13ACEHPRT

Searching and Sorting

13.1 Prologue

Searching and *sorting* are vast and important subjects. At the practical level they are important because they are what many large computer systems do much of the time. At the theoretical level they help distill the general properties and interesting theoretical questions about algorithms and data structures and offer rich material on which to study and compare them. We will consider these topics in the context of working with arrays, along with other common algorithms that work with arrays.

Searching tasks in computer applications range from finding a particular character in a string of a dozen characters to finding a record in a database of 100 million records. In the abstract, searching is a task involving a set of data elements represented in some way in computer memory. Each element includes a *key* that can be tested against a target value for an exact match. A successful search finds the element with a matching key and returns its location or some information associated with it: a value, a record, or the address of a record.

Searching refers to tasks where matching the keys against a specified target is straightforward and unambiguous. If, by comparison, we had to deal with a database of fingerprints and needed to find the best match for a given specimen, that application would fall into the category of *pattern recognition* rather than searching. It would also be likely to require the intervention of some human experts.

To *sort* means to arrange a list of data elements in ascending or descending order. The data elements may be numeric values or some records ordered by keys. In addition to preparing a data set for easier access (such as Binary Search), sorting has many other applications. One example is matching two data sets. Suppose we want to merge two large mailing lists and eliminate the duplicates. This task is straightforward when the lists are alphabetically sorted by name and address but may be unmanageable otherwise. Another use may be simply presenting information to a user in an ordered manner. A list of the user's files on a personal computer, for example, may be sorted by name, date, or type. A word processor sorts information when it automatically creates an index or a bibliography for a book. In large business systems, millions of transactions (for example, bank checks or credit card charges) are sorted daily before they are posted to customer accounts or forwarded to other payers.

In this chapter we first consider different ways of comparing objects in Java. We then look at two searching algorithms, Sequential Search and Binary Search, and several common sorting algorithms: Selection Sort, Insertion Sort, and two faster ones, Mergesort and Quicksort.

13.2 `equals`, `compareTo`, and `compare`

Java offers three ways for comparing objects:

```
public boolean equals(Object other)
public int compareTo(T other)
public int compare(T obj1, T obj2)
```

where *T* is the name of your class. The `boolean` method `equals` compares `this` object to `other` for equality. The `int` method `compareTo` compares `this` object to another object of the same type and returns an integer that indicates whether `this` is greater than, equal to, or less than `other`. The `int` method `compare` compares two objects of the same type and returns an integer that indicates which of the two objects is greater than the other. Let us take a closer look at each of these methods: where they come from and how we can benefit from them.

1. `equals`

The `equals` method —

```
public boolean equals(Object other)
```

— is a method of the class `Object`. It compares the addresses of `this` and `other` and returns `true` if they are the same and `false` otherwise. Since every class has `Object` as an ancestor, every class inherits this method from `Object`. However, we are more often interested in comparing the contents of objects rather than their addresses (and we have the `==` operator to compare the addresses). So programmers often override `Object`'s `equals` method in their classes.

We have already seen `equals` in the `String` class. There strings are compared character by character for an exact match. Consider another example, the class `Country` in Figure 13-1. The `equals` method in `Country` compares `this` country to `other` based on their names: two countries with the same name are considered equal. It is common for an `equals` method in a class to employ calls to `equals` for one or several fields.

> **In order to override `Object`'s `equals` method in your class, the signature of your `equals` method must be exactly the same as the signature of `equals` in `Object`. In particular, the declared type of the parameter `other` must be `Object`.**

```
public class Country implements Comparable<Country>
{
  private String name;
  private int population;

  public Country(String nm) { name = nm; population = 0; }
  public Country(String nm, int pop) { name = nm; population = pop; }
  public String getName() { return name; }
  public int getPopulation() { return population; }

  public boolean equals(Object other)
  {
    if (other != null)
      return name.equals(((Country)other).getName());
    else
      return false;
  }

  public int compareTo(Country other)
  {
    return name.compareTo(other.getName());
  }

  public String toString()
  {
    return name + ": " + population;
  }
}
```

Figure 13-1. `JM\Ch13\Compare\Country.java`

That's why we had to cast `other` into `Country`. If you write

```
public boolean equals(Country other)  // Error
{
  if (other != null)
    return name.equals(other.getName());
  else
    return false;
}
```

you will define a different `equals` method and not override the one in `Object`. You might say: "So what? I only intend to compare countries to each other, not to other types of objects." The problem is that certain library methods, such as `contains` and `indexOf` in `ArrayList`, call your `equals` method polymorphically when they need to compare objects for equality. So if you plan to store your objects in an

`ArrayList` (or another Java collection), you have to override the `Object` class's `equals` properly.

"Then," you might wonder, "What happens if I accidentally pass an incompatible type of parameter to `equals`?" For example,

```
Country country = new Country("USA");
...
if (country.equals("USA"))
  ...
```

Since `"USA"` is an object, this code compiles with no errors, but at run time the equals method throws a `ClassCastException`, because it cannot cast a `String` into a `Country`. It would be better to catch such errors at compile time, but better late than never. The correct comparison would be

```
if (coutry.getName().equals("USA"))
  ...
```

or

```
Country usa = new Country("USA");
if (coutry.equals(usa))
  ...
```

Some programmers make `equals` simply return `false` if the parameter is of an incompatible type. This may be necessary if a programmer plans to mix different types of objects in the same array or list or another collection. For example:

```
public boolean equals(Object other)
{
  if (other instanceof Country)
    return name.equals(((Country)other).getName());
  else
    return false;
}
```

Java's `boolean` operator

```
x instanceof T
```

returns `true` if an only if *x* IS-A *T*. More precisely, if *x*'s class is *X*, *x* `instanceof` *T* returns `true` when *X* is exactly *T*, or *X* is a subclass of *T* or has *T* as an ancestor, or *T* is an interface and *X* implements *T*.

