndex])
11                {
12                    minIndex = j;
13                }
14            }
15            int temp = arr[i];
16            arr[i] = arr[minIndex];
17            arr[minIndex] = temp;
```

```

18         System.out.print("After pass " + (i + 1) + ": ");
19         printArray(arr);
20     }
21 }
22
23 public static void printArray(int[] arr)
24 {
25     System.out.print(arr[0]);
26     for (int k = 1; k < arr.length; k++)
27     {
28         System.out.print(" " + arr[k]);
29     }
30     System.out.println();
31 }
32
33 public static void main(String[] args)
34 {
35     int[] scores = {29, 12, 45, 7, 18};
36     System.out.print("Start:      ");
37     printArray(scores);
38     selectionSort(scores);
39 }
40 }

```

```

Start:      29 12 45 7 18
After pass 1: 7 12 45 29 18
After pass 2: 7 12 45 29 18
After pass 3: 7 12 18 29 45
After pass 4: 7 12 18 29 45

```

These are exactly the states you found by hand. Now take the code apart:

- **Line 5, the outer loop.** `i` is the index of the first unsorted position, the spot this pass will fill. It runs from 0 while `i < arr.length - 1`, so for 5 elements it takes the values 0, 1, 2, 3: four passes, $n - 1$, because the last element needs no pass of its own.
- **Line 7.** `minIndex` remembers **where** the smallest element seen so far is. It starts at `i`, the first unsorted element, because before looking further, that is the smallest one found. It stores an index, not a value, because the swap needs to know which position to trade with.
- **Lines 8–14, the inner loop.** `j` visits every other unsorted position, from `i + 1` to the end. Whenever `arr[j]` is smaller than the current smallest, `minIndex` moves to `j`. The comparison is `<`, so on a tie the first one found is kept.
- **Lines 15–17, the swap.** This is the three-step swap from Topic 4.5: save `arr[i]` in `temp`, copy the smallest into position `i`, then put the saved value where the smallest used to be. When `minIndex` equals `i`, the swap trades an element with itself, which changes nothing.

Here is the inner loop during pass 1 (`i` is 0), showing how `minIndex` hunts for the smallest element:

j	arr[j]	arr[j] < arr[minIndex]?	minIndex after
—	—	(starts at i)	0 (value 29)
1	12	12 < 29: yes	1 (value 12)

2	45	45 < 12: no	1
3	7	7 < 12: yes	3 (value 7)
4	18	18 < 7: no	3

After the inner loop, `minIndex` is 3, so the swap trades `arr[0]` (29) with `arr[3]` (7).

To select the **largest** element instead, only the comparison changes: `arr[j] > arr[maxIndex]` keeps the position of the biggest value seen so far. (The variable is renamed `maxIndex` so the name says what it holds.) This program runs the descending sort you traced by hand in 4.15.2:

SelectionDescending.java

```

1  public class SelectionDescending
2  {
3      public static void sortHighToLow(int[] arr)
4      {
5          for (int i = 0; i < arr.length - 1; i++)
6          {
7              int maxIndex = i;
8              for (int j = i + 1; j < arr.length; j++)
9              {
10                 if (arr[j] > arr[maxIndex])
11                 {
12                     maxIndex = j;
13                 }
14             }
15             int temp = arr[i];
16             arr[i] = arr[maxIndex];
17             arr[maxIndex] = temp;
18             System.out.print("After pass " + (i + 1) + ": ");
19             printArray(arr);
20         }
21     }
22
23     public static void printArray(int[] arr)
24     {
25         System.out.print(arr[0]);
26         for (int k = 1; k < arr.length; k++)
27         {
28             System.out.print(" " + arr[k]);
29         }
30         System.out.println();
31     }
32
33     public static void main(String[] args)
34     {
35         int[] points = {14, 30, 9, 30, 22};
36         System.out.print("Start: ");
37         printArray(points);
38         sortHighToLow(points);
39     }
40 }
```

```
Start:      14 30 9 30 22
After pass 1: 30 14 9 30 22
After pass 2: 30 30 9 14 22
After pass 3: 30 30 22 14 9
After pass 4: 30 30 22 14 9
```

One character in the comparison decides the order. Everything else, the passes, the inner search, and the swap, is the same.

WATCH OUT

Swapping without a temporary variable. This version replaces the three-line swap with two lines:

SwapNoTemp.java

```
1 public class SwapNoTemp
2 {
3     public static void selectionSort(int[] arr)
4     {
5         for (int i = 0; i < arr.length - 1; i++)
6         {
7             int minIndex = i;
8             for (int j = i + 1; j < arr.length; j++)
9             {
10                if (arr[j] < arr[minIndex])
11                {
12                    minIndex = j;
13                }
14            }
15            arr[i] = arr[minIndex];
16            arr[minIndex] = arr[i];
17        }
18    }
19
20    public static void printArray(int[] arr)
21    {
22        System.out.print(arr[0]);
23        for (int k = 1; k < arr.length; k++)
24        {
25            System.out.print(" " + arr[k]);
26        }
27        System.out.println();
28    }
29
30    public static void main(String[] args)
31    {
32        int[] scores = {29, 12, 45, 7, 18};
33        selectionSort(scores);
34        printArray(scores);
35    }
36 }
```

7 7 7 7 18

In pass 1, line 15 copies 7 into `arr[0]`, and the 29 that was there is gone for good. Line 16 then copies `arr[0]`, which is now 7, into `arr[3]`. Instead of trading places, both positions hold 7. Each pass destroys another value, and the array ends up full of copies. Students write this because "a gets b, b gets a" sounds like a swap in English, but each assignment overwrites a value before the next line can use it. The fix is the `temp` variable from lines 15–17 of `SelectionSort.java`: save one value before overwriting it.

PAUSE AND TRY

Question 5 Tracing

This code runs only the inner loop of pass 1 on the same array, counting how many times `minIndex` changes. What does it print?

