Automatic Array Algorithm Animation Lab

Overview

In this laboratory, you will program several array algorithms in an object-oriented framework. You will define your algorithms to operate on a class `IntArray` that encapsulates an ordinary Java array `int[]` of integer. You will test your algorithms by running bar chart animations. You will not however have to write any animation code. Because the class `BarChart` inherits from the class `IntArray`, the animations will work automatically as soon as you install your algorithms in the `Algorithms Test` program.

Below is a screen snapshot of the `Algorithms Test` program:

The algorithms that you will program are `RotateLeft`, `RotateRight`, `Reverse`, `InsertionSort`, `SelectionSort`, `QuickSort`, `QuickInsertSort`, and `MergeSort`. You will be given 2 algorithms, `InvertValues` and `BubbleSort`, to serve as examples. You should be aware that `BubbleSort` is one of the slowest sort algorithms known and should never be used in a practical program. Its value is purely to illustrate the Java code needed to define an algorithm in this exercise.
The Critical Functions of the Class `IntArray`

Since the data arrays used for testing will be created in the `Algorithms Test` program, there are only three critical functions that you need to aware of in the class `IntArray`:

```java
public int getValue(int i);
public void setValue(int i, int value);
public int length();
```

The first two functions allow you to extract the value of an element in the encapsulated data array and to set such an element to a new value. The third function returns the length of the encapsulated array. These three functions form the basis for all of the array algorithms.

The Algorithmic Framework Provided by `IntArrayAlgorithm`:

The class `IntArrayAlgorithm` provides a framework so that you do not need to repeat standard code when defining each individual algorithm.

The first facility offered by `IntArrayAlgorithm` is to maintain a `name` for each algorithm that may be used, for example, to name the button for the algorithm in the user interface. The `name` is passed as a `String` to the constructor.

The second facility offered by `IntArrayAlgorithm` is to define an `algorithmic template` to simplify the algorithm definition process. Let us explain this by illustrating the method calls. Let `data` be an object of class `IntArray` and `algorithm` be an object of class `IntArrayAlgorithm`. Then there are two ways we might want to use the `algorithm` on the `data`. One is to apply the `algorithm` to all of the `data` and the other is to apply the `algorithm` to a specific range of values within the `data`. This corresponds to two possible method calls:

```java
algorithm.perform(data);            // perform on all of the data
algorithm.perform(data, min, max);  // perform on min <= i <= max
```

The `IntArrayAlgorithm` class supports these variations with two methods as follows:

```java
public final void perform (IntArray data) {
    if (data != null)        perform (data, 0, data.length() - 1);
}
public abstract void perform(IntArray data, int min, int max);
```
Notice that the `perform` method with one parameter is set up to call the `perform` method with three parameters where `min` is 0 and `max` is `data.length() - 1`. In other words, the one-parameter method sets up the range of values to cover the whole `data` array and then uses the three-parameter method. This means that your task is much simpler. *You need only program the three-parameter `perform` method.*

The one-parameter `perform` method is declared `final`. This means that *you may not alter its definition in a derived class*. The relationship between the one-parameter `perform` method and the three-parameter `perform` method is frozen once and for all.

In contrast, the three-parameter `perform` method is declared `abstract`. This means that the class `IntArrayAlgorithm` *makes no pretense of knowing what algorithm to define*. If you use this class, you *must provide the definition* of the three-parameter `perform` method. If you wish, you may also define additional methods to help with the definition of the three-parameter `perform` method.