If you define `equals` this way, then you have to be extra careful when you use it because the program won't tell you when you pass a wrong type of parameter to `equals` — the call will just return `false`.

Note that overriding the `Object`'s `equals` method does not change the meaning of the `==` operator for the objects of your class: it still compares addresses of objects.

2. `compareTo`

The `compareTo` method is an abstract method defined in the `java.util.Comparable<T>` (pronounced *com-'parable*) interface, so there is no default implementation for it. To implement `Comparable<T>` in your class, you need to supply the following method:

```
public int compareTo(T other)
```

where *T* is the name of your class. For example:

```
public class Country implements Comparable<Country>
{
  ...
  public int compareTo(Country other)
  {
    return name.compareTo(other.getName());
  }
  ...
}
```

`compareTo` returns an `int`: a positive value indicates that `this` is "greater than" `other`; a zero indicates that they are "equal," and a negative value indicates that `this` is "less than" `other`. So `x.compareTo(y)` is sort of like "`x - y`."

It is the programmer who decides what is "greater" and what is "less." `compareTo` is said to define a *natural ordering* among the objects of your class. In the above example, the "natural ordering" among countries is alphabetical, by name.

Note that `compareTo` takes a parameter of your class type, not `Object`.

Why do we need the `Comparable` interface and why would we want our classes to implement it? The reason is the same as for the `equals` method: certain library methods expect objects to be `Comparable` and polymorphically call `compareTo` for your objects. For example, the `java.util.Arrays` class has a

`sort(Object[] arr)` method that expects the elements of `arr` to be `Comparable`. So if there is a reasonable ordering among the objects of your class, and you plan to use library methods or classes that deal with `Comparable` objects, it makes sense to make the objects of your class `Comparable`.

`String`, `Integer`, `Double`, and several other library classes implement `Comparable`.

If you do define a `compareTo` method in your class, don't forget to state in the header of your class

```
... implements Comparable<YourClass>
```

If your class implements `Comparable`, then it is a good idea to define the `equals` method, too, and to make `compareTo` and `equals` agree with each other, so that `x.equals(y)` returns `true` if and only if `x.compareTo(y)` returns 0. Otherwise, some of the library methods (and you yourself) might get confused.

Very well. You've made `Countries` comparable. You can sort them by name using `Arrays.sort`. But suppose that sometimes you need to sort them by population. What can you do? Then you need a *comparator*.

3. compare

A *comparator* is an object that specifies a way to compare two objects of your class. A comparator belongs to a class that implements the `java.util.Comparator<T>` interface and has a method

```
public int compare(T obj1, T obj2)
```

where `T` is the name of a class.

If `compare` returns a positive integer, `obj1` is considered greater than `obj2`; if the returned value is 0, they are considered equal; if the returned value is negative, `obj1` is considered less than `obj2`. So `compare(obj1, obj2)` is sort of like "`obj1 - obj2`."

The purpose of comparators is to be passed as parameters to constructors and methods of certain library classes (or your own classes). By creating different types of comparators, you can specify different ways of comparing objects of your class. You can create different comparators for ordering objects by different fields in ascending or descending order.

For example, the `PopulationComparator` class in Figure 13-2 defines comparators that compare countries by population. You can create an "ascending" comparator and a "descending" comparator by passing a `boolean` parameter to `PopulationComparator`'s constructor.

```
// Comparator for Country objects based on population

import java.util.Comparator;

public class PopulationComparator implements Comparator<Country>
{
  private boolean ascending;

  // Constructors
  public PopulationComparator() { ascending = true; }
  public PopulationComparator(boolean ascend) { ascending = ascend; }

  public int compare(Country country1, Country country2)
  {
    int diff = country1.getPopulation() - country2.getPopulation();
    if (ascending)
      return diff;
    else
      return -diff;
  }
}
```

Figure 13-2. **JM\Ch13\Compare\PopulationComparator.java**

The `Arrays` class has an overloaded version of the `sort` method that takes a comparator as a parameter. Now we can either rely on the natural ordering or create different comparators and pass them to `Arrays.sort` (Figure 13-3). The output from the main method in the figure is

```
[Brazil: 190, China: 1321, India: 1110, Indonesia: 249, USA: 301]
[Brazil: 190, Indonesia: 249, USA: 301, India: 1110, China: 1321]
[China: 1321, India: 1110, USA: 301, Indonesia: 249, Brazil: 190]
```

```
import java.util.Arrays;

public class ComparatorTest
{
  public static void main(String[] args)
  {
    Country[] countries =
      { // population in millions
        new Country("China", 1321),
        new Country("India", 1110),
        new Country("USA", 301),
        new Country("Indonesia", 249),
        new Country("Brazil", 190),
      };

    // Sort by name:
    Arrays.sort(countries);
    System.out.println(Arrays.toString(countries));

    // Sort by population ascending:
    Arrays.sort(countries, new PopulationComparator(true));
    System.out.println(Arrays.toString(countries));

    // Sort by population descending:
    Arrays.sort(countries, new PopulationComparator(false));
    System.out.println(Arrays.toString(countries));
  }
}
```

