Array Algorithms

Maintaining a collection of related data makes it especially easy to perform analytical operations on that data. This note describes some of the most basic and most common operations. You should master all of these techniques -- these are tools you need to have at your fingertips!

The algorithms in this note are demonstrated using arrays, but other types of collections could be used with only minor modifications. Focus on the fundamental principles of each algorithm as much as on the array-specific syntax.

Basic Iteration

Iterating through a collection -- examining each element in sequence -- is the most basic operation and is at the core of many algorithms. The following short bit of code illustrates the use of a counting loop to vary the array index over the full range of valid values.

```
int[ ] grades = { 95, 88, 99, 74, Math.round(77.8f), 65, 96 };

for ( int i = 0; i < grades.length; i++ ) {
    System.out.println( grades[i] );
}
```

Notice the pattern used in the *for* loop. This is a pattern you should memorize. The counting loop begins with the number 0 and counts as long as the count is less than the length of the array. Because this pattern uses the array length property rather than a magic number, it will work with an array of any length. This example also illustrates array initialization.

Sums and Averages

Adding things up or finding the average value are related operations. Both require a complete pass through the collection, 'touching' every element. Here is a method that takes an int array as a parameter and returns the average of all the values in the array, rounded to the nearest integer value:

```
public static int averageInts( int[ ] x ) // assumes length > 0
{
    int sum = 0;
    for ( int i = 0; i < x.length; i++ ) {
        sum += x[i];
    }
    return Math.round( sum / (double)x.length );
}
```

Notice the following:

- This method is declared ***static***. When can you do that? The simple rule is that any method that does not make use of instance variables may be declared *static*, and there is really no reason not to do so.
- Notice the data type of the parameter: **int[ ]** This means that any *int* array may be provided, and it says nothing about the length of the array. You always need to check the length of an array parameter -- you cannot make any assumptions about what it will be.
- The accumulator **sum** must be **initialized** before the beginning of the loop.
- The standard pattern for iterating through all elements in an array is used.

Lowest or Highest Value (or its Index)

Another very common algorithm determines the lowest (or highest) value in a collection. Here's an example:

```
public static int findLowestInt( int[ ] x ) // assumes length > 0
{

    int lowest = x[0]; //initialization -- assume first element is lowest
    //notice initialization on next line
    for ( int i = 1; i < x.length; i++ ) {
        if ( x[i] < lowest )
            lowest = x[i];
        }
    return lowest;

}
```

Notice the following:

- This algorithm also requires an accumulator, and like all accumulators, it must be initialized. However, the initialization here can be a little bit trickier. If you know that the array provided will always contain at least one element, you may initialize to the value of the first element, as has been done here. Otherwise, you need to use another strategy.
- The counting loop skips the first element of the array, since this has been handled with the initialization.
- An *if* statement inside the body of the loop determines whether or not the accumulator should be updated.
- Apart from changing the names of identifiers so they are still meaningful, only one character in this program needs to be changed to make it find the highest value instead of the lowest. What is it?

Many variations on any of these algorithms are possible. Suppose you wanted to find the element in an array of String objects that contains the fewest characters. Or perhaps you want to find the element in a String array that begins with the "highest" character (i.e. the letter closest to the end of the alphabet if the String values are all words). How would you adapt the algorithms to solve these problems? Why not try it and post your solutions in the forum!

One very important variation involves finding the **index** of the lowest (or highest) value instead of the value itself. This is especially important when working with an array of objects, as we will soon learn. By finding the index rather than the value itself, it becomes possible to retrieve a reference to the object itself, which can then be used in any way desired. We will soon discover that finding the lowest/highest index is also an important part of some sorting algorithms. Here is the method above modified to return the index of the lowest element:

```
public static int findLowestIntIndex( int[ ] x ) // assumes length > 0
{

	int lowestIndex = 0; //initialization -- assume first element is lowest
	//notice initialization on next line
	for ( int i = 1; i < x.length; i++ ) {
		if ( x[i] < x[lowestIndex] )
			lowestIndex = i;
		}
	return lowestIndex;

}
```

Make a point of noticing the several ways in which this method differs from the previous version.

Sequential Search

Another common operation is a simple, or sequential, search. Here's an example:

```
public static boolean findIntValue( int[ ] x, int target )
{

	for ( int i = 0; i < x.length; i++ ) {
		if ( x[i] == target )
			return true; // we don't need to search any more
	}
	return false; // we only get here if search failed

}
```

Notice the following:

- No extra initialization required; standard iteration loop used.
- The method terminates (with a return statement) as soon as a match for the target is found -- there is no need to continue searching.
- If the loop finishes normally, the search must have failed.
- A variation on this might return the index of the first element that matches the target, or some special code (frequently -1) if the target is not present. What would you need to change to make this happen? Give it a try!

Counting Things

A few changes to this search method yields a method that counts the occurrences of the target:

```
public static int findIntValueCount( int[ ] x, int target )
{
	int count = 0; // initialize the count
	for ( int i = 0; i < x.length; i++ ) {
		if ( x[i] == target )
			count++; // found one -- count it
	}
	return count;

}
```

Notice that no early exit is possible. We need to examine every element in order to ensure that the count is complete.

