Chapter Fourteen Bonus Lessons: Algorithms and Efficiency

The following lessons take a deeper look at Chapter 14 topics regarding algorithms, efficiency, and "Big O" measurements. They can be completed by AP students after Chapter 14.

Lesson One: Common Algorithms

When you are given a problem, how do you choose a method for solving it? Let's say your history teacher tells you to write a paper on the history of the soybean. You aren't even sure what a soybean looks like! You could approach the problem in a number of different ways, including:

- Go to the library and read a book on soybeans.
- Find a local soybean farmer and interview him about his crops
- Talk to the Food Science Department at your local university
- Search for the word "soybean" on the Internet.

Although each of these methods solves the problem in a different way, they are all likely to produce reasonable answers. You might get slightly different results in each case, or one method might take more time than the others. The approach you usually select will be the one that gives you the best results for the least amount of time and effort. You will be faced with many problem-solving challenges as a computer programmer!

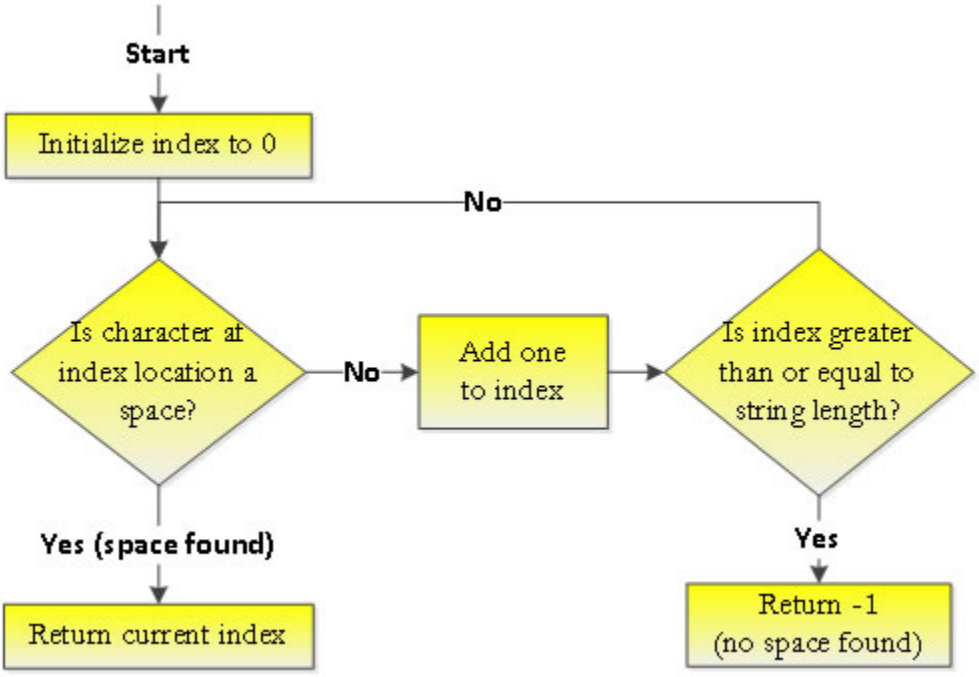
Algorithms and Flowcharts

It's important for computer programmers to learn to solve problems. Programmers are often given a problem description and will then need to develop a program to produce an answer. The exact sequence of steps your program takes to solve a problem is called an *algorithm*.

Sometimes things that seem natural and easy to a human are a little difficult to translate into computer code. A *flowchart* is a useful picture that helps break down your algorithm into a series of easily understood questions, answers, and actions. Simple flowcharts use three symbols:

- Diamonds contain questions or represent decision points in your code.
- Arrows represent program flow and contain text describing why that branch is taken
- Rectangles represent one or more actions to be taken

There are many other more advanced symbols used in formal flowcharts, but you can get quite a bit of use out of these three simple shapes! Let's consider a flowchart for an algorithm to find the first space in a string.



