Chapter 12
Arrays II

In a previous chapter we looked at arrays with an eye towards using them to manage multiple objects in a game. In this chapter we’ll learn more array fundamentals and examine algorithms for searching and sorting arrays.

12.1 More Array Basics

Let's begin with more foundation material on arrays, followed by some simple and very useful algorithms.

12.1.1 Details, details...

In Chapter 9 we used simple int values to specify a position in an array. If `grectArray` is an array of GRects (clever name, eh?) we can display it with

```java
for(int i=0; i<grectArray.length; i++)
    println(grectArray[i]);
```

grectArray[i] is a GRect, so apply the GRect `toString` method and display the result

The index can also be specified with an int valued expression. For example, let's assume we have

```
intArray
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and display it in reverse order.

```java
for(int i=0; i<intArray.length; i++)
    println(intArray[intArray.length-i-1]);
```

Here's a trace table.

```
<table>
<thead>
<tr>
<th>i</th>
<th>intArray.length</th>
<th>i&lt;intArray.length</th>
<th>intArray.length-i-1</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>true</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>true</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
```
An array index may also be referred to as a **subscript**.

Remember the difference between an array and an element in the array. In our discussion above `intArray` refers to the **entire array**, which is an object and is treated as such. `intArray[i]` is a particular `int` in the array and is treated like any other `int`.

The **length** value of an array is a constant and so cannot be changed once the array is instantiated.

```
intArray.length = 100;
```

The size of an array can be set at run time.

```
double[ ] doubArray;
doubArray = new double[readInt("Size?")];
```

and the array variable can be reassigned by creating a new array.

```
doubArray = new double[50];
```

### 12.1.2 Initializing an Array

Arrays are automatically initialized to values appropriate for the data or object type—numeric values to 0, `bools` to `false`, `chars` to the ASCII character with value 0 and objects to `null`. We can also manually initialize if needed in the declaration statement, though this is cumbersome for big arrays. The array's **length** value will automatically be set.

```
double[ ] temps = {8.3, -14.6, 34.7};
Color[ ] colors = {Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE};
char[ ] secretCode = {'!', 'd', 'a', 'm', 's', '?', 'r', 'e', 't', 's', 'u', 'h', 'c', 's'};
```

And of course loops may be used to set an existing array to a fixed value.

```
for(int i=0; i<boolArray.length; i++)
{    boolArray[i] = true; }
```

or to a random value

```
for(int j=0; j<doubArray.length; j++)
{    doubArray[j] = rg.nextDouble(0, 1); } //assume rg is a RandomGenerator
```
12.1.3 Fundamental Array Algorithms

Now let's examine a few array related algorithms.

**Summing Array Values**
Summing the values in an array is very similar to summing algorithms we've used before. Here's a sample solution.

```java
double sum = 0;
for(double d : doubArray)
{    sum = sum + d;
}
```

**Finding the Maximum Value**
The algorithm for finding the maximum (or minimum) value in an array can be useful but it is very easy to write incorrectly. Below is an example of just such an incorrect implementation.