Sifting

Another variation on a simple search is a sifting operation. The key difference is that a sifting operation can find more than one match -- it's job is to find all values that match the specified criteria. Here's an example:

```
/**
* This method takes an array of ints and a threshold value.
* Prints to system.out the values of all elements from
* the array that are greater than or equal to the threshold.
*/
public static void printHighValues( int[ ] x, int threshold )
{
	System.out.print( "Values >= " + threshold + ": " );
	for ( int i = 0; i < x.length; i++ ) {
		if ( x[i] >= threshold )
			System.out.print( x[i] + " " );
	}
	System.out.println(); // go to next line

}
```

As with counting, no early exit is possible. The standard pattern for iterating through an entire array is used.

Methods that Return an Array

Many algorithms that operate on collections produce a new collection as a result. In these cases, the method needs to return an array as its result. Here's an example:

```
/**
* This method takes 2 double array parameters and returns
* a double array result. Each element of the result is the
* sum of the corresponding elements of the parameters. If
* the parameter arrays are of different lengths, the result
* has the length of the longer of the two.
*/
public static double[ ] addArrays ( double[ ] x, double[ ] y ) {

        int size = Math.max( x.length, y.length );
        int smaller = Math.min( x.length, y.length );
        double[ ] result = new double[size];

        double[ ] longer;
        if ( x.length > y.length ) {
                longer = x;
        } else {
                longer = y;
        }

        for ( int i = 0; i < smaller; i++ ) {
                result[i] = x[i] + y[i];
        }
        for ( int i = smaller; i < size; i++ ) {
                result[i] = longer[i];
        }
        return result;

}
```

Notice the following:

- The method's return type is **double[ ]** -- it returns an array. Nothing in this data type specifies the length of the array to be returned. That is determined by the arrays provided as parameters. However, the length needs to be known before the array to be returned can be constructed, because arrays are fixed-length collections.
- Another array variable named **longer** is used as an **alias** for the longer of the two array parameters. This makes the code that fills in the elements at the end of the **result** array easier to write.
- The two loops operating in sequence, one after the other, fill in every element in the **result** array.
- The *return* statement simply uses the variable **result**. This variable represents the entire array. Notice that **result** is declared with data type **double[ ]**, which matches the method's return type.

Swap Two Elements

An important basic array operation is swapping, or exchanging, two array elements. This simple operation is used in a number of different sorting algorithms. Here's a method that performs a swap:

```
public static void swap( int[ ] x, int a, int b )
{
    if ( a < 0 || a >= x.length || b < 0 || b >= x.length )
        throw new IndexOutOfBoundsException( "Illegal index" );

    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;

}
```

Notice the following:

- The parameters **a** and **b** represent **index values** -- the indexes of the elements to be swapped. You may wish to **validate** the indexes and do something appropriate if they are not valid, as illustrated.
- A temporary variable is always required to perform a swap.
- The change takes place inside the array object provided as the actual parameter of the method call. No return value is required because the array parameter itself is modified.

Using Computed Array Indexes

In every example so far, each array index is either the value of a variable or a literal number. However, many situations require the array index to be **calculated** in some way. One common type of calculation is an **offset** -- the index used is the value of a counter offset by some amount.

The following example rotates the elements in the array one position to the left. This is accomplished in the following way:

1. Copy element 0 into a temporary variable.
2. Shift each of the other elements one position to the left.
3. Copy the value from the temporary variable into the last element of the array.

For example, here's an array and the result after a rotate-left operation:

```
{ 32, 18, 27, 12, 56, 15, 49 } // original array
{ 18, 27, 12, 56, 15, 49, 32 } // result after rotate-left
```

The following method implements this operation:

```
public static void rotateLeft( int[ ] x ) // Assumes length > 0
{
    int temp = x[0]; // Initialization
    for ( int i = 1; i < x.length; i++ ) {
        x[i-1] = x[i];
```

```
        }
        x[x.length-1] = temp; // Finalization

}
```

Notice the following:

- The value of **x[0]** is stored in the variable **temp**.
- Shifting the remaining elements to the left requires **length-1** operations, so the loop count is started at 1.
- **x[i-1]** is an example of a subscripted variable with a computed index value.
- A finalization step is needed to get the last value in place.
- This algorithm is a cousin of the swap algorithm we examined above. Can you see the similarities? Like the swap method, this is a *void* method -- the changes are made right in the array itself.
- The comment says that the method assumes the length of the array is > 0. Take a minute to trace through this method in the case where the length of the array is 1. Convince yourself that the method works properly in this 'degenerate' case.

What changes would you need to make to this method to produce a rotateRight method? This is a little bit tricky. To figure it out, sketch an array on paper and then plan out what you would need to do in order to rotate the elements to the right. Hint: the loop will need to count down. Try writing and testing the code!