

the progress of the search by using diagrams similar to those in Example 1.

(a) 72                      (b) 41                      (c) 62

2. What changes would have to be made to `binSearch` so that it will search an array that is sorted in *descending* order.
3. Rewrite `binSearch` so that, if a search is unsuccessful, the method will return the index of the value *nearest* to `item`, instead of returning `-1`. (If there is a tie, return the smaller index.)
4. What is the maximum number of comparisons that might be necessary to perform a binary search on a list containing seven items?
5. Repeat the previous question for lists with the indicated sizes.
 

|         |         |           |            |
|---------|---------|-----------|------------|
| (a) 3   | (b) 15  | (c) 31    | (d) 63     |
| (e) 100 | (f) 500 | (g) 1 000 | (h) 10 000 |

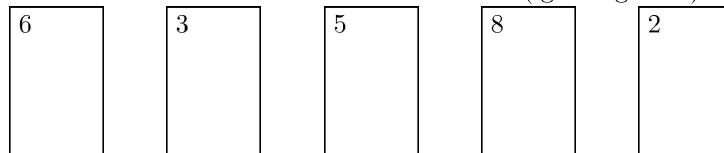
## 10.3 Insertion Sort

To use a binary search, we noted that it was necessary to have the items sorted. We now turn our attention to finding ways of sorting data.

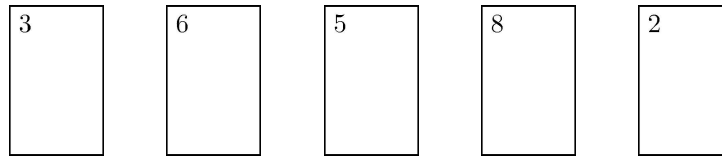
To develop algorithms of any kind, it is often useful to examine the ways that we solve similar problems in everyday life. To develop a solution to the problem of sorting data, let us look at how we might proceed if we were playing a card game and we wanted to put our hand in order, from smallest to largest. A common way of sorting such a hand is to examine cards from left to right, rearranging them if necessary by placing cards where they belong in the sequence of cards on their left. The next example illustrates the process.

### Example 1

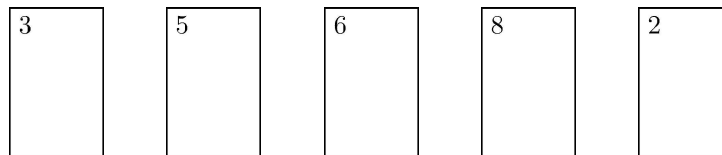
Suppose that we have the five cards shown below (ignoring suits).



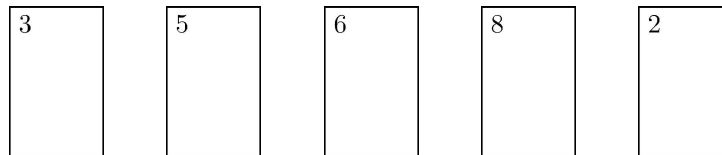
We could begin to sort these cards by looking at the 3, the second card from the left. Since  $3 < 6$ , the 3 should be inserted to the left of the 6. This will produce the following arrangement in which the two values on the left are in order.



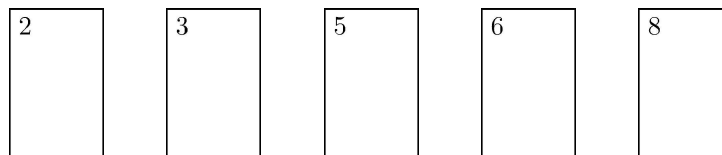
The next card from the left, the 5, should be inserted between the 3 and the 6. Doing this gives us the next arrangement in which the first three cards are guaranteed to be in order.



Now we examine the 8. Because it is greater than any of the values to its left, it should stay where it is and still give us the four leftmost values in order.



Finally, looking at the 2, we can see that it must be inserted before any of the other values. Doing this gives us our final, ordered arrangement.