Both this flowchart and the paragraph below give the same description of the algorithm:

"To search for a space in a string, we first initialize some index to zero. We then use this index to check the character at that location within the string. If that location contains a space, we are done, so return that index value as the location of the space. If not, add one to the index. Then check the index to see if we have passed the end of the string. If not, go back and check the next characters. Otherwise, we have searched the entire string and not found any space, so just return -1."

Which description is easier to read? It depends on your personal preference, but many prefer the visual approach. Flowcharts are great tools for developing step-by-step algorithms that can then be easily translated into computer code.

Sometimes you might figure out more than one way to attack a problem. That means you can write more than one algorithm that will give you the correct answer. However, each algorithm may take a different amount of time and computer memory or space and may work better on certain kinds of input data. Often, it's best to choose the method that works the most efficiently with your data. In

other words, you want to choose an algorithm that solves the problem in the least amount of time and with the least amount of drain on the computer's resources.

Let's take a quick look at some common algorithms that can be solved with a computer program. As we develop each algorithm, try to think of some way to improve the proposed solution.

Factoring

Factors are numbers that can be multiplied together to result in a target value. For example, if our target value is 6 then the possible factors are 1 and 6 or 2 and 3. Put another way, factors are numbers that divide evenly (without remainder) into a target value.

Here is an algorithm to find the factors of a target value. Can you tell how it works?

```
public static void findFactors(int targetValue)
{
   System.out.println("The Factors of the target value are: ");
   for (int i = 1; i <= targetValue; i++)
   {
      if ( targetValue % i == 0)
      {
         System.out.print( i + " ");
      }
   }
}
```

If the target value is 12, the function will give us the following results:

```
The Factors of the target value are:
1 2 3 4 6 12
```

Summing a Series

In mathematics, a **series** is sum of values, where each value is determined by some function of an integer number. For example, if our function is "**f(n) = n**", then for "n" ranging from 1 through 10 the series is 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55.

The actual function of "n" may of course be more interesting like the examples shown to the right.

$$f(n) = \frac{1}{2n} \qquad f(n) = \frac{1}{2^n}$$

A series can be written with a special summation symbol as shown to the right. The range of "n" will start from the initial value specified on the bottom and run to the limit "m" shown on the top. The actual function of "n" will be displayed to the right of the summation symbol.

$$\sum_{n=1}^{m} f(n)$$

In pure mathematics it's often interesting to calculate the sum where "m" equals infinity (gets larger without bound). While it's possible to calculate the theoretical limits of some infinite sums, computers will need to stop at some point and declare themselves finished. So we are going to write an algorithm that accepts a lower bound (starting point) and upper bound (ending point) and calculate the series sum of the function "**f(n) = 1.0 / Math.pow(2 , n)**".

```
public static void findSeriesSum(int lowerBound, int upperBound)
{
   double sum = 0;
   for (int n=lowerBound; n<= upperBound; n++)
   {
      double function_n = 1.0 / Math.pow(2, n);
      sum += function_n;
   }

   System.out.println("Series result is " + sum);
}
```

If we call **findSeriesSum**() with the bounds (1, 5), (1, 10), and (1, 100), here are the results:

```
Series result is 0.96875
Series result is 0.9990234375
Series result is 1.0
```

Are these answers surprising? It looks like we aren't making any progress beyond 1.0 as a result no matter how large the upper bound becomes. This series is said to *converge* to a particular result, even if the upper bound is infinite.

Roots of a Quadratic Equation

A *quadratic equation* is a formula in the form "**$ax^2 + bx + c = 0$**". The constants **a**, **b**, and **c** are the *coefficients* of the powers of **x**. There are generally 0, 1, or 2 real values of **x** that will solve the equation. We can use the *Quadratic Formula* to solve for these values of **x**.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Fortunately, mathematical expressions like this are straightforward to write in computer code. You just need to make sure that no invalid operations will happen regardless of the data input. In this case, the value "**b^2 - 4ac**", which is called the *discriminant*, must be positive in order for the square root function to succeed with a real number. If the discriminant is positive there will be two real roots. If the discriminant equals zero there will be one real root. If the discriminant is negative there will be no real roots (though there can be *imaginary* roots).