The Examples *InvertValues* and *BubbleSort*

Let us now examine the code of the two examples that are provided in the Java source file `IntArrayAlgorithms.java`. We start first with *InvertValues*.

```java
public static IntArrayAlgorithm invertValues
    = new IntArrayAlgorithm("Invert Values")
    {
        public void perform(IntArray data, int min, int max) {
            if (data == null) return;

            int sum = data.getMinimum() + data.getMaximum();

            for (int i = min; i <= max; i++)
                data.setValue(i, sum - data.getValue(i));
    }
};
```

The *algorithmic object* `invertValues` is defined to be an `IntArrayAlgorithm`. To create this object, we use the constructor that accepts the name of the algorithm as a `String`, in this case, “Invert Values”. We must also define the three-parameter `perform` method. To do this, we use the technique of *on-the-fly method definition* by opening a pair of braces `{ }` and placing the method definition inside.

The `perform` method begins with an error check to see if the `data` parameter is `null`. Then the algorithm begins. The key idea is that to invert values we wish to take an element of *minimum value* and make it *maximum*, an element of *maximum value* and make it *minimum*, and linearly interpolate in between. This can be done as follows:
Replace each element value \( v \) with \( \text{minimum} + \text{maximum} - v \).

The details of the method perform just this replacement for each array element in the range \( \text{min} \leq i \leq \text{max} \). Notice that we use \textit{getValue} to obtain the current value of the \( i \)-th element and \textit{setValue} to replace that value with the new value.

Let us next examine \textit{BubbleSort}.

```java
public static IntArrayAlgorithm bubbleSort = new IntArrayAlgorithm("Bubble Sort")
{
    public void perform(IntArray data, int min, int max) {
        if (data == null)
            return;
        for (int i = min; i < max; i++)
            bubbleDown(data, i, max);
    }
    protected void bubbleDown(IntArray data, int i, int max) {
        int v = data.getValue(max);
        int w;
        for (int j = max - 1; j >= i; j--)
        {
            w = data.getValue(j);
            if (w <= v) {
                v = w;
            } else {
                data.setValue(j + 1, w);
                data.setValue(j, v);
            }
        }
    }
};
```

The three-parameter \textit{perform} method has as its main task the execution of an \textit{ascending loop} that hands over the work of the \( i \)-th stage to the helper method \textit{bubbleDown}. This technique of having the \textit{perform} method simply establish the main loop and having the helper method define any nested loops simplifies the structure of the algorithm and makes it much more readable.

The idea of \textit{BubbleSort} is that, at the \( i \)-th stage, one attempts to “bubble” the rightmost element in the array downwards from \( \text{max} \) towards position \( i \) except that if a smaller element is encountered then that element takes over as the “bubbling element”. The reason that \textit{BubbleSort} is especially inefficient is that the bubbling process takes two \textit{setValue} calls each time one elements bubbles past another. Once you program \textit{SelectionSort}, you will see how the idea of moving the smallest element in the range from \( i \) to \( \text{max} \) into position \( i \) can be implemented much more efficiently.

The main reason for presenting \textit{BubbleSort} is to illustrate the technique of using a main loop with a helper method. An auxiliary reason is to warn you that \textit{Bubble Sort} is one of the slowest sort algorithms known and should never be used in a practical program.
The Simple Algorithms

In this section, we will discuss the five simple algorithms RotateLeft, RotateRight, Reverse, InsertionSort, and SelectionSort that you must program. We will discuss the three more advanced algorithms later.

RotateLeft

Save the value in position min. Then for i in the range min < i <= max, move the value in position i down (or left) into position (i-1). Finally, put the value saved at the beginning into position max. This process is illustrated by the before and after snapshots below.

RotateRight

Save the value in position max. Then for i in the range max > i >= min, move the value in position i up (or right) into position (i+1). Finally, put the value saved at the beginning into position min. This process is illustrated by the before and after snapshots below.
Reverse
Swap the elements in positions min and max. Then swap the elements in positions (min+1) and (max-1). Continue this way until there are no more pairs to swap. This process is illustrated by the before and after snapshots below.

InsertionSort
The three-parameter perform method should run an increasing loop. The goal of the work at stage i is to have the elements from min to i sorted. You may assume that when stage i is begun the elements from min to (i-1) are already sorted. Thus, the work of stage i is to move the i-th element into its sorted position in the range from min to i and to move those elements to the left of position i that are larger than element at i one cell to the right. In other words, you must rotate right on the proper sub-range of min to i to achieve the sort.