To implement this algorithm for an array of values, we must examine and correctly insert each of the values in the array from the second position up to the last one. This requires a loop like the following, for an array called `list`.

```

for (int top = 1; top < list.length; top++)
    // insert element at top into its correct position
    // among the elements from 0 to top - 1

```

At each stage or *pass* of the sort, we first copy the element that we want to insert into a temporary location. We then move from right to left through the (already sorted) items on the left of the value that we are inserting. If a value in the list is larger than the new one, it is moved one space to the right (to provide room for the new item). Once the correct location of the new value is found, it is then inserted back into the array from the temporary location in which it had been saved. These ideas are incorporated into the method of the next example.

## Example 2

The method `insertSort` uses an insertion sort to arrange an array of double values in ascending order.

```

public static void insertSort (double[] list)
{
    for (int top = 1; top < list.length; top++)
    {
        double item = list[top];
        int i = top;
        while (i > 0 && item < list[i-1])
        {
            list[i] = list[i-1];
            i--;
        }
        list[i] = item;
    }
}

```

## Exercises 10.3

1. An insertion sort is to be used to put the values

6    2    8    3    1    7    4

in ascending order. Show the values as they would appear after each pass of the sort.

2. What changes would have to be made to the `insertSort` method in Example 2 in order to sort the values in descending order?
3. What might happen if, in Example 2, the `while` statement's first line were written in the following form?  

```
while (item < list[i-1] && i > 0)
```
4. Write a program that initializes an array with the names of the planets ordered by their distances from the sun (Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, and Pluto) and prints them in that order on one line. The program should then use an insertion sort to arrange the names alphabetically. To trace the progress of the sort, have it print the list after each pass.
5. The *median* of an ordered list of numerical values is defined in the following way. If the number of values is odd, the median is the middle value. If the number of values is even, the median is the average of the two middle values. Write a program that first prompts the user for the number of items to be processed, reads that many real values, and then finds their median.
6. A sort is said to be *stable* if it always leaves values that are considered to be equal in the same order after the sort. Is the insertion sort stable? Justify your answer.

## 10.4 Selection Sort

In using an insertion sort, values are constantly being moved. A sorting technique that reduces the amount of data movement is the *selection sort*. Here, we again perform a number of passes through the data and, after each pass, we again place one more item into an ordered sequence. Now, however, once an item has been placed in its position in the ordered sequence, it is never moved again.

We can begin a selection sort by scanning all the values, finding the largest one, and placing it at the top of the list, exchanging it with the item that was originally in this position.

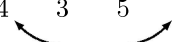
**Example 1**

To begin sorting the data

7    1    9    3    5    4

we find the largest element, 9, and swap it with 4, the element at the right end of the list. The result, after the first pass of a selection sort is

7    1    4    3    5    9



On the second pass, all the items except the last are examined to see which of these is the largest and this item is then placed at the right end of this sublist. This pattern continues on subsequent passes; on each one, the largest value among the unsorted items is placed at the top of the sublist.

**Example 2**

Using the set of data shown in Example 1, the table shows the results of successive passes of selection sort. The vertical bars indicate the division points between the unsorted items and the sorted items.

After Pass

|   |   |          |          |          |          |          |          |
|---|---|----------|----------|----------|----------|----------|----------|
| 1 | 7 | 1        | 4        | 3        | 5        |          | <b>9</b> |
| 2 | 5 | 1        | 4        | 3        |          | <b>7</b> | <b>9</b> |
| 3 | 3 | 1        | 4        |          | <b>5</b> | <b>7</b> | <b>9</b> |
| 4 | 3 | 1        |          | <b>4</b> | <b>5</b> | <b>7</b> | <b>9</b> |
| 5 |   | <b>1</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>7</b> | <b>9</b> |

Notice that only five passes are required to sort six items. Once all but one of the items are placed in their correct positions, the remaining item must also be in its correct position.

To code this algorithm in Java, we note that, for an array called `list`, we must successively find the largest item in sublists of sizes `list.length`, `list.length-1`, ..., 2. Since the upper bound of an array has index that is one less than the size of the array, the loop that controls the passes of the sort will have the form

```
for (int top = list.length - 1; top > 0; top--)  
    // locate largest item and then  
    // swap it with item at list[top]
```

We have already seen how to find the largest element in an array and how to swap elements in an array. The next example shows the selection sort that results when we put these pieces together.