We will check for these three special cases (positive, zero, and negative) in our coded algorithm.

```
public static void findRoots(double a, double b, double c)
{
   double discriminant = (b * b) - (4 * a * c);
   if ( discriminant > 0 )
   {
      double root1 = (- b + Math.sqrt(discriminant)) / (2 * a);
      double root2 = (- b - Math.sqrt(discriminant)) / (2 * a);
      System.out.println("The roots are unequal: "
                         + root1 + " and " + root2);
   }
   else if ( discriminant == 0 )
   {
      double root = - b / (2 * a);
      System.out.println("The roots are equal: "
                         + root + " and " + root );
   }
   else // discriminant < 0
   {
      System.out.println("The roots are imaginary");
   }
}
```

We can test our algorithm with a variety of inputs to exercise each case:

```
findRoots(2,4,2);
findRoots(3,5,1);
findRoots(1,1,4);
```

The results are:

```
The roots are equal: -1.0 and -1.0
The roots are unequal: -0.2324081207560018 and -1.434258545910665
The roots are imaginary
```

Generating Fibonacci Numbers

The last common algorithm we'll investigate is one to generate *Fibonacci numbers*. A Fibonacci series is a set of ordered numbers where the next number is equal to the sum of the previous two numbers. Fibonacci series normally start with 0 and 1 as the first two numbers. So you can easily calculate other numbers in the series:

```
0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

Of course a computer algorithm can't generate an infinite list of anything, so we will need to decide how many numbers in the series to generate. The number of entries is therefore the input parameter to our function. We can use a **for**() loop to conveniently generate the requested number of Fibonacci values.