In the snapshot, the first 5 bars have been sorted but the remaining bars have not even been inspected so their values will have to be inserted into position by moving earlier bars to the right.

SelectionSort
The three-parameter perform method should run an increasing loop. The goal of the work at stage i is to have the elements from min to i be sorted and contain the smallest (min-i+1) elements in the array. You may assume that this condition is true for (i-1) when the i-th stage is entered. Thus, the work of stage i is to locate the next smallest element in the range from i to max and to swap this element into position i.
The key to an efficient algorithm is to search for the next minimum element but to do no swaps until the search process is complete. The way to accomplish this is to run a loop and maintain in the variable \texttt{minvalue} the smallest value found so far and in another variable \texttt{minindex} the location at which \texttt{minvalue} was found. When the search loop is done, simply swap the values at \texttt{i} and \texttt{minindex}.

In the snapshot, the first 5 bars have been sorted and the remaining bars all have values larger than those in the first 5 bars. Hence, the first 5 bars never need to be examined again during the sort and will remain fixed.

\textbf{Note on Understanding the Sorts}

It is impossible to provide screen snapshots that can fully explain the two sorts. To get a better understanding of these sorts, run the sample solution program and see how each of the sorts works. We recommend small array sizes (25 -50) so that you can see clearly what is going on. Since, in animation, \texttt{InsertionSort} is slower than \texttt{SelectionSort}, we recommend not running \texttt{InsertionSort} on an array of size larger than 100.

\textbf{Installation of the Algorithms}

When you finish each algorithm, you must install the algorithm in the graphical user interface of the main program. We have simplified this process by encapsulating the work in a method \texttt{installActions()} and several helper methods. Look for the method \texttt{installActions()} in \texttt{AlgorithmsTest.java}. In this method, look for the lines below:

\begin{verbatim}
// installAlgorithm(IntArrayAlgorithms.rotateLeft, 0, 0);
// installAlgorithm(IntArrayAlgorithms.rotateRight, 1, 0);
// installAlgorithm(IntArrayAlgorithms.reverse, 2, 0);
installAlgorithm(IntArrayAlgorithms.invertValues, 3, 0);
installAlgorithm(IntArrayAlgorithms.bubbleSort, 0, 1);
// installAlgorithm(IntArrayAlgorithms.insertionSort, 1, 1);
// installAlgorithm(IntArrayAlgorithms.selectionSort, 2, 1);
// installAlgorithm(IntArrayAlgorithms.quickSort, 3, 1);
// installAlgorithm(IntArrayAlgorithms.quickInsertSort, 4, 1);
// installAlgorithm(IntArrayAlgorithms.mergeSort, 5, 1);
\end{verbatim}
As you complete each algorithm in `IntArrayAlgorithms.java`, remove the comment from the corresponding line in `installActions()` and the algorithm’s button will then appear in the GUI. You will only need to hand in the file `IntArrayAlgorithms.java` since the grader will use a main program with the comments removed for testing your work.

### The More Advanced Sorting Algorithms

After you have completed the simple algorithms, you will program two major algorithms, *QuickSort* and *MergeSort*, plus a variation of *QuickSort* that uses *InsertionSort* to sort ranges within the large array that have at most 16 elements. We want to prepare for this with some general considerations.

#### Recursive Split-Merge Algorithms

One of the grand principles in the field of algorithms is the method of *divide and conquer*. This method recommends solving a problem by *dividing the data into smaller parts*, solving the problem on these smaller parts, and then somehow using the information to solve the original problem on the full data set.

In this portion of this laboratory, you will sort a full array by *splitting* the array into two parts, sorting each part, and then *merging* the two parts in such a way that the full array becomes sorted. We will call such sorting algorithms *split-merge algorithms*.