```java
public int findMaxWrong(int[ ] d)
{
    int maxValue = 0
    for(int i=0; i<d.length; i++)
    {    if (d[i] > maxValue) maxValue = d[i];
    return maxValue;
}
} //findMaxWrong
```

To figure out why it's wrong let's assume we have the array `d` below and look at a trace of the execution.

<table>
<thead>
<tr>
<th>d</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>18</td>
<td>2</td>
<td>20</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

and here is the trace of the execution of `findMaxWrong(d)`

<table>
<thead>
<tr>
<th>maxValue</th>
<th>i</th>
<th>d.length</th>
<th>i&lt;d.length</th>
<th>d[i]</th>
<th>d[i] &gt; maxValue</th>
<th>returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>true</td>
<td>7</td>
<td>true</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>true</td>
<td>18</td>
<td>true</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>5</td>
<td>true</td>
<td>2</td>
<td>false</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>5</td>
<td>true</td>
<td>20</td>
<td>true</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>5</td>
<td>true</td>
<td>12</td>
<td>false</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>5</td>
<td>false</td>
<td></td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

Hey, it works perfectly, returning the maximum value of 20. So we're good to go, right? Not so fast! Now try it with this data set.
Not so great! The maximum value in the array was actually -2, yet a zero was returned because that was our initial \texttt{maxValue} and all of the array data was negative. The algorithm doesn't work when the data is a bit different than we might have expected.

Now let's get it right. The key is to initially assign \texttt{maxValue} a value from the array, thus ensuring that it is a valid value. We have almost exactly the same code.

\begin{verbatim}
public int findMaxRight(int [ ] d)
{
    int maxValue = d[0];
    for(int i=1; i<d.length; i++)
    {
        if (d[i] > 0) maxValue = d[i];
    }
    return maxValue;
}
\end{verbatim}

Now run a trace table with either data set above. You'll find that this algorithm works correctly.

There's a larger principle to be learned here.

\begin{center}
\textbf{Testing can find errors, but it cannot prove correctness.}
\end{center}

\textbf{Copying an Array}

Remember that an array is an object. That means that array assignments do \textit{not} make a separate copy. If \texttt{x} and \texttt{y} are arrays then the statement

\begin{verbatim}
y = x;
\end{verbatim}

would give us

\begin{verbatim}
Wrong!
\end{verbatim}
Clearly this isn't correct as \( y \) is merely an alias of \( x \). This is a \textit{shallow copy}, which is a copy where the objects share data by sharing a reference to the location of the data.

Instead we must explicitly create a \textit{new} array and fill it, without sharing references. This is known as a \textit{deep copy}.

Here's an example for arrays of \texttt{ints}.

\begin{verbatim}
double[ ] y = copy1(x);

public double[ ] copy1(double [ ] a)
{
    double[ ] temp = new double[a.length];
    for(int i=0; i<temp.length; i++)
    {
        temp[i] = a[i];
    }
    return temp;
} //copy1
\end{verbatim}

If the arrays contained objects we would need to replace the

\begin{verbatim}
temp[i] = a[i];
\end{verbatim}

assignment statement with use of a copy constructor. For example, if we had an array of \texttt{Strings} then the array copy method would become

\begin{verbatim}
public String[ ] copy2(String [ ] a)
{
    String[ ] temp = new String[a.length];
    for(int i=0; i<temp.length; i++)
    {
        temp[i] = new String(a[i]);
    }
    return temp;
} //copy2
\end{verbatim}

12.2 Searching an Array

Arrays often consist of very large amounts of data—thousands or even millions of elements may be stored in the array. Searching is a very important task that must be done efficiently when we have a lot of data. We consider three search algorithms.
12.2.1 Linear Searching

A linear search is a search that begins at the beginning of the array and sequentially examines positions until the search target is found or the end of the array is reached.

Linear Searching an Unsorted Array

The simplest algorithm assumes that the elements of the array are not guaranteed to be in order. Searching becomes very simple

```java
traverse through every element of the array
{
    if the target element is found
        return the position at which the target was found (thus exiting the loop)
    }
return -1 to indicate target not found
```

Assume that we have the unsorted array lyrics below.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I'll</td>
<td>be</td>
<td>the</td>
<td>roundabout</td>
<td>the</td>
<td>words</td>
</tr>
</tbody>
</table>

Here's a sample call and the search code.

```java
int pos = linearSearch1(stringArray, "way");
if (pos != -1)
{
    println("Found at: " + pos);
}
else
{
    println("Not found");
}
```

```java
public int linearSearch1(String[] v, String t)
{
    //DO NOT ASSUME the array is in order
    for(int i=0; i<v.length; i++)
    {
        if (v[i].equalsIgnoreCase(t) == true) return i;  //found it!
    }
    return -1;
}
```

This search finds the first and only the first instance of the target value.

Linear Searching a Sorted Array

This algorithm assumes that the elements of the array are in order. That allows us to make the search more efficient.
For example, if our target was "sunday" then searching should stop after position 3 is examined because the ordered characteristic of the array means that "sunday" can't possibly be stored after the first "the". In general the assumption of ordering allows us to search as far as necessary and no farther.

Our algorithm is

traverse through every element of the array
{
    if the target element is found
        return position at which the target was found (exiting the loop)
    else if the target element is after the element of the array
        return -1 (indicating target not found)
}
return -1 to indicate target not found

And here's example code.

