Chapter 12

Searching and Sorting Algorithms

With an Introduction to Algorithm Efficiency

Outline

Minimum and maximum algorithms
Linear and binary search algorithm
Running times for linear and binary search
Selection and insertion sort
Running time for selection and insertion sort
Simulation to compare selection and insertion sort
Mergesort
File merge example
Quicksort
Running time for mergesort and quicksort
Simulation to compare mergesort and quicksort
Generic sorting
Sorting strings in lexicographical order
Comparing bank account objects
12.1 Introduction

In this chapter we study some important searching and sorting algorithms with an emphasis on efficiency and recursion. Searching and sorting are two of the most important processing operations performed by computers so it is important to have efficient algorithms. Some of the material from Chapter 8, Section 8.4, is repeated here in a more general form.

First we consider some simple variations of algorithms for finding the minimum or maximum values in an array of \( n \) elements and for doing a linear search of an array for a given value. To measure the efficiency of an algorithm we introduce a special mathematical order notation that can be used to measure the “running time” without worrying about the effects of particular computer hardware and software. Using this notation we see that these simple algorithms are \( O(n) \) which means that for large \( n \) an upper bound on their running time is proportional to \( n \), the number of elements to search. For each algorithm we can have three kinds of behavior: best case, average case, and worst case.

Next we consider both recursive and non-recursive versions of the binary search algorithm for a sorted array and determine the running time. It will be clear that binary search is much more efficient for large arrays than linear search since we can show it is \( O(\log n) \), whereas linear search is \( O(n) \).

Next we consider the efficiency of four popular sorting algorithms. Two of these, selection sort and insertion sort, are called quadratic sorting algorithms because they are \( O(n^2) \). The other two, mergesort and quicksort are \( O(n \log n) \). This means that they are much more efficient for large arrays than the quadratic algorithms. The mergesort and quicksort algorithms are naturally recursive. We also show how to empirically compare sorting algorithms using a simulation.

Initially our algorithms are developed using arrays of type \( \text{int[]} \) but there are many other kinds of arrays that we might want to search or sort. Rather than rewrite each algorithm for different types of arrays we develop generic searching and sorting classes that work with arrays of type \( \text{Object[]} \) using the \text{Comparable} and \text{Comparator} interfaces to define an ordering for the elements.

12.2 Minimum and maximum algorithms

We have considered these algorithms in Chapter 8, Section 8.4, and now want to generalize them so that a specified subarray, rather than the entire array, is searched. A subarray of an array \( \langle a_0, \ldots, a_{n-1} \rangle \) is defined to be any sequence of elements \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \) with \( 0 \leq \text{start} \leq \text{end} \leq n - 1 \). The entire array is the subarray having \( \text{start} = 0 \) and \( \text{end} = n - 1 \).

The minimum problem can be stated formally as follows:

“Given the array \( \langle a_0, \ldots, a_{n-1} \rangle \) and the index values \( \text{start} \) and \( \text{end} \) defining a subarray, determine an index \( i \) such that \( \text{start} \leq i \leq \text{end} \) and \( a_i \leq a_k \) for all \( k \) such that \( \text{start} \leq k \leq \text{end} \).”

The index \( i \) is not unique since the minimum value may occur at several places in the subarray. Even for a simple algorithm like this there are several variations. Do we return the minimum value or the position at which the minimum value occurs? If we return the index do we return the index for the first occurrence of the minimum or the last occurrence?
Let us first design the algorithm to return the position of the first occurrence of the minimum. We begin by assuming the minimum value is at index `start` and then use a loop to process the remaining elements in the subarray from `start + 1` to `end`. Each time a smaller value is obtained the index is updated. When the loop terminates the index will be the position of the first occurrence of the minimum. The pseudo-code algorithm is given in Figure 12.1. The final value of `index` is the smallest index of the subarray such that \( a_{index} \) is the minimum value of the elements in the subarray.

Here are three variations of `findMinimum` in Java.

(a) Return index of first occurrence:

```java
int findMinimum(int[] a, int start, int end)
{
    int index = start;
    for (int k = start + 1; k <= end; k++)
    {
        if (a[k] < a[index])
        {
            index = k;
        }
    }
    return index;
}
```

(b) Return index of last occurrence:

```java
int findMinimum(int[] a, int start, int end)
{
    int index = start;
    for (int k = start + 1; k <= end; k++)
    {
        if (a[k] <= a[index])
        {
            index = k;
        }
    }
    return index;
}
```
The only difference is the use of <= instead of < in the comparison of array elements.

(c) Return the minimum value itself. It is the simplest algorithm if the position is not required.

```java
int findMinimum(int[] a, int start, int end) {
    int min = a[start];
    for (int k = start + 1; k <= end; k++) {
        if (a[k] < min)
            min = a[k];
    }
    return min;
}
```

To obtain a findMaximum method for finding the maximum simply reverse the inequality in the if-statement.

**EXAMPLE 12.1** (Using findMinimum method, versions (a) and (b)) The statements

```java
int[] scores = {56, 32, 27, 98, 27, 57, 68, 28, 45, 65};
int pos = findMinimum(score, 0, score.length - 1);
System.out.println("Position of minimum is " + pos);
System.out.println("Minimum value is " + scores[pos]);
```

find the minimum value in the scores array. For version (a) the values printed are 2 for pos and 27 for the minimum value. For version (b) the value 4 instead of 2 for pos is printed. This method can be tested in BeanShell by entering the method into the workspace editor and choosing "Eval in workspace".

We have used an array of integers here although an array of any type for which elements can be compared can be used. For example, to find the minimum balance in an array of BankAccount objects and return the index of its first occurrence use the method

```java
int findMinimumBalance(BankAccount[] a, int start, int end) {
    int index = start;
    for (int k = start + 1; k <= end; k++) {
        if (a[k].getBalance() < a[index].getBalance())
            index = k;
    }
    return index;
}
```
12.3 Running time of an algorithm

In order to compare two algorithms to see if one is more efficient than the other we need to measure the running time of each algorithm. One way to do this is to implement the algorithms and write a program that calculates their running time for various arrays and array sizes. The problem here is that the running time will depend on the particular hardware and software (processor, operating system, compiler, language). Faster computers or more efficient compilers will give lower running times.

We need a hardware/software-independent theoretical way to measure the running time of an algorithm in terms of the size of the problem and the number of times selected statements in the algorithm are executed.

For example, the running time of \texttt{findMinimum} depends on the number of array elements that are searched to find the minimum value. It is clear that finding the minimum in a million element array will take longer than for a 10 element array. If you time \texttt{findMinimum} for various array sizes, \( n \), you will find that as \( n \) increases the running time increases linearly with \( n \): searching a 10000 element array takes about twice as long as a 5000 element array which takes about twice as long as a 2500 element array, and so on. We say that \texttt{findMinimum} is a linear algorithm.

This means that for large \( n \) the running time will have the form \( T(n) = an + b \) for some constants \( a \) and \( b \). If \( n \) is large we can omit \( b \) in comparison to \( an \). It is the fact that \( T(n) \) is proportional to \( n \), for large \( n \), that is important here, not the constants \( a \) and \( b \) which depend on the particular hardware/software environment. To remove this dependency on \( a \) and \( b \) we simply say that all algorithms whose running time is at most \( an + b \), such as \texttt{findMinimum}, are of order \( n \) and we write \( T(n) = O(n) \) to indicate that an upper bound on the running time is proportional to \( n \). This is often called the “Big Oh” notation.

To derive this result mathematically we need some representative statements to count in the \texttt{findMinimum} algorithm. In this way we reduce the running time calculation to a counting problem. For example, we can use the number of times the if-statement inside the for loop is executed (the number of comparisons). This is the number of times the for loop is executed, namely \( \texttt{end} - \texttt{start} + 1 \), and this is just the number, \( n \), of elements in the subarray. Therefore \( T(n) = n \). We could choose other measures such as the total number of comparisons and the total number of assignment statements but then we would have difficulty calculating the exact number of operations since the number of times the assignment statement inside the if-statement is executed depends on the elements in the array. In any case this number would still have an upper bound of the form \( an + b \) so \( T(n) = O(n) \) although this is more difficult to show.

We can also define the best, average, and worst case behavior of an algorithm. For the linear search algorithm each can be shown to be \( O(n) \). For other algorithms the worst case behavior is often easy to determine but the average behavior can be difficult since it depends on the probabilistic distribution of the elements in the array. In any case the worst case behavior gives an upper bound on the running time.

\textbf{Example 12.2} (Mathematical definition of \( O \)) We can give a precise definition of the \( O \) notation as follows: Given two functions \( f \) and \( g \) we say that \( f(n) = O(g(n)) \) if there exist constants \( c > 0 \) and \( N > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n > N \). It is important to realize that \( O(n) \) is not a function. It represents an infinite set of functions. Therefore you should really interpret
$T(n) = O(n)$ to mean $T(n) \in O(n)$. There are other measures of the rate of growth of a function denoted by $\Omega(g(n))$, for a lower bound, and $\Theta(g(n))$, for both upper and lower bounds, that we will not consider here.

**Example 12.3** (Example of $O$ calculations) It is easy to show that $an + b$ is $O(n)$. Let $c = a + 1$. Then $an + b \leq cn$ if $b \leq n$. Therefore let $N$ be any integer greater than $b$ and we will have $an + b \leq cn$ whenever $n > N$. This shows for example that $n, 2n, 3n + 1/n$ are all $O(n)$.

**Example 12.4** (Example of $O$ calculations) Let us show that $n^2 + 3n + 1$ is $O(n^2)$. Let $c = 2$. Then $n^2 + 3n + 1 \leq 2n^2$ if $n^2 \geq 3n + 1$ which is true if $n > 4$. Therefore let $N = 4$ and $n^2 + 3n + 1$ is $O(n^2)$ for all $n > 4$.

### 12.4 Searching algorithms

The linear search algorithms was briefly discussed in Chapter 8, Section 8.4. Here we consider the linear search algorithm and both recursive and non-recursive versions of the much superior binary search algorithm and we obtain upper bounds on their running times. We will develop these algorithms for arrays of type `int[]` although it is easy to modify them to search other types of arrays.

The Java method for each of these three algorithms has a prototype of the form

```java
public static int search(int[] a, int x, int start, int end)
```

where $a$ is the array to search, $x$ is the element to search for in the subarray defined by $start$ and $end$, and the return value is either the position at which $x$ is found or $-1$ if $x$ is not found.

#### 12.4.1 Linear search algorithm

The `findMinimum` and `findMaximum` algorithms are examples of search algorithms since we are searching for the smallest or largest value. An important variation is to search for a given value. We now generalize the results in Chapter 8, Section 8.4 so that the linear search is applied to a subarray rather than the entire array.

In a linear search of a subarray we are looking for a given value $x$ among the array elements in the subarray. If we find it then we can return the array index at which it is found, otherwise we can return the invalid index value $-1$. The linear search problem can be stated as follows:

“Given the array $\langle a_0, \ldots, a_{n-1} \rangle$, the index values $start$ and $end$ defining a subarray $\langle a_{start}, \ldots, a_{end} \rangle$, and a value $x$ to find, determine an index $i$ such that $a_i = x$ and $start \leq i \leq end$. If such an index cannot be found let the index be $-1$.”

A while loop is appropriate here since we do not know how many times the body of the loop will be executed. We need to stop executing the body if the element we are looking for is found. The loop continues as long as we are within the subarray and as long as we have not found the element we are looking for. The pseudo-code algorithm is given in Figure 12.2. There are two ways the while loop can terminate. If $index \leq end$ is false then we have “gone off the end” of the subarray
12.4 Searching algorithms

![Figure 12.2: Pseudo-code linear search algorithm](image)

**Algorithm** LinearSearch\((\langle a_0, a_1, \ldots, a_{n-1} \rangle, x, \text{start, end} \rangle)\)

\[
\text{index} \leftarrow \text{start} \\
\text{WHILE} \ \text{index} \leq \text{end} \land a_{\text{index}} \neq x \ \text{DO} \\
\quad \text{index} \leftarrow \text{index} + 1 \\
\text{END WHILE} \\
\text{IF} \ \text{index} > \text{end} \ \text{THEN} \\
\quad \text{RETURN} -1 \\
\text{ELSE} \\
\quad \text{RETURN} \ \text{index} \\
\text{END IF}
\]

Figure 12.2: Pseudo-code linear search algorithm

![Figure 12.3: Alternate Pseudo-code linear search algorithm](image)

**Algorithm** LinearSearch\((\langle a_0, a_1, \ldots, a_{n-1} \rangle, x, \text{start, end} \rangle)\)

\[
\text{index} \leftarrow \text{start} \\
\text{WHILE} \ \text{index} \leq \text{end} \ \text{DO} \\
\quad \text{IF} \ a_{\text{index}} = x \ \text{THEN} \\
\quad \quad \text{RETURN} \ \text{index} \\
\quad \text{END IF} \\
\quad \text{index} \leftarrow \text{index} + 1 \\
\text{END WHILE} \\
\text{RETURN} -1
\]

Figure 12.3: Alternate Pseudo-code linear search algorithm

and the entire boolean expression is false so the loop will exit. The expression \(a_{\text{index}} \neq x\) will not be evaluated in this case, assuming short-circuit evaluation as in Java (in an expression like \(a \land b\) if \(a\) is false \(b\) is never evaluated). Otherwise the array index could be out of range. If the element \(x\) is found then the expression \(a_{\text{index}} \neq x\) will be false and the loop will exit. When the loop exits we can test \(\text{index}\) to see which exit was taken. If \(\text{index} > \text{end}\) then we did not find \(x\) so \(-1\) is returned. Otherwise \(x\) was found and \(\text{index}\) is returned.