*QuickSort* is one instance of this technique in which the main effort is in the split phase and the merge phase is entirely trivial. *MergeSort* is another instance of this technique in which the split phase is one line of code and the main effort is in the merge phase.

In both *QuickSort* and *MergeSort*, the work done on any subarray has exactly the same structure as the work done on the full array. This means that the process is recursive, that is, the steps of the process occur over and over again on smaller and smaller scales of data. In this situation, the method that implements the algorithm can invoke itself on the smaller and smaller data segments. On each subarray, it will invoke itself on even smaller subarrays. This will continue until some *if-statement* says that the subarray is small enough to be trivial (one element) or at least small enough to be handled by simpler means (such as *InsertionSort*) at which point the recursive process will stop.

To give an outline of how a *split-merge algorithm* is programmed, recall that we have set up an `IntArrayAlgorithm` so that you must supply the method:

```java
public abstract void perform(IntArray data, int min, int max);
```

Since this method uses the endpoints of a subarray of the full data array, it is set up to fit into a recursive framework. Hence we may outline the *split-merge* code as follows:
public abstract void perform(IntArray data, int min, int max) {
    if (data == null)
        return;
    if (max <= min)
        return;
    int s = split(data, min, max);
    perform(data, min, s);
    perform(data, s+1, max);
    merge(data, min, s, max);
}

The first if-statement is a standard null check. The second if-statement checks whether the recursive process has reached the trivial state where we are sorting one element. If not, we then invoke the algorithm’s split-process and get back an index $s$ that will be used to divide the array into two subarrays $min ... s$ and $s+1 ... max$. We then recursively invoke the perform method on these subarrays. Finally, after the recursion, we invoke the algorithm’s merge-process to complete the work.

In QuickSort, the split process is called partition and is generally provided as a separate static function since it is used in related algorithms. Furthermore, there is no merge process so the last line is omitted.

In QuickInsertSort, the second if-statement is replaced by

    if (max <= (min + cutoff)) {
        insertionSort.perform(data, min, max);
        return;
    }

so that the recursive process is terminated more quickly than in regular QuickSort and is replaced by the use of insertionSort on small subarrays.

In MergeSort, the split process computes the average of $min$ and $max$ and does nothing to the data array. Hence, the split method is replaced by inline code. However, the merge process is more complex since it requires a spare array to perform its work. This means that yet another helper perform method must be defined. We will explain this below.

Before we discuss the separate algorithms in greater detail, let us discuss the use of recursion in the above outline more carefully. The recursion occurs in two lines.

    perform(data, min, s);
    perform(data, s+1, max);

Notice how simple this looks. The subtle aspect of this simple code is that each invocation of perform uses a separate data area for its parameters and its local variables. Thus, multiple invocations of perform do not interfere with one another but instead do the tasks of the algorithm on different variables.
Thus, for example, in the call

\[
\text{perform}(\text{data}, \text{min}, s);
\]

the formal parameters of the called method (that we will name the *callee*) are set as follows:

- *data* in the callee is set to be identical to *data* in the caller;
- *min* in the callee is set to be identical to *min* in the caller;
- *max* in the callee is set to be the split position *s* in the caller.

Thus, the callee is ready to work on the caller’s sub-range *min* to *s*. The callee will of course introduce its own local variable *s* that will be quite distinct from the caller’s local variable *s*.

Similarly, in the call

\[
\text{perform}(\text{data}, s+1, \text{max});
\]

the formal parameters of the callee are set as follows:

- *data* in the callee is set to be identical to *data* in the caller;
- *min* in the callee is set to be the split position *s+1* in the caller;
- *max* in the callee is set to be identical to *max* in the caller.

Thus, the callee is ready to work on the caller’s sub-range *s+1* to *max*. The callee will of course introduce its own local variable *s* that will be quite distinct from the caller’s local variable *s*.