```
1 int[] arr = {29, 12, 45, 7, 18};
2 int i = 0;
3 int minIndex = i;
4 int updates = 0;
5 for (int j = i + 1; j < arr.length; j++)
6 {
7     if (arr[j] < arr[minIndex])
8     {
9         minIndex = j;
10        updates++;
11    }
12 }
13 System.out.println("minIndex updated " + updates + " times; minIndex is " +
    minIndex);
```

Question 6 Multiple choice

A student changes the outer loop of `selectionSort` from `i < arr.length - 1` to `i < arr.length`. What happens when the method sorts 29 12 45 7 18?

- A. The array is still sorted correctly; there is one extra pass that changes nothing.
- B. An `ArrayIndexOutOfBoundsException` occurs on the last pass.
- C. The last element is left out of order.
- D. The array ends up in descending order.

4.15.4 Insertion Sort by Hand

Think about how many people sort a hand of playing cards. They pick up cards one at a time. Each new card gets slid into the right spot among the cards already in their hand, so the hand stays in order the whole time. They never search the rest of the deck for the smallest card. That is insertion sort.

Like selection sort, insertion sort keeps a sorted portion at the front. The difference is where the next element comes from. Insertion sort does not hunt for the smallest remaining element. It simply takes the **next** element, the first one in the unsorted portion, and **inserts** it into the right place among the sorted elements.

To make room, the larger sorted elements **shift** one position to the right. This is the shift from Topic 4.5: each element moves over one spot, and the element being inserted drops into the gap.

1. At the start, the first element alone counts as the sorted portion (one element is always in order).
2. Take the first element of the unsorted portion. Call it the current element.
3. Moving left through the sorted portion, shift every element that is larger than the current element one place right.
4. Put the current element into the gap. The sorted portion has grown by one. Repeat.

Here is the key difference from selection sort: the element insertion sort places is in its **correct position within the sorted portion, but not necessarily its final position**. A smaller value that has not been looked at yet may come along later and push it further right.

KEY IDEA

Each pass of insertion sort takes the next element from the unsorted portion and inserts it into its correct (but not necessarily final) place in the sorted portion, shifting larger sorted elements one place right to make room.

WORKED EXAMPLE Insertion sort on the same five scores

Start again with 29 12 45 7 18. At the start, the sorted portion is just 29.

Pass	Element inserted	Sorted elements shifted right	Array after the pass
1	12	29	12 29 45 7 18
2	45	none	12 29 45 7 18
3	7	45, 29, 12	7 12 29 45 18
4	18	45, 29	7 12 18 29 45

- **Pass 1:** 12 is smaller than 29, so 29 shifts right and 12 goes in front.
- **Pass 2:** 45 is bigger than 29, the largest sorted element, so nothing shifts. 45 stays where it is. This pass is very quick.
- **Pass 3:** 7 is smaller than every sorted element, so all three shift right and 7 goes to index 0.
- **Pass 4:** 18 is smaller than 45 and 29, so those two shift. It is bigger than 12, so the shifting stops and 18 goes right after 12.

Like selection sort, five elements take four passes: the first element starts out sorted, and each pass adds one.

Follow the 29. After pass 1 it sits at index 1, in the correct place **among the sorted elements so far**. After pass 3 it has been pushed to index 2, and it finishes at index 3. In selection sort, once an element joined the sorted portion, it never moved again. In insertion sort, sorted elements keep moving right whenever a smaller value is inserted in front of them.

Now zoom in on pass 3 to see the shifting one step at a time. Before the pass, the array is 12 29 45 7 18, and 7 is the current element. Java cannot shift everything at once; each step copies one element one place to the right:

```
start:           12 29 45  7 18    current = 7 (saved)
shift 45 right:  12 29 45 45 18
shift 29 right:  12 29 29 45 18
shift 12 right:  12 12 29 45 18
insert 7:        7 12 29 45 18
```

In the middle steps, one value appears twice. That is expected: shifting copies a value into the next spot, and the old copy is overwritten on the following step (or by the inserted element at the end). The 7 is not lost because it was saved before the shifting started; its spot at index 3 is the first one overwritten. You will see why that saving step matters in the code.

WATCH OUT

Thinking each inserted element is in its final position. After pass 1 of the worked example, a student might say "12 and 29 are done." Only the first part is guaranteed: 12 and 29 are in order **relative to each other**. The smaller 7 had not been looked at yet, and it ended up in front of both. Students make this mistake because selection sort really does finalize each element, and both sorts grow a sorted portion at the front. The difference: selection sort's sorted portion holds the smallest values of the **whole** array; insertion sort's sorted portion holds the elements that **started** at the front, rearranged into order.

PAUSE AND TRY

Question 7 Multiple choice

Insertion sort is applied to 40 10 30 20 50. What does the array look like after **two** passes?

- A. 10 30 40 20 50
- B. 10 20 30 40 50
- C. 10 40 30 20 50
- D. 10 20 40 30 50

Question 8 Short answer

Partway through insertion sort, the sorted portion is 8 15 23 and the next element to insert is 11. Which elements shift, and what does the sorted portion look like after this pass? Explain why the shifting stops where it does.

4.15.5 Insertion Sort in Java

By hand, you could see where the current element belonged and slide everything over at once. Java has to work its way left through the sorted portion one element at a time, deciding at each step whether to shift or stop. Because the number of shifts is not known in advance (zero in pass 2, three in pass 3), the inner loop is a `while` loop from Topic 2.7 rather than a counting `for` loop.

WORKED EXAMPLE Insertion sort, pass by pass

Match each part to a step from 4.15.4: line 7 saves the current element, lines 9–13 shift, and line 14 inserts.

InsertionSort.java

```
1 public class InsertionSort
2 {
3     public static void insertionSort(int[] arr)
4     {
5         for (int i = 1; i < arr.length; i++)
6         {
7             int current = arr[i];
8             int j = i - 1;
9             while (j >= 0 && arr[j] > current)
10            {
11                arr[j + 1] = arr[j];
12                j--;
13            }
14            arr[j + 1] = current;
15            System.out.print("After pass " + i + ": ");
16            printArray(arr);
17        }
18    }
19
20    public static void printArray(int[] arr)
21    {
22        System.out.print(arr[0]);
23        for (int k = 1; k < arr.length; k++)
24        {
25            System.out.print(" " + arr[k]);
26        }
27        System.out.println();
28    }
29
30    public static void main(String[] args)
31    {
32        int[] scores = {29, 12, 45, 7, 18};
33        System.out.print("Start: ");
34        printArray(scores);
35        insertionSort(scores);
36    }
37 }
```