Figure 13-3. **JM\Ch13\Compare\ComparatorTest.java**

13.3 Sequential and Binary Search

Suppose we have an array of a certain size and we want to find the location of a given "target" value in that array (or ascertain that it is not there). If the elements of the array are in random order, we have no choice but to use *Sequential Search*, that is, to check the value of each consecutive element one by one until we find the target element (or finish scanning through the whole array). For example:

```
String[] words = { < ... some words> };
String target = < ... a word >;

for (int k = 0; k < words.length; k++)
{
  if (target.equals(words[k]))
    return k;
}
...
```

This may be time-consuming if the array is large. For an array of 1,000,000 elements, we will examine an average of 500,000 elements before finding the target (assuming that the target value is always somewhere in the array). This algorithm is called an $O(n)$ ("order of n") algorithm because it takes an average number of operations roughly proportional to n, where n is the size of the array. ($O(...)$ is called the "big-O" notation. We explain it more formally in Chapter 18.)

In Chapter 4 we introduced a more efficient algorith, called *Binary Search*, which can be used if the elements of the array are arranged in ascending or descending order (or, as we say, the array is *sorted*). Let's say our array is sorted in ascending order and we are looking for a target value x. Take the middle element of the array and compare it with x. If they are equal, the target element is found. If x is smaller, the target element must be in the left half of the array, and if x is larger, the target must be in the right half of the array. In any event, each time we repeat the same procedure we narrow the range of our search by half (see Figure <...> on page <...>). This sequence stops when we find the target or get down to just one element, which happens very quickly.

A binary search in an array of 3 elements requires at most 2 comparisons to find the target value or establish that it is not in the array. An array of 7 elements requires at most 3 comparisons. An array of 15 elements requires at most 4 comparisons, and so on. In general, an array of $2^n - 1$ (or fewer) elements requires at most n comparisons. So an array of 1,000,000 elements will require at most 20 comparisons ($2^{20} - 1 = 1{,}048{,}575$), which is much better than 500,000. That is why such methods are called "divide and conquer." Binary Search is an $O(\log n)$ algorithm because the number of operations it takes is roughly $\log_2 n$.

The `binarySearch` method in Figure 13-4 implements the Binary Search algorithm for an array of `Comparable` objects sorted in ascending order.

```
// Uses Binary Search to look for target in an array a, sorted in
// ascending order, If found, returns the index of the matching
// element; otherwise returns -1.
public static <T> int binarySearch(T[] a, Comparable<? super T> target)

   // Wow!  We wanted to show you this bizarre syntax once!
   // <T> indicates that this method works for an array of Comparable
   // objects of any type T.  Comparable<? super T> ensures that
   // the method will work not only for a class T that implements
   // Comparable<T> but also for any subclass of such a class.

{
  int left = 0, right = a.length - 1, middle;

  while (left <= right)
  {
    // Take the index of the middle element between
    //   "left" and "right":

    middle = (left + right) / 2;

    // Compare this element to the target value
    //  and adjust the search range accordingly:

    int diff = target.compareTo(a[middle]);

    if (diff > 0)  // target > a [middle]
      left = middle + 1;
    else if (diff < 0) // target < a[middle]
      right = middle - 1;
    else   // target is equal to a[middle]
      return middle;
  }

  return -1;
}
```

Figure 13-4. **JM\Ch13\BinarySearch\BinarySearch.java**

One way to understand and check code is to *trace* it manually on some representative examples. Let us take, for example:

```
Given:
    int[] a = {8, 13, 21, 34, 55, 89};
      // a[0] =  8; a[1] = 13; a[2] = 21; a[3] = 34;
      // a[4] = 55; a[5] = 89;
    target = 34

Initially:
    left =  0; right = a.length-1 = 5

First iteration:
    middle = (0+5)/2 = 2;
    a[middle] = a[2] = 21;
    target > a[middle] (34 > 21)
      ==> Set left = middle + 1 = 3; (right remains 5)

Second iteration:
    middle = (3+5)/2 = 4;
    a[middle] = a[4] = 55;
    target < a[middle] (34 < 55)
      ==> Set right = middle - 1 = 3; (left remains 3)

Third iteration:
    middle = (3+3)/2 = 3;
    a[middle] = a[3] = 34;
    target == a[middle] (34 = 34)
      ==> return 3
```

A more comprehensive check should also include tracing special situations (such as when the target element is the first or the last element, or is not in the array) and "degenerate" cases, such as when `a.length` is equal to 0 or 1.

We also have to make sure that the method terminates — otherwise, the program may hang. This is better accomplished by logical or mathematical reasoning than by tracing specific examples, because it is hard to foresee all the possible paths of an algorithm. Here we can reason as follows: our `binarySearch` method must terminate because on each iteration the difference `right - left` decreases by at least `1`. So eventually we either quit the loop via `return` (when the target is found), or reach a point where `right - left` becomes negative and the condition in the `while` loop becomes false.