*The principle of recursion is to perform the same task on different data.* This is achieved by giving each call of a recursive function its own data area for its parameters and local variables. This guarantees that different recursive calls do not interfere with one another. Of course, *Java does not have to do anything different for recursive calls than it does for any other method calls. It is always the case that for any method call there is set up a separate data area for its parameters and local variables.* Thus, recursion simply exploits this general fact in a clever fashion.

**Details of QuickSort**

*QuickSort* has the fastest average sorting time of all algorithms designed for internal sorting of random data. This description is a mouthful so let us explain some of the phrases:

*fastest average sorting time*

This means that if you average the sort times over a large number of arrays of random test data of the same size, then *QuickSort* will score the best in terms of sort speed. This also means that *QuickSort* does not guarantee the best time in all
cases. Indeed in the worst cases, \textit{QuickSort} can be extremely slow. Hence, efforts must be taken to ensure that these worst cases are avoided if possible.

\textit{internal sorting}

This means that the sort uses internal computer memory and does not access external storage devices such as disk drives or tapes to assist in the sort.

\textit{random data}

This means that the data is assumed to have no special properties that would make the sort easier.

As an example of a special property that could make sorting easier, consider the situation in which you had an array of 900 numbers between 1 and 1000. There are clever schemes that could take advantage of the fact that you have a tight bound on the values of the numbers. \textit{QuickSort} uses no such special tricks.

The \textit{QuickSort} algorithm was discovered by C. A. R. Hoare in 1962, was studied by Robert Sedgwick in his 1978 thesis, and was tuned for maximum robustness and speed by Jon Bentley and Douglas McIlroy in 1993. The central idea of \textit{QuickSort} is the \textit{partition process} that you will implement in the \texttt{partition} method. After the partition process, the array is divided into 2 parts from \texttt{min} to \texttt{s} and from \texttt{s+1} to \texttt{max} where \texttt{s} is the split position. The critical condition that is achieved by the partition process is stated succinctly in the opening comments to the \texttt{partition} function:

```java
/**
 * The partition helper function for quick sort
 * and related algorithms.
 * If min < max then partition the data between
 * min and max and return s such that:
 * min <= s < max
 * if min <= u <= s and s+1 <= v <= max
 * then data.getValue(u) <= pivot <= data.getValue(v)
 */
public static int partition(IntArray data, int min, int max)
```

In other words, after the partition process, \textit{all elements in the range from min to s have values less than or equal to the values of all elements in the range from s+1 to max}. This means that if we now sort the range from \texttt{min} to \texttt{s} and the range from \texttt{s+1} to \texttt{max} then the full range will be sorted with no additional work. This is the genius of \textit{QuickSort}.

The \textit{partition process} may be described as follows. By some random means, choose an element in the array data to be sorted. \textit{Copy this element} to a spare location named \texttt{pivot}. We will call \texttt{pivot} the \texttt{partition} pivot. Although \texttt{pivot} is not returned from the \texttt{partition} method, its algorithm will in fact guarantee the stronger constraint:

```java
* if min <= u <= s and s+1 <= v <= max
  then data.getValue(u) <= pivot <= data.getValue(v)
```
The **partition process** consists of one or more *search-and-swap* steps that swap elements less than or equal to **pivot** to the left and elements greater than or equal to **pivot** to the right.

Prepare for this process by initializing search indices **i** and **j** with **i** set to **min** and **j** set to **max**. During the *search-and-swap* loop, increase **i** (via **i++**) and decrease **j** (via **j--**). This main loop will stop when **i** and **j** cross, that is, when **i** > **j**. Hence this loop may be controlled using a structure: ```while(i <= j) { ... }```.

The *search* phase consists of a pair of nested *while loops* within the main loop. In the first *nested while loop*, we increase **i** until we achieve the condition:

```
data.getValue(i) >= pivot
```

We say that the **data** is **large** at **i** meaning that it is at least as large as the **pivot**.