```
Start:      29 12 45 7 18
After pass 1: 12 29 45 7 18
After pass 2: 12 29 45 7 18
After pass 3: 7 12 29 45 18
After pass 4: 7 12 18 29 45
```

Taking it apart:

- **Line 5, the outer loop.** `i` is the index of the current element, the first unsorted one. It starts at 1, not 0, because the element at index 0 alone is already a sorted portion. It runs to the last index, so 5 elements give 4 passes.
- **Line 7.** `current` saves a copy of `arr[i]`. This matters: the first shift writes over `arr[i]`, so without the copy the value being inserted would be lost.
- **Line 8.** `j` starts at `i - 1`, the last element of the sorted portion, and moves left.
- **Line 9, the condition.** Keep shifting while **both** parts are true: `j >= 0` (there is still a sorted element to look at) and `arr[j] > current` (that element is bigger, so it must move right).
- **Lines 11–12.** Copy `arr[j]` one place right into `arr[j + 1]`, then step `j` left.
- **Line 14.** When the loop stops, `j` is just left of the gap, so the current element goes into `arr[j + 1]`.

The loop on line 9 can stop for two different reasons. Compare pass 3 and pass 4.

Pass 3 (`i` is 3, `current` is 7, array 12 29 45 7 18):

<code>j</code>	<code>j >= 0</code>	<code>arr[j] > current</code>	Action
2	true	45 > 7: true	shift 45 to index 3; <code>j</code> becomes 1
1	true	29 > 7: true	shift 29 to index 2; <code>j</code> becomes 0
0	true	12 > 7: true	shift 12 to index 1; <code>j</code> becomes -1
-1	false	not checked	stop; <code>arr[0] = 7</code>

Pass 4 (`i` is 4, `current` is 18, array 7 12 29 45 18):

<code>j</code>	<code>j >= 0</code>	<code>arr[j] > current</code>	Action
3	true	45 > 18: true	shift 45 to index 4; <code>j</code> becomes 2
2	true	29 > 18: true	shift 29 to index 3; <code>j</code> becomes 1
1	true	12 > 18: false	stop; <code>arr[2] = 18</code>

In pass 3, the current element was smaller than everything, so the loop ran off the left end: `j` reached -1 and `j >= 0` stopped it. In pass 4, the loop found a smaller element (12) and stopped there. Notice "not checked" in the last row of pass 3. Because `&&` short-circuits (Topic 2.5), when `j >= 0` is false, Java never evaluates `arr[-1] > current`. That is what keeps the code safe.

KEY IDEA

Insertion sort's inner loop shifts while `j >= 0` and `arr[j] > current`. It stops either at the front of the array or at the first element that is not larger, and the saved element goes into `arr[j + 1]`.

WATCH OUT

Writing the two conditions in the wrong order. This version swaps the two sides of `&&`:

InsertionAndOrder.java

```
1 public class InsertionAndOrder
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = {29, 12, 45, 7, 18};
6         for (int i = 1; i < scores.length; i++)
7         {
8             int current = scores[i];
9             int j = i - 1;
10            while (scores[j] > current && j >= 0)
11            {
12                scores[j + 1] = scores[j];
13                j--;
14            }
15            scores[j + 1] = current;
16        }
17        System.out.println("Sorted!");
18    }
19 }
```

It crashes with a message like this (wording can vary):

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 5
    at InsertionAndOrder.main(InsertionAndOrder.java:10)
```

In pass 1, 29 shifts right and `j` becomes `-1`. Then line 10 checks `scores[-1] > current` **first**, before it ever gets to `j >= 0`. Index `-1` does not exist, so Java throws the exception, and `Sorted!` never prints. Students write it this way because `&&` means "and," and in everyday logic "A and B" is the same as "B and A." In Java the order matters: the left side is checked first, and the right side is skipped when the left is false. The fix is to put the bounds check first: `while (j >= 0 && scores[j] > current)`.

PAUSE AND TRY

Question 9 Debugging

This insertion sort leaves out the `current` variable and uses `arr[i]` directly. It prints
29 29 45 45 45 (one per line) instead of the sorted scores. Find the error and correct it.

InsertionLostValue.java

```

1 public class InsertionLostValue
2 {
3     public static void insertionSort(int[] arr)
4     {
5         for (int i = 1; i < arr.length; i++)
6         {
7             int j = i - 1;
8             while (j >= 0 && arr[j] > arr[i])
9             {
10                arr[j + 1] = arr[j];
11                j--;
12            }
13            arr[j + 1] = arr[i];
14        }
15    }
16
17    public static void main(String[] args)
18    {
19        int[] scores = {29, 12, 45, 7, 18};
20        insertionSort(scores);
21        for (int value : scores)
22        {
23            System.out.println(value);
24        }
25    }
26 }

```

Question 10 Multiple choice

Which change to line 9 of `InsertionSort.java` makes it sort into **descending** order (largest first)?

- A. `while (j >= 0 && arr[j] < current)`
- B. `while (j >= 0 && arr[j] >= current)`
- C. `while (j < 0 && arr[j] > current)`
- D. `while (j >= 0 || arr[j] > current)`

4.15.6 Sorting an ArrayList

Both algorithms work on an `ArrayList` as well as an array, and the steps are exactly the same. Only the syntax for reading and writing elements changes, as you saw when you moved array algorithms over to `ArrayList` in Topic 4.10:

Array	ArrayList
<code>arr.length</code>	<code>list.size()</code>
<code>arr[j]</code>	<code>list.get(j)</code>
<code>arr[j] = x;</code>	<code>list.set(j, x);</code>

Sorting an `ArrayList` uses `set` to put values into positions that already exist. It does not use `add` or `remove`, because the list never changes size: elements only trade or shift places.

A second change comes from the element type. An `ArrayList<Integer>` holds `Integer` objects, but `list.get(j) < list.get(minIndex)` still works because each `Integer` unboxes to an `int` before comparing (Topic 4.7). An `ArrayList<String>` is different: `<` and `>` do not work on `String` objects at all. To compare strings in dictionary order, use `compareTo` from Topic 1.15. The test "a comes after b" is written `a.compareTo(b) > 0`.

WORKED EXAMPLE Selection sort on lap times, insertion sort on names

`selectionSort` is the array version with `get`, `set`, and `size` swapped in. `insertionSort` sorts `String` values, so line 29 uses `compareTo` where the array version used `>`.

SortArrayList.java

```

1  import java.util.ArrayList;
2
3  public class SortArrayList
4  {
5      public static void selectionSort(ArrayList<Integer> list)
6      {
7          for (int i = 0; i < list.size() - 1; i++)
8          {
9              int minIndex = i;
10             for (int j = i + 1; j < list.size(); j++)
11             {
12                 if (list.get(j) < list.get(minIndex))
13                 {
14                     minIndex = j;
15                 }
16             }
17             int temp = list.get(i);
18             list.set(i, list.get(minIndex));
19             list.set(minIndex, temp);
20         }
21     }
22
23     public static void insertionSort(ArrayList<String> list)
24     {
25         for (int i = 1; i < list.size(); i++)
26         {
27             String current = list.get(i);
28             int j = i - 1;
29             while (j >= 0 && list.get(j).compareTo(current) > 0)
30             {
31                 list.set(j + 1, list.get(j));
32                 j--;
33             }
34             list.set(j + 1, current);
35             System.out.println("After pass " + i + ": " + list);
36         }
37     }
38
39     public static void main(String[] args)