13.4 *Lab*: Keeping Things in Order

In this lab you will write a class `SortedWordList`, which represents a list of words, sorted alphabetically (case blind). `SortedWordList` should extend `ArrayList<String>`. You have to redefine several of `ArrayList`'s methods to keep the list always alphabetically sorted and with no duplicate words.

1. Provide a no-args constructor and a constructor with one `int` parameter, the initial capacity of the list.
2. Redefine the `contains` and `indexOf` methods: make them use Binary Search.
3. Redefine the `set(i, word)` method so that it first checks whether `word` fits alphabetically between the (*i* - 1)-th and (*i* + 1)-th elements and is not equal to either of them. If this is not so, `set` should throw an `IllegalArgumentException`, as follows:

```
if (...)
  throw new IllegalAgrumentException("word = " + word + " i = " + i);
```

4. Redefine the `add(i, word)` method so that it first checks whether `word` fits alphabetically between the (*i* - 1)-th and *i*-th elements and is not equal to either of them. If this is not so, throw an `IllegalArgumentException`.
5. Redefine the `add(word)` method so that it inserts `word` into the list in alphabetical order. If `word` is already in the list, `add` should not insert it and should return `false`; otherwise, if successful, `add` should return `true`. Use Binary Search to locate the place where word should be inserted.
6. Define a new method `merge(SortedWordList additionalWords)`. This method should insert into this list in alphabetical order all the words from `additionalWords` that are not already in this list. `merge` should be efficient. You may not use any temporary arrays or lists. Each element from this list should move at most once, directly into its proper location. To achieve this while avoiding `IndexOutOfBounds` errors, you first need to add some dummy elements to the list. Save the current size of the list, then append to it *n* arbitrary strings, where *n* = `additionalWords.size()`. Call `super.add("")` to append an empty string or just call `addAll(additionalWords)` once. Now merge the lists, starting at the end of each list and at the end of the added space. At each step decide which of the two lists should supply the next element for the next vacant location.

7. Combine your class with `SortedListTest.java` in `JM\Ch13\SortedList` and test the program.

13.5 Selection Sort

The task of rearranging the elements of an array in ascending or descending order is called *sorting*. We are looking for a general algorithm that works for an array of any size and for any values of its elements. There exist many sorting algorithms for accomplishing this task, but the most straightforward one is probably *Selection Sort*. We have already mentioned it in Chapter <...>; we present it here again for convenience:

Selection Sort

1. Initialize a variable *n* to the size of the array.
2. Find the largest among the first *n* elements.
3. Make it swap places with the *n*-th element.
4. Decrement *n* by 1.
5. Repeat steps 2 - 4 while $n \geq 2$.

On the first iteration we find the largest element of the array and swap it with the last element. The largest element is now in the correct place, from which it will never move again. We decrement *n*, pretending that the last element of the array does not exist anymore, and repeat the procedure until we have worked our way through the entire array. The iterations stop when there is only one element left, because it has already been compared with every other element and is guaranteed to be the smallest.

The `SelectionSort` class in Figure 13-5 implements this algorithm for an array of the type `double`. A similar procedure will sort the array in descending order; instead of finding the largest element on each iteration, we can simply find the smallest element among the first *n*.

Sorting is a common operation in computer applications and a favorite subject on which to study and compare algorithms. Selection Sort is an $O(n^2)$ algorithm because the number of comparisons in it is $n \cdot (n-1)/2$, which is roughly proportional to n^2. It is less efficient than other sorting algorithms considered here, but more predictable: it always takes the same number of comparisons, regardless of whether the array is almost sorted, randomly ordered, or sorted in reverse order.

```
public class SelectionSort
{
  // Sorts a[0], ..., a[size-1] in ascending order
  //   using Selection Sort
  public static void sort(double[] a)
  {
    for (int n = a.length; n > 1; n--)
    {
      // Find the index iMax of the largest element
      //   among a[0], ..., a[n-1]:

      int iMax = 0;
      for (int i = 1; i < n; i++)
      {
        if (a[i] > a[iMax])
          iMax = i;
      }

      // Swap a[iMax] with a[n-1]:

      double aTemp = a[iMax];
      a[iMax] = a[n-1];
      a[n-1] = aTemp;

      // Decrement n (accomplished by n-- in the for loop).
    }
  }
}
```

Figure 13-5. **JM\Ch13\Benchmarks\SelectionSort.java**

13.6 Insertion Sort

The idea of the *Insertion Sort* algorithm is to keep the beginning part of the array sorted and insert each next element into the correct place in it. It involves the following steps:

Insertion Sort

1. Initialize a variable *n* to 1 (keep the first *n* elements sorted).
2. Save the next element and find the place to insert it among the first *n* so that the order is preserved.
3. Shift the elements as necessary to the right and insert the saved one in the created vacant slot.
4. Increment *n* by 1.
5. Repeat steps 2 - 4 while *n* < array length.

The `InsertionSort` class in Figure 13-6 implements this algorithm for an array of `doubles`.

```
public class InsertionSort
{
  // Sorts a[0], ..., a[size-1] in ascending order
  //   using Insertion Sort
  public static void sort(double[] a)
  {
    for (int n = 1; n < a.length; n++)
    {
      // Save the next element to be inserted:
      double aTemp = a[n];

      // Going backwards from a[n-1], shift elements to the
      //   right until you find an element a[i] <= aTemp:

      int i = n;
      while (i > 0 && aTemp < a[i-1])
      {
        a[i] = a[i-1];
        i--;
      }

      // Insert the saved element into a[i]:
      a[i] = aTemp;

      // Increment n (accomplished by n++ in the for loop).
    }
  }
}
```

Figure 13-6. **JM\Ch13\Benchmarks\InsertionSort.java**

Insertion Sort is also on average an $O(n^2)$ algorithm, but it can do better than Selection Sort when the array is already nearly sorted. In the best case, when the array is already sorted, Insertion Sort just verifies the order and becomes an $O(n)$ algorithm.