In the second *nested while loop*, we decrease **j** until we achieve the condition:

```
data.getValue(j) <= pivot
```

We say that the **data** is **small** at **j** meaning that it is at least as small as the **pivot**.

In the first main loop cycle, both of these nested loops must stop since the **pivot** is a value in the array and therefore the loops must stop at the index of the **pivot** if not sooner. After the first main loop cycle, there will be a **small element on the left** side of the array and a **large element on the right** side of the array and the presence of these elements prevents the indices **i** and **j** from leaving the range **min** ... **max** during the subsequent nested while loops.

After the two nested while loops, we enter the *swap* phase. If **i** <= **j** then we have a **large** element at **i** that is to the left of a **small** element at **j**. In this case, we need to swap these elements to achieve the goal of moving small elements to the left and large elements to the right.

Therefore, if **i** <= **j** we swap the data in positions **i** and **j** using a combination of **getValue** and **setValue** calls. After this swap, we must also increment **i** using **i++** and decrement **j** using **j--** so that we do not examine the data at positions **i** and **j** over and over again during the partition process. *What is the problem?* If the values at positions **i** and **j** both happen to equal the **pivot** value then we could fall into the trap of swapping these values back and forth endlessly and never finish the partition process. Failure to increment **i** and decrement **j** after the swap may cause an infinite loop.

When the main loop is complete, **j** < **i**, all elements to the left of **i** are **small**, and all elements to the right of **j** are **large**. More precisely:

```
For min <= u < i: data.getValue(i) <= pivot
For j < v <= max: pivot <= data.getValue(j)
```
The critical question is then: **What index should we return to specify the split position?** The temptation is to always return \( j \) but the correct answer is subtle. The problem is that it is possible for the value of \( j \) to equal \( \min - 1 \) after the execution of the main \texttt{while} loop! To return \( j \) in this case would violate the constraints on the \texttt{partition} function.

To understand this, here is an example in which this situation happens. Consider the following array with 5 elements:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
<td>56</td>
<td>41</td>
<td>92</td>
<td>28</td>
</tr>
</tbody>
</table>

Take \( \min = 0 \) and \( \max = 4 \) and assume that the random process that selects the pivot has led to the pivot value of 17. Now initialize \( i \) to 0 (\( \min \)) and \( j \) to 4 (\( \max \)). Since the value at \( i \) is the pivot itself, this value is large and the nested loop for \( i \) does not change \( i \). On the other hand, since each value to the right of 0 is strictly larger than the pivot, the nested loop for \( j \) decrements \( j \) until \( j \) equals 0 also. Since \( i \leq j \) is true, we swap 17 with itself and then increment \( i \) to 1 and decrement \( j \) to -1. At this point, the main loop terminates. Unfortunately, \( j \) is now an invalid index since it is outside of the range from \( \min \) to \( \max \). Hence, in this case, we cannot return \( j \).

By mathematical-parameter, it is possible to show that this example is representative of the situations that lead to \( j < \min \) after the main loop. These situations can occur only if the value at \( \min \) equals the pivot and all values to the right of \( \min \) are strictly larger than the pivot. Hence, if we return \( \min \) as the split index in this case, we will satisfy the constraints on the \texttt{partition} function.

Thus, the return statement of the \texttt{partition} function should read:

```java
if (j >= min)
    return j;
else
    return min;
```

The final point we need to make about the \texttt{partition} algorithm is the choice of the pivot value. It has been found that if the pivot value is chosen in a systematic manner (such as the value in the middle of the range \( \min \) to \( \max \)) then certain arrays that occur in practice will exhibit the worst case behavior for \textit{QuickSort}. Therefore, it is recommended that a random index position be used to minimize the chance of this behavior. Here is how to accomplish this:

```java
int index = MathUtilities.randomInt(min, max);
int pivot = data.getValue(index);
```

The class \texttt{MathUtilities} is part of the \textit{Java Power Tools}. 