```java
public int linearSearch2(String[] v, String t)
{
    // MUST ASSUME the array in increasing order
    for(int i=0; i<v.length; i++)
    {
        if (v[i].equalsIgnoreCase(t) == true) return i;
        else if (v[i].compareToIgnoreCase(t) > 0) return -1;
    }
    return -1;
} // linearSearch2
```

This search also finds the first and only the first instance of the target value.

### 12.2.2 Binary Searching

The assumption of order in our array allows us another approach, the binary search, that vastly improves the efficiency of a search.

To understand the binary search let's think about doing a manual search of a drawer full of folders. The folder tabs have names on them, and the entire drawer is in alphabetical order front to back. We might have
Assume our search target is "hernandez" and pretend that we can't really see all of the names because the drawer is crowded.

Remember that the file drawer is in order and take a guess at the middle of the drawer. Let's say we pick position 7. That's "jordan", which is after "hernandez" in the alphabetical ordering of the files. We didn't find "hernandez" but we eliminated everything from the "jordan" file and afterwards in the file drawer. "hernandez" has to be somewhere in the first half of the file drawer, so we only have to search positions 0 through 6. Effectively we've cut the search space, the number of items to search, in half. Conceptually the relevant part of the array is now

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>able</td>
<td>carlson</td>
<td>davidson</td>
<td>edward</td>
<td>hernandez</td>
<td>jordan</td>
<td>keith</td>
</tr>
</tbody>
</table>

Pick the middle of folders 0 through 6—the "davidson" folder at position 3. That's too far forward in the ordering but we've now eliminated everything in positions 0 through 3. We have

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>edward</td>
<td>fini</td>
<td>hernandez</td>
</tr>
</tbody>
</table>

Continuing this process of halving the search space will eventually find the target or tell us that it is not in the array, as shown here. Here's an illustration of the process
The search space has one element. At this point either we've found our target or the target isn't in the array.

Our algorithm is

\[
\text{while (the array search space has at least 1 element)} \\
\{ \\
\quad \text{find middle of array} \\
\quad \text{if (middle value in the array is target)} \\
\quad \quad \text{return middle} \\
\quad \text{else if the middle value is greater than target value then} \\
\quad \quad \text{shrink array search space to the front half of current array search space} \\
\quad \text{else if} \\
\quad \quad \text{shrink array search space to the rear half of current array search space} \\
\}\]

\text{return -1 to indicate failure to find target}

and the code for the binary search of an array of \textbf{Strings} is
public int binarySearch(String [] v, String t) {
    int start = 0, end = v.length - 1, middle;
    while (start <= end) {
        middle = (start + end)/2;
        if (v[middle].equalsIgnoreCase(t) == true) return middle;
        else if (v[middle].compareToIgnoreCase(t) > 0) end = middle - 1;
        else start = middle + 1;
    }
    return -1;
} //binarySearch

Note that in this particular example the code searches without consideration of upper or lower case, using the 'IgnoreCase' versions of the String comparison methods. Thus 'dog', 'Dog' and 'DOG' would all be considered the same values.

It is very easy to write a binary search that doesn't work, perhaps not finding targets at the beginning or end of an array. Be sure to thoroughly test any code you write.

12.2.3 Searching Efficiency

A linear search of a sorted array cuts the size of the search space from N to N-1 with each comparison. A binary search of the same array cuts the size of the search space from N to N/2 with each comparison. The binary search is thus much faster.

The table below shows the maximum number of comparisons needed for linear and binary searches.

<table>
<thead>
<tr>
<th>array size</th>
<th>linear search</th>
<th>binary search</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>7</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>9</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>10</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>11</td>
</tr>
</tbody>
</table>

Math types might recognize the binary search numbers as $\log_2 2+1$, $\log_2 4+1$, $\log_2 8+1$, $\log_2 16+1$ and so on. The maximum number of comparisons grows logarithmically, which is pretty slow and that's why the binary search is so fast.
Below is a graph of this data, with the array sizes extended to more than one million elements.

The speed advantage of the binary search doesn't show up until the number of elements is large, but it really pays off then.

### 12.3 Sorting an Array

Sorting the contents of an array into ascending or descending order is a fundamental topic in computer science. There are many algorithms for sorting, from the simple to the very complex. The choice of algorithm is very important—one algorithm might take several minutes (or even hours) where another might run in fractions of a second. We'll look at three basic algorithms.

#### 12.3.1 The Bubble Sort

One of the simplest algorithms is the bubble sort. The bubble sort is not presented here as an attractive solution to the problem of sorting. In fact, it is one the slowest algorithms available that is not artificially clumsy, and it probably shouldn't be used except as an example. Please don't tell anyone you learned it from this book! The bubble short does have one advantage—*it is very easy to understand.*
Consider an array $x$ of 7 chars that we wish to put in ascending order, 'a' through 'z'. Here's the original array.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>u</td>
<td>e</td>
<td>v</td>
<td>a</td>
<td>p</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

We will sort this into order using several passes. Each pass traverses through the array, comparing adjacent elements to each other and switching them if they are out of order relative to each other. For an array of $n$ elements we describe the algorithm as