### Example 3

The method `selectSort` uses a selection sort to arrange an array of double values in ascending order.

```
public static void selectSort (double[] list)  
{  
    for (int top = list.length - 1; top > 0; top--)  
    {  
        int largeLoc = 0;  
        for (int i = 1; i <= top; i++)  
            if (list[i] > list[largeLoc])  
                largeLoc = i;  
  
        double temp = list[top];  
        list[top] = list[largeLoc];  
        list[largeLoc] = temp;  
    }  
}
```

### Exercises 10.4

1. If a selection sort were to be used to sort the data shown below in alphabetical order, show the data after each pass of the sort.

Renée      Brien      Vincent      Doris      Scarlett

2. In the `selectSort` method shown in Example 3, what would happen if the expression `list[i] > list[largeLoc]` were to be changed to `list[i] < list[largeLoc]`?

3. In our version of selection sort, if the largest item is already at location `top` in the list, then the method still swaps that value with itself, even though that is not necessary.
  - (a) How could the method be changed to avoid this unnecessary swapping?
  - (b) Why might it be better to leave the method as it is in the text?
4. On each pass of our version of selection sort, the *largest* value among the remaining unsorted items was placed in its correct position. An alternate form of the algorithm uses each pass to place the *smallest* value among the remaining unsorted values in its correct position.
  - (a) Given the set of data  

8      9      6      1      2      4

show the data as they would appear after each pass of a selection sort using this algorithm.
  - (b) Write a Java method that implements this algorithm to sort an array of `int` values.
5. Sometimes we are only interested in knowing the values that would occupy one end of the list if the list were sorted. As an example, we may want to know the scores of only the top ten competitors in a contest. Modify the selection sort of Example 3 so that, instead of sorting the entire array, it puts the  $k$  largest values in order in the last  $k$  positions in the array. The value of  $k$  should be a parameter of the method.

## 10.5 Bubble Sort

Our next (but not our last) sorting algorithm, called a *bubble sort*, is an example of a class of sorts known as *exchange sorts*. With the bubble sort, the basic idea is to compare adjacent values and exchange them if they are not in order.

### Example 1

Suppose that we want to use a bubble sort to arrange the names

Phil      Ivan      Sara      Jack      Gina

in alphabetical order. We start by comparing Phil to Ivan. Since they are not in order, we exchange them to give

Ivan      Phil      Sara      Jack      Gina

Next we compare Phil to Sara. Since they are in order, we leave them that way giving

Ivan      Phil      Sara      Jack      Gina

Now, comparing Sara to Jack, we see that they must be exchanged. Doing this gives

Ivan      Phil      Jack      Sara      Gina

We then compare Sara to Gina. Again, we must perform an exchange to obtain

Ivan      Phil      Jack      Gina      Sara

The result of all this comparing and exchanging is that, after one pass, the largest value (Sara, in our example) will be at the upper end of the list. Like a bubble rising in a liquid, the largest value has risen to the top of the list.

As with the selection sort shown in the previous section, we now repeat the process used in the first pass on all but the last element. As we proceed, the passes deal with shorter and shorter subsequences of the original list until all values are in their correct positions.

### Example 2

The following tables show the actions of a bubble sort in ordering the names

Phil      Ivan      Sara      Jack      Gina



The colons indicate the values currently being compared. The vertical bars indicate the division points between the unsorted data and the sorted data.

|             |             |   |             |      |             |             |
|-------------|-------------|---|-------------|------|-------------|-------------|
|             | Phil        | : | Ivan        | Sara | Jack        | Gina        |
|             | Ivan        |   | Phil        | :    | Sara        | Jack        |
| First Pass  | Ivan        |   | Phil        |      | Sara        | :           |
|             | Ivan        |   | Phil        | Jack | Sara        | :           |
|             | Ivan        |   | Phil        | Jack | Sara        | :           |
|             | Ivan        |   | Phil        | Jack | Gina        |             |
|             |             |   |             |      |             | <b>Sara</b> |
|             | Ivan        | : | Phil        | Jack | Gina        |             |
| Second Pass | Ivan        |   | Phil        | :    | Jack        | Gina        |
|             | Ivan        |   | Jack        |      | Phil        | :           |
|             | Ivan        |   | Jack        | Gina |             | <b>Phil</b> |
|             |             |   |             |      |             | <b>Sara</b> |
|             | Ivan        | : | Jack        | Gina |             | <b>Phil</b> |
| Third Pass  | Ivan        |   | Jack        | :    | Gina        |             |
|             | Ivan        |   | Gina        |      | <b>Jack</b> | <b>Phil</b> |
|             |             |   |             |      |             | <b>Sara</b> |
| Fourth Pass | Ivan        | : | Gina        |      | <b>Jack</b> | <b>Phil</b> |
|             | <b>Gina</b> |   | <b>Ivan</b> |      | <b>Jack</b> | <b>Phil</b> |
|             |             |   |             |      |             | <b>Sara</b> |