```

```
40     {
41         ArrayList<Integer> laps = new ArrayList<Integer>();
42         laps.add(62);
43         laps.add(58);
44         laps.add(71);
45         laps.add(55);
46         selectionSort(laps);
47         System.out.println("Laps: " + laps);
48
49         ArrayList<String> names = new ArrayList<String>();
50         names.add("Mia");
51         names.add("Ava");
52         names.add("Zoe");
53         names.add("Eli");
54         insertionSort(names);
55     }
56 }
```

Laps: [55, 58, 62, 71]

After pass 1: [Ava, Mia, Zoe, Eli]

After pass 2: [Ava, Mia, Zoe, Eli]

After pass 3: [Ava, Eli, Mia, Zoe]

- **Swapping with get and set (lines 17–19).** The three-step swap is the same: save `list.get(i)` in `temp`, copy the smallest into position `i` with `set`, then `set` the saved value where the smallest was.
- **Pass 1 of the names.** current is "Ava". `"Mia".compareTo("Ava") > 0` is true (M comes after A), so "Mia" shifts right and "Ava" goes to index 0.
- **Pass 2.** current is "Zoe". `"Mia".compareTo("Zoe")` is negative, so nothing shifts.
- **Pass 3.** current is "Eli". "Zoe" and "Mia" both come after it and shift right. `"Ava".compareTo("Eli")` is negative, so the shifting stops and "Eli" goes to index 1.

WATCH OUT

Comparing String values with >. Copying the array version of insertion sort onto an `ArrayList<String>` without changing the comparison does not compile:

StringLessThan.java

```

1  import java.util.ArrayList;
2
3  public class StringLessThan
4  {
5      public static void insertionSort(ArrayList<String> list)
6      {
7          for (int i = 1; i < list.size(); i++)
8          {
9              String current = list.get(i);
10             int j = i - 1;
11             while (j >= 0 && list.get(j) > current)
12             {
13                 list.set(j + 1, list.get(j));
14                 j--;
15             }
16             list.set(j + 1, current);
17         }
18     }
19 }

```

```

StringLessThan.java:11: error: bad operand types for binary operator '>'
        while (j >= 0 && list.get(j) > current)
                                   ^

```

```

first type: String
second type: String

```

The relational operators work only on numbers (and on `Integer` values, by unboxing). A `String` is an object, so Java has no meaning for `>` between two of them. Students make this mistake because "comes later alphabetically" feels like "greater than." The fix is `list.get(j).compareTo(current) > 0`, as on line 29 of `SortArrayList.java`.

PAUSE AND TRY

Question 11 Multiple choice

An insertion sort for an `ArrayList<String>` named `words` should put the words in dictionary order. Which condition correctly replaces `arr[j] > current` from the array version?

- A. `words.get(j).compareTo(current) > 0`
- B. `words.get(j) > current`
- C. `words[j].compareTo(current) > 0`
- D. `words.get(j).compareTo(current) < 0`

4.15.7 Telling the Two Sorts Apart

You now know two algorithms that end with the same sorted list. A common exam question shows a list partway through a sort and asks which algorithm produced it. Another asks which one does more work. Both questions come down to what each pass does.

Question	Selection sort	Insertion sort
What does each pass do?	Finds the smallest unsorted element and swaps it	Takes the next element and shifts larger ones to insert it
Which part does the pass search?	The unsorted portion	The sorted portion
Is the placed element final?	Yes	Not necessarily
After k passes, the front k (or k + 1) elements are...	The k smallest values of the whole list	The first k + 1 original elements, put in order
Does the work depend on the data?	No: it always scans the whole unsorted portion	Yes: a nearly sorted list needs very few shifts

The fourth row is the fastest way to identify a sort. After pass k of selection sort, the front holds the smallest values **in the whole list**. After pass k of insertion sort, the front holds whatever values **started** in the first k + 1 positions, now in order, and the rest of the list is untouched.

WORKED EXAMPLE Which sort made this state?

The array 50 20 40 10 30 is being sorted. After one pass, the result is 10 20 40 50 30. After one pass of the other algorithm, it would be 20 50 40 10 30. Which is which?

- 10 20 40 50 30: the smallest value in the whole array, 10, is already at the front, and 50 jumped from index 0 to index 3. That is a swap of the minimum into place: **selection sort**.
- 20 50 40 10 30: only the first two original elements, 50 and 20, have been put in order, and indices 2–4 are unchanged. That is **insertion sort**, after inserting 20.

Both sorts take 4 passes on this array. In this case insertion sort's in-between states look less sorted, but that tells you nothing about which algorithm is faster. To compare work, count, as in Topic 2.12.

The program below counts the main work of each sort on three 5-element arrays: how many times selection sort compares two elements (the `if` inside its inner loop), and how many times insertion sort shifts an element (the body of its `while` loop). Each sort gets its own copy of the data, because sorting changes the array.

CountWork.java

```

1 public class CountWork
2 {
3     public static int selectionComparisons(int[] arr)
4     {
5         int count = 0;
6         for (int i = 0; i < arr.length - 1; i++)
7         {
8             int minIndex = i;
9             for (int j = i + 1; j < arr.length; j++)
10            {
11                count++;
12                if (arr[j] < arr[minIndex])
13                {
14                    minIndex = j;
15                }
16            }
17        }
18    }
19 }

```

```

17         int temp = arr[i];
18         arr[i] = arr[minIndex];
19         arr[minIndex] = temp;
20     }
21     return count;
22 }
23
24 public static int insertionShifts(int[] arr)
25 {
26     int count = 0;
27     for (int i = 1; i < arr.length; i++)
28     {
29         int current = arr[i];
30         int j = i - 1;
31         while (j >= 0 && arr[j] > current)
32         {
33             count++;
34             arr[j + 1] = arr[j];
35             j--;
36         }
37         arr[j + 1] = current;
38     }
39     return count;
40 }
41
42 public static void report(String label, int[] forSelection, int[] forInsertion)
43 {
44     System.out.println(label + " selection comparisons " +
45 selectionComparisons(forSelection)
46     + ", insertion shifts " + insertionShifts(forInsertion));
47 }
48
49 public static void main(String[] args)
50 {
51     int[] mixedA = {29, 12, 45, 7, 18};
52     int[] mixedB = {29, 12, 45, 7, 18};
53     int[] sortedA = {7, 12, 18, 29, 45};
54     int[] sortedB = {7, 12, 18, 29, 45};
55     int[] reverseA = {45, 29, 18, 12, 7};
56     int[] reverseB = {45, 29, 18, 12, 7};
57     report("Mixed: ", mixedA, mixedB);
58     report("Sorted: ", sortedA, sortedB);
59     report("Reverse:", reverseA, reverseB);
60 }

```

```

Mixed:   selection comparisons 10, insertion shifts 6
Sorted:  selection comparisons 10, insertion shifts 0
Reverse: selection comparisons 10, insertion shifts 10

```

Selection sort makes 10 comparisons no matter what: its inner loop runs $4 + 3 + 2 + 1 = 10$ times, the shrinking nested loop you counted in Topic 2.12. Even on an array that is already sorted, it still has to scan to **prove** each element is the smallest. Insertion sort's work changes with the data. On the sorted array, every current element is already larger than its left neighbor, so the `while`

condition is false right away and nothing shifts. On the reversed array, every element must shift all the way to the front: $1 + 2 + 3 + 4 = 10$ shifts. The mixed array falls in between: $1 + 0 + 3 + 2 = 6$.