13.7 Mergesort

The tremendous difference in efficiency between Binary Search and Sequential Search hints at the possibility of faster sorting, too, if we could find a "divide and conquer" algorithm for sorting. Mergesort is one such algorithm. It works as follows:

Mergesort

1. If the array has only one element, do nothing.
2. (Optional) If the array has two elements, swap them if necessary.
3. Split the array into two approximately equal halves.
4. Sort the first half and the second half.
5. Merge both halves into one sorted array.

This recursive algorithm allows us to practice our recursive reasoning. Step 4 tells us to sort half of the array. But how will we sort it? Shall we use Selection Sort or Insertion Sort for it? Potentially we could, but then we wouldn't get the full benefit of faster sorting. For best performance we should use Mergesort again!

Thus it is very convenient to implement Mergesort in a recursive method, which calls itself. This fact may seem odd at first, but there is nothing paradoxical about it. Java and other high-level languages use a *stack* mechanism for calling methods. When a method is called, a new frame is allocated on the stack to hold the return address, the arguments, and all the local variables of a method (see Section <...>). With this mechanism there is really no difference whether a method calls itself or any other method.

Recall that any recursive method must recognize two possibilities: a *base case* and a *recursive case*. In the base case, the task is so simple that there is little or nothing to do, and no recursive calls are needed. In Mergesort, the base case occurs when the array has only one or two elements. The recursive case must reduce the task to similar but smaller tasks. In Mergesort, the task of sorting an array is reduced to sorting two smaller arrays. This ensures that after several recursive calls the task will fall into the base case and recursion will stop.

Figure 13-7 shows a `Mergesort` class that can sort an array of `double`s. This straightforward implementation uses a temporary array into which the two sorted halves are merged. The `sort` method calls a recursive helper method that sorts a particular segment of the array.

```
public class Mergesort
{
  private static double[] temp;

  // Sorts a[0], ..., a[size-1] in ascending order
  //   using the Mergesort algorithm
  public static void sort(double[] a)
  {
    int n = a.length;
    temp = new double[n];
    recursiveSort(a, 0, n-1);
  }

  // Recursive helper method: sorts a[from], ..., a[to]
  private static void recursiveSort(double[] a, int from, int to)
  {
    if (to - from < 2)       // Base case: 1 or 2 elements
    {
      if (to > from && a[to] < a[from])
      {
        // swap a[to] and a[from]
        double aTemp = a[to]; a[to] = a[from]; a[from] = aTemp;
      }
    }
    else                     // Recursive case
    {
      int middle = (from + to) / 2;
      recursiveSort(a, from, middle);
      recursiveSort(a, middle + 1, to);
      merge(a, from, middle, to);
    }
  }
```

Figure 13-7 `Mergesort.java` Continued

```
  // Merges a[from] ... a[middle] and a[middle+1] ... a[to]
  //   into one sorted array a[from] ... a[to]
  private static void merge(double[] a, int from, int middle, int to)
  {
    int i = from, j = middle + 1, k = from;

    // While both arrays have elements left unprocessed:
    while (i <= middle && j <= to)
    {
      if (a[i] < a[j])
      {
        temp[k] = a[i];   // Or simply temp[k] = a[i++];
        i++;
      }
      else
      {
        temp[k] = a[j];
        j++;
      }
      k++;
    }

    // Copy the tail of the first half, if any, into temp:
    while (i <= middle)
    {
      temp[k] = a[i];     // Or simply temp[k++] = a[i++]
      i++;
      k++;
    }

    // Copy the tail of the second half, if any, into temp:
    while (j <= to)
    {
      temp[k] = a[j];     // Or simply temp[k++] = a[j++]
      j++;
      k++;
    }

    // Copy temp back into a
    for (k = from; k <= to; k++)
      a[k] = temp[k];
  }
}
```

Figure 13-7. **JM\Ch13\Benchmarks\Mergesort.java**

The `merge` method is not recursive. To understand how it works, imagine two piles of cards, each sorted in ascending order and placed face up on the table. We want to merge them into the third, sorted, pile. On each step we take the smaller of the two exposed cards and place it face down on top of the destination pile. When one of the original piles is gone, we take all the remaining cards in the other one (the whole pile

or one by one — it doesn't matter) and place them face down on top of the destination pile. We end up with the destination pile sorted in ascending order.

Mergesort is an $O(n \log n)$ algorithm — much better than the $O(n^2)$ performance of Selection Sort and Insertion Sort.

13.8 Quicksort

Quicksort is another $O(n \log n)$ sorting algorithm. The idea of Quicksort is to pick one element, called the pivot, then rearrange the elements of the array in such a way that all the elements to the left of the pivot are smaller than or equal to it, and all the elements to the right of the pivot are greater than or equal to it. The pivot element can be chosen arbitrarily among the elements of the array. This procedure is called *partitioning*. After partitioning, Quicksort is applied (recursively) to the left-of-pivot part and to the right-of-pivot part, which results in a sorted array. Figure 13-8 shows a Java implementation, in which the pivot is the element in the middle of the array.