```
public static void findFibonacci(int n)
{
   System.out.print("0 1 ");
   int nMinus2 = 0;
   int nMinus1 = 1;
   for (int i=2; i<n; i++)
   {
      int sum = nMinus2 + nMinus1 ;
      System.out.print(sum + " ");
      nMinus2 = nMinus1;
      nMinus1 = sum;
   }
}
```

We can call **findFibonacci**(20) to generate the first 20 numbers in the series:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Fibonacci series have a variety of interesting applications in computing, mathematics, biology, art, and architecture (the golden ratio).

Lesson Two: Algorithm Performance

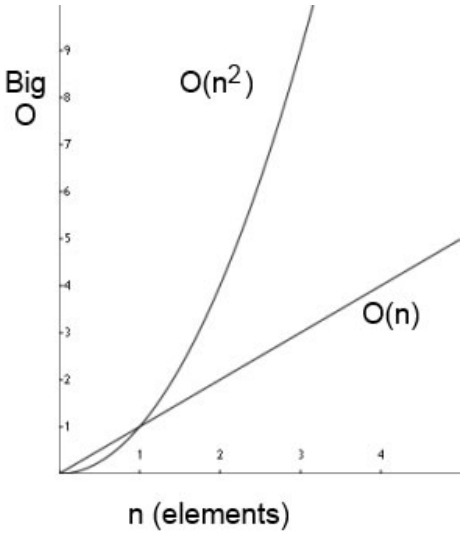
You have now written a variety of algorithms including sorting, searching, and other common mathematical algorithms. In each case we focused on writing code that was simple and clear. However, that does not mean the code was necessarily written in the most optimal manner! There may also be completely different ways to achieve the same results, as we saw with the different sorting algorithms.

Which code implementation should you select when trying to solve an algorithm? Which algorithm is most appropriate to apply to a particular data set? These questions generally revolve around the *performance* or *efficiency* of the code implementation. A function's performance and efficiency can be measured in *time* that it takes to run the function. It can also be measured in the *space* or *memory* it takes to hold all of the data needed. You would like to write code that takes as little time and space as possible.

Time and Space Efficiency in "Big O" Notation

A description of the computing resources required for a function can usually be given based on the size of the problem it's trying to solve. Most functions don't take much time or space in order to solve small problems with reasonably few data points. But the same functions could take a very long time as the number of data points increase.

Given a number of data points "n", the performance of a function can be described by "Big O" notation. Big O notation is a function **f(n)** that shows roughly how well a function will perform in time or space based on the number of data elements "n". Lower values for f(n) mean the algorithm performs better with that number of data elements, while higher values of f(n) means the algorithm takes more time or space.



The "O" in Big "O" notation is short for "order of" and is a rough approximation of the time and resources it would take to complete an algorithm.

"Big O" notation is written as **O(f(n))**, where the "n" is the number of elements that need to be sorted, and "f(n)" is some function of that number. For instance, if "f(n) = n" then the Big O notation is simply **O(n)**. This is called a *linear* function, because there is a direct relationship between the number of elements and the time and resources it takes to finish the algorithm. This means the more elements you

have, the more time it takes! A "Big O" notation of **O(n^2)** would mean that as the number of data elements increases, the time it takes to sort them increases exponentially by a factor of n-squared.

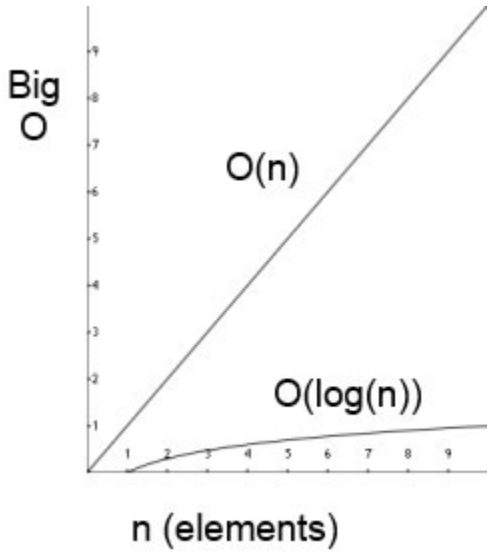
As you can see, for larger values of "n", the **O(n^2)** algorithm would take much more time than the **O(n)** sorting algorithm. If you had 1000 data elements (n = 1000), then **O(n)** would equal 1000, but **O(n^2)** would equal 1,000,000. That's a thousand times slower! This is why it's good to understand the time it takes to perform an algorithm using the "Big O" notation.

"Big O" Is an Approximation

It's hard to calculate exactly what sort of performance an algorithm will have in all cases. Often the nature of the input data, especially in sorting tasks, will impact the speed or efficiency of an algorithm. For this reason it's important to understand that Big O functions are approximations and not an exact science. In fact, Big O formulas are deliberately simplified by dropping all but the highest exponent and discarding any constant coefficients! So if you determined that a particular algorithm would complete in "$3n^2 + 2n$" time, you would drop the "2n" completely, as well as the 3 coefficient, and write the Big O formula as **O(n^2)**.

Sorting and Searching Efficiency

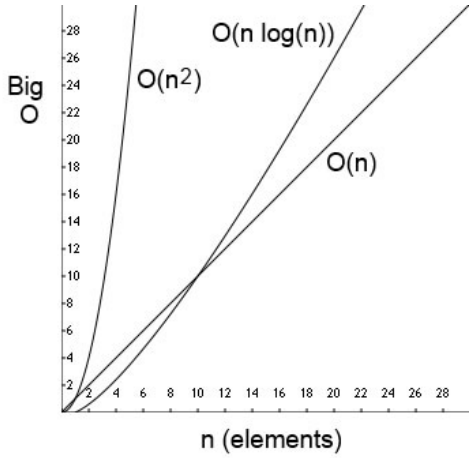
Searching algorithms are tasked with finding one element in a sorted list. The simplest sequential search will check every item in the list, and this is a worst case scenario. As you add more items to the list, the number of checks you need to make also goes up by the same amount. Therefore the Big O notation for a sequential search on average is **O(n)**.



The average "Big O" formula of the binary search algorithm is **O(log (n))**. This means it is considerably more efficient than the sequential search for collections with many elements. We get this efficiency because we can toss out half of the remaining elements at each step in the search process.

Most sorting algorithms will have some "Big O" function that falls in between **O(n)** and **O(n^2)**. The following table describes the average Big O behavior of the bubble, selection, insertion, and merge sorts.

Algorithm	Average Big O
Bubble Sort	O(n^2)
Selection Sort	O(n^2)
Insertion Sort	O(n^2)
Merge Sort	O(n * log(n))



Best, Average, and Worst Case Data

Some algorithms, especially sorting, are very sensitive to the arrangement of the input data. A sorting algorithm might on average have an **O(n^2)** efficiency for a completely random initial data set. But what if your initial data set was already sorted? How long would it take for your sorting algorithm to decide that it's done? The bubble sort and insertion sorts only need to make a single simple pass through the array without any significant data movement, so they would complete in **O(n)** time for pre-sorted data! However some other sorting algorithms might actually get worse if the input data was pre-sorted, or sorted in reverse order.

Based on your knowledge of the bubble, selection, insertion, and merge sort algorithms, can you describe the best and worst case data inputs for each? What kind of input data would make the algorithm run the slowest or fastest? Discuss these possibilities in your class. We've filled out a couple of answers to get you started.

Algorithm	Best Case	Worst Case
Bubble Sort	Data is pre-sorted	
Selection Sort		
Insertion Sort	Data is pre-sorted	
Merge Sort		

Lesson Three: Measuring Sorting Efficiency

The Big O formulas are one way to theoretically compare algorithm performance. But the formulas are theoretical; how do we prove or disprove them? We need to actually measure algorithms running over a variety of data to get a feel for the actual performance as compared to theoretical predictions. There are two reasonably easy ways to measure performance: measuring statement counts and measuring elapsed time.

Counting Statements

One good way to understand an algorithm's performance is to carefully count the number of statements executed within the code. If your algorithm has a loop, then you can count the number of iterations within the loop. Your execution time will typically be proportional to the number of loops you have to make. How many times will the innermost loop statements be run in this example with 10 elements?