KEY IDEA

Selection sort always does the same amount of comparing for a list of a given size. Insertion sort does very little work on a list that is already nearly sorted and the most work on a list in reverse order.

WATCH OUT

Judging the algorithm by how sorted the list looks. Students often guess "selection sort" for whichever state looks more sorted. That works sometimes, but not reliably. Look for the evidence instead: are the front values the smallest in the **whole** list (selection), or are they the original front values rearranged, with the back of the list untouched (insertion)? Did two elements trade places (selection), or did a block slide right (insertion)?

PAUSE AND TRY

Question 12 Multiple choice

The array `6 2 9 1 5` is being sorted in ascending order. After two passes, it is `1 2 9 6 5`. Which algorithm could have produced this?

- A. Selection sort only
- B. Insertion sort only
- C. Either selection sort or insertion sort
- D. Neither selection sort nor insertion sort

Question 13 Short answer

Both sorts are run on an array of 6 elements that is **already** in ascending order. How many comparisons does selection sort make, and how many shifts does insertion sort make? Explain the difference.

Putting it together

A race organizer has five finishing times in seconds, including a tie, and wants them in order so the top three can be announced. This program sorts one copy with selection sort and another with insertion sort, printing every pass, then reads the top three from the sorted array. Predict both sets of states before reading the output.

WORKED EXAMPLE Race results, sorted two ways

RaceResults.java

```
1 public class RaceResults
2 {
3     public static void selectionSort(int[] arr)
4     {
```

```
5         for (int i = 0; i < arr.length - 1; i++)
6         {
7             int minIndex = i;
8             for (int j = i + 1; j < arr.length; j++)
9             {
10                 if (arr[j] < arr[minIndex])
11                 {
12                     minIndex = j;
13                 }
14             }
15             int temp = arr[i];
16             arr[i] = arr[minIndex];
17             arr[minIndex] = temp;
18             System.out.print("  pass " + (i + 1) + ": ");
19             printArray(arr);
20         }
21     }
22
23     public static void insertionSort(int[] arr)
24     {
25         for (int i = 1; i < arr.length; i++)
26         {
27             int current = arr[i];
28             int j = i - 1;
29             while (j >= 0 && arr[j] > current)
30             {
31                 arr[j + 1] = arr[j];
32                 j--;
33             }
34             arr[j + 1] = current;
35             System.out.print("  pass " + i + ": ");
36             printArray(arr);
37         }
38     }
39
40     public static void printArray(int[] arr)
41     {
42         System.out.print(arr[0]);
43         for (int k = 1; k < arr.length; k++)
44         {
45             System.out.print(" " + arr[k]);
46         }
47         System.out.println();
48     }
49
50     public static void main(String[] args)
51     {
52         int[] timesA = {61, 48, 55, 48, 52};
53         int[] timesB = {61, 48, 55, 48, 52};
54         System.out.println("Selection sort:");
55         selectionSort(timesA);
56         System.out.println("Insertion sort:");
57         insertionSort(timesB);
```

```

58         System.out.println("Top three times: " + timesB[0] + ", " + timesB[1] + ",
59         " + timesB[2]);
60     }

```

Selection sort:

```

pass 1: 48 61 55 48 52
pass 2: 48 48 55 61 52
pass 3: 48 48 52 61 55
pass 4: 48 48 52 55 61

```

Insertion sort:

```

pass 1: 48 61 55 48 52
pass 2: 48 55 61 48 52
pass 3: 48 48 55 61 52
pass 4: 48 48 52 55 61

```

Top three times: 48, 48, 52

Selection sort, starting from 61 48 55 48 52:

- Pass 1: the smallest is 48. There are two, at indices 1 and 3; the `<` comparison keeps the **first** one found, index 1. Swap with 61: 48 61 55 48 52.
- Pass 2: in 61 55 48 52, the smallest is the other 48, at index 3. Swap with 61 at index 1: 48 48 55 61 52.
- Pass 3: in 55 61 52, the smallest is 52 at index 4. Swap with 55: 48 48 52 61 55.
- Pass 4: in 61 55, the smallest is 55. Swap: 48 48 52 55 61.

Insertion sort, from the same start:

- Pass 1: insert 48; 61 shifts. 48 61 55 48 52. (Same as selection sort's pass 1, by coincidence: inserting the second element and selecting the minimum happen to agree here.)
- Pass 2: insert 55; 61 shifts, 48 does not. 48 55 61 48 52. Now the two algorithms differ: selection sort already had both 48s in front.
- Pass 3: insert the second 48; 61 and 55 shift. The check `48 > 48` is false, so shifting stops and the new 48 goes right after the old one: 48 48 55 61 52.
- Pass 4: insert 52; 61 and 55 shift, 48 does not. 48 48 52 55 61.

Both sorts finish with the same array, four passes each. Selection sort placed a final value on every pass. Insertion sort's values kept moving: 61 was at index 1 after pass 1 and ended at index 4. The tied 48s caused no trouble for either algorithm; equal values simply end up next to each other. Finally, once the array is sorted, the top three times are simply `timesB[0]`, `timesB[1]`, and `timesB[2]`, one reason sorting is so useful.