```plaintext
for n-1 passes
{
    for every adjacent pair of elements in the unsorted part of the array
    {
        if the elements are out of order relative to each other then swap them
    }
}
```

We consider each pass in excruciating detail.

<table>
<thead>
<tr>
<th>Pass</th>
<th>values compared</th>
<th>out of order</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td>original $\rightarrow$ t u e v a p f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t u e v a p f</td>
<td>no</td>
<td>t u e v a p f</td>
</tr>
<tr>
<td>2</td>
<td>t u e v a p f</td>
<td>yes</td>
<td>t e u v a p f</td>
</tr>
<tr>
<td>3</td>
<td>t e u v a p f</td>
<td>no</td>
<td>t e u v a p f</td>
</tr>
<tr>
<td>4</td>
<td>t e u v a p f</td>
<td>yes</td>
<td>t e u a v p f</td>
</tr>
<tr>
<td>5</td>
<td>t e u a v p f</td>
<td>yes</td>
<td>t e u a p v f</td>
</tr>
<tr>
<td>6</td>
<td>t e u a v p f</td>
<td>yes</td>
<td>t e u a p v f</td>
</tr>
</tbody>
</table>

Summary

After pass 1, the largest value has been placed at the end of the array. The remainder of the array is possibly out of order.

Pass 2 proceeds in exactly the same manner, except that we compare only up to position 4, comparing its value with the value in position 5. Why can we skip comparing position 5 with position 6?
## Pass 2

<table>
<thead>
<tr>
<th>values compared</th>
<th>out of order</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>e t u a p f v</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>no change</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>e t a u p f v</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>e t a p f u v</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>e t a p f u v</td>
</tr>
</tbody>
</table>

Summary: After pass 2, the largest 2 values have been placed at the end of the array, in order. The remainder of the array is possibly out of order.

And so on...

## Pass 3

<table>
<thead>
<tr>
<th>values compared</th>
<th>out of order</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>no change</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>e a t p f u v</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>e a p t f u v</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>e a p f t u v</td>
</tr>
</tbody>
</table>

Summary: After pass 3, the largest 3 values has been placed at the end of the array, in order. The remainder of the array is possibly out of order.

## Pass 4

<table>
<thead>
<tr>
<th>values compared</th>
<th>out of order</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>a e p f t u v</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>no change</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>a e f p t u v</td>
</tr>
</tbody>
</table>

Summary: After pass 4, the largest 4 values have been placed at the end of the array, in order. The remainder of the array is possibly out of order.

## Pass 5

<table>
<thead>
<tr>
<th>values compared</th>
<th>out of order</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>no change</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>no change</td>
</tr>
</tbody>
</table>

Summary: After pass 5, the largest 5 values have been placed at the end of the array, in order. The remainder of the array is possibly out of order.
Pass 6 values compared

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>e</td>
<td>f</td>
<td>p</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
</tbody>
</table>

out of order | result
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary

After pass 6 the largest 6 values have been placed at the end of the array, in order. The remaining value, at the beginning of the array, is the smallest value. The array is now sorted.

And finally here's the code that implements the algorithm on an array of `chars`.