An alternate version that may be easier to understand is shown in Figure 12.3. As for the \text{findMinimum} method there are other variations. We could run the loop backwards and find the last occurrence of \(x\) rather than the first occurrence. Or, if the position is not required, a boolean return type can have the value true if \(x\) is found and false otherwise.

**Order of linear search**

Since the while loop exits immediately if \(x\) is found, the best case behavior will occur if \(x\) is the first element in the subarray. The running time will not depend at all on the size, \(n\), of the array. To indicate that the running time does not depend on \(n\) we write \(T(n) = O(1)\).
Searching and Sorting Algorithms

<table>
<thead>
<tr>
<th>Step</th>
<th>Left subarray</th>
<th>Middle value</th>
<th>Right subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 5 7 11</td>
<td>21</td>
<td>47 56 63 84 89</td>
</tr>
<tr>
<td>2</td>
<td>47 56</td>
<td>63</td>
<td>84 89</td>
</tr>
<tr>
<td>3</td>
<td>empty</td>
<td>47</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>empty</td>
<td>56</td>
<td>empty</td>
</tr>
</tbody>
</table>

Table 12.1: Binary search example

The worst case behavior will occur if \( x \) is found at the last position in the subarray or \( x \) is not found. In this case every element in the subarray is examined. Defining \( n = end - start + 1 \), the body of the while loop will be executed \( n \) times. Therefore the worst case behavior is \( T(n) = O(n) \). The average case behavior will depend on the values of the array elements and a probabilistic argument shows that \( T(n) = O(n) \) in this case too.

12.4.2 Recursive binary search algorithm

The linear search algorithm is an \( O(n) \) algorithm so it is not useful for large \( n \). If we can assume that the array elements are sorted in increasing order (or decreasing order) then we can write a much better algorithm called the binary search algorithm because it continually divides the problem size (number of elements to search) in half until the element is found. This division process is often called bisection. On the other hand each iteration of the linear search algorithm would eliminate only one element.

To develop the recursive algorithm we start with an array \( \langle a_0, \ldots, a_{n-1} \rangle \) sorted in increasing order. The half to be searched at each step will be a subarray of the form

\[
\langle a_{start}, \ldots, a_{end} \rangle, \text{ where } a_{start} \leq a_{start+1} \leq \cdots \leq a_{end}
\]

We want to determine if \( x \) is found in the subarray. To begin the bisection process we look at the middle element, \( a_{mid} \), of the subarray which we define by the index \( mid = (start + end)/2 \). This will be the middle element if there is an odd number of elements in the subarray and the leftmost of the two middle elements if there is an even number of elements. There are two base cases and two recursive cases to consider:

1. If \( start > end \), the subarray is empty and \( x \) is not found (base case).
2. If \( x = a_{mid} \), then \( x \) has been found (base case).
3. If \( x < a_{mid} \), search left subarray \( \langle a_{start}, \ldots, a_{mid-1} \rangle \) for \( x \) (recursive case).
4. If \( x > a_{mid} \), search right subarray \( \langle a_{mid+1}, \ldots, a_{end} \rangle \) for \( x \) (recursive case).

As an example, lets try to find 56 in the ten element array \( \langle 3,5,7,11,21,47,56,63,84,89 \rangle \). The bisection results are shown in Table 12.1. The important feature of this algorithm is that the number of elements remaining to be searched is cut in half at each step.

The running time is proportional to the total number of bisections that are made and this is proportional to the number of comparisons of \( x \) with \( a_{mid} \). The best case behavior occurs when we...
12.4 Searching algorithms

<table>
<thead>
<tr>
<th>n</th>
<th>(\log_2 n = (\ln n / \ln 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>(10^6)</td>
<td>20</td>
</tr>
<tr>
<td>(10^9)</td>
<td>30</td>
</tr>
<tr>
<td>(10^{40})</td>
<td>133</td>
</tr>
</tbody>
</table>

Table 12.2: Comparison of \(n\) and \(\log_2 n\)

are searching for the element at the middle. Then the algorithm is \(O(1)\). In the worst case, which we now consider, the element is either found at the last bisection step, when both left and right subarrays are empty, or it is not found.

For a 10 element array there are never more than 4 bisections: the number of elements remaining to search at each step is 10 \(\rightarrow\) 5 \(\rightarrow\) 3 \(\rightarrow\) 2 \(\rightarrow\) 1. For linear search there could be as many as 10 comparisons. For a 1000 element array the number of elements remaining to be searched at each step is

\[
1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 63 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1
\]

so there are never more than 10 bisections versus as many as 1000 comparisons for linear search. Similarly, for a 1,000,000 element array at most 20 bisections are needed versus as many as 1,000,000 comparisons for linear search. It is clear that binary search is far superior to linear search. Later we will show that the order of binary search is \(O(\log_2 n)\). Table 12.2 shows how much faster \(n\) increases compared to \(\log_2 n\): if it took 30 units of time to search a billion element array then a linear search could take as many as \(10^9\) units of time. Calculators usually have buttons for the base 10 logarithms, \(\log_{10} n\), or base \(e\) logarithms, \(\log_e n = \ln n\), but it is easy to calculate logarithms in other bases using the identity

\[
\log_b x = \log_e x / \log_e b
\]

Therefore, using \(b = 2\), \(c = e\), and \(x = n\) we have \(\log_2 n = \log_e n / \log_e 2 = \ln n / \ln 2\) as shown in the table.

We can now write a pseudo-code binary search algorithm. The bisection process terminates when the indices \(\text{start}\) and \(\text{end}\) satisfy \(\text{start} > \text{end}\). This corresponds to an empty subarray. The algorithm is given in Figure 12.4. Recalling that recursion is a problem solving technique for which a problem is solved in terms of one or more smaller versions of itself and one or more non-recursive base cases, we see that this algorithm is naturally recursive: the problem of searching a subarray is expressed in terms of the two smaller problems of searching either the left half subarray or the right half subarray. The base case occurs when the subarray is empty and this occurs when \(\text{start} > \text{end}\).

12.4.3 Non-recursive binary search algorithm

It is also easy to develop a non-recursive version of binary search. For a subarray \(\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle\) we need to keep two indices \(\text{low}\) and \(\text{high}\) that define the half \(\langle a_{\text{low}}, \ldots, a_{\text{high}} \rangle\) to be searched next.
### 12.4.4 Running time of binary search algorithm

The results in Table 12.2 suggest that the worst case behavior of binary search is $O(\log_2 n)$ (which is also $O(\log_b n)$ for any base $b$). We now sketch the proof using the recursive version. Let $n$ be the number of elements to search and let $T(n)$ be the running time for $n$ elements defined as the number of bisections required. This will be the number of times the if-statement is executed. To simplify the proof we also assume that $n$ is a power of 2: $n = 2^m$ for some $m$. Then we can search a subarray of $n$ elements using one bisection to determine which half to use and $T(n/2)$ bisections...
12.4 Searching algorithms

ALGORITHM NRBinarySearch((a_{start}, \ldots, a_{end}), x)
low ← start
high ← end
WHILE low ≤ high DO
  mid ← (start + end) / 2
  IF x < a_{mid} THEN
    high ← mid − 1
  ELSE IF x > a_{mid} THEN
    low ← mid + 1
  ELSE
    RETURN mid
  END IF
END WHILE
RETURN −1

Figure 12.5: Pseudo-code non-recursive binary search algorithm

to search this half. This gives the following recurrence relation connecting \( T(n) \) and \( T(n/2) \):

\[
T(n) = T(n/2) + 1, \text{ with initial condition } T(1) = 1.
\]

Substituting \( n/2 \) for \( n \) gives \( T(n/2) = T(n/2^2) + 1 \). Continuing

\[
T(n) = T(n/2) + 1
= T(n/2^2) + 1 + 1 = T(n/2^2) + 2
= T(n/2^3) + 1 + 2 = T(n/2^3) + 3
\]
\[
\ldots
= T(n/2^m) + m
= T(1) + m
= 1 + m
= 1 + \log_2 n, \text{ since } n = 2^m
= O(\log_2 n)
\]

Therefore binary search is an \( O(\log n) \) algorithm and, as Table 12.2 shows, is much superior to linear search. If \( n \) is not a power of two we can use \( \lceil n/2^k \rceil \) at each stage instead of \( n/2^k \), noting that \( 2^k \leq n \leq 2^{k+1} \) for some \( k \) (\( \lceil x \rceil \) denotes the smallest integer greater than or equal to \( x \)).

12.4.5 Class of static searching methods

The three search algorithms can easily be translated into Java. We place them in a class called IntArraySearch as static methods:
package chapter12.searching;

/**
 * Static methods for searching an integer array
 */
public class IntArraySearch
{
    /**
     * Search subarray for a given element using linear search algorithm.
     * @param a The array to search
     * @param x The value to search for
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     * @return If x is found the first array index such that x = a[i] else -1 to indicate failure.
     */
    public static int linearSearch(int[] a, int x, int start, int end)
    {
        int i = start;
        while ((i <= end) && (a[i] != x))
            i++;
        if (i <= end)
            return i;
        else
            return -1;
    }

    /**
     * Search subarray for a given element using the recursive binary search algorithm. It is assumed that the array is sorted in increasing order.
     * @param a The array to search
     * @param x The value to search for
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     * @return If x is found the first array index such that x = a[i] else -1 to indicate failure.
     */
    public static int rBinarySearch(int[] a, int x, int start, int end)
    {
        if (start <= end)
        {
            int mid = (start + end) / 2;
            if (x < a[mid])
                return rBinarySearch(a, x, start, mid - 1); // search a[start] to a[mid-1]
            else if (x > a[mid])
                return rBinarySearch(a, x, mid + 1, end); // search a[mid+1] to a[end]
            else
                return mid; // x found and x = a[mid]
        }
    }
}
12.4 Searching algorithms

```java
/**
 * Search a subarray for a given element using the non-recursive binary search algorithm. It is assumed that the array is sorted in increasing order.
 * @param a The array to search
 * @param x The value to search for
 * @param start Index defining start of subarray
 * @param end Index defining end of subarray
 * @return If <code>x</code> is found the first array index such that <code>x = a[i]</code> else <code>-1</code> to indicate failure.
 */
public static int nrBinarySearch(int[] a, int x, int start, int end)
{
    int low = start;
    int high = end;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (x < a[mid])
            high = mid - 1; // search left half a[low] to a[high-1]
        else if (x > a[mid])
            low = mid + 1; // search right half a[mid+1] to a[high]
        else
            return mid; // x found and x = a[mid]
    }
    return -1; // x not found
}
```

12.4.6 Testing the search algorithms

To test these classes we use the following IntArraySearchTester class:

```java
package chapter12.searching;
import java.util.Scanner;

/**
 * Test the linear and binary searching algorithms.
 */
public class IntArraySearchTester
{
    /** Test the search algorithms.
     */
    public void doTest()
```
Scanner input = new Scanner(System.in);

// read the array

System.out.print("Enter number of integers in array: ");
int size = input.nextInt(); input.nextLine();
int[] testArray = new int[size];
for (int k = 0; k < testArray.length; k++)
{
   System.out.print("Enter element "+ k + ": ");
   testArray[k] = input.nextInt(); input.nextLine();
}

// Read element to find and the subarray start, end indices

System.out.print("Enter element to find: ");
int x = input.nextInt(); input.nextLine();
System.out.print("Enter start index for subarray: ");
int start = input.nextInt(); input.nextLine();
System.out.print("Enter end index for subarray: ");
int end = input.nextInt(); input.nextLine();

// Search the array and display results for each algorithm

int pos;
pos = IntArraySearch.linearSearch(testArray, x, start, end);
displayResult(pos, x);
pos = IntArraySearch.rBinarySearch(testArray, x, start, end);
displayResult(pos, x);
pos = IntArraySearch.nrBinarySearch(testArray, x, start, end);
displayResult(pos, x);

public void displayResult(int pos, int x)
{
   if (pos < 0)
      System.out.println("Element " + x + " was not found");
   else
      System.out.println("Element " + x + " was found at position " + pos);
}

public static void main(String[] args)
{
   new IntArraySearchTester().doTest();
}
12.5 Sorting algorithms

We now consider four well-known sorting algorithms applied to a subarray of an integer array. The first two, selection sort and insertion sort, are quadratic algorithms. Their worst and average case behavior is $O(n^2)$. The other two, quicksort and mergesort, are much superior for large arrays since they have average case behavior $O(n \log n)$. Mergesort also has worst case behavior $O(n \log n)$ but requires a temporary array for storage, and quicksort has worst case behavior $O(n^2)$ but requires no temporary array.

We design our algorithms to sort arrays of integers in increasing order. For example, the array $\langle 5, 3, 8, 4, 2, 2 \rangle$ is not sorted. In increasing order the sorted array is $\langle 2, 2, 3, 4, 5, 5, 8 \rangle$. Later we show how to do generic sorting using arrays of type `Object[]`.

The Java methods for each of these four algorithms has a prototype of the form

```
public static void sort(int[] a, int start, int end)
```

where `a` is the array $\langle a_0, \ldots, a_{n-1} \rangle$ and the subarray $\langle a_{start}, \ldots, a_{end} \rangle$ to sort is defined by `start` and `end`.