Vocabulary

Term	Definition	Example
sorted	Arranged in order by some rule	3 7 7 12 is sorted in ascending order
ascending order	Each element is less than or equal to the next (smallest first)	2 4 4 8

descending order	Each element is greater than or equal to the next (largest first)	20 12 7 3
sorting algorithm	A step-by-step process that rearranges a collection into sorted order	Selection sort, insertion sort
iterative	Using loops to repeat the steps	Both sorts use nested loops
sorted portion	The front part of the list that is already in order during a sort	7 12 in 7 12 45 29 18
unsorted portion	The part of the list the sort has not dealt with yet	45 29 18 in 7 12 45 29 18
pass	One repetition of a sort's outer loop, which adds one element to the sorted portion	5 elements take 4 passes
selection sort	Each pass finds the smallest (or largest) element not yet placed and trades it into the next open spot, where it stays	29 12 45 7 18 → 7 12 45 29 18 after pass 1
insertion sort	Repeatedly inserts the next unsorted element into its correct (not necessarily final) place in the sorted portion by shifting larger elements right	29 12 45 7 18 → 12 29 45 7 18 after pass 1
swap	Two elements trade places, using a temporary variable	<code>temp = a[i]; a[i] = a[m]; a[m] = temp;</code>
shift	Copying an element one position over to make room	<code>arr[j + 1] = arr[j];</code>
final position	The position an element will occupy when the whole list is sorted	Selection sort places each element there

Chapter summary

- A list is sorted in ascending order when every element is less than or equal to the one after it; checking each neighboring pair is enough to tell.
- Selection sort and insertion sort are iterative sorts for arrays and `ArrayList`s. Both grow a sorted portion at the front by one element per pass, so n elements take $n - 1$ passes.
- Selection sort scans the unsorted portion for the smallest (or largest) element and swaps it into the first unsorted position, which is its final position. Its code is a find-the-minimum loop inside a pass loop, followed by a three-step swap.
- Insertion sort saves the next unsorted element, shifts larger sorted elements one place right, and drops the saved element into the gap. Its place is correct among the sorted elements but not necessarily final. The `while` condition must check `j >= 0` before `arr[j]`.
- On an `ArrayList`, use `get`, `set`, and `size`; compare `String` values with `compareTo`. To tell the sorts apart, check whether the front holds the smallest values overall (selection) or the original front values in order (insertion). Selection sort's comparisons never change with the data; insertion sort's shifts do.

Review questions

Question 14 Multiple choice

Insertion sort has completed 3 passes on an array of 7 different numbers. Which statement **must** be true?

- A. The elements at indices 0–3 are in ascending order, and they are the same four values that started at indices 0–3.
- B. The elements at indices 0–3 are the four smallest values in the array.
- C. The elements at indices 0–2 are in their final positions.
- D. The elements at indices 4–6 are in ascending order.

Question 15 Tracing

This code runs selection sort on an `ArrayList` of fruit names. What does it print?

SelectionWords.java

```
1 import java.util.ArrayList;
2
3 public class SelectionWords
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> fruits = new ArrayList<String>();
8         fruits.add("pear");
9         fruits.add("fig");
10        fruits.add("kiwi");
11        fruits.add("apple");
12        fruits.add("date");
13        for (int i = 0; i < fruits.size() - 1; i++)
14        {
15            int minIndex = i;
16            for (int j = i + 1; j < fruits.size(); j++)
17            {
18                if (fruits.get(j).compareTo(fruits.get(minIndex)) < 0)
19                {
20                    minIndex = j;
21                }
22            }
23            String temp = fruits.get(i);
24            fruits.set(i, fruits.get(minIndex));
25            fruits.set(minIndex, temp);
26            System.out.println(fruits);
27        }
28    }
29 }
```

Question 16 Debugging

This selection sort should print 7 12 18 29 45, but it prints 7 18 12 45 29. Find the error and correct it.

MinIndexBug.java

```
1 public class MinIndexBug
2 {
3     public static void main(String[] args)
4     {
5         int[] arr = {29, 12, 45, 7, 18};
6         int minIndex = 0;
7         for (int i = 0; i < arr.length - 1; i++)
8         {
9             for (int j = i + 1; j < arr.length; j++)
10            {
11                if (arr[j] < arr[minIndex])
12                {
13                    minIndex = j;
14                }
15            }
16            int temp = arr[i];
17            arr[i] = arr[minIndex];
18            arr[minIndex] = temp;
19        }
20        for (int value : arr)
21        {
22            System.out.print(value + " ");
23        }
24        System.out.println();
25    }
26 }
```

Question 17 Multiple choice

This code counts the shifts insertion sort makes. What does it print?

```
1 int[] arr = {3, 1, 4, 2};
2 int shifts = 0;
3 for (int i = 1; i < arr.length; i++)
4 {
5     int current = arr[i];
6     int j = i - 1;
7     while (j >= 0 && arr[j] > current)
8     {
9         arr[j + 1] = arr[j];
10        shifts++;
11        j--;
12    }
13    arr[j + 1] = current;
14 }
15 System.out.println(shifts);
```

- A. 3
- B. 2
- C. 6
- D. 4

Question 18 Tracing

This code is insertion sort with one change: the outer loop stops at `i < 3`. What does it print?

```
1 int[] arr = {9, 4, 7, 1, 5};
2 for (int i = 1; i < 3; i++)
3 {
4     int current = arr[i];
5     int j = i - 1;
6     while (j >= 0 && arr[j] > current)
7     {
8         arr[j + 1] = arr[j];
9         j--;
10    }
11    arr[j + 1] = current;
12 }
13 for (int value : arr)
14 {
15     System.out.print(value + " ");
16 }
17 System.out.println();
```

Question 19 Multiple choice

Selection sort makes 45 comparisons when it sorts a list of 10 elements. About how many comparisons does it make for a list of 20 elements?

- A. 190
- B. 90
- C. 400
- D. 65

Question 20 Short answer

Selection sort's outer loop is `for (int i = 0; i < arr.length - 1; i++)`, and insertion sort's is `for (int i = 1; i < arr.length; i++)`. Both make $n - 1$ passes on n elements. Explain why selection sort can skip the **last** element and insertion sort can skip the **first**.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can explain what ascending order means and check whether an array is sorted by comparing neighbors. Reread: 4.15.1 What It Means to Be Sorted
- ☐ I can carry out selection sort by hand and write the list after each pass. Reread: 4.15.2 Selection Sort by Hand
- ☐ I can trace the selection sort code, including how `minIndex` changes and how the swap works. Reread: 4.15.3 Selection Sort in Java
- ☐ I can carry out insertion sort by hand and show which elements shift on each pass. Reread: 4.15.4 Insertion Sort by Hand
- ☐ I can trace the insertion sort code and explain why `j >= 0` must come first in the while condition. Reread: 4.15.5 Insertion Sort in Java

☐ I can follow both sorts on an ArrayList, including comparing String values with compareTo.

Reread: 4.15.6 Sorting an ArrayList

☐ I can tell which sort produced a given in-between state and compare how much work each sort does. Reread: 4.15.7 Telling the Two Sorts Apart

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

A. Every element is less than or equal to the one after it: $2 \leq 4$, $4 \leq 4$, $4 \leq 8$. Repeated values are allowed in ascending order.

- **B** is wrong: it is sorted, but in **descending** order, largest first.
- **C** is wrong: the pair (3, 2) is out of order. Most of the array looks sorted, but one bad pair is enough to make it unsorted.
- **D** is wrong: the last pair (9, 8) is out of order. Checking only the first few elements would miss it.

Reread 4.15.1 *What It Means to Be Sorted* if you chose differently.

