That result makes intuitive sense. Suppose that \( n \) is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because \( \log_2(100) \approx 6.64386 \), and indeed the next larger power of 2 is \( 2^7 = 128 \).

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you search the array only once, then it is more efficient to pay for an \( O(n) \) linear search than for an \( O(n \log(n)) \) sort and an \( O(n \log(n)) \) binary search. But if you will be making many searches in the same array, then sorting it is definitely worthwhile.

The Arrays class contains a static binarySearch method that implements the binary search algorithm, but with a useful enhancement. If a value is not found in the array, then the returned value is not \(-1\), but \(-k - 1\), where \( k \) is the position before which the element should be inserted. For example,

```java
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// Returns -3; v should be inserted before position 2
```

**Self Check**

13. Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?

14. Why is it useful that the Arrays.binarySearch method indicates the position where a missing element should be inserted?

15. Why does Arrays.binarySearch return \(-k - 1\) and not \(-k\) to indicate that a value is not present and should be inserted before position \( k \)?

### 14.8 Sorting Real Data

In this chapter we have studied how to search and sort arrays of integers. Of course, in application programs, there is rarely a need to search through a collection of integers. However, it is easy to modify these techniques to search through real data.

The Arrays class supplies a static sort method for sorting arrays of objects. However, the Arrays class cannot know how to compare arbitrary objects. Suppose, for example, that you have an array of Coin objects. It is not obvious how the coins should be sorted. You could sort them by their names, or by their values. The Arrays.sort method cannot make that decision for you. Instead, it requires that the objects belong to classes that implement the Comparable interface. That interface has a single method:

```java
public interface Comparable
{
```

int compareTo(Object otherObject);
}

The call

    a.compareTo(b)

must return a negative number if \(a\) should come before \(b\), 0 if \(a\) and \(b\) are the same, and a positive number otherwise.

Several classes in the standard Java library, such as the String and Date classes, implement the Comparable interface.

You can implement the Comparable interface for your own classes as well. For example, to sort a collection of coins, the Coin class would need to implement this interface and define a compareTo method:

    public class Coin implements Comparable {
        public int compareTo(Object otherObject) {
            Coin other = (Coin) otherObject;
            if (value < other.value) return -1;
            if (value == other.value) return 0;
            return 1;
        }
    }

When you implement the compareTo method of the Comparable interface, you must make sure that the method defines a total ordering relationship, with the following three properties:

- **Antisymmetric**: If \(a\).compareTo\(b\) \(\leq 0\), then \(b\).compareTo\(a\) \(\geq 0\)
- **Reflexive**: \(a\).compareTo\(a\) = 0
- **Transitive**: If \(a\).compareTo\(b\) \(\leq 0\) and \(b\).compareTo\(c\) \(\leq 0\), then \(a\).compareTo\(c\) \(\leq 0\)

Once your Coin class implements the Comparable interface, you can simply pass an array of coins to the Arrays.sort method:

    Coin[] coins = new Coin[n];
    // Add coins
    Arrays.sort(coins);

If the coins are stored in an ArrayList, use the Collections.sort method instead; it uses the merge sort algorithm:

    ArrayList<Coin> coins = new ArrayList<Coin>();
    // Add coins
    Collections.sort(coins);

As a practical matter, you should use the sorting and searching methods in the Arrays and Collections classes and not those that you write yourself. The library algorithms have been fully debugged and optimized. Thus, the primary purpose of this chapter was not to teach you how to implement practical sorting and searching
algorithms. Instead, you have learned something more important, namely that different algorithms can vary widely in performance, and that it is worthwhile to learn more about the design and analysis of algorithms.

**Self Check**

16. Why can't the Arrays.sort method sort an array of Rectangle objects?
17. What steps would you need to take to sort an array of BankAccount objects by increasing balance?

**Common Error 14.1**

The compareTo Method Can Return Any Integer, Not Just -1, 0, and 1

The call `a.compareTo(b)` is allowed to return any negative integer to denote that `a` should come before `b`, not necessarily the value -1. That is, the test

```
if (a.compareTo(b) == -1) // ERROR!
```

is generally wrong. Instead, you should test

```
if (a.compareTo(b) < 0) // OK
```

Why would a compareTo method ever want to return a number other than -1, 0, or 1? Sometimes, it is convenient to just return the difference of two integers. For example, the compareTo method of the String class compares characters in matching positions:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

If the characters are different, then the method simply returns their difference:

```
if (c1 != c2) return c1 - c2;
```

This difference is a negative number if `c1` is less than `c2`, but it is not necessarily the number -1.

**Advanced Topic 14.4**

The Parameterized Comparable Interface

As of Java version 5.0, the `Comparable` interface is a parameterized type, similar to the `ArrayList` type:

```java
public interface Comparable<T>
{
    int compareTo(T other)
}
```

The type parameter specifies the type of the objects that this class is willing to accept for comparison. Usually, this type is the same as the class type itself. For example, the `Coin` class would implement `Comparable<Coin>`, like this:
The type parameter has a significant advantage: You need not use a cast to convert an Object parameter into the desired type.

**Advanced Topic 14.5**

**The Comparator Interface**

Sometimes, you want so sort an array or array list of objects, but the objects don’t belong to a class that implements the Comparable interface. Or, perhaps, you want to sort the array in a different order. For example, you may want to sort coins by name rather than by value.

You wouldn’t want to change the implementation of a class just in order to call Arrays.sort. Fortunately, there is an alternative. One version of the Arrays.sort method does not require that the objects belong to classes that implement the Comparable interface. Instead, you can supply arbitrary objects. However, you must also provide a comparator object whose job is to compare objects. The comparator object must belong to a class that implements the Comparator interface. That interface has a single method, compare, which compares two objects.

As of Java version 5.0, the Comparator interface is a parameterized type. The type parameter specifies the type of the compare parameters. For example, Comparator&lt;Coin&gt; looks like this:

```java
public interface Comparator&lt;Coin&gt;
{
    int compare(Coin a, Coin b);
}
```

The call

```java
comp.compare(a, b)
```

must return a negative number if a should come before b, 0 if a and b are the same, and a positive number otherwise. (Here, comp is an object of a class that implements Comparator&lt;Coin&gt;.)

For example, here is a Comparator class for coins:

```java
public class CoinComparator implements Comparator&lt;Coin&gt;
{
    public int compare(Coin a, Coin b)
    {
        if (a.getValue() < b.getValue()) return -1;
        if (a.getValue() == b.getValue()) return 0;
        return 1;
    }
}
```
To sort an array of coins by value, call

\[
\text{Arrays.sort(coins, new CoinComparator());}
\]

CHAPTER SUMMARY

1. The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

2. Computer scientists use the big-Oh notation \( f(n) = O(g(n)) \) to express that the function \( f \) grows no faster than the function \( g \).

3. Selection sort is an \( O(n^2) \) algorithm. Doubling the data set means a fourfold increase in processing time.

4. Insertion sort is an \( O(n^2) \) algorithm.

5. The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

6. Merge sort is an \( O(n \log(n)) \) algorithm. The \( n \log(n) \) function grows much more slowly than \( n^2 \).

7. The Arrays class implements a sorting method that you should use for your Java programs.

8. A linear search examines all values in an array until it finds a match or reaches the end.

9. A linear search locates a value in an array in \( O(n) \) steps.

10. A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

11. A binary search locates a value in an array in \( O(\log(n)) \) steps.

12. The sort method of the Arrays class sorts objects of classes that implement the Comparable interface.

13. The Collections class contains a sort method that can sort array lists.

FURTHER READING