12.5.1 Selection sort algorithm

The selection sort algorithm is one of the easiest sorting algorithms to understand because of its intuitive nature. We apply it to the subarray $\langle a_{start}, \ldots, a_{end} \rangle$ of the array $\langle a_0, \ldots, a_{n-1} \rangle$ whose elements have an order defined for them. The algorithm for increasing order is

1. Find the smallest of the elements in $\langle a_{start}, \ldots, a_{end} \rangle$ and exchange (swap) it with the element $a_{start}$ at position `start`. Now $a_{start}$ is the smallest element in the subarray and it is in the correct position.

2. Find the smallest of the elements in the remaining subarray $\langle a_{start+1}, \ldots, a_{end} \rangle$ and exchange (swap) it with the element $a_{start+1}$ at position `start+1`. Now $a_{start}$ and $a_{start+1}$ are the two smallest elements in the original subarray and they are in the correct position ($a_{start} \leq a_{start+1}$).

3. Repeat, using smaller subarrays until the last subarray to sort is $\langle a_{end-1}, a_{end} \rangle$.

This gives the top level pseudo-code algorithm

```
FOR i ← start TO end − 1 DO
    Find the index k of the smallest element in subarray $\langle a_i, \ldots, a_{end} \rangle$
    Exchange (swap) elements at positions i and k.
END FOR
```

As an example, consider the array $\langle a_0, \ldots, a_7 \rangle$ given by $\langle 44, 55, 12, 42, 94, 18, 6, 67 \rangle$ and use `start = 0, end = 7`. The steps are shown in Table 12.5 where underlined elements are in their correct position. The complete pseudo-code algorithm for selection sort is given in Figure 12.6.

Note carefully in the outer for loop that the last value of `i` is `end − 1` since the last subarray to examine is $\langle a_{end-1}, a_{end} \rangle$, but in the inner for loop the last value of `j` is `end` since we must search
**Searching and Sorting Algorithms**

Table 12.4: Selection sort example

<table>
<thead>
<tr>
<th>Step</th>
<th>$a_0$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>$a_5$</th>
<th>$a_6$</th>
<th>$a_7$</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>swap 6 with 44</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>44</td>
<td>67</td>
<td>swap 12 with 55</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
<td>55</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>44</td>
<td>67</td>
<td>swap 18 with 55</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>94</td>
<td>55</td>
<td>44</td>
<td>67</td>
<td>swap 42 with itself</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>94</td>
<td>55</td>
<td>44</td>
<td>67</td>
<td>swap 44 with 94</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>67</td>
<td>swap 55 with itself</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>45</td>
<td>94</td>
<td>67</td>
<td>swap 67 with 94</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
<td>(done)</td>
</tr>
</tbody>
</table>

**ALGORITHM** selectionSort($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, start, end)

FOR $i$ ← start TO end − 1 DO

$k$ ← $i$

FOR $j$ ← $i$ + 1 TO end DO

IF $a_j < a_k$ THEN

$k$ ← $j$

END IF

END FOR

END FOR

$\text{temp} \leftarrow a_k$

$a_k \leftarrow a_i$

$a_i \leftarrow \text{temp}$

END FOR

Figure 12.6: Pseudo-code selection sort algorithm
each subarray until the last element. Also, three assignment statements are needed to exchange (swap) two values, since it is necessary to use a temporary variable to save the first element of the pair being swapped.

As we did for the searching algorithms, it is easy to translate this pseudo-code algorithm into the following static method called selectionSort.

```java
public static void selectionSort(int[] a, int start, int end)
{
    for (int i = start; i < end; i++)
    {
        // find position k of minimum element among
        // the elements a[i] to a[end]

        int k = i;
        for (int j = i+1; j <= end; j++)
        {
            if (a[j] < a[k])
                k = j;
        }

        // swap the smallest element found (it’s a[k]) with a[i]

        int temp = a[k];
        a[k] = a[i];
        a[i] = temp;
    }
}
```

This sort method is placed in a class called IntArraySort (see Section 12.5.13, page 696).

### 12.5.2 Running time for selection sort

One measure of the running time is the number of times the if statement is executed in the inner loop. This is the number of times the loop is executed and this in turn depends on the outer loop index. We can assume that there are \( n \) array elements so \( n = end - start + 1 \). We can use the results in Table 12.5 to do the counting. Adding the entries in the last column gives the running time

\[
T(n) = n - 1 + n - 2 + n - 3 + \cdots + 1 = 1 + 2 + 3 + \cdots + n - 1.
\]

If you don’t know the formula for this common sum write it twice, once forward and once backward, as follows

\[
T(n) = 1 + 2 + 3 + \cdots + n - 1 = n - 1 + n - 2 + n - 3 + \cdots + 1.
\]

Now add to obtain \( 2T(n) = n + n + \ldots + n = n(n - 1) \) since \( n \) is repeated \( n - 1 \) times. Therefore

\[
T(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2).
\]
outer loop index | inner loop index | inner loop executions
---|---|---
start | start + 1, ..., end | end − start = n − 1
start + 1 | start + 2, ..., end | end − start − 1 = n − 2
start + 2 | start + 3, ..., end | end − start − 2 = n − 3
... | ... | ...
end − 1 | end, ..., end | 1

Table 12.5: Counting inner loop executions for selection sort

Since the inner and outer for loop indices do not depend on the actual array elements, according to our definition of the running time, the best, average, and worst case behavior of selection sort are all \( O(n^2) \). Of course the actual running time is sensitive to the distribution of elements in the array since this affects the number of times the assignment statement is executed in the if statement.

### 12.5.3 Insertion sort algorithm

Insertion sort is another intuitive algorithm and is performed by poker or bridge players when they arrange their cards in order. To understand this algorithm think of dividing the subarray to be sorted into two subarrays, a left one called \( L \) which is already sorted and a right one called \( R \) which is unsorted. If the initial subarray is \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \) then its two parts are

\[
L = \langle a_{\text{start}} \rangle, \quad R = \langle a_{\text{start}+1}, \ldots, a_{\text{end}} \rangle
\]

despite a one-element part is always sorted. We begin by taking the first element of \( R \) and moving it to its proper place in the left part to give

\[
L = \langle a_{\text{start}}, a_{\text{start}+1} \rangle, \quad R = \langle a_{\text{start}+2}, \ldots, a_{\text{end}} \rangle
\]

where \( a_{\text{start}} \leq a_{\text{start}+1} \). Next we move \( a_{\text{start}+2} \) to its proper place in \( L \). After several steps we arrive at the general situation

\[
L = \langle a_{\text{start}}, \ldots, a_{i-1} \rangle, \quad R = \langle a_i, \ldots, a_{\text{end}} \rangle
\]

for which \( a_{\text{start}} \leq a_{\text{start}+1} \leq \cdots \leq a_{i-1} \). Rather than moving \( a_i \) left, by swapping, until it is in its proper position in \( L \) we can move \( a_{i-1} \) to the right one place if it is larger than \( a_i \), \( a_{i-2} \) to the right one place if it is larger than \( a_{i-1} \) and so on until we obtain a “hole” in \( L \) into which we can drop \( a_i \). This “move and drop” technique is more efficient than swapping since it involves less data movement.

The following example illustrates this for the array \( A = \langle 44, 55, 12, 42, 94, 18, 6, 67 \rangle \). Starting with \( L = \langle 44 \rangle \) and \( R = \langle 55, 12, 42, 94, 18, 6, 67 \rangle \) we see that 55 is already in its correct position with respect to 44 so the next step is \( L = \langle 44, 55 \rangle \) and \( R = \langle 12, 42, 94, 18, 6, 67 \rangle \). Continuing we obtain Figure 12.7. The vertical bars indicate the division between the subarrays \( L \) and \( R \) at each step. The complete pseudo-code algorithm for insertion sort is given in Figure 12.8. At step \( i \) the sorted part is \( L = \langle a_{\text{start}}, \ldots, a_{i-1} \rangle \) and the unsorted part is \( R = \langle a_i, \ldots, a_{\text{end}} \rangle \). In this step a hole for \( x = a_i \) is opened up by moving \( a_{i-1}, a_{i-2}, \ldots \) to the right in the while loop if they are greater than \( x \).

We can translate this pseudo-code algorithm to the following method called `insertionSort` which will later be placed in the `IntArraySort` class along with the other sorting algorithm.
12.5 Sorting algorithms

<table>
<thead>
<tr>
<th>step</th>
<th>array</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44</td>
<td>55 12 42 94 18 6 67</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>55 12 42 94 18 6 67</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>44 55 42 94 18 6 67</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>42 44 55 94 18 6 67</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>42 44 55 94 18 6 67</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18 42 44 55 94 6 67</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>12 18 42 44 55 94 67</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>12 18 42 44 55 67 94</td>
</tr>
</tbody>
</table>

Figure 12.7: An insertion sort example for \((44, 55, 12, 42, 94, 18, 6, 67)\).

```
ALGORITHM insertionSort((a₀, a₁, ..., aₙ₋₁), start, end)
FOR i ← start + 1 TO end DO
    x ← aᵢ
    j ← i - 1
    WHILE j ≥ start ∧ x < aⱼ DO
        aⱼ₊₁ ← aⱼ
        j ← j - 1
    END WHILE
    aⱼ₊₁ ← x
END FOR
```

Figure 12.8: Pseudo-code insertion sort algorithm

```java
public static void insertionSort(int[] a, int start, int end)
{
    for (int i = start+1; i <= end; i++)
    {
        // Sorted part of array is a[start], ..., a[i-1]
        // Unsorted part is a[i], ..., a[end]

        int x = a[i]; // left element of unsorted part
        int j = i-1;  // right index of sorted part

        // move elements right until position for x is found.
        while ( (j >= start) && (x < a[j]) )
        {
            a[j+1] = a[j]; // move a[j] one place to the right
            j--;           // move a[j] one place to the right
        }
    }
}
```
### Table 12.6: Counting inner loop executions for worst case insertion sort

<table>
<thead>
<tr>
<th>outer loop index</th>
<th>inner loop index</th>
<th>inner loop executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>start + 1</td>
<td>start, . . . , start</td>
<td>1</td>
</tr>
<tr>
<td>start + 2</td>
<td>start, . . . , start + 1</td>
<td>2</td>
</tr>
<tr>
<td>start + 3</td>
<td>start, . . . , start + 1</td>
<td>3</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>end</td>
<td>start, . . . , end – 1</td>
<td>end – start</td>
</tr>
</tbody>
</table>

This sort method is placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

#### 12.5.4 Running time for insertion sort

For this algorithm let us use the number of times the statements inside the while loop are executed to determine the running time. This is a more difficult problem than for selection sort since the number of times the while loop is executed depends on the array. Therefore we will calculate the worst case behavior. The inner loop will be executed the maximum number of times if \( j \) is decremented to the beginning of the array each time. This occurs when the array is sorted in reverse order. For example, \( \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle \). In this case we obtain the results in Table 12.6 again using \( n = end – start + 1 \). Adding the entries in the last column gives

\[
T(n) = 1 + 2 + \cdots + end – start = \frac{n(n – 1)}{2} = \mathcal{O}(n^2).
\]

Clearly the worst case running time is \( \mathcal{O}(n^2) \). It can be shown that the average time is also \( \mathcal{O}(n^2) \). The best case behavior occurs when the initial array is already sorted. Then the statements in the while loop are never executed. Since the outer loop is executed \( n – 1 \) times the best case behavior is \( \mathcal{O}(n) \).

#### 12.5.5 Simulation to compare selection and insertion sort

Simulations are important in the analysis of algorithms and computer systems since it is not always possible to obtain good theoretical estimates. Also it may be necessary to optimize an algorithm for a particular computer system. Even though we can analyze selection sort and insertion sort and determine their average behavior is \( \mathcal{O}(n^2) \), it is useful to compare them on a given computer system to see which is faster. Since the worst case behavior of insertion sort compares with the average case behavior of selection sort we expect that insertion sort is a little faster even though both algorithms are \( \mathcal{O}(n^2) \) (a more careful theoretical analysis confirms this).