```java
public void bubbleSort(char[] x)
{
    for(int pass=0; pass<x.length-1; pass++)
    {
        for(int pos=0; pos<x.length-pass-1; pos++)
        {
            if (x[pos] > x[pos+1])
            {
                char temp = x[pos];
                x[pos] = x[pos+1];
                x[pos+1] = temp;
            }
        }
    } //complete one pass
} //complete all passes
} //bubbleSort
```

Could the bubble sort be improved? Think about passes 5 and 6 in the example above and possible reasons for stopping the loop early. But the improvement would be pretty minor!

The comparison statement `x[pos] > x[pos+1]` will work as expected for the primitive numeric data types but note that `char` values will be compared using their ASCII values and `boolean` values cannot be compared with `>`, `>=`, `<` or `<=`. If we compare objects then a suitable comparison method, similar to the `String.compareTo( )` method will be required. If we’ve created our own class then we’ll probably need to write our own comparison method.

The data that is compared for the purposes of sorting is the sort key. In the examples above the sort key was the entire record of data to be rearranged in the array. If we have an array of objects, the sort key will consist of a value or combination of values derived from the object.

### 12.3.2 The Selection Sort

The selection sort algorithm is a modest improvement on the bubble sort. In each pass of this algorithm we identify the largest value in the unsorted portion of the array. That value is put at the end of the unsorted portion of the array. Then the unsorted portion is shrunk by one element and the process is repeated.
The algorithm is

*scan entire array for largest, put in last spot, swapping with what is there*

*scan remainder of array for largest, swapping with what is in second-to-last spot*

*continue in this manner till done*

or we might describe it as

*for all unsorted portions of the array*

*{*

*scan that portion to find the largest value*

*put that value at the end of that portion, swapping with what is there*

*}*

Below is a representation of the array as it is processed by the passes of the selection sort.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>t</td>
<td>u</td>
<td>e</td>
<td>v</td>
<td>a</td>
<td>p</td>
<td>f</td>
</tr>
<tr>
<td>Pass 1</td>
<td>t</td>
<td>u</td>
<td>e</td>
<td>v</td>
<td>a</td>
<td>p</td>
<td>f</td>
</tr>
<tr>
<td>result</td>
<td>t</td>
<td>u</td>
<td>e</td>
<td>f</td>
<td>a</td>
<td>p</td>
<td>v</td>
</tr>
<tr>
<td>Pass 2</td>
<td>t</td>
<td>u</td>
<td>e</td>
<td>f</td>
<td>a</td>
<td>p</td>
<td>v</td>
</tr>
<tr>
<td>result</td>
<td>t</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>a</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>Pass 3</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>result</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>Pass 4</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>result</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>Pass 5</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>result</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>Pass 6</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>result</td>
<td>a</td>
<td>p</td>
<td>e</td>
<td>f</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td><strong>After pass 6 the largest 6 values have been placed in order at the end of the array. The remaining value, at the beginning of the array, is the smallest value. The array is now sorted.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And finally we have the implementation.

```java
public void selectionSort(char [] x)
{
    char temp;
    //examine each value
    for(int i=0; i<x.length; i++)
    {
        //find largest element in array
        int maxPos = 0;

        for ( int j=1; j<x.length-i; j++)
        {
            if ( x[j]>x[maxPos] ) maxPos = j;
        }

        //swap max value with value in current position
        temp = x[maxPos];
        x[maxPos] = x[x.length-i-1];
        x[x.length-i-1] = temp;
    } //complete all passes
} //selectionSort
```

### 12.3.3 The Insertion Sort

The insertion sort algorithm is another improvement on the bubble sort. The basic algorithm is