---


©2001, College of Computer Science, Northeastern University, Boston MA 02115
Details of MergeSort

In the MergeSort algorithm, the data array is split simply be computing the average of min and max. Since no values within data are changed in this step, all of the sorting must be accomplished in the merge step.

The philosophy of the merge step is that, by the recursive processes, the values in the two subranges min ... s and s+1 ... max of data will already be sorted by the time the merge step is called. At this point, what one wishes to do is to repeatedly look at the next element in each subrange, pick the smaller of these two elements, and move this smaller element to the left. Unfortunately, there is no way to do this inside data itself without messing up the two subranges. Hence, it is necessary to make a spare copy of the values being sorted so that we can choose elements from the copy to insert back into the original data array.

The implications of these considerations is that we must build the copy array in the three-parameter perform method and then pass this to a four-parameter perform method that will actually do the recursive work! Since this is subtle, we will give you the exact recursive code:

```java
public void perform(IntArray data, int min, int max) {
    if ((data == null) || (max <= min))
        return;
    IntArray copy = new IntArray(data.length());
    perform(data, copy, min, max);
}

protected void perform (IntArray data, IntArray copy, int min, int max)
{
    if (max <= min)
        return;
    int s = min + (max - min)/2;
    perform(data, copy, min, s );
    perform(data, copy, s+1, max);
    merge(data, copy, min, s, max);
}
```

Notice how the three-parameter perform method makes the copy array and then hands off the real work to the four-parameter perform method. Notice further that the split is computed in the four-parameter perform method by simple arithmetic. Hence the heart of MergeSort is in the merge method. The signature of the merge method is:

```java
protected void merge(IntArray data, IntArray copy, int min, int s, int max)
```

We now describe the steps in the merge method.

Step 1: Copy the values in the data array in the range from min to max to the corresponding locations in the copy array. This is a simple for-loop using getValue and setValue.
Step 2: Prepare for the copy-back phase in which we must scan the 2 sub-ranges of the copy array (min ... s and s+1 ... max) and each time pick out the next smallest element to copy back into the data array. We must introduce 3 index variables to control the loops and the copying operations:

```c
int i = min;    // next spot in range min <= i <= s  to copy from
int j = s+1;    // next spot in range s+1 <= j <= max to copy from
int k = min;    // next spot in range min <= k <= max to copy into
```

Step 3: The while loop when both \( i \leq s \) and \( j \leq max \) are valid.

The idea is that we compare the value in copy at positions \( i \) and \( j \) and then copy the smaller of these values to position \( k \) in the data array. Then, depending on which value we have just copied, we increment \( i \) or \( j \). In all cases, we must increment \( k \).

When this loop terminates, we know that either \( i > s \) or \( j > max \) but without a further test we do not know which of these condition holds. It is easiest to ignore this question and just write two more loops with appropriate control conditions.

Step 4: The follow-up while loop when \( i \leq s \) is still valid

Now copy from position \( i \) in copy to position \( k \) in data and then increment \( i \) and \( k \).

Step 5: The follow-up while loop when \( j \leq max \) is still valid

Now copy from position \( j \) in copy to position \( k \) in data and then increment \( j \) and \( k \).

### Mixed QuickSort and InsertionSort

Some people believe that QuickSort is slow on small sub-ranges and that it is better to use InsertionSort on such sub-ranges. It is possible to measure the size of the interval between min and max and to switch to InsertionSort if this size is smaller than a cutoff value, say, 16. One way to do this has been suggested above in the introductory remarks to the more advanced algorithms. However you do this, you should copy and modify your QuickSort algorithm to implement this mixed algorithm that switches from a QuickSort strategy on large ranges to an InsertionSort strategy on small ranges.

### Thought Questions

InsertionSort and SelectionSort are usually programmed with loops. Is there any way to use recursion for these algorithms and is there any benefit to doing this (either in terms of ease of understanding or in efficiency)?