To partition the array, you proceed from both ends of the array towards the meeting point comparing the elements with the pivot. If the element on the left is not greater than the pivot, you increment the index on the left side; if the element on the right is not less than the pivot, you decrement the index on the right side. When you reach a deadlock (the element on the left is greater than pivot and the element on the right is less than pivot), you swap them and update both indices. When the left and the right side meet at a certain position, you swap the pivot with one of the elements that have met.

The Quicksort algorithm was invented by C.A.R. Hoare in 1962. Although its performance is less predictable than Mergesort's, it averages a faster time for random arrays.

```
public class Quicksort
{
  // Sorts a[0], ..., a[size-1] in ascending order
  //   using the Quicksort algorithm
  public static void sort(double[] a)
  {
    recursiveSort(a, 0, a.length - 1);
  }
```

Figure 13-8 *Quicksort.java Continued*

```
  // Recursive helper method: sorts a[from], ..., a[to]
  private static void recursiveSort(double[] a, int from, int to)
  {
    if (from >= to)
      return;

    // Choose pivot a[p]:
    int p = (from + to ) / 2;
      // The choice of the pivot location may vary:
      //   you can also use p = from or p = to or use
      //   a fancier method, say, the median element of the above three.

    // Partition:
    int i = from;
    int j = to;
    while (i <= j)
    {
      if (a[i] <= a[p])
        i++;
      else if (a[j] >= a[p])
        j--;
      else
      {
        swap (a, i, j);
        i++;
        j--;
      }
    }

    // Place the pivot in its correct position:
    if (p < j)
    {
      swap (a, j, p);
      p = j;
    }
    else if (p > i)
    {
      swap (a, i, p);
      p = i;
    }

    // Sort recursively:
    recursiveSort(a, from, p - 1);
    recursiveSort(a, p + 1, to);
  }

  private static void swap (double[] a, int i, int j)
  {
    double temp = a[i]; a[i] = a[j]; a[j] = temp;
  }
}
```

Figure 13-8. `JM\Ch13\Benchmarks\Quicksort.java`

13.9 *Lab:* Benchmarks

A benchmark is an empirical test of performance. The program in Figure 13-9 is designed to compare running times for several sorting methods. Enter the array size, select one of the four sorting methods in the "combo box" (pull-down list) and click the "Run" button. The program fills the array with random numbers and sorts them. This test is repeated many times for more accurate timing. (The default number of repetitions is set to 20, but you can enter that number as a command-line argument if you wish.) Then the program displays the total time it took to run all the trials.

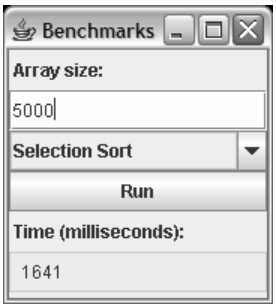


Figure 13-9. The *Benchmarks* program

The trials are performed by the `runSort` method, which returns the total time spent sorting an array filled with random values. Your task is to write this method.

First you need to learn how to generate a sequence of random numbers. (To be precise, such numbers are called *pseudo-random*, because they are generated according to some algorithm.) We have already used the `Math.random` method, but this time we won't use it because we want to have more control over how the random number generator is "seeded." A "seed" is a value that is used to initialize the random number generator. If you seed the random number generator with the same seed, you will get the same sequence of random numbers. If you want different sequences on different runs of your program, you need different seeds. A common technique is to use the current system time as a seed. In our program we call the `currentTimeMillis` method of the `System` class to obtain a value for the seed. (Recall that `System` is imported into all programs and has only static fields and

methods.) Once we obtain a seed value, we initialize the random number generator with that seed at the beginning of `runSort` because we want to run all the sorting methods on exactly the same data.

Java's `util` package provides a class `Random` for generating random numbers. `Random` has a no-args constructor that initializes the random number generator with an unpredictable seed, different each time. However, it has another constructor that takes the seed as a parameter. For example:

```
Random generator = new Random(seed);
```

You should create a `Random` object at the beginning of `runSort` using this constructor. After that you can generate the next random `double` by calling this object's `nextDouble` method. For example:

```
a[k] = generator.nextDouble();
```

The `runSort` method fills the array with random numbers and then sorts it. This is repeated `RUNS` times, and `runSort` returns the total time it took in milliseconds. Call the system time before and after each trial and add the elapsed time to the total.

Set up a project with the files `SelectionSort.java`, `InsertionSort.java`, `Mergesort.java`, `Quicksort.java`, and `Benchmarks.java`, provided in `JM\Ch13\Benchmarks`. Fill in the blanks in the `Benchmarks` class. Once your program is working, collect benchmarks for each sorting algorithm for arrays of various sizes: ranging, say, from 10,000 to 50,000 elements (to 500,000 for Mergesort and Quicksort). Plot the running time vs. the array size for each of the four sorting methods. You can do this on your graphing calculator, manually, or by entering the results into a spreadsheet or another data analysis program. See how well your experimental results fit with parabolas for Selection Sort and Insertion Sort and with an n log n curve for Mergesort and Quicksort.