```plaintext
for every element in the array
{

    insert that element it into the already sorted part of the array that is in front of
    that element, shuffling the elements of the array to make room

}
```

The table below shows the ascending order insertion sort applied to the same array of characters.
<table>
<thead>
<tr>
<th>Pass</th>
<th>Action</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insert u into [t]</td>
<td>t u e v a p f</td>
</tr>
<tr>
<td>2</td>
<td>Insert e into [t, u]</td>
<td>e t u v a p f</td>
</tr>
<tr>
<td>3</td>
<td>Insert v into [e, t, u]</td>
<td>e t u v a p f</td>
</tr>
<tr>
<td>4</td>
<td>Insert a into [e, t, u, v]</td>
<td>a e t u v p f</td>
</tr>
<tr>
<td>5</td>
<td>Insert p into [a, e, t, u, v]</td>
<td>a e p t u v f</td>
</tr>
<tr>
<td>6</td>
<td>Insert f into [a, e, p, t, u, v]</td>
<td>a e f p t u v</td>
</tr>
</tbody>
</table>

**Summary**

After pass 6 the largest 6 values have been placed at the end of the array, in order. The remaining value, at the beginning of the array, is the smallest value. *The array is now sorted.*

The code for an insertion sort is left for an assignment or internet search.

### 12.3.4 Sorting Efficiency

Choosing the right sorting algorithm is essential for ensuring satisfactory performance. But in truth it usually makes no difference if we're working with small amounts of data, perhaps 100 or 1,000 things to be sorted. With realistic amounts of data the choice of algorithm can make an enormous difference.

Below is data recently compiled on various sorting algorithms, including two that we haven't discussed, from a testing program run on a moderate performance desktop computer running Windows XP. Each array was filled with random strings of 10 characters A through Z.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Run Time (nearest second)</th>
<th>Array Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10,000</td>
<td>20,000</td>
</tr>
<tr>
<td>bubble</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>selection</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>insertion</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>shell</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>quicksort</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that the shell and quick sorts are radically faster than the other algorithms. The bubble, selection and insertion sorts are probably quick enough if the array size is small, but they rapidly become unacceptable.

Below is a graph of run time vs. number of elements.

So why not just use the either the shell or quick sort since they are so obviously faster than the other algorithms? The answer is a little complex and just a bit beyond CSI but here are some reasons:

- the shell and particularly the quick sort are more difficult to write
- not every algorithm is stable and not every algorithm can be used as a partial sort. In this book we'll ignore what the terms stable sort and partial sort mean, but they are important in some applications
- the quick sort is usually implemented with recursion, and not all programming languages support recursion, though Java does
when sorting something besides an array (disk files or dynamic data structures for example) not all of these algorithms are efficient or reasonable to implement

The take away is that there are many different sorting algorithms and we must choose wisely for a given task.

**Problem Set**

**Easy** – use functions for operations such as load, display, etc.

**ShowEvenPositions** Keyboard load an array of 10 characters. Display the values in positions 0, 2, ..., 8 in the array.

**TooBig1** Modify the program in showEvenPositions so that it attempts to print 20 characters from the array, which has room for only 10 characters. What happens?

**TooBig2** Modify the program in showEvenPositions so that it attempts to keyboard load 20 characters, even though the array is defined for 10. What happens?

**Greater10** Modify the program in showEvenPositions to load integers and then output only those integers > 10.

**ShowOdds** Modify the program in showEvenPositions to output only the odd integers in the array. Note: odd values in the array, such as 93, not odd indexes.

**LoadSquares** Load an array with the integers 12, 22, 32, ..., 1002 using a loop. Display the array.

**FindLargeSmall** Keyboard load an array of 10 integers. Then traverse the array once, finding the largest and smallest values in the array, as well as the sum of the integers. Display the max, min, sum and average.

**arrayEqual** Keyboard load two arrays of characters and test them for equality, using a

```c
bool isEqual(char [] array1, char [] array2); function that you write.
```

**Palindrome** A palindrome is a word that is spelled the same forward or backward, such as "radar". Remembering that a **String** can be treated much like an array, write a function