As we saw with selection sort, the number of passes needed to sort the items is one less than the number of items. After all but one of the items have been ordered, the remaining one must also be in its correct position.

To create a Java method for a bubble sort is fairly easy if we use some of our previous techniques. To perform all the passes we need a loop like that of the selection sort.

```
for (int top = list.length-1; top > 0; top--)
    // compare adjacent values of the unsorted sublist
    // from 0 to top, exchanging as necessary
```

Within each pass, the code for comparing and exchanging values in a sublist is easily written.

```
for (int i = 0; i < top; i++)
    if (list[i].compareTo(list[i+1]) > 0)
    {
        temp = list[i];
```

```
list[i] = list[i+1];  
list[i+1] = temp;  
}
```

Combining these fragments of code and inserting them into a method gives us a basic bubble sort. We can improve this sort's performance by noting that, on each pass, we often do more than simply place the greatest value at the upper end of a sublist; the comparisons and exchanges tend to push *all* values toward their final positions in the list. It may happen that this shuffling of values puts all the values in their final positions before all passes have been completed. In such a situation, we should stop working as soon as possible, avoiding any unnecessary passes. As it turns out, this is not too difficult to do.

Before we incorporate this modification, we must first answer the following question: how do we know if there is no more work to be done? The answer is: if *no* exchanges are performed in an entire pass, then the items must be fully sorted. The reasoning here is that, if no exchanges are needed, then each value must be correctly ordered between its immediate neighbours. It follows that the entire set of values must, therefore, be completely ordered. Thus we need to perform another pass only if the previous one carried out one or more exchanges.

To encode this idea in Java, the `for` statement that controls the number of passes should now incorporate a test to see if another pass is necessary. We do this with a `boolean` variable called `sorted`. Initially, `sorted` is set to `false`; we do not start a pass unless `sorted` is `false`. Once we have started a pass, we (optimistically) set `sorted` to `true` because no exchanges have yet been performed during that pass. If any exchanges are required during a pass, we reset `sorted` to `false`. Processing continues until `sorted` remains `true` after a full pass or we have completed the maximum number of passes.

### Example 3

This method uses a bubble sort to place an array of strings in ascending order. It uses a boolean flag, `sorted`, to make the bubble sort more efficient by stopping the sort after a full pass in which there are no exchanges.

```
public static void bubbleSort (String[] list)  
{  
    boolean sorted = false;
```

```

for (int top = list.length-1; top > 0 && !sorted; top--)
{
    sorted = true;
    for (int i = 0; i < top; i++)
        if (list[i].compareTo(list[i+1]) > 0)
        {
            sorted = false;
            String temp = list[i];
            list[i] = list[i+1];
            list[i+1] = temp;
        }
}
}

```

Although bubble sort uses an interesting technique and has a cute name, we do not recommend it. It is almost always slower than either of the sorts that we have already examined because it usually involves far more data movement than they require.

## Exercises 10.5

1. Make tables like those shown in Example 2 to show the comparisons and exchanges that would take place in using a bubble sort to put the following data in ascending order.

3    8    3    2    7    5

2. What changes would have to be made to the `bubbleSort` method in order to make it sort values in descending order?
3. A modification of the bubble sort is the *cocktail shaker sort* in which, on odd-numbered passes, large values are carried to the top of the list while, on even-numbered passes, small values are carried to the bottom of the list. Make tables like those shown in Example 2 to show the first two passes of a cocktail-shaker sort on the following data.

2    9    4    6    1    7