It is easy to write a simulation program to estimate the running time of a sorting algorithm. We simply generate random arrays and average the times over a certain number of trials. We can use the `Random` class in `java.util` to generate random integers. For example, if we define...
Random rand = new Random(1234);

then each call rand.nextInt() generates the next random number in the sequence determined by the seed 1234. Each seed generally produces a different sequence of integers. To time the selection sort algorithm we can use the statements

```java
long startTime = System.nanoTime();
IntArraySort.selectionSort(array, 0, array.length - 1);
long endTime = System.nanoTime();
double sortTime = (endTime - startTime) / 1E9; // seconds
```

Here is a class that can be used to time the selection sort and insertion sort algorithms algorithm. Note that we have run one extra trial and not counted the first one since the Java compiler will do some "just in time" compiling after the first run of a block of code:

```java
package chapter12.sorting;
import java.util.Scanner;
import java.util.Random;

public class QuadraticSortTimer {

    public void doTest() {
        Scanner input = new Scanner(System.in);
        String sortType = "";

        // Get the sorting algorithm

        System.out.println("Selection sort: 1");
        System.out.println("Insertion sort: 2");
        System.out.println("Enter 1 or 2");
        int choice = input.nextInt(); input.nextLine();

        // Get size of array and number of trials to average

        System.out.print("Enter size of array: ");
        int size = input.nextInt(); input.nextLine();
        System.out.print("Enter number of trials to average: ");
```
```
int numTrials = input.nextInt(); input.nextLine();
Random rand = new Random(1234);
int[] array = new int[size];

double timeSum = 0.0;
for (int trial = 1; trial <= numTrials + 1; trial++)
{
    // fill array with random integers
    for (int k = 0; k < array.length; k++)
        array[k] = rand.nextInt();

    // compute the time for this trial
    long endTime = 0L, startTime = 0L;
    if (choice == 1)
    {
        sortType = "Selection sort";
        startTime = System.nanoTime(); // nanoseconds
        IntArraySort.selectionSort(array, 0, array.length - 1);
        endTime = System.nanoTime();
    }
    else
    {
        sortType = "Insertion sort";
        startTime = System.nanoTime(); // nanoseconds
        IntArraySort.insertionSort(array, 0, array.length - 1);
        endTime = System.nanoTime();
    }

    if (trial > 1) // skip first trial (do numTrials trials)
    {
        double sortTime = (endTime - startTime)/ 1E9; // seconds
        System.out.println("Sort time: " + sortTime + " seconds");
        timeSum = timeSum + (double) sortTime;
    }
}

// Compute average time over all trials
double averageTime = timeSum / (double) numTrials;

// Display the results
System.out.println();
System.out.println(sortType);
System.out.println("Array size (n): "+ size);
System.out.println("Number of trials: " + numTrials);
System.out.println("Average running time: " + averageTime + " seconds");
double nSquared = (double) size * (double) size;
double ratio = averageTime / nSquared;
System.out.println("average / n^2: " + ratio);
```
This class can be used to calculate the average running time of these algorithms. This is done by generating random arrays of integers, calculating the time it takes to sort them, adding up the times and dividing by the total number of trials, and displaying the final result for the average time.

Since the $O(n^2)$ algorithms have running time proportional to $n^2$ for large $n$ it would also be useful to estimate the constant $a$ such that

$$T(n) = an^2,$$

approximately for large $n$.

We can do this by displaying the ratio $T(n)/n^2$, which should be a good estimate of the constant $a$ for large $n$.

Some results of running this programs on a particular computer system are shown in Table 12.7. To estimate the average running time 10 trials were averaged. If we didn’t know that both of these algorithms were $O(n^2)$ we could use this data to check it empirically. For selection sort the ratio $T(n)/n^2$ varies from $1.64 \times 10^{-8}$ to $2.13 \times 10^{-8}$. These values are approximately constant so we have the empirical formula

$$T(n) = 2.1 \times 10^{-8} n^2,$$

approximately for large $n$

using $2.1 \times 10^{-8}$ as an estimate for the constant $a$. Similarly for insertion sort the ratio $T(n)/n^2$ varies from $1.01 \times 10^{-8}$ to $1.25 \times 10^{-8}$. This gives the empirical formula

$$T(n) = 1.2 \times 10^{-8} n^2,$$

approximately for large $n$

In this range insertion sort is almost twice as fast as selection sort on this computer system.

<table>
<thead>
<tr>
<th>size $n$</th>
<th>selection sort</th>
<th>$T(n)/n^2$</th>
<th>insertion sort</th>
<th>$T(n)/n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.41 sec</td>
<td>$1.64 \times 10^{-8}$</td>
<td>0.25 sec</td>
<td>$1.01 \times 10^{-8}$</td>
</tr>
<tr>
<td>10,000</td>
<td>2.07 sec</td>
<td>$2.07 \times 10^{-8}$</td>
<td>1.02 sec</td>
<td>$1.02 \times 10^{-8}$</td>
</tr>
<tr>
<td>20,000</td>
<td>8.50 sec</td>
<td>$2.12 \times 10^{-8}$</td>
<td>4.36 sec</td>
<td>$1.09 \times 10^{-8}$</td>
</tr>
<tr>
<td>30,000</td>
<td>19.5 sec</td>
<td>$2.16 \times 10^{-8}$</td>
<td>10.0 sec</td>
<td>$1.11 \times 10^{-8}$</td>
</tr>
<tr>
<td>40,000</td>
<td>34.7 sec</td>
<td>$2.17 \times 10^{-8}$</td>
<td>18.8 sec</td>
<td>$1.17 \times 10^{-8}$</td>
</tr>
<tr>
<td>50,000</td>
<td>53.6 sec</td>
<td>$2.15 \times 10^{-8}$</td>
<td>29.0 sec</td>
<td>$1.16 \times 10^{-8}$</td>
</tr>
<tr>
<td>100,000</td>
<td>213.0 sec</td>
<td>$2.13 \times 10^{-8}$</td>
<td>125.0 sec</td>
<td>$1.25 \times 10^{-8}$</td>
</tr>
</tbody>
</table>

Table 12.7: Comparing average times for selection sort and insertion sort.
12.5.6 Mergesort

Mergesort is a naturally recursive algorithm for sorting a subarray by dividing it into two halves. If the initial subarray is \( (a_{start}, \ldots, a_{end}) \) then the two halves are

\[
L = (a_{start}, \ldots, a_{mid}), \quad R = (a_{mid+1}, \ldots, a_{end})
\]

where \( mid = (start + end) / 2 \). Like the recursive binary search algorithm it is a “divide and conquer” algorithm. We apply mergesort to each half (smaller versions of the problem) and then we assume that there is some function called \( \text{merge} \) that will merge the two sorted halves together into a sorted subarray. The base case for the recursion occurs if \( start = end \) since a one element subarray is already sorted. This gives the following simple Java method:

```java
public static void mergeSort(int[] a, int start, int end)
{
    if (start == end) // one-element subarray is already sorted
        return;
    int mid = (start + end) / 2;
    mergeSort(a, start, mid); // mergesort the left half
    mergeSort(a, mid+1, end); // mergesort the right half
    merge(a, start, mid, end); // merge the two halves
}
```

Before we consider how to solve the merge problem let us assume that the \( \text{merge} \) method has been written and look at the recursive process. The two recursive calls to \( \text{mergeSort} \) simply divide the array into smaller and smaller subarrays until they have only one element. Then merge recombines each pair of sorted subarrays into a larger sorted subarray. An example for the array \( \langle 8, 1, 6, 4, 10, 5, 3, 2, 22 \rangle \) is shown in Figure 12.9. The top half of the diagram shows how the recursive calls break the array into subarrays. The first value of \( mid \) is \((0 + 8) / 2 = 4\) so we get the subarrays \( \langle 8, 1, 6, 4, 10 \rangle \) and \( \langle 5, 3, 2, 22 \rangle \). The second subarray is not generated until the left subarray has been fully subdivided in the left part of the diagram. The top half of the diagram shows how the recursive process divides the initial array into one element subarrays and the bottom half shows how the merge method recombines these subarrays to eventually produce the original array in sorted form. For example, the first merge step on the left combines the one-element subarrays \( \langle 8 \rangle \) and \( \langle 1 \rangle \) to obtain the sorted subarray \( \langle 1, 8 \rangle \) and the last merge step combines the sorted subarrays \( \langle 1, 4, 6, 8, 10 \rangle \) and \( \langle 2, 3, 5, 22 \rangle \) to give the sorted array \( \langle 1, 2, 3, 4, 5, 6, 8, 10, 22 \rangle \).

12.5.7 Merge algorithm for two sorted subarrays

The merge problem is interesting by itself. We can state it for arrays as follows:

“Given a subarray \( (a_{start}, \ldots, a_{end}) \) partitioned by an index \( \text{split} \) with \( start \leq \text{split} \leq end \), such that the subarrays \( (a_{start}, \ldots, a_{\text{split}}) \) and \( (a_{\text{split}+1}, \ldots, a_{end}) \) are each sorted in increasing order, sort the entire subarray \( (a_{start}, \ldots, a_{end}) \) into a temporary array \( (t_0, \ldots, t_{end-\text{start}+1}) \) such that \( t_0 \leq t_1 \leq \ldots \leq t_{end-\text{start}+1} \)”
Figure 12.9: Mergesort example for the array \(\langle 8, 1, 6, 4, 10, 5, 3, 2, 22 \rangle\)
Figure 12.10: Merging the subarrays \( \langle 1, 8, 12, 15, 17 \rangle \) and \( \langle 2, 9, 10, 19, 21, 23, 25 \rangle \) to obtain the sorted subarray \( \langle 1, 2, 8, 9, 10, 12, 15, 17, 19, 21, 23, 25 \rangle \).

For example, the unsorted array \( \langle 1, 8, 12, 15, 17, 2, 9, 10, 19, 21, 23, 25 \rangle \) with \( \text{split} = 4 \) is such that the left subarray \( \langle 1, 8, 12, 15, 17 \rangle \) and the right subarray \( \langle 2, 9, 10, 19, 21, 23, 25 \rangle \) are each sorted. We want to obtain the sorted subarray \( \langle 1, 2, 8, 9, 10, 12, 15, 17, 19, 21, 23, 25 \rangle \). The temporary array will be \( \langle t_0, \ldots, t_{11} \rangle \). The merge process for this example is quite simple, as shown in Figure 12.10. Here the left subarray is shown vertically on the left and the right subarray is shown vertically on the right. The temporary array is shown in the center. The arrows indicate the data movement and the numbers above the horizontal lines indicate the step number.

The algorithm simply compares the current element from the left subarray with the current element from the right subarray and moves the smaller into the current position in the temporary array. First 1 is compared with 2 so 1 is moved into the first position in the temporary array. Then the next element, 8, of the first subarray is compared with 2 so 2 is moved into the next position in the subarray. This process continues until one or both subarrays are exhausted. In the example this occurs first for the left array at step 8. Then the remaining elements 19, 21, 23, and 25 in the right subarray are copied to the temporary array. Finally the temporary array is copied back to the original array. A top level pseudo-code algorithm is shown in Figure 12.11.

Since the first while loop will exhaust at least one of the left or right subarrays, only one of the remaining while loops will be executed to copy any remaining elements. The running time of merge is \( O(n) \).

To write a Java method for this algorithm we need three index variables, \( i \) initialized to start indexes elements of the left subarray, \( j \) initialized to \( \text{split} + 1 \) indexes elements of the right subarray, and \( k \) initialized to 0 indexes elements of the temporary array \( t \). Assuming these initial-
12.5 Sorting algorithms

**ALGORITHM** merge(⟨a₀,a₁,...,aₙ₋₁⟩, start, split, end)

\[
t ← ⟨t₀,...,t_{end−start+1}⟩
\]

**WHILE** neither ⟨a_start,...,a_split⟩ nor ⟨a_split+1,...,a_end⟩ is exhausted **DO**

- Compare pairs of elements and copy smallest to temp array \( t \)
- \( t ← ⟨t₀,...,t_{end−start+1}⟩ \)

**END WHILE**

**WHILE** elements remain in ⟨a_start,...,a_split⟩ **DO**

- Copy them to \( t \)

**END WHILE**

**WHILE** elements remain in ⟨a_split+1,...,a_end⟩ **DO**

- Copy them to \( t \)

**END WHILE**

⟨a_start,...,a_end⟩ ← ⟨t₀,...,t_{end−start+1}⟩

---

Figure 12.11: Pseudo-code merge algorithm for two subarrays

izations the first while loop can be expressed as

```java
while (i <= split && j <= end)
{
    if (a[i] < a[j]) // element in left subarray is smaller
    {
        t[k] = a[i]; // move it to temp array t
        i++; // index next element in left subarray
    }
    else // element in right subarray is smaller
    {
        t[k] = a[j]; // move it to temp array t
        j++; // index next element in right subarray
    }
    k++; // in either case index next position in t
}
```

After this loop executes either both subarrays are exhausted (if they have the same size) or there are elements in one of them that remain to be copied. The following loop will execute only if there are elements remaining in the left subarray:

```java
while (i <= split)
{
    t[k] = a[i];
    k++;
    i++;
}
```

and the following loop will execute only if there are elements remaining in the right subarray:

```java
while (j <= end)
```
Finally the temporary array can be copied back to the original array using the for loop

\[
\text{for } (k = 0; k < \text{end} - \text{start} + 1; k++) \\
\text{a[start + k]} = t[k];
\]

This can be done a little more efficiently using the special `arraycopy` method in the `System` class:

```
System.arraycopy(t, 0, a, start, end - start + 1);
```

Later the `mergeSort` and `merge` methods will be placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

```java
public static void mergeSort(int[] a, int start, int end)
{
    if (start == end) // one-element subarray is already sorted
        return;
    int mid = (start + end) / 2;
    mergeSort(a, start, mid); // merge sort left half
    mergeSort(a, mid+1, end); // merge sort right half
    merge(a, start, mid, end); // merge the two halves
}

public static void merge(int[] a, int start, int split, int end)
{
    int n = end - start + 1; // number of elements to merge
    int[] t = new int[n]; // temporary storage required
    int i = start; // index of elements in left subarray
    int j = split + 1; // index of elements in right subarray
    int k = 0; // index into temporary storage

    // merge elements from left and right subarray to temp array
    // until one or both of the subarrays are exhausted
    while (i <= split && j <= end)
    {
        if (a[i] < a[j]) // element in left subarray is smaller
        {
            t[k] = a[i]; // move it to temp array t
            i++; // index next element in left subarray
        }
        else // element in right subarray is smaller
        {  
```
12.5 Sorting algorithms

```java
  t[k] = a[j]; // move it to temp array t
  j++;
  } // index next element in right subarray
  k++; // in either case index next position in t
}

// copy any remaining elements from left subarray to t
while (i <= split)
{
  t[k] = a[i];
  i++;
  k++;
}

// copy any remaining elements from right subarray to t
while (j <= end)
{
  t[k] = a[j];
  j++;
  k++;
}

// copy elements to a from temporary array t. Can also use
// System.arraycopy(t, 0, a, start, end-start+1);
for (k = 0; k < n; k++)
  a[start+k] = t[k];
```

If you like compact code the three loops in the merge method can be expressed as

```java
while (i <= split && j <= end)
  if (a[i] < a[j]) t[k++] = a[i++];
  else t[k++] = a[j++]
while (i <= split) t[k++] = a[i++];
while (j <= end) t[k++] = a[j++];
for (k=0; k < n; k++) a[start+k] = t[k];
```

Here we make use of the fact that an expression like \(a[i++]\) means to first use \(i\) as an index and then increment it. Therefore an assignment such as

```java
t[k++] = a[i++];
```

is equivalent to the three assignment statements

```java
t[k] = a[i];
k = k + 1;
i = i + 1;
```
It should also be noted that \(a[i++]\) is quite different from \(a[++i]\) which increments \(i\) before using it as an array index.