```c
bool isPalindrome(String s);
```

The function returns **true** if the **String** is a palindrome, **false** otherwise. Test the function on the words "mamma", "dad", "a", "dood", "doxoc" and "cool".

**StandardDev** The standard deviation of a list of numbers is a measure of how much the numbers deviate from the average. The standard deviation, \( r \), of a list of \( N \) numbers is defined as
Keyboard load an array with up to 10 integers. Then, using the above formula, calculate the standard deviation of the numbers and display the result.

**Moderate**

**SortComparison** – Write a program that creates an array of 50,000 random Strings. Sort the array using the bubbleSort and timing the execution. Sort the *same data set* using the selectionSort and the insertionSort, again timing the execution. To get and record the current system time we use a standard Java method as shown here.

```
long start = System.currentTimeMillis(); //get current time
```

`long` is a standard Java primitive data type that can hold a very large integer.

**InsertAndDelete** - write a program that declares an array of 10 strings. Now load the array of with 6 strings from the keyboard, putting them in positions 0 through 5 in the array, leaving positions 6 though 9 empty. Now present the user with a menu.

a) delete string – the user inputs an integer representing a position to delete. If the position is currently occupied then move all of the strings after this position one "forward", overwriting the values already there. Empty spots are filled with an empty string. If the position is not currently occupied the deletion is denied.

b) insert string – the user specifies a position and a string. If the position is already occupied, then the strings are all moved one position "backward", making room for the new string, which is inserted. The last item "drops off" the array if necessary. If the position is not already occupied the insertion is denied.

c) display array – neatly display the entire array.

Here’s a sample of the action on an array.

<table>
<thead>
<tr>
<th>position</th>
<th>after loading 6 names</th>
<th>delete 3</th>
<th>delete 7</th>
<th>insert 2 circle</th>
<th>after inserting 4 more at various locations</th>
<th>insert 5 spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dab tony george whiff snort box</td>
<td>dab tony george snort box</td>
<td>Error: Can’t delete position 7</td>
<td>The array is unchanged.</td>
<td>dab tony circle george snort box</td>
<td>dab line tony window circle yellow george snort star box</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hard**
IPA Clearly an array is a useful item but we can make it even more useful if we package it inside an object that also has useful methods. In this exercise you'll create an object suitable for working with arrays of ints. We'll call this object IPA for int power array. This object will provide many methods for for manipulating the IPA.

Write the code for the IPA as described, then write a thorough test program. In the interest of brevity, test only the methods your instructor specifies.

### the IPA object

#### fields

- **private int [ ] body** - an array for holding the ints of the IPA
- **int count** – an int for holding the number of ints actually present in body

#### constructors

- **IPA( )**
  - construct an IPA with a body that has capacity of 10
  - Example: `IPA ipa1 = new IPA( );`
- **IPA(n)**
  - construct an IPA with a body that has a capacity of n
  - Example: `IPA ipa21 = new IPA(40);`
- **IPA (IPA x)**
  - copy constructor for IPA object
  - Example: `IPA ipa3 = new IPA(ipa1);`

#### add methods – methods that add elements to the IPA, putting them into body

- **void add(int v)**
  - add v at the end of the body of the IPA.
  - Example: `ipa1.add(33);`
- **void add(int p, int v)**
  - add v at position p in the body of the IPA, shuffling array elements as necessary. If p is not valid halt the program with the statement `System.exit(0);`
  - Example: `ipa1.add(7, 33);`
- **void randomLoad(int minVal, int maxVal)**
  - load the body of the IPA with random ints in the range [minVal, maxVal]
  - Example: `ip1.randomLoad(1, 6);`

#### set and get methods – change or retrieve the value of a position in the body

- **void set(int p, int v)**
  - set position p in the body of the IPA to have the value v. If p is not valid halt the program
  - Example: `ipa1.set(7, 20);`
- **int get(int p)**
  - return the value at position p in the body of the IPA. If p is not valid halt the program
  - Example: `println(ipa2.get(4));`