Question 2

2

The loop checks the four neighboring pairs:

i	Pair	heights[i] > heights[i + 1]?	outOfOrder
0	(4, 9)	no	0
1	(9, 6)	yes	1
2	(6, 8)	no	1
3	(8, 2)	yes	2

Two pairs are out of order. The loop stops after `i` is 3 because `i < heights.length - 1` means `i < 4`.

Reread 4.15.1 *What It Means to Be Sorted* if your answer differed.

Question 3

A. Pass 1: the smallest is 10 at index 3; swap it with 50 at index 0, giving 10 20 40 50 30. Pass 2: the unsorted portion is 20, 40, 50, 30; the smallest is 20, already first, so it swaps with itself and nothing changes.

- **B** is wrong: that is the fully sorted array. Two passes place only the two smallest values; selection sort needs four passes for five elements.
- **C** is wrong: it shifts 50, 20, and 40 right to make room for 10. Selection sort swaps 10 with 50 and leaves the others alone.
- **D** is wrong: it only puts the first two elements in order. That is the state after one pass of **insertion** sort, which you will meet in 4.15.4.

Reread 4.15.2 *Selection Sort by Hand* if you chose differently.

Question 4

Model answer: Indices 0, 1, and 2 hold the three smallest values in the whole array, in ascending order. Each pass selected the smallest value left and swapped it into the next position, so pass 1

placed the smallest overall, pass 2 the second smallest, and pass 3 the third smallest. They are in their final positions: every later pass only searches the unsorted portion (indices 3 to 7) and swaps within it, so these three never move again.

A complete answer must:

- say they are the three smallest values, in order,
- say they are in their final positions, and
- explain that later passes only search and swap inside the unsorted portion.

Reread 4.15.2 *Selection Sort by Hand* if your answer differed.

Question 5

```
minIndex updated 2 times; minIndex is 3
```

`minIndex` starts at 0 (value 29). When `j` is 1, `12 < 29`, so `minIndex` becomes 1 (first update). When `j` is 2, `45 < 12` is false. When `j` is 3, `7 < 12`, so `minIndex` becomes 3 (second update). When `j` is 4, `18 < 7` is false. Two updates, ending at index 3, the position of 7.

Reread the inner-loop trace in 4.15.3 *Selection Sort in Java* if your answer differed.

Question 6

A. On the extra pass, `i` is 4. `minIndex` starts at 4, and the inner loop starts at `j = 5`, which already fails `j < arr.length`, so the inner loop does not run at all. The swap trades `arr[4]` with itself. The array still ends as 7 12 18 29 45. The extra pass is harmless but wasted, which is why the standard code stops at `arr.length - 1`.

- **B** is wrong: the inner loop's condition is checked before `arr[5]` is ever used, so no bad index is accessed.
- **C** is wrong: the last element was already correct after pass 4; the extra pass does not disturb it.
- **D** is wrong: the order depends on the comparison `<`, which did not change.

Reread 4.15.3 *Selection Sort in Java* if you chose differently.

Question 7

A. Pass 1 inserts 10: 40 shifts right, giving 10 40 30 20 50. Pass 2 inserts 30: 40 shifts right, 10 does not, so 30 goes between them, giving 10 30 40 20 50. The 20 and 50 have not been touched yet.

- **B** is wrong: that is what **selection** sort produces after two passes (it selects 10, then 20). Insertion sort only works with the elements it has reached, and 20 has not been reached.
- **C** is wrong: that is the state after only one pass.
- **D** is wrong: it treats the passes as swaps. Insertion sort shifts elements; it does not trade 20 with 40.

Reread 4.15.4 *Insertion Sort by Hand* if you chose differently.

Question 8

Model answer: 23 shifts right, then 15 shifts right. 8 does not shift, because 8 is not larger than 11. 11 goes into the gap after 8, so the sorted portion becomes 8 11 15 23. Shifting stops at the first sorted element (moving left) that is not larger than the current element, because everything to its left is smaller still.

A complete answer must:

- say 23 and 15 shift (two elements) and 8 does not,
- give 8 11 15 23, and
- explain that shifting stops at the first element that is not larger than 11.

Reread 4.15.4 *Insertion Sort by Hand* if your answer differed.

Question 9

The value being inserted is overwritten by the first shift. In pass 1 (*i* is 1), the loop compares `arr[0]` (29) with `arr[1]` (12), so it shifts: line 10 copies 29 into `arr[1]`. Now `arr[i]` is 29, not 12. The 12 is gone. Line 13 then puts `arr[i]`, which is 29, into `arr[0]`, giving 29 29 45 7 18. Later passes lose 7 and 18 the same way, ending with 29 29 45 45 45. The program compiles and runs; only the wrong output reveals the bug.

Corrected code: save the element before any shifting, and use the saved copy in the condition and the insert:

```
1 int current = arr[i];
2 int j = i - 1;
3 while (j >= 0 && arr[j] > current)
4 {
5     arr[j + 1] = arr[j];
6     j--;
7 }
8 arr[j + 1] = current;
```

With this change the method matches `InsertionSort.java` and prints 7, 12, 18, 29, 45.

Reread 4.15.5 *Insertion Sort in Java* (the note on line 7) if you missed it.

Question 10

A. For descending order, an element should shift right when it is **smaller** than the current element, so larger values stay in front. With this change the array 29 12 45 7 18 sorts to 45, 29, 18, 12, 7.

- **B** is wrong: it still sorts ascending. It also shifts elements equal to `current`, which just does extra work.
- **C** is wrong: `j` starts at `i - 1`, which is at least 0, so `j < 0` is false at once and nothing ever shifts; the array stays unsorted.
- **D** is wrong: with `||`, whenever `j >= 0` is true the whole condition is true without comparing anything, so the loop shifts blindly. Once `j` becomes `-1`, the left side is false, so Java evaluates `arr[-1]` and throws an `ArrayIndexOutOfBoundsException` during the very first pass.

Reread 4.15.5 *Insertion Sort in Java* if you chose differently.

Question 11

A. An element should shift right when it comes **after** `current` in dictionary order, and `compareTo` returns a positive number in exactly that case.

- **B** is wrong: `>` cannot compare two `String` objects. It does not compile (bad operand types for binary operator '`>`').
- **C** is wrong: square brackets work only on arrays. On an `ArrayList`, Java reports `array required, but ArrayList<String> found. Use get(j).`
- **D** is wrong: it compiles, but it shifts elements that come **before** `current`, which sorts the words into reverse dictionary order.

Reread 4.15.6 *Sorting an ArrayList* if you chose differently.

Question 12

A. Selection sort: pass 1 swaps the smallest value, 1, with 6, giving 1 2 9 6 5; pass 2 finds 2 already in place. That matches. Insertion sort after two passes would only have put the first three original values, 6, 2, and 9, in order: 2 6 9 1 5. It could not have moved 1 to the front yet, because it has not reached index 3.