### 12.5.8 Running time for mergesort

The calculation of the running time for mergesort is similar to the previous calculation for the recursive binary search. Let \(T(n)\) be the running time for a subarray of size \(n\). Then the `mergeSort` method shows that

\[
T(n) = T(n/2) + T(n/2) + \text{time to do merge}
\]

The merge algorithm is \(O(n)\) since each while loop executes in \(O(n)\) time. The number of times each loop is executed depends only on the array size and not the actual array elements. Therefore let us assume that its running time is \(an + b\) and that \(n = 2^m\) for some \(m\). Then

\[
T(n) = 2T(n/2) + an + b
= 2 \left[ 2T(n/2^2) + a\frac{n}{2} + b \right] + an + b
= 2^2 T(n/2^2) + 2an + (1 + 2)b
= 2^2 \left[ 2T(n/2^3) + a\frac{n}{2^2} + b \right] + 2an + (1 + 2)b
= 2^3 T(n/2^3) + 3an + (1 + 2 + 2^2)b
\]

\[...
= 2^m T(n/2^m) + am + (1 + 2 + \cdots + 2^m - 1)b
= 2^m T(n/2^m) + am + (2^m - 1)b
\]

Substituting \(n = 2^m\) and using \(T(1) = 1\) we obtain

\[
T(n) = n + an \log_2 n + (n - 1)b
= O(n \log_2 n)
= O(n \log n)
\]

for any logarithmic base. It can be shown that the best, worst, and average behavior are all \(O(n \log n)\) for mergesort. The only disadvantage is that a temporary array is required, of the same size as the subarray to be sorted.

So far we have seen algorithms that are \(O(f(n))\) for \(f(n) = \log n\), \(f(n) = n\), \(f(n) = n \log n\), and \(f(n) = n^2\), in order of fastest increase as \(n \to \infty\), as shown in Table [12.8]

### 12.5.9 File merge example

The merge part of mergesort is useful by itself. For example, it can be used to merge two sorted files into a single larger sorted file. The algorithm in this case is simpler than the array merge used in mergesort. No temporary storage is required. We simply read a record from each file, compare the fields that define the order of the records, and write the smaller record to the output file. This
## 12.5 Sorting algorithms

Table 12.8: Growth rates for $\log_2 n$, $n$, $n \log_2 n$, and $n^2$.

<table>
<thead>
<tr>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.32</td>
<td>10</td>
<td>33.22</td>
<td>$10^4$</td>
</tr>
<tr>
<td>6.64</td>
<td>$10^2$</td>
<td>664.39</td>
<td>$10^4$</td>
</tr>
<tr>
<td>9.97</td>
<td>$10^3$</td>
<td>$9.97 \times 10^3$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>13.29</td>
<td>$10^4$</td>
<td>$1.33 \times 10^5$</td>
<td>$10^8$</td>
</tr>
<tr>
<td>16.61</td>
<td>$10^5$</td>
<td>$1.66 \times 10^6$</td>
<td>$10^{10}$</td>
</tr>
<tr>
<td>19.93</td>
<td>$10^6$</td>
<td>$1.99 \times 10^7$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>23.25</td>
<td>$10^7$</td>
<td>$2.33 \times 10^8$</td>
<td>$10^{14}$</td>
</tr>
<tr>
<td>26.58</td>
<td>$10^8$</td>
<td>$2.66 \times 10^9$</td>
<td>$10^{16}$</td>
</tr>
<tr>
<td>29.90</td>
<td>$10^9$</td>
<td>$2.99 \times 10^{10}$</td>
<td>$10^{18}$</td>
</tr>
</tbody>
</table>

continues until one or both files are exhausted. Then if there are any records remaining in one of the input files they are written to the output file.

As an example consider files of `BankAccount` objects, using the single-line colon-separated format discussed in Chapter 12, that are sorted in order of increasing account number. For example, if the first file is `accounts1.dat`:

175:Linda Kerr:8008.43
424:Mary Barber:35135.5
932:Harry Garfield:4723.1
1134:Alfred Vaillancourt:51914.93
1345:Amy Flintstone:81507.31
2489:Barney Lafreniere:66568.5
7123:Jean Ebert:53361.25
7845:Marc Gardiner:7541.32
9243:Dan Sinclair:4151.34
9546:Peter Jensen:16146.29

and the second file is `accounts2.dat`:

310:Don Laing:12337.39
417:Marc Tyler:3455.42
811:Amanda Schryer:3541.15
1219:Gilles Olivier:84285.23
1765:Patricia Innes:4494.43
9623:Remi Martin:19732.12
9754:Patricia Schneider:40066.76
9912:Mike Laforge:5798.0

then the merged output file is

175:Linda Kerr:8008.43
310:Don Laing:12337.39
417:Marc Tyler:3455.42
424: Mary Barber: 35135.5
811: Amanda Schryer: 3541.15
932: Harry Garfield: 4723.1
1134: Alfred Vaillancourt: 51914.93
1219: Gilles Olivier: 84285.23
1345: Amy Flintstone: 81507.31
1765: Patricia Innes: 4494.43
2489: Barney Lafreniere: 66568.5
7123: Jean Ebert: 53361.25
7845: Marc Gardiner: 7541.32
9243: Dan Sinclair: 4151.34
9546: Peter Jensen: 16146.29
9623: Remi Martin: 19732.12
9754: Patricia Schneider: 40066.76
9912: Mike Laforge: 5798.0

Here is a class that does this file merge using command line arguments for the names of the two input files and the merged output file.

```java
package chapter12.merge;
import custom_classes.BankAccount;
import java.io.IOException;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;

/**<n
   * This class uses the merge algorithm to read two bank account
   * files that are each sorted in order of increasing account number
   * and merge them into a single sorted file.
   */
public class FileMerger
{
    private File inFile1;
    private File inFile2;
    private File outFile;

    /** Create an object for given filenames.
     * @param inFile1 the first input file object
     * @param inFile2 the second input file object
     * @param outFile the output file object
     */
    public FileMerger(File inFile1, File inFile2, File outFile)
    {
        this.inFile1 = inFile1;
        this.inFile2 = inFile2;
    }
```
/** Merge the two input files according to increasing account number. * @throws IOException */
public void mergeFiles() throws IOException {
    BufferedReader in1 =
        new BufferedReader(new FileReader(inFile1));
    BufferedReader in2 =
        new BufferedReader(new FileReader(inFile2));
    PrintWriter out =
        new PrintWriter(new FileWriter(outFile));

    // Read records until end of file is reached on one or both files
    // and merge them to the output file.

    BankAccount b1 = in1.readAccount(); // get account from 1st file
    BankAccount b2 = in2.readAccount(); // get account from 2nd file

    while (b1 != null && b2 != null)
    {
        if (b1.getNumber() < b2.getNumber())
        {
            out.writeAccount(b1); // write account from 1st file
            b1 = in1.readAccount(); // read next account from 1st file
        }
        else
        {
            out.writeAccount(b2); // write account from 2nd file
            b2 = in2.readAccount(); // read next account from 2nd file
        }
    }

    // If there are records remaining in 1st file write them to output file.

    while (b1 != null)
    {
        out.writeAccount(b1);
        b1 = in1.readAccount();
    }

    // If there are records remaining in 2nd file write them to output file.

    while (b2 != null)
    {
        out.writeAccount(b2);
        b2 = in2.readAccount();
    }

    in1.close();
public static void main(String[] args) throws IOException
{
    if (args.length == 3)
    {
        File inFile1 = new File(args[0]);
        File inFile2 = new File(args[1]);
        File outFile = new File(args[2]);
        FileMerger merger = new FileMerger(inFile1, inFile2, outFile);
        merger.mergeFiles();
    }
    else
    {
        System.out.println("args: inFileName1 inFileName2 outFileName");
        System.exit(1);
    }
}

12.5.10 Quicksort

The “divide and conquer” method used in binary search and mergesort is also the basis of the quicksort algorithm. Dividing is easy for binary search. The middle element is always used, an $O(1)$ operation, and the dividing is an $O(\log n)$ operation. This gives the $O(\log n)$ running time. For mergesort the middle element is also used. Here the merging is an $O(n)$ operation and again the dividing is $O(\log n)$. This gives the $O(n \log n)$ running time.

For the quicksort algorithm the dividing is $O(\log n)$ but it is based on a partitioning method for dividing the array into two parts determined by choosing a pivot element. Like merge, this partitioning is an $O(\log n)$ operation but the partitioning operation is not intuitive at all compared to the one defined by merge. For quicksort the two parts are not necessarily of equal size. In fact it is possible to have one part contain one element and the other part to contain the remaining elements at each stage in the recursive division process. This becomes an $O(n^2)$ operation and is the reason that quicksort has an $O(n \log n)$ average case behavior but only an $O(n^2)$ worst case behavior.

Partitioning an array

To understand quicksort we first define a partition of the subarray $\langle a_{start}, \ldots, a_{end} \rangle$ with respect to some arbitrary element $x$ in this subarray called the pivot element as

$$L = \langle a_{start}, \ldots, a_{split} \rangle, \quad R = \langle a_{split+1}, \ldots, a_{end} \rangle, \quad x = a_{split}$$

where the left and right subarrays are such that each element $y$ in $L$ satisfies $y \leq a_{split}$ and each element $y$ in $R$ satisfies $y \geq a_{split}$. Thus, every element in $L$ is less than or equal to the pivot element and every element in $R$ is greater than or equal to the pivot element. We have included the pivot
element in $L$ for convenience, although it is in its correct position and will not be included in $L$ in the recursive algorithm. Unlike mergesort this does not mean that $L$ and $R$ are each sorted and the pivot element does not have to be the middle element of the subarray.

For example, consider the array $\langle 8, 1, 6, 4, 10, 5, 3, 2, 22 \rangle$ and let us choose the middle element 10 as the pivot at each stage defined by the position $\text{mid} = (\text{start} + \text{end})/2$. Then an initial partition is defined by $L = \langle 2, 1, 6, 4, 8, 5, 3, 10 \rangle$, $R = \langle 22 \rangle$. Each element in $L$ is less than or equal to every element in $R$. Since $R$ has only one element it is already sorted. We now partition $L$ using its middle element 4 as pivot. This gives the subarrays $\langle 3, 1, 2, 4 \rangle$ and $\langle 8, 5, 6, 10 \rangle$. Continuing we obtain the results shown in Figure 12.12. Since the partitioning is done in place, then 1 is in position 0, 2 is in position 1 and so on until we arrive at 22 in position 8. If we read the one-element arrays from left to right we see that the array is now sorted.

Given an algorithm for partitioning a subarray, it is easy to quicksort the array recursively: partition the subarray into the $L$ and $R$ subarrays and recursively quicksort $L$ and $R$ until any remaining subarrays have one element (base case). Because of the defining property of a partition the result will be a sorted array. Therefore the recursive quicksort method for a subarray $\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle$ has the top level recursive structure given by the Java method

```java
public void quickSort(int[] a, int start, int end)
{
    if (start < end)
    {
        int split = partition(a, start, end);
        quickSort(a, start, split-1); // sort left part a[start] to a[split-1]
        quickSort(a, split+1, end); // sort right part a[split+1] to a[end]
    }
}
```
where the partition method returns the index, split, defining the two parts of the subarray. Since the pivot is in its correct position it is not included in the subarrays for either of the recursive calls to sort.

An implementation of partition

Designing an algorithm to perform partitioning is not easy. Essentially we need to scan the array from left to right and swap elements that are in the wrong subarray. There are several versions. Some use two index pointers, one beginning at the left and moving to the right and the other beginning at the right and moving to the left. The elements at these positions are compared and swapped if necessary. The simplest version uses only one index and can be written in Java as

```java
int partition(int a[], int start, int end)
{
    // choose middle element as pivot and move it to
    // the start of the subarray temporarily.
    swap(a, (start + end)/2, start);
    int pivot = a[start];

    // partition the elements a[start+1] to a[end].
    // lastLeft is the index of the last element in the left subarray.
    // The elements a[start] to a[lastLeft] are
    // less than or equal to the pivot value.
    int lastLeft = start;
    for (int j = start+1; j <= end; j++)
    {
        if (a[j] < pivot)
        {
            lastLeft++;  // move partition index left
            swap(a, j, lastLeft);  // and swap element there with a[j]
        }
    }

    // move the pivot back to its correct position
    swap(a, start, lastLeft);
    return lastLeft;
}
```

where the swap method is defined as

```java
void swap(int[] a, int i, int j)
{
```
Initially the pivot element is swapped with the first element of the subarray. Then the index pointer, `lastLeft`, is initialized. It will move to the right and at the end of the loop will be the correct position of the pivot element. The last step is to swap the pivot into this position.