13.10 The `Arrays` and `Collections` Classes

No matter what you want to do in Java, it has been done before. Sure enough, `java.util` package has a class `Arrays` and a class `Collections` with methods that implement Binary Search and sorting (using a fast version of the Mergesort algorithm). You will need to import `java.util.Arrays` or `java.util.Collections` in order to use their methods (unless you are willing to type in the full names every time). All of `Arrays`'s and `Collections`'s methods are static, and you cannot create objects of these classes.

`Arrays`'s `binarySearch` method is called as follows:

```
int pos = Arrays.binarySearch(a, target);
```

`Arrays` has overloaded versions of `binarySearch` for arrays of `char`s, `int`s, and other primitive data types. There is also a version for any comparable objects. In particular, it can be used with `String`s.

`Arrays.sort` methods can sort an array of `char`s, `int`s, `double`s, `String`s, and so on, either the whole array or a segment within specified limits. For example:

```
String[] dictionary = new String[maxWords];
int wordsCount;
< ... other statements >

Arrays.sort(dictionary, 0, wordsCount – 1);
```

As we have seen, there is also a version that works with any comparable objects and a version that takes a comparator as a parameter.

The `Arrays` class also offers a set of `fill` methods for different data types. A `fill` method fills an array or a portion of an array with a specified value. Another useful method is `toString`. For example,

```
String[] names = {"Ann", "Kate", "Zoe"};
System.out.println(Arrays.toString(names));
```

produces the output

```
[Ann, Kate, Zoe];
```

The `asList(T[] arr)` method returns a representation of the array `arr` as a fixed-length `List`. You can't add elements to this list, but you can call other methods and pass it as a parameter to library methods that expect a `List` or a `Collection`. For example, to shuffle the elements of the array `names` you can write

```
Collections.shuffle(Arrays.asList(names));
```

You can also print out the list:

```
System.out.println(Arrays.asList(names));
```

produces the same output as

```
System.out.println(Arrays.toString(names));
```

The `Collections` class works with Java collections, such as `ArrayList`. For example, to find a target word in `ArrayList<String> words`, you can call

```
Collections.binarySearch(words, target);
```

The `Collections` class also has methods to sort a `List` of comparable objects (or to sort a `List` using a comparator), to shuffle a list, to copy a list, to fill a list with a given value, to find max or min, and other methods.

13.11 Summary

The `boolean` method `equals(Object other)` in the `Object` class compares the address of `this` object to `other`. The correct way to override the `equals` method in your class is:

```
public boolean equals(Object other)
{
  if (other != null)
    ...
  else
    return false;
}
```

The `int` method `compareTo` of the `java.util.Comparable<T>` interface has the following signature:

```
public int compareTo(T other)
```

`compareTo` returns an `int` value that indicates whether `this` is larger than, equal to, or less than `other`. It is like "`this - other`." To implement the `Comparable` interface in `SomeClass`, state

```
public class SomeClass implements Comparable<SomeClass>
```

and provide a method

```
public int compareTo(SomeClass other) { ... }
```

that returns a positive integer if `this` is greater than `other`, 0 if they are equal, and a negative integer if `this` is less than `other` (sort of like `"this - other"`). The main reason for making objects of your class `Comparable` is that certain library methods and data structures work with `Comparable` objects. If provided, the `compareTo` method should agree with the `equals` method in your class.

A comparator is an object dedicated to comparing two objects of `SomeClass`. The comparator's class implements the `java.util.Comparator<SomeClass>` interface and provides a method

```
public int compare(SomeClass obj1, SomeClass obj2) { ... }
```

The `compare` method returns an integer, sort of like "`obj1 - obj2`." Comparators are passed as parameters to constructors and to methods of library classes, telling them how to compare the objects of your class.

Sequential Search is used to find a target value in an array. If the array is sorted in ascending or descending order, Binary Search is a much more efficient searching method. It is called a "divide and conquer" method because on each step the size of the searching range is cut in half.

The four sorting algorithms discussed in this chapter work as follows:

Selection Sort

Set *n* to the size of the array. While *n* is greater than 2, repeat the following: find the largest element among the first *n*, swap it with the *n*-th element of the array, and decrement *n* by one.

Insertion Sort

Keep the first *n* elements sorted. Starting at $n = 2$, repeat the following: find the place of `a[n]` in order among the first $n - 1$ elements, shift the required number of elements to the right to make room, and insert `a[n]`. Increment *n* by one.

Mergesort

If the array size is less than or equal to 2, just swap the elements if necessary. Otherwise, split the array into two halves. Recursively sort the first half and the second half, then merge the two sorted halves.

Quicksort

Choose a pivot element and partition the array, so that all the elements on the left side are less than or equal to the pivot and all the elements on the right side are greater than or equal to the pivot; then recursively sort each part.

Selection Sort is the slowest and most predictable of the four: each element is always compared to every other element. Insertion Sort works quickly on arrays that are almost sorted. Mergesort and Quicksort are both "divide and conquer" algorithms. They work much faster than the other two algorithms on arrays with random values.

Java's `Arrays` and `Collections` classes from the `java.util` package have `binarySearch`, `sort`, and other useful methods. All `Arrays` and `Collections` methods are static. The `Arrays` methods work with arrays, with elements of primitive data types, as well as with `String`s and other comparable objects. It also offers the convenient methods `fill`, `toString(`*`T`*`[] arr)`, where *`T`* is a primitive or a class data type, and `asList(Object[] arr)`. The `Collections` methods work with `List`s, such as `ArrayList`, and other Java collections.