- **B** is wrong: insertion sort only rearranges the elements it has reached. The 1 at index 3 is not reached until pass 3.
- **C** is wrong: the two algorithms give different states here, as shown above.
- **D** is wrong: the state matches selection sort exactly.

Reread 4.15.7 *Telling the Two Sorts Apart* if you chose differently.

Question 13

Model answer: Selection sort makes $5 + 4 + 3 + 2 + 1 = 15$ comparisons, and insertion sort makes 0 shifts. Selection sort's inner loop always scans the entire unsorted portion, because it cannot know an element is the smallest without comparing it to everything left. Insertion sort compares each current element to its left neighbor, finds the neighbor is not larger, and stops right away without shifting anything.

A complete answer must:

- give 15 comparisons and 0 shifts,
- explain that selection sort scans the whole unsorted portion on every pass regardless of order, and
- explain that insertion sort's `while` loop stops immediately when the left neighbor is not larger.

Reread 4.15.7 *Telling the Two Sorts Apart* if your answer differed.

Question 14

A. The first element starts as the sorted portion, and each pass inserts one more, so after 3 passes the sorted portion is indices 0–3. Insertion sort only rearranges elements it has reached, so those are the four original front values, now in order.

- **B** is wrong: that describes selection sort. A smaller value might still be waiting at index 4, 5, or 6.
- **C** is wrong: inserted elements are not necessarily final. A smaller value from later in the array can push them right.
- **D** is wrong: insertion sort has not touched indices 4–6 yet, so they are in their original order, whatever that was.

Reread 4.15.4 *Insertion Sort by Hand* and the comparison table in 4.15.7 if you chose differently.

Question 15

```
[apple, fig, kiwi, pear, date]
[apple, date, kiwi, pear, fig]
[apple, date, fig, pear, kiwi]
[apple, date, fig, kiwi, pear]
```

The comparison `compareTo(...) < 0` means "comes earlier in dictionary order," so each pass selects the alphabetically first word in the unsorted portion.

Pass	Unsorted portion	First alphabetically	Swap
1	pear fig kiwi apple date	apple (index 3)	pear ↔ apple
2	fig kiwi pear date	date (index 4)	fig ↔ date
3	kiwi pear fig	fig (index 4)	kiwi ↔ fig
4	pear kiwi	kiwi (index 4)	pear ↔ kiwi

Notice that in pass 1 the list looks almost sorted, [apple, fig, kiwi, pear, date], but the swap sends fig to the end in pass 2. Selection sort only swaps; it never shifts.

Reread 4.15.6 *Sorting an ArrayList* and 4.15.3 *Selection Sort in Java* if your output differed.

Question 16

minIndex is declared once, before the outer loop, so it is never reset at the start of each pass.

Pass 1 works by luck, because `minIndex` starts at 0, which equals `i`. After pass 1 the array is 7 12 45 29 18 and `minIndex` is still 3. In pass 2 (`i` is 1), the search should start with 12 at index 1, but it starts with the leftover position 3 (value 29). The inner loop only looks at indices 2 to 4, so 12 is never considered. It finds 18 at index 4, which is smaller than 29, and swaps 18 into index 1, even though 12 belongs there. The later passes go wrong the same way, ending at 7 18 12 45 29.

Corrected code: remove line 6 and start each pass by setting `minIndex` to the first unsorted position:

```
1 for (int i = 0; i < arr.length - 1; i++)
2 {
3     int minIndex = i;
4     for (int j = i + 1; j < arr.length; j++)
```

Each pass must search the unsorted portion starting from its first element, so `minIndex` has to begin at `i` every time. The program then prints 7 12 18 29 45, as `SelectionSort.java` does.

Reread the note on line 7 in 4.15.3 *Selection Sort in Java* if you missed it.

Question 17

A. Pass 1 inserts 1: 3 shifts (1 shift), giving 1 3 4 2. Pass 2 inserts 4: 3 is not larger, so 0 shifts. Pass 3 inserts 2: 4 shifts, 3 shifts, then 1 is not larger (2 shifts), giving 1 2 3 4. Total $1 + 0 + 2 = 3$.

- **B** is wrong: it misses one shift, probably by stopping pass 3 after shifting only 4. The 3 is also larger than 2, so it shifts too.
- **C** is wrong: 6 is $1 + 2 + 3$, the number of shifts for a list in **reverse** order. This list is partly sorted, so fewer shifts are needed.
- **D** is wrong: it counts a shift for 4 in pass 2. Since 3 is not larger than 4, the `while` condition is false at once and nothing moves.

Reread 4.15.5 *Insertion Sort in Java* and the counting in 4.15.7 if you chose differently.

Question 18

4 7 9 1 5

The outer loop runs only for `i = 1` and 2, so it makes two passes.

Pass (i)	current	Shifted	Array after
----------	---------	---------	-------------

1	4	9	4 9 7 1 5
2	7	9	4 7 9 1 5

The first three elements are in order, but 1 and 5 were never reached. Even though 1 is the smallest value, it is still at index 3, because insertion sort only rearranges the elements it has reached.

Reread 4.15.5 *Insertion Sort in Java* if your output differed.

Question 19

A. The inner loop runs $19 + 18 + \dots + 1 = 190$ times for 20 elements, just as it ran $9 + 8 + \dots + 1 = 45$ times for 10. Doubling the size roughly quadrupled the work, because both loops got longer: the same growth you saw for nested loops in Topic 2.12.

- **B** is wrong: doubling the work is how a **single** loop grows. Selection sort has a loop inside a loop.
- **C** is wrong: $20 \times 20 = 400$ assumes the inner loop always runs 20 times. It starts at $i + 1$ and gets shorter each pass.
- **D** is wrong: it adds 20 to the old count, ignoring that every pass also gets longer.

Reread the counting in 4.15.7 *Telling the Two Sorts Apart* if you chose differently.

Question 20

Model answer: In selection sort, after $n - 1$ passes the $n - 1$ smallest values are in their final positions at the front. The one element left must be the largest, and it is already at the end, so a pass for it would only swap it with itself. In insertion sort, the first element by itself already forms a sorted portion of one element, so there is nothing to insert it into. Insertion starts with the second element, at index 1, and inserts each later element into the growing sorted portion.

A complete answer must:

- say the last element in selection sort is already the largest and in place after $n - 1$ passes,
- say the first element in insertion sort is a sorted portion by itself, and
- connect each to the loop's starting or ending value.

Reread 4.15.2 *Selection Sort by Hand* and 4.15.5 *Insertion Sort in Java* if your answer differed.