As an example consider partitioning the array \(\langle 2, 1, 6, 4, 8, 5, 3, 10 \rangle\), using the pivot element 4, to obtain \(\langle 3, 1, 2, 4, 8, 5, 6, 10 \rangle\) which gives the partition \(L = \langle 3, 1, 2, 4 \rangle, R = \langle 8, 5, 6, 10 \rangle\). The steps are shown in Figure 12.13. Here the first line of the table shows the initial array. The next line shows the array after the pivot element 4 in position 3 is swapped with the element, 2, in position 0. The next 7 lines correspond to the seven iterations of the for loop at the end of each iteration showing only elements that have been swapped. The last line corresponds to swapping the pivot element to its correct position given by the final value of `lastLeft` to obtain the partition \(\langle 3, 1, 2, 4 \rangle\) and \(\langle 8, 5, 6, 10 \rangle\).

The quickSort and partition methods are placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

### 12.5.11 Running time for quicksort

Since the partition algorithm is \(O(n)\) and the dividing should be \(O(\log n)\) as in mergesort, we expect quicksort to be an \(O(n \log n)\) algorithm. On average this is true, as shown below, but there are cases where the algorithm degenerates to \(O(n^2)\). This occurs when the partitions at each step have one element in one part and the remaining elements in the other part. For example, since we are using the middle element as the pivot at each step, suppose the pivot is always the smallest element in the subarray. Then, since the elements of the left part are always less than or equal to the pivot, we will always have a left part with just one element (assuming there are no duplicates among the array elements). In this worst case the partitioning and the division processes are each
\(O(n)\) so quicksort becomes an \(O(n^2)\) algorithm.

The analysis of the average running time is the same as for mergesort. On average we expect that the partitioning of an \(n\) element subarray produces left and right parts with \(n/2\) elements in each. This gives the recurrence relation

\[
T(n) = T(n/2) + T(n/2) + \text{time to partition} = 2T(n/2) + an + b
\]

This is the same recurrence relation obtained for mergesort so the average case behavior of quicksort is also \(O(n \log n)\).

12.5.12 Simulation to compare mergesort and quicksort

We have compared the running times of the \(O(n^2)\) selection and insertion sort algorithms and we now do the same for the \(O(n \log n)\) algorithms, mergesort and quicksort, using the following class:

```
package chapter12.sorting;
import java.util.Scanner;
import java.util.Random;

/**
 * Estimate the average time of the merge and quick
 * sort algorithms using a given array size and number of trials
 * to average
 */
public class FasterSortTimer
{
    /**
     * Perform a sorting simulation to compare insertion
     * sort and selection sort and display results.
     */
    public void doTest()
    {
        Scanner input = new Scanner(System.in);
        String sortType = "";

        // Get the sorting algorithm
        System.out.println("Merge sort: 1");
        System.out.println("Quick sort: 2");
        System.out.println("Enter 1 or 2");
        int choice = input.nextInt(); input.nextLine();

        // Get size of array and number of trials to average
        System.out.print("Enter size of array: ");
```
int size = input.nextInt(); input.nextLine();
System.out.print("Enter number of trials to average: ");
int numTrials = input.nextInt(); input.nextLine();
Random rand = new Random(1234);
int[] array = new int[size];

double timeSum = 0.0;
for (int trial = 1; trial <= numTrials + 1; trial++)
{
    // fill array with random integers
    for (int k = 0; k < array.length; k++)
        array[k] = rand.nextInt();

    // compute the time for this trial
    long endTime = 0L, startTime = 0L;
    if (choice == 1)
    {
        sortType = "Merge sort";
        startTime = System.nanoTime(); // nanoseconds
        IntArraySort.mergeSort(array, 0, array.length - 1);
        endTime = System.nanoTime();
    }
    else
    {
        sortType = "Quick sort";
        startTime = System.nanoTime(); // nanoseconds
        IntArraySort.quickSort(array, 0, array.length - 1);
        endTime = System.nanoTime();
    }

    if (trial > 1) // skip first trial (do numTrials trials)
    {
        double sortTime = (endTime - startTime) / 1E9; // seconds
        System.out.println("Sort time: "+sortTime + " seconds");
        timeSum = timeSum + (double) sortTime;
    }
}

// Compute average time over all trials

double averageTime = timeSum / (double) numTrials;

// Display the results
System.out.println();
System.out.println(sortType);
System.out.println("Array size (n): "+size);
System.out.println("Number of trials: "+numTrials);
System.out.println("Average running time: "+averageTime + " seconds");
double nLogn = size * Math.log(size);
Some results of running these programs on the same computer are shown in Table 12.9. For large \( n \) we have approximately

\[
T(n) = \frac{5.1 \times 10^{-7}}{n \ln n}, \text{ for mergesort}
\]

\[
T(n) = \frac{1.6 \times 10^{-7}}{n \ln n}, \text{ for quicksort}
\]

showing that quicksort, on average, is more than 3 times faster than mergesort. Because mergesort requires a temporary array we ran out of memory trying it with \( n = 10^7 \). Either algorithm is considerably faster than selection or insertion sort as a comparison with Table 12.7 shows.

### 12.5.13 Class of static sorting methods

Here is the IntArraySort class containing the four sorting algorithms that we have developed.

```java
class IntArraySort
```

```java
package chapter12.sorting;
```
/**
* Sorting methods for arrays of integers
*/

public class IntArraySort
{
    /**
     * Sort a subarray in increasing order using selection sort.
     * @param a The array
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     */
    public static void selectionSort(int[] a, int start, int end)
    {
        for (int i = start; i < end; i++)
        {
            // find position k of minimum element among
            // the elements a[i] to a[end]
            int k = i;
            for (int j = i+1; j <= end; j++)
            {
                if (a[j] < a[k])
                {
                    k = j;
                }
            }
            // swap the smallest element found (it’s a[k]) with a[i]
            int temp = a[k];
            a[k] = a[i];
            a[i] = temp;
        }
    }

    /**
     * Sort a subarray in increasing order using insertion sort.
     * @param a The array
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     */
    public static void insertionSort(int[] a, int start, int end)
    {
        for (int i = start+1; i <= end; i++)
        {
            // Sorted part of array is a[start], ..., a[i-1]
            // Unsorted part is a[i], ..., a[end]
            int x = a[i]; // left element of unsorted part
            int j = i-1; // right index of sorted part

            // move elements right until position for x is found.
            while ( (j >= start) && (x < a[j]) )
            {
                // move elements right until position for x is found.
                // Sorted part of array is a[start], ..., a[i-1]
                // Unsorted part is a[i], ..., a[end]
                int x = a[i]; // left element of unsorted part
                int j = i-1; // right index of sorted part

                // move elements right until position for x is found.
                while ( (j >= start) && (x < a[j]) )
                {
```java
{   a[j+1] = a[j]; // move a[j] one place to the right
    j--;
}  
a[j+1] = x; // drop x into the hole found
}

/**  
* Sort a subarray in increasing order using merge sort.  
* @param a The array  
* @param start Index defining start of subarray  
* @param end Index defining end of subarray  
*/
public static void mergeSort(int[] a, int start, int end)
{
    if (start == end)  
        return; // one-element subarray is already sorted
    int mid = (start + end) / 2;
    mergeSort(a, start, mid); // merge sort left half
    mergeSort(a, mid+1, end); // merge sort right half
    merge(a, start, mid, end); // merge the two halves
}

/**  
* Merge two sorted subarrays into a sorted subarray.  
* The merge part of merge sort that takes the sorted subarrays  
* a[start] to a[mid] and a[split+1] to a[end] and merges them into  
* the sorted subarray a[start] to a[end].  
* @param a The array  
* @param start Index defining start of left subarray a[start] to a[mid].  
* @param split Index defining end of left subarray  
* @param end Index defining end of right subarray a[mid+1] to a[end]  
*/
public static void merge(int[] a, int start, int split, int end)
{
    int n = end - start + 1; // number of elements to merge
    int[] t = new int[n]; // temporary storage required
    int i = start; // index of elements in left subarray
    int j = split + 1; // index of elements in right subarray
    int k = 0; // index into temporary storage
    // merge elements from left and right subarray to temp array
    // until one or both of the subarrays are exhausted
    while (i <= split && j <= end)
    {
        if (a[i] < a[j]) // element in left subarray is smaller
        {
            t[k] = a[i]; // move it to temp array t
            i++;
        }
    }
```
12.5 Sorting algorithms

else // element in right subarray is smaller
{
    t[k] = a[j]; // move it to temp array t
    j++;       // index next element in right subarray
}
    k++;       // in either case index next position in t

// copy any remaining elements from left subarray to t
while (i <= split)
{
    t[k] = a[i];
    i++;
    k++;
}

// copy any remaining elements from right subarray to t
while (j <= end)
{
    t[k] = a[j];
    j++;
    k++;
}

// copy elements to a from temporary array t. Can also use
// System.arraycopy(t, 0, a, start, end-start+1);
for (k = 0; k < n; k++)
a[start+k] = t[k];

/**
 * Sort a subarray in increasing order using quicksort.
 * @param a The array
 * @param start Index defining start of subarray
 * @param end Index defining end of subarray
 */
public static void quickSort(int[] a, int start, int end)
{
    if (start < end)
    {
        int split = partition(a, start, end);
        quickSort(a, start, split-1); // sort left part a[start] to a[split-1]
        quickSort(a, split+1, end); // sort right part a[split+1] to a[end]
    }
}

/**
 * Partition a subarray using the middle element as pivot.
 * This version of partition is due to Lomuto.
 *
public static int partition(int a[], int start, int end)
{
    // choose middle element as pivot and move it to
    // the start of the subarray temporarily.
    swap(a, (start + end)/2, start);
    int pivot = a[start];

    // partition the elements a[start+1] to a[end].
    // lastLeft is the index of the last element in the left
    // subarray. The elements a[start] to a[lastLeft] are
    // less than or equal to the pivot value.
    int lastLeft = start;
    for (int j = start+1; j <= end; j++)
    {
        if (a[j] < pivot)
        {
            lastLeft++; // move partition index right
            swap(a, j, lastLeft); // and swap element there with a[j]
        }
    }
    swap(a, start, lastLeft); // move pivot to its correct position
    return lastLeft;
}

/*
 * Swap two array elements given by their indices i, j.
 * 
 */
private static void swap(int[] a, int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

12.5.14 Testing the sorting algorithms

The following SortTester class can be used to test the sorting algorithms.
package chapter12.sorting;
import java.util.Scanner;

/**
 * Test the sort algorithms.
 */
public class SortTester
{
    public void doTest()
    {

        Scanner input = new Scanner(System.in);

        // Read the array to sort
        System.out.print("Enter number of integers in array: ");
        int size = input.nextInt(); input.nextLine();
        int[] testArray = new int[size];
        for (int k = 0; k < testArray.length; k++)
        {
            System.out.print("Enter element " + k + ": ");
            testArray[k] = input.nextInt(); input.nextLine();
        }

        // Read the indices that define the array slice
        System.out.print("Enter start index for subarray: ");
        int start = input.nextInt(); input.nextLine();
        System.out.print("Enter end index for subarray: ");
        int end = input.nextInt(); input.nextLine();

        // sort the array using each method and display the sorted array
        int[] testArrayCopy;
        testArrayCopy = arrayCopy(testArray);
        IntArraySort.selectionSort(testArrayCopy, start, end);
        System.out.println("Selection sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);

        testArrayCopy = arrayCopy(testArray);
        IntArraySort.insertionSort(testArrayCopy, start, end);
        System.out.println("Insertion sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);

        testArrayCopy = arrayCopy(testArray);
        IntArraySort.mergeSort(testArrayCopy, start, end);
        System.out.println("Merge sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);
    }
}
testArrayCopy = arrayCopy(testArray);
IntArraySort.quickSort(testArrayCopy, start, end);
System.out.println("Quick sort: Sorted subarray is");
printArray(testArrayCopy, start, end);
}
private void printArray(int[] a, int start, int end)
{
    System.out.print("[");
    for (int k = start; k <= end; k++)
    {
        System.out.print(a[k]);
        if (k < end) System.out.print(",");
    }
    System.out.print("]");
    System.out.println();
}
private int[] arrayCopy(int[] a)
{
    int[] copy = new int[a.length];
    for (int k = 0; k < a.length; k++)
        copy[k] = a[k];
    return copy;
}
public static void main(String[] args)
{
    new SortTester().doTest();
}

12.6 Generic object sorting

So far our sorting algorithms in IntArraySort work only with arrays of type int[]. To sort arrays of double numbers we would have to rewrite all of them to use type double[]. Also we have assumed that the arrays are to be sorted in increasing order but we may want to sort in decreasing order which would require another set of classes.

The solution for versions of Java up to 1.4 is to write generic object algorithms that sort arrays of type Object[]. Then an array of elements of any object type can be sorted since the elements are also of type Object. The only problem is that we must ensure that only objects of the desired type are stored in the array and we must typecast to obtain the actual type from the object type.