#### delete methods – methods that delete values from body

- **void remove(int p)**
  - remove the value at position p in the body of IPA, shuffling the array elements as necessary. If p is not valid halt the program
  - Example: `ipa2.remove(4);`
- **void removeRange(int fromP, int toP)**
  - remove the values from position fromP to position toP in the body of the IPA, shuffling the array elements as necessary. If either fromP or toP is not valid or fromP > toP halt the program
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void clear()</code></td>
<td>remove all elements from the body of the IPA</td>
<td><code>ipa1.clear();</code></td>
</tr>
<tr>
<td><strong>search methods</strong> – methods that check for presence of a value in the body of the IPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>boolean contains(int v)</code></td>
<td>returns true if v is within the body of the IPA, false otherwise</td>
<td><code>if (ipa2.contains(x-32) == true) doSomething;</code></td>
</tr>
<tr>
<td><code>int indexOf(int v)</code></td>
<td>returns first position at which v is found within the body of the IPA, -1 if not found</td>
<td><code>int foundPos = ipa1.indexOf(m);</code></td>
</tr>
<tr>
<td><code>int lastIndexOf(int v)</code></td>
<td>returns last position at which v is found within the body of the IPA, -1 if not found</td>
<td><code>int lastFoundPos = ipa1.indexOf(t);</code></td>
</tr>
<tr>
<td><strong>size methods</strong> – methods related to the size of the body in the IPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>return true if there are no elements within the body of the IPA, false otherwise</td>
<td><code>if (ipa1.isEmpty() == false) doSomething;</code></td>
</tr>
<tr>
<td><code>int length()</code></td>
<td>return the length of the body of the IPA</td>
<td><code>int len = ipa1.length();</code></td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>return the number of elements in the body of the IPA</td>
<td><code>int sz = ipa1.size();</code></td>
</tr>
<tr>
<td><strong>miscellaneous methods</strong> – <code>equals</code>, <code>copy</code> and <code>toString</code> methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>boolean equals(IPA x)</code></td>
<td>return true if the body of this IPA is equal to (same elements in the same order) the body of the IPA x, false otherwise</td>
<td><code>if (ipa1.equals(ipa2) == false) doSomething;</code></td>
</tr>
<tr>
<td><code>static boolean equals(IPA x, IPA y)</code></td>
<td>static method that returns true if the body of IPA x is equal to (same elements in the same order) the body of the IPA y, false otherwise</td>
<td><code>if (equals(ipa1, ipa2) == true) doSomething;</code></td>
</tr>
<tr>
<td><code>IPA copy()</code></td>
<td>return a separate in memory copy of this IPA</td>
<td><code>IPA ipa4 = ipa1.copy();</code></td>
</tr>
<tr>
<td><code>static IPA copy(IPA x)</code></td>
<td>static method that returns a separate in memory copy of IPA x</td>
<td><code>IPA ipa5 = copy(ipa1);</code></td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>returns a String representation of the length of the body of this IPA, the count of elements actually within the the body of this IPA and the first three elements in the IPA. Of course, there may not be as many as 3 elements, so you must handle this inability.</td>
<td><code>println(ipa1);</code></td>
</tr>
</tbody>
</table>

Hey, guess what! If you've successfully implemented IPA then you've done a lot of the work of implementing a Java ArrayList object.

**LargelInt** Though the long data type provides for very large integers, it is certainly possible that a particular application might need even larger integers. Perhaps integers with as many as 100 digits. Create a largelInt object for the storage and the operations input, output, add and
subtracting largeInt (we'll use only 15 digits for ease of testing), using the general scheme shown below. Use functions for each of the major operations.

Store 1,234,567,890 as

<table>
<thead>
<tr>
<th>array position</th>
<th>...</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Operations that create values too large or too small to be stored should result in display of an error message and halting of the program. This is called integer overflow.

**VisualSortDemo**

Write a program to visual demonstrate the bubble sort algorithm. Represent integer values as rectangles with large values represented by taller rectangles (duh). As the array is sorted move the boxes. Below is a sample visual sort, perhaps of the array [15, 25, 33, 19, 8]

![Visual Sort Diagram]