IN THIS CHAPTER

» Looking at a slow way to sort

» Finding a faster way to sort

Chapter 3 Sorting

The world is full of computer science majors who don't know anything more about computers than you do, but they once attended a class in which the instructor explained how sorting algorithms worked. They may have earned a C in that class, but it was good enough to graduate.

Now you have a chance to find out what you missed by not majoring in computer science. I'm going to show you two popular algorithms for sorting data. The first, called Bubble Sort, is the simplest, but it isn't very efficient. The second, called Quicksort, is much more efficient and makes very clever use of recursion, which you learned about in Chapter 2 of this minibook.

REMEMBER

For most of us, figuring out how sorting algorithms such as Quicksort work is merely an intellectual exercise. The Java application programming interface (API) has sorting already built in (check out the `Arrays.sort` method, for example). Those sort routines are way better than any that you or I will ever write.

Looking at the Bubble Sort Algorithm

The simplest way to sort a set of values into order is to use a venerable algorithm called *Bubble Sort.* A Bubble Sort works by looking at the values to be sorted two at a time and swapping them if they aren't already in order. It's not very efficient, but it is easy to understand.

Here's a basic version of Bubble Sort for sorting an array of integer values:

1. **Start with the first and second numbers in the array; if the first number is greater than the second number, swap them.**
2. **Now look at the second and third numbers; if the second number is greater than the third number, swap them.**
3. **Continue comparing and swapping pairs of numbers until you've reached the end of the array.**
4. **Return to the start of the array and repeat the entire process, except this time stop at the next-to-last pair of numbers in the array, because the first pass through Steps 1 through 3 guarantees that the largest value in the array will now be in the final position.**
5. **Continue repeating the process, ignoring the the last *n* numbers for each pass through the array.**

   Eventually, you'll have sorted all the numbers in the array.

Here's how you can express the Bubble Sort algorithm in Java:

```
public static int[] SortArray(int[] a)
{
    int n = a.length;
    for (int i = 0; i < n-1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
    return a;
}
```

In the preceding example, the `SortArray` method accepts an array of integers and returns a copy of the array in sorted order. The method consists of two `for` loops, one nested inside the other. The outer loop touches all the elements in the array; the inner loop touches the first *j* elements, where *j* is the length of the array minus *i*.

In the innermost loop, a pair of values in the array — the current value and the next value — are compared. If they're out of order (that is, if the first element is greater than the second element), the elements are swapped. A temporary variable is used to hold the value of the first element in the pair while the elements are swapped.

Here's a snippet of code that calls the `SortArray` method to sort a small array of just ten elements:

```
int[]array = {23, 5, 94, 74, 10, 31, 58, 66, 42, 81};
PrintArray(array);
int[] sorted = SortArray(array);
PrintArray(sorted);
```

Here, the array is initialized to a set of ten integer values that are not in order. The array is printed by a call to a method named `PrintArray`. Then, the `SortArray` method is called to sort the array, saving the result in a new array named `sorted`. Finally, `PrintArray` is called again to print the array.

Here's the `PrintArray` method used to print the array:

```
public static void PrintArray(int[] a)
{
    for (int i = 0; i < a.length; i++)
    {
        System.out.print(a[i]);
        if (i < a.length - 1)
        {
            System.out.print(", ");
        }
    }
    System.out.println();
}
```

The `PrintArray` method prints the values of the array separated by commas on a single line. When the previous code snippet is run, the following appears on the console:

```
23, 5, 94, 74, 10, 31, 58, 66, 42, 81
5, 10, 23, 31, 42, 58, 66, 74, 81, 94
```

The first line shows the contents of the array before it's sorted; the second line shows the sorted array.

Earlier in this section, I mention that the Bubble Sort algorithm is not very efficient. To demonstrate, here's a revised version of the SortArray method that keeps count of how many steps are required to sort the array passed to it. Prior to returning the sorted array, this version prints the size of the array and the number of steps required to sort it on the console:

```
public static int[] SortArray(int[] a)
{
    int steps = 0;
    int count = a.length;
    int n = a.length;
    for (int i = 0; i < n-1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
            steps++;
        }
    }
    System.out.println("Sorted " + count + " integers in "
                       + steps + " steps.");
    return a;
}
```

Before I show you the code that tests the SortArray method with these additions, I want to show you another method I added to initialize an array of any given size with random integers ranging between 0 and 999:

```
public static int[] RandomizeArray(int size)
{
    int[] array = new int[size];
    for (int i = 0; i < size; i++)
    {
        array[i] = (int) (Math.random() * 999 + 1);
    }
    return array;
}
```

And finally, here's a snippet of code that calls these methods to create and sort arrays with 1, 10, 100, 1,000, and 10,000 elements to see how many steps are necessary to sort them:

```
SortArray(RandomizeArray(1));
SortArray(RandomizeArray(10));
SortArray(RandomizeArray(100));
SortArray(RandomizeArray(1000));
SortArray(RandomizeArray(10000));
```

When this code is run, the following output is displayed on the console:

```
Sorted 1 integers in 0 steps.
Sorted 10 integers in 45 steps.
Sorted 100 integers in 4950 steps.
Sorted 1000 integers in 499500 steps.
Sorted 10000 integers in 49995000 steps.
```

It's hard to gain perspective on these numbers in this format, so here they are properly aligned:

Number of Elements	Steps Required to Sort
1	0
10	45
100	4,950
1,000	499,500
10,000	49,995,000

In approximate numbers, when you increase the size of the array by a multiple of 10, the number of steps increases by a multiple of 100.

As a general rule, the Bubble Sort algorithm scales with the square of the input size.

Introducing the Quicksort Algorithm

Quicksort is an improvement over Bubble Sort, which performs much faster. The Quicksort technique sorts an array of values by using recursion. Its basic steps are as follows:

1. **Pick an arbitrary value that lies within the range of values in the array.**

   This value is the *pivot point.* The most common way to choose the pivot point is to simply pick the first value in the array. Folks have written doctoral degrees