Exercises

1. Describe the difference between searching and pattern recognition.

2. Suppose a class `Person` implements `Comparable<Person>`. A `Person` has the `getFirstName()` and `getLastName()` methods; each of them returns a `String`. Write a `compareTo` method for this class. It should compare this person to another by comparing their last names; if they are equal, it should compare their first names. The resulting order should be alphabetical, as in a telephone directory.

3.■ Make the `Fraction` objects, defined in Chapter 9, `Comparable`. The natural ordering for `Fraction`s should be the same as for the rational numbers that they represent. Also redefine the `equals` method to make it agree with this ordering.

4. (a) Write a class `QuadraticFunction` that represents a quadratic function $ax^2 + bx + c$, with integer coefficients *a*, *b*, and *c*. Provide a constructor with three `int` parameters for *a*, *b*, and *c*. Provide a method double `valueAt(double x)` that returns the value of this quadratic function at *x*. Also provide a `toString` method. For example,

```
System.out.println(new QuadraticFunction(1, -5, 6));
```

should display

```
x^2-5x+6
```

Continued

(b) Override the `equals` method in the `QuadraticFunction` class. Two `QuadraticFunction`s should be considered equal if their respective coefficients are equal.

(c) Make the `QuadraticFunction` objects `Comparable`. The `compareTo` method should first compare the *a*-coeffients; if equal, then compare the *b*-coefficients; if also equal, compare the *c*-coefficients. (This ordering basically defines which function will have greater values for very large *x*.)

(d)■ Define a comparator class for comparing two `QuadraticFunction` objects. Provide two constructors: a no-args constructor and a constructor that takes one `double` parameter. When a comparator is created by the no-args constructor, it should compare two `QuadraticFunction`s based on their values at $x = 0$; when a comparator is created by the constructor with a parameter *x*, it should compare `QuadraticFunction`s based on their values at *x*.

5. Describe a situation where the performance of Sequential Search on average is better than $O(n)$. ≤ Hint: different target values in the array do not have to come with the same probability. ≥ ✓

6. What is the number of comparisons in Binary Search required in the worst case to find a target value in a sorted array of 80 elements? Consider two scenarios:

(a) We know for sure that the target is always in the array; ✓
(b) The target may be not in the array. ✓

7.■ A string contains several X's followed by several O's. Devise a divide-and-conquer method that finds the number of X's in the string in $\log_2 n$ steps, where *n* is the length of the string.

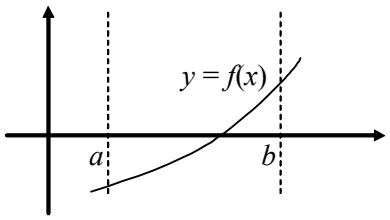
8.■ Write a recursive method that tries to find `target` among the elements `a[m]`, ..., `a[n-1]` of a given array (any array, not necessarily sorted).

```
public static int search(int[] a, int m, int n, int target)
```

If found, the method should return the position of the target value; otherwise it should return `-1`. The base case is when the searching range is empty ($m \geq n$). For the recursive case, split the searching range into two approximately equal halves and try to find the target in each of them.

9.■ Write a recursive implementation of Binary Search.

10.■ A divide-and-conquer algorithm can be used to find a zero (root) of a function. Suppose a function $f(x)$ is a continuous function on the interval $[a, b]$. Suppose $f(a) < 0$ and $f(b) > 0$:



The graph of the function must cross the x-axis at some point. We can split the segment into two halves and continue our search for a zero in the left or the right half, depending on the value of $f(x)$ in the middle point $x = (a + b) / 2$.

Write a program that finds x (to the nearest .001), such that $x = \cos(x)$.
≤ Hint: consider the function $f(x) = x - \cos(x)$ on the interval $[0, \pi/2]$. ≥ ✓

11.◆ An array originally contained different numbers in ascending order but may have been subsequently rotated by a few positions. For example, the resulting array may be:

```
21 34 55 1 2 3 5 8 13
```

Is it possible to adapt the Binary Search algorithm for such data?

12. Mark true or false and explain:

(a) If the original array was already sorted, 190 comparisons would be performed in a Selection Sort of an array containing 20 elements. _____ ✓

(b) Mergesort works faster than Insertion Sort on any array. _____ ✓

13. Write a method `void shuffle(Object[] arr)` that shuffles the objects in `arr` in random order. ≤ Hint: the algorithm is very similar to Selection Sort, only on each iteration, instead of finding the largest element, you choose a random one. ≥

14. An array of six integers — 6, 9, 74, 10, 22, 81 — is being sorted in ascending order. Show the state of the array after two iterations through the outer `for` loop in Selection Sort (as implemented in Figure 13-5).

15. What are the values stored in an array `a` after five iterations in the `for` loop of the Insertion Sort (as implemented in Figure 13-6) if the initial values are 3, 7, 5, 8, 2, 0, 1, 9, 4, 3? ✓

16. What is the state of the array after the partitioning phase of the Quicksort algorithm, at the top level of recursion, if its initial values are

```
int[] a = {6, 9, 74, 10, 22, 81, 2, 11, 54};
```

and the middle element is chosen as a pivot? ✓

17. Add `Arrays`'s built-in sorting method to the *Benchmarks* program to see how it compares to our own sorting methods.