WORKED EXAMPLE 14.1 Enhancing the Insertion Sort Algorithm

In this Worked Example, we will implement an improvement of the insertion sort algorithm of Special Topic 14.1, called *Shell sort* after its inventor, Donald Shell.

Shell sort is an enhancement of insertion sort that takes advantage of the fact that insertion sort is an $O(n)$ algorithm if the array is already sorted. Shell sort brings parts of the array into sorted order, and then runs an insertion sort over the entire array, so that the final sort doesn't do much work.

A key step in Shell sort is to arrange the sequence into rows and columns, and then to sort each column separately. For example, if the array is

65	46	14	52	38	2	96	39	14	33	13	4	24	99	89	77	73	87	36	81

and we arrange it into four columns, we get

65	46	14	52
38	2	96	39
14	33	13	4
24	99	89	77
73	87	36	81

Now we sort each column:

14	2	13	5
24	33	14	39
38	46	36	52
65	87	89	77
73	99	96	81

Put together as a single array, we get

14	2	13	5	24	33	14	39	38	46	36	52	65	87	89	77	73	99	96	81

Note that the array isn't completely sorted, but many of the small numbers are now in front, and many of the large numbers are in the back.

We will repeat the process until the array is sorted. Each time, we use a different number of columns. Shell had originally used powers of two for the column counts. For example, on an array with 20 elements, he proposed using 16, 8, 4, 2, and finally one column. With one column, we have a plain insertion sort, so we know the array will be sorted. What is surprising is that the preceding sorts greatly speed up the process.

However, better sequences have been discovered. We will use the sequence of column counts

$$\begin{aligned} c_1 &= 1 \\ c_2 &= 4 \\ c_3 &= 13 \\ c_4 &= 40 \\ &\ldots \\ c_{i+1} &= 3c_i + 1 \end{aligned}$$

That is, for an array with 20 elements, we first do a 13-sort, then a 4-sort, and then a 1-sort. This sequence is almost as good as the best known ones, and it is easy to compute.

We will not actually rearrange the array, but compute the locations of the elements of each column.

For example, if the number of columns c is 4, the four columns are located in the array as follows:

65				38				14				24				73			
	46				2				33				99				87		
		14				96				13				89				36	
			52				39				4				77				81

Note that successive column elements have distance c from another. The *k*th column is made up of the elements a[k], a[k + c], a[k + 2 * c], and so on.

Now let's adapt the insertion sort algorithm to sort such a column. The original algorithm was

```
for (int i = 1; i < a.length; i++)
{
   int next = a[i];
   // Move all larger elements up
   int j = i;
   while (j > 0 && a[j - 1] > next)
   {
      a[j] = a[j - 1];
      j--;
   }
   // Insert the element
   a[j] = next;
}
```

The outer loop visits the elements a[1], a[2], and so on. In the *k*th column, the corresponding sequence is a[k + c], a[k + 2 * c], and so on. That is, the outer loop becomes

```
for (int i = k + c; i < a.length; i = i + c)
```

In the inner loop, we originally visited a[j], a[j - 1], and so on. We need to change that to a[j], a[j - c], and so on. The inner loop becomes

```
while (j >= c && a[j - c] > next)
{
   a[j] = a[j - c];
   j = j - c;
}
```

Putting everything together, we get the following method:

```
/**
   Sorts a column, using insertion sort.
   @param a the array to sort
   @param k the index of the first element in the column
   @param c the gap between elements in the column
*/
public static void insertionSort(int[] a, int k, int c)
{
   for (int i = k + c; i < a.length; i = i + c)
   {
      int next = a[i];
      // Move all larger elements up
      int j = i;
      while (j >= c && a[j - c] > next)
      {
            a[j] = a[j - c];
            j = j - c;
         }
         // Insert the element
         a[j] = next;
      }
   }
```

Now we are ready to implement the Shell sort algorithm. First, we need to find out how many elements we need from the sequence of column counts. We generate the sequence values until they exceed the size of the array to be sorted.

```
   ArrayList<Integer> columns = new ArrayList<Integer>();
   int c = 1;
   while (c < a.length)
   {
      columns.add(c);
      c = 3 * c + 1;
   }
```

For each column count, we sort all columns:

```
   for (int s = columns.size() - 1; s >= 0; s--)
   {
      c = columns.get(s);
      for (int k = 0; k < c; k++)
      {
         insertionSort(a, k, c);
      }
   }
```

How good is the performance? Let's compare with the `Arrays.sort` method in the Java library.

```
   int[] a = ArrayUtil.randomIntArray(n, 100);
   int[] a2 = Arrays.copyOf(a, a.length);

   StopWatch timer = new StopWatch();

   timer.start();
   ShellSorter.sort(a);
   timer.stop();

   System.out.println("Elapsed time with Shell sort: "
         + timer.getElapsedTime() + " milliseconds");

   timer.reset();
   timer.start();
   Arrays.sort(a2);
   timer.stop();

   System.out.println("Elapsed time with Arrays.sort: "
         + timer.getElapsedTime() + " milliseconds");

   if (!Arrays.equals(a, a2))
   {
      throw new IllegalStateException("Incorrect sort result");
   }
```

We make sure to sort the same array with both algorithms. Also, we check that the result of the Shell sort is correct by comparing it against the result of `Arrays.sort`

Finally, we compare with the insertion sort algorithm.

The results show that Shell sort is a dramatic improvement over insertion sort:

```
Enter array size: 1000000
Elapsed time with Shell sort: 205 milliseconds
Elapsed time with Arrays.sort: 101 milliseconds
Elapsed time with insertion sort: 148196 milliseconds
```

However, quicksort (which is used in `Arrays.sort`) outperforms Shell sort. For this reason, Shell sort is not used in practice, but it is still an interesting algorithm that is surprisingly effective.

You may also find it interesting to experiment with Shell's original column sizes. In the sort method, simply replace

```
c = 3 * c + 1;
```

with

```
c = 2 * c;
```

You will find that the algorithm is about three times slower than the improved sequence. That is still much faster than plain insertion sort.

You will find a program to demonstrate and compare Shell sort to insertion sort in the `worked_example_1` folder of the book's companion code.