In Java 5 generic parametric types were introduced so we can use a type parameter such as E for the element type and use an array specified as E[]. The advantage is that the compiler can now check that only elements of type E are stored in the array. When the checking is complete the type E can be converted by the compiler to type Object.

In either case (Java 1.4 or Java 5) it is necessary to have some generic object comparison
method that can compare two objects to define which comes first (total ordering). For example, in the InsertionSort method there is the if-statement

```java
if (a[j] < a[k])
    k = j;
```

The boolean expression \(a[j] < a[k]\) will not work if we want to compare objects. For int and double numbers there are predefined operators such as \(<\) that define an order. For objects we need to define the order ourselves. For example, we could sort an array of BankAccount objects in order of increasing bank balance, or we could do it in order of increasing account number, or even in lexicographical order using the owner name.

### 12.6.1 The Comparator interface

There is an interface called Comparator in package java.util that has a compare method that can be used to define a total order on objects of some type \(T\) in Java 5:

```java
public interface Comparator<T>
{
    public int compare(T obj1, T obj2);
}
```

Prior to Java 5 this interface was given by

```java
public interface Comparator
{
    public int compare(Object obj1, Object obj2);
}
```

Here `compare` returns a negative value if `obj1` is “less than” `obj2`, 0 if `obj1` is equal to `obj2`, and a positive value if `obj1` is “greater than” `obj2`. This is the same convention used by the `compareTo` method in the `String` class. Then, if \(f\) is an object from a class that implements this interface the generic form of the above if-statement is

```java
if (f.compare(a[j], a[k]) < 0)
    k = j;
```

The opposite order can be defined simply by reversing the negative and positive return values.

Now we can rewrite all our sorting methods to use an argument array of type `Object[]` (type `E[]` in Java 5) and an extra argument to specify an object implementing the Comparator interface (`Comparator<E>` in Java 5).

For example, in Java 1.4 the selection sort method would now look like:

```java
public static void selectionSort(Object[] a, int start, int end, Comparator f)
{
    for (int i = start; i < end; i++)
    {
        // find position k of minimum element among
```
// the elements a[i] to a[end]

int k = i;
for (int j = i+1; j <= end; j++)
{
    if (f.compare(a[j],a[k]) < 0)
        k = j;
}

// swap the smallest element found (it’s a[k]) with a[i]

Object temp = a[k];
a[k] = a[i];
a[i] = temp;
}
}

and in Java 5 it would look like

public static <E> void selectionSort(E[] a, int start, int end, Comparator<E> f)
{
    for (int i = start; i < end; i++)
    {
        // find position k of minimum element among
        // the elements a[i] to a[end]

        int k = i;
        for (int j = i+1; j <= end; j++)
        {
            if (f.compare(a[j],a[k]) < 0)
                k = j;
        }

        // swap the smallest element found (it’s a[k]) with a[i]

        E temp = a[k];
a[k] = a[i];
a[i] = temp;
    }
}

Here we specify the type before the void keyword (this is always done for static methods) and then we specify the array type E[] and use Comparator<E> to specify the comparator type. Also, when swapping two elements we use the type E for the temporary variable.
12.6 Generic object sorting

12.6.2 GenericArraySort class

We can put all our generic sorting algorithms in a static class called GenericArraySort. Here we show only the Java 5 version.

```java
class GenericArraySort
{
    /**
     * Sort a subarray in a specified order using selection sort.
     * @param a The array
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     * @param f A Comparator object defining the sort order
     */
    public static <E> void selectionSort(E[] a, int start, int end, Comparator<E> f)
    {
        for (int i = start; i < end; i++)
        {
            int k = i;
            for (int j = i+1; j <= end; j++)
            {
                if (f.compare(a[j], a[k]) < 0)
                    k = j;
            }
            swap(a, k, i);
        }
    }

    /**
     * Sort a subarray in a specified order using insertion sort.
     * @param a The array
     * @param start Index defining start of subarray
     * @param end Index defining end of subarray
     * @param f A Comparator object defining the sort order
     */
}
```

public static <E> void insertionSort(E[] a,
   int start, int end, Comparator<E> f)
{
   for (int i = start+1; i <= end; i++)
   {
      // Sorted part of array is a[start], ..., a[i-1]
      // Unsorted part is a[i], ..., a[end]

      E x = a[i]; // left element of unsorted part
      int j = i-1; // right index of sorted part

      // move elements right until position for x is found.
      while ( (j >= start) && (f.compare(x,a[j]) < 0) )
      {
         a[j+1] = a[j]; // move a[j] one place to the right
         j--;
      }
      a[j+1] = x; // drop x into the hole found
   }
}

/**
 * Sort a subarray in a specified order using merge sort.
 * @param a The array
 * @param start Index defining start of subarray
 * @param end Index defining end of subarray
 * @param f A Comparator object defining the sort order
 */
public static <E> void mergeSort(E[] a,
   int start, int end, Comparator<E> f)
{
   if (start == end)
      return; // one-element subarray is already sorted
   int mid = (start + end) / 2;
   mergeSort(a, start, mid, f); // merge sort left half
   mergeSort(a, mid+1, end, f); // merge sort right half
   merge(a, start, mid, end, f); // merge the two halves
}

/**
 * Merge two sorted subarrays into a sorted subarray.
 * The merge part of merge sort that takes the sorted subarrays
 * a[start] to a[split] and a[split+1] to a[end] and merges them into
 * the sorted subarray a[start] to a[end].
 * @param a The array
 * @param start Index defining start of left subarray a[start] to a[mid].
 * @param split Index defining end of left subarray
 * @param end Index defining end of right subarray a[mid+1] to a[end]
 * @param f A Comparator object defining the sort order
 */
public static <E> void merge(E[] a,
   int start, int split, int end, Comparator<E> f)
{
   int n = end - start + 1; // number of elements to merge
   E[] t = (E[]) new Object[n]; // temporary storage required
   int i = start; // index of elements in left subarray
   int j = split + 1; // index of elements in right subarray
   int k = 0; // index into temporary storage

   // merge elements from left and right subarray to temp array
   // until one or both of the subarrays are exhausted
   while (i <= split && j <= end)
   {
      if (f.compare(a[i],a[j]) < 0) // element in left subarray is smaller
      {
         t[k] = a[i]; // move it to temp array t
         i++; // index next element in left subarray
      }
      else // element in right subarray is smaller
      {
         t[k] = a[j]; // move it to temp array t
         j++; // index next element in right subarray
      }
      k++; // in either case index next position in t
   }

   // copy any remaining elements from left subarray to t
   while (i <= split)
   {
      t[k] = a[i];
      i++;
      k++;
   }

   // copy any remaining elements from right subarray to t
   while (j <= end)
   {
      t[k] = a[j];
      j++;
      k++;
   }

   // copy elements to a from temporary array t. Can also use
   // System.arraycopy(t, 0, a, start, end-start+1);

   for (k = 0; k < n; k++)
      a[start+k] = t[k];
}
/**
* Sort a subarray in a specified order using quicksort.
* @param a The array
* @param start Index defining start of subarray
* @param end Index defining end of subarray
* @param f A Comparator object defining the sort order
*/
public static <E> void quickSort(E[] a,
        int start, int end, Comparator<E> f)
{
    if (start < end)
    {
        int split = partition(a, start, end, f);
        quickSort(a, start, split-1, f); // sort left part a[start] to a[split-1]
        quickSort(a, split+1, end, f); // sort right part a[split+1] to a[end]
    }
}

/**
* Partition a subarray using the middle element as pivot.
* This version of partition is due to Lomuto.
* @param a The array
* @param start Index of first subarray element
* @param end Index of last subarray element
* @param f A Comparator object defining the sort order
* @return index <code>split</code> such that the left
* subarray is <code>a[start]</code> to <code>a[split]</code>,
* with <code>a[split]</code> being the pivot,
* and the right subarray is <code>a[split+1]</code>
* to <code>a[end]</code>.
*/
public static <E> int partition(E a[],
        int start, int end, Comparator<E> f)
{
    // choose middle element as pivot and move it to
    // the start of the subarray temporarily.
    swap(a, (start + end)/2, start);
    E pivot = a[start];

    // partition the elements a[start+1] to a[end].
    // lastLeft is the index of the last element in the left
    // subarray. The elements a[start] to a[lastLeft] are
    // less than or equal to the pivot value.
    int lastLeft = start;
    for (int j = start+1; j <= end; j++)
    {
        if (f.compare(a[j],pivot) < 0)
        {
            lastLeft++; // move partition index right
            swap(a, j, lastLeft); // and swap element there with a[j]
        }
    }
    return lastLeft;
}
12.6 Generic object sorting

swapped(a, start, lastLeft); // move pivot to its correct position
return lastLeft;

/**
 * Swap two array elements given by their indices i, j.
 */
private static <E> void swap(E[] a, int i, int j)
{
    E temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

12.6.3 Sorting strings in lexicographical order

As an example, suppose you want to sort an array of strings in increasing lexicographical order. First define this order in the following simple class that implements the Comparator interface:

```java
package chapter12.generic;
import java.util.Comparator;

/**
 * An implementation of the Comparator interface for strings
 * in lexicographical order.
 */
public class StringComparator implements Comparator<String>
{
    /**
     * Compare two strings lexicographically
     * @param s1 first string
     * @param s2 second string
     * @return -1 if s1 precedes s2, 0 if s1 is equal
     * to s2, and 1 if s1 follows s2
     */
    public int compare(String s1, String s2)
    {
        return s1.compareTo(s2);
    }
}
```

Note that we have used Comparator<String> to indicate that the comparator object is comparing two strings and we have used the compareTo method in the String class to do the comparison.
Then the following statement can be used to sort a subarray of a string array called \( s \) using selection sort.

\[
\text{GenericArraySort.selectionSort}(s, \text{start}, \text{end}, \text{new StringComparator}());
\]

The entire string array can be sorted using

\[
\text{GenericArraySort.selectionSort}(s, 0, s.\text{length} - 1, \text{new StringComparator}());
\]

To sort an array of strings in decreasing lexicographical order we could use the following class:

```java
public class StringDecreasingComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s2.compareTo(s1);
    }
}
```

Here is a complete program that sorts an array of strings using all of the sorting methods.

```java
public class GenericStringSortTester {
    public static void main(String[] args) {
        String[] array = {"banana", "apple", "cherry", "date"};
        StringDecreasingComparator comparator = new StringDecreasingComparator();
        GenericArraySort.selectionSort(array, 0, array.length - 1, comparator);
        System.out.println("Array sorted in decreasing order:");
        for (String s : array) {
            System.out.println(s);
        }
    }
}
```
public class GenericStringSortTester
{
    public void doTest(Comparator<String> comp)
    {
        Scanner input = new Scanner(System.in);

        // Read the array

        System.out.print("Enter number of strings in array: ");
        int size = input.nextInt(); input.nextLine();
        String[] testArray = new String[size];
        for (int k = 0; k < testArray.length; k++)
        {
            System.out.print("Enter string element " + k + ": ");
            testArray[k] = input.nextLine();
        }

        // Read the indices that define the array slice

        System.out.print("Enter start index for subarray: ");
        int start = input.nextInt(); input.nextLine();
        System.out.print("Enter end index for subarray: ");
        int end = input.nextInt(); input.nextLine();

        // sort the array using each method and display the sorted array

        String[] testArrayCopy;
        testArrayCopy = arrayCopy(testArray);
        GenericArraySort.selectionSort(testArrayCopy, start, end, comp);
        System.out.println("Selection sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);

        testArrayCopy = arrayCopy(testArray);
        GenericArraySort.insertionSort(testArrayCopy, start, end, comp);
        System.out.println("Insertion Sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);

        testArrayCopy = arrayCopy(testArray);
        GenericArraySort.mergeSort(testArrayCopy, start, end, comp);
        System.out.println("Merge sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);

        testArrayCopy = arrayCopy(testArray);
        GenericArraySort.quickSort(testArrayCopy, start, end, comp);
        System.out.println("Quick sort: Sorted subarray is");
        printArray(testArrayCopy, start, end);
    }

    private <E> void printArray(E[] a, int start, int end)
    {
        System.out.print("[");
    }
}
for (int k = start; k <= end; k++)
{
    System.out.print(a[k]);
    if (k < end) System.out.print(",");
}
System.out.print("]");
System.out.println();

private String[] arrayCopy(String[] a)
{
    String[] copy = new String[a.length];
    for (int k = 0; k < a.length; k++)
    {
        copy[k] = a[k];
    }
    return copy;
}

public static void main(String[] args)
{
    System.out.println("Output for normal sort order");
    new GenericStringSortTester().doTest(new StringComparator());
    System.out.println("Output for reverse sort order");
    new GenericStringSortTester().doTest(new StringDecreasingComparator());
}

Some output is

Output for normal sort order
Enter number of strings in array: 4
Enter string element 0: one
Enter string element 1: two
Enter string element 2: three
Enter string element 3: four
Enter start index for subarray: 0
Enter end index for subarray: 3
Selection sort: Sorted subarray is [four,one,three,two]
Insertion Sort: Sorted subarray is [four,one,three,two]
Merge sort: Sorted subarray is [four,one,three,two]
Quick sort: Sorted subarray is [four,one,three,two]
Output for reverse sort order
Enter number of strings in array: 4
Enter string element 0: one
Enter string element 1: two
Enter string element 2: three
Enter string element 3: four
Enter start index for subarray: 0
Enter end index for subarray: 3
Selection sort: Sorted subarray is [two,three,one,four]
Insertion Sort: Sorted subarray is [two,three,one,four]
Merge sort: Sorted subarray is [two,three,one,four]
Quick sort: Sorted subarray is [two,three,one,four]

12.6.4 Comparing BankAccount objects

As another example, if you want to sort an array of BankAccount objects in order of increasing account number, the following class defines the order.

```java
package chapter12.generic;
import custom_classes.BankAccount;
import java.util.Comparator;

public class AccountNumberComparator implements Comparator<BankAccount>
{
    public int compare(BankAccount b1, BankAccount b2)
    {
        return b1.getNumber() - b2.getNumber();
    }
}
```

Here we use a subtraction so that a negative number is returned in case b1 has a smaller account number than b2.

Similarly, if you want to sort in order of increasing balance use the class

```java
package chapter12.generic;
import custom_classes.BankAccount;
import java.util.Comparator;

public class AccountBalanceComparator implements Comparator<BankAccount>
{
    public int compare(BankAccount b1, BankAccount b2)
    {
        double diff = b1.getBalance() - b2.getBalance();
    }
}
if (diff < 0.0) return -1;
else if (diff == 0.0) return 0;
else return 1;
}

Here is class to test these sort orders.

```java
package chapter12.generic;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

public class BankAccountSortTester {
    public void doTest() {
        BankAccount[] b = new BankAccount[3];
        b[0] = new BankAccount(321, "Fred", 150.0);
        b[1] = new BankAccount(234, "Gord", 350.0);
        b[2] = new JointBankAccount(123, "Jack", "Jill", 200.0);

        GenericArraySort.selectionSort(b, 0, 2,
                                        new AccountNumberComparator);
        System.out.println("Order: number");
        for (int k = 0; k < 3; k++)
            System.out.println(b[k]);

        System.out.println("Order: balance");
        GenericArraySort.selectionSort(b, 0, 2,
                                        new AccountBalanceComparator);
        for (int k = 0; k < 3; k++)
            System.out.println(b[k]);
    }
    public static void main(String[] args) {
        new BankAccountSortTester().doTest();
    }
}
```

### 12.7 The Arrays Class For Searching and Sorting

There is a class called Arrays in package java.util that has many static methods for searching and sorting arrays. Normally you should use these methods. However, it is important to know how to develop, test, compare and determine the efficiency of searching and sorting algorithms so the algorithms developed in this chapter have pedagogical value.
12.7 Arrays class

The generic versions of these searching and sorting methods require that a total order be defined on the array elements. This can be done in two ways:

- The array elements belong to some class that implements the Comparable interface. The order defined by this interface is called the natural order. The String class implements Comparable.

- If the class does not implement the Comparable interface or a different order is desired then we use a class that implements the Comparator interface and use an object from this class as a method argument to specify the order.

We have considered the Comparator interface in the GenericArraySort class on page 705.

12.7.1 Comparable interface

The Comparable interface resides in package java.lang (not java.util like Comparator):

```java
public interface Comparable{
    public int compareTo(Object obj)
}
```

In Java 5 this interface is defined as

```java
public interface Comparable<E>{
    public int compareTo(E element)
}
```

This interface, like Comparator, defines a total ordering on the objects of some class. However it is used in a different way. To use Comparable to order the objects of some class it is necessary that this class implement the Comparable interface (String for example).

12.7.2 Searching algorithms in the java.util.Arrays class

The Arrays class in package java.util contains the following searching algorithms:

- **static int binarySearch(double[] a, double key)**
  Search the double array `a` for the value `key` and return the index at which the value is found. If the value is not found then return the negative value $-k - 1$ where $k$ is the position at which `key` would need to be inserted to keep the array in sorted order.
  There are similar methods for arrays of all primitive types (boolean, char, byte, short, int, long, float, and double)

- **static int binarySearch(Object[] a, double key)**
  This is the generic object version with array type `Object[]`. The actual array element type must implement the Comparable interface.
• static <E> int binarySearch(E[] a, E key,
   Comparator<? super E> c)

This is the Java 5 version that uses a generic method with an array of type E and a comparator for E or any superclass of E. If the comparator argument c is null then the natural sort order provided by E is used, assuming that E implements the Comparable<E> interface.

[EXAMPLE 12.5] (Searching using Arrays.binarySearch) Using a null object for the comparator the statements

    String[] names = {"Bill", "Fred", "Gord", "Harry"};
    System.out.println(Arrays.toString(names));
    int k = Arrays.binarySearch(names, "Fred", null);
    System.out.println("Index for Fred is " + k);
    k = Arrays.binarySearch(names, "Bob", null);
    System.out.println("Index for Bob is " + k);

produce the output

    [Bill, Fred, Gord, Harry]
    Index for Fred is 1
    Index for Bob is -2

which shows that Fred was found at index 1. Bob was not found but the correct sorted location would be $k$ where $-k - 1 = -2$, so $k = -1 + 2 = 1$ (at position 1).

12.7.3 Sorting algorithms in the java.util.Arrays class

The Arrays class in package java.util also contains the following sorting algorithms:

• static void sort(double[] a)
  Sort the double array a in increasing numerical order. There are similar methods for arrays of all primitive types (boolean, char, byte, short, int, long, float, and double)

• static void sort(double[] a, int fromIndex, int toIndex)
  Sort the double array a in increasing numerical order using the subarray beginning at index fromIndex and ending at index toIndex - 1, not toIndex. There are similar methods for arrays of all primitive types (boolean, char, byte, short, int, long, float, and double)

• static void sort(Object[] a)
  static void sort(Object[] a, int fromIndex, int toIndex)
  These are the generic object version with array type Object[]. The actual element type must implement the Comparable interface.
12.8 Exercises

- static <E> void sort(E[] a, Comparator<? super E> c)
  static <E> void sort(E[] a, int fromIndex, int toIndex, Comparator<? super E> c)

These are the Java 5 versions that uses a generic method with an array of type E and a comparator for any type E or any superclass of E.

**EXAMPLE 12.6** (Sorting using Arrays.sort) Using a StringDecreasingComparator (see page 710) and the natural order the statements

System.out.println(Arrays.toString(names));
Arrays.sort(names, null);
System.out.println(Arrays.toString(names));
Arrays.sort(names, new StringDecreasingComparator());
System.out.println(Arrays.toString(names));

produce the output

[Harry, Gord, Fred, Bill]
[Bill, Fred, Gord, Harry]
[Harry, Gord, Fred, Bill]

which shows that the original string array (ordered in decreasing order) has been sorted in the natural increasing order and then this string array has been sorted in decreasing order to give back the original string array.

12.8 Exercises

- **Exercise 12.1** (Recursive version of findMinimum)
  Write a recursive version of the findMinimum method that has the prototype

  int findMinimum(int[] a, int start, int end)

  and returns the first index at which the minimum occurs

- **Exercise 12.2** (Another recursive version of findMinimum)
  Write a recursive version of the findMinimum method that returns the minimum rather than the index.

- **Exercise 12.3** (Searching for all occurrences of a given element)
  Write a pseudo-code algorithm that finds all occurrences of a given element in a subarray of a given array. The algorithm should return an array containing the indices at which the element occurs. Write a Java method with prototype

  int[] findAllElements(int[] a, int x, int start, int end)
that implements this algorithm, where a is the array, x is the element to search for, and start and end define the subarray. For example, if the array is \(\{4, 3, 8, 65, 32, 65, 17, 65\}\), the subarray is the entire array, and the element to search for is 65, then the array of indices returned is \((3, 5, 7)\).

**Exercise 12.4 (Recursive binary search for String arrays)**
Write a recursive binary search method with prototype

```java
static int binarySearch(String[] s, String target, int start, int end)
```

that searches the subarray \(s[start]\) to \(s[end]\) of a String array \(s\) for a given target string. Assume that the string array is sorted in increasing lexicographical order.

**Exercise 12.5 (Modified binary search algorithm)**
Modify the recursive and non-recursive binary search algorithms so they work like the ones in the 
Arrays class in case the element is not found. This means that instead of returning \(-1\) when the element is not found the algorithm should return \(-k - 1\) where \(k\) is the array position at which the element could be inserted to keep the array sorted. The value \(-k - 1\) is chosen so that the result returned is always negative when an element is not found (the minimum value of \(k\) would be 0) and always non-negative when the element is found.

Implement this algorithm as a Java method for searching an array of strings with prototype

```java
static int binarySearch(String[] a, String key)
```

Implement this algorithm as a Java 5 method for searching an ArrayList<String> of strings with prototype

```java
static int binarySearch(ArrayList<String> a, String key)
```

**Exercise 12.6 (Insertion into an ordered list)**
Use the Java 5 version of binary search from Exercise [12.5] to write a method with prototype

```java
static int insertInOrderedList(ArrayList<String> a, String key)
```

If \(key\) is not found then it should be inserted into the list in the correct position to keep the list sorted. The return value is the position of \(key\) in the list. Hint: the ArrayList\(<\text{T}>\) class has a method with prototype

```java
public void add(int k, T element)
```

that inserts an element of type \(T\) into the list at index \(k\).

**Exercise 12.7 (Reversing the sort order)**
Suppose you have a subarray of type int[] that is already sorted in increasing order. Write an \(O(n)\) algorithm that will sort it in decreasing order. Write a Java method called reverseSortOrder with prototype

```java
void reverseSortOrder(int[] a, int start, int end)
```

for this algorithm.
12.8 Exercises

► Exercise 12.8 (Comparing running times on your computer)
Reproduce the running time results in Table 12.7 and Table 12.9 using your computer.

► Exercise 12.9 (A version of GenericArraySort for ArrayList<E> objects)
Write a version of the GenericArraySort class called GenericArrayListSort with methods that sort an ArrayList<E> object instead of an array object of type E[].

► Exercise 12.10 (A GenericArraySearch class)
Using the GenericArraySort class as a guide, write a class called GenericArraySearch that implements generic versions of the linear, recursive binary, and non-recursive binary search algorithms that were written for arrays of type int[].

► Exercise 12.11 (A version of GenericArraySearch for ArrayList<E> objects)
Write a version of the GenericArraySearch class called GenericArrayListSearch with methods that search an ArrayList<E> object instead of an array object of type E[].

► Exercise 12.12 (Sorting a file of BankAccount objects)
Using the GenericArrayListSort class write a class called AccountSorter that reads a file of BankAccount objects in the single-line colon-separated format into an ArrayList object and writes a new file that is sorted in order of increasing account number using the quicksort method in the GenericArrayListSort class from Exercise 12.9.

► Exercise 12.13 (Sorting a phone book file)
Using the GenericArraySort class write a program class called PhoneBookSorter that reads a file of PhoneBookEntry objects (see Chapter 11 exercises) in the single-line colon-separated format into an ArrayList<PhoneBookEntry> object and writes a new file that is sorted in lexicographical order on the name using the quicksort method in the GenericArrayListSort class from Exercise 12.9.

► Exercise 12.14 (Using GenericArraySort to sort arrays of primitive type)
How can you use GenericArraySort to sort an array of type int[] or of type double[], noting that int and double are primitive types, not Object types? Write a Java program class called DoubleSortTester that reads an array of type double[] and sorts it using one of the sort methods in GenericArraySort. HINT: Use the Double wrapper class.

► Exercise 12.15 (Sorting arrays of rectangles)
Write a Rectangle class that has private data fields for the height and width of the rectangle, a constructor for a rectangle, given its width and height, get methods for the width and height, a toString method, and a method to return the area of the rectangle.

The class should implement the Comparable<Rectangle> interface using the area to define the order: one rectangle is less than another if it has a smaller area. Two rectangles are equal if they have the same area.

Also write a RectangleComparator class that implements the Comparator<Rectangle> interface to define the following ordering of rectangles. One rectangle is less than another if its width is smaller. In case the widths are the same then use the height and define one rectangle to be less
than another of the same width if the height is smaller. In case both the height and the width are the same then the rectangles are equal.

Write a tester class called RectangleSortTester that constructs an array of Rectangle objects and uses the sorting methods in the Arrays class to sort the array using the order defined by the Comparable<Rectangle> interface. Use the order defined by RectangleComparator.

**Exercise 12.16 (Comparing two versions of partition)**
Here is a different version of the partition method for arrays of type int[] than the one we have used in quicksort.

```java
public static int partition(int a[], int start, int end)
{
    int left = start;
    int right = end;

    // choose middle element as pivot and move it to
    // the end of the array temporarily
    swap(a, (start + end)/2, end);
    int pivot = a[end];

    while (left < right)
    {
        // search left part for element larger than pivot
        while (left < right && a[left] <= pivot) left++;

        // search right part for element smaller than pivot
        while (left < right && a[right] >= pivot) right--;

        // if we find a pair of elements in the wrong parts
        // swap them and look for more
        if (left < right)
        {
            swap(a, left, right);
            left++; // look for more
        }
    }
    swap(a, left, end); // put pivot back in correct position
    return left;
}
```

Test the running time of the two versions to see if one is faster than the other.

**Exercise 12.17 (Graphical simulation of sorting algorithms)**
Write a graphical simulation for one of the sorting algorithms. First choose an initial random array of n integers. Next scale the values so that the largest one is the height of the highest vertical bar.
that will fit. Then each integer in the array can be drawn as a vertical bar using the array index as a horizontal coordinate. Now apply a sorting algorithm and after each iteration update the bar graph by calling the `repaint` method. When the sort finishes the bar heights should increase from left to right.