```
int numElements = 10;
for (int i=0; i<numElements; i++)
{
    for (int j=0; j<numElements; j++)
    {
       // do something useful here
    }
}
```

That's right, there are 100 iterations of the innermost logic and since $10^2 = 100$, you have **$O(n^2)$** performance!

Sometimes you can't necessarily predict the actual loop iterations because **break** statements or other logic will short-circuit some iterations. So to reliably measure the statement count, we can add special code just to track the loops. The code won't change the behavior of the loop, but will simply measure the performance.

```
int numLoops = 0;
int numElements = 10;
for (int i=0; i<numElements; i++)
{
    for (int j=0; j<numElements; j++)
    {
       numLoops++; // count actual loop iterations
       // do something useful here
    }
}
System.out.println("Number of iterations: " + numLoops");
```

Now we can print out the number of loop iterations that actually took place after the algorithm finishes.

Measuring Elapsed Time

Counting the number of loop iterations is a reasonably good way to measure algorithm performance. But we can also measure the actual elapsed time; that is what we really care about! Let's add some time measurement code to our nested **for**() loop example.

```
long startTime = System.currentTimeMillis();
int numElements = 10;
for (int i=0; i<numElements; i++)
{
    for (int j=0; j<numElements; j++)
    {
       // do something useful here
    }
}
long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("Elapsed time: " + elapsedTime + "(ms)");
```

You can call the **System.currentTimeMillis**() to get the current time in milliseconds. We don't care about the absolute time of day, but we are looking for the difference between the time when the algorithm begins and ends. A millisecond is actually a long time in computer terms, so short algorithms or small data sets may measure a zero elapsed time. In that case it's hard to draw any conclusions about the performance, so you'll need to work with larger data sets.

Generating Large Data Sets

The **SortDemo** sample project uses a very simple data set by default:

```
int[] myNumbers = {5, 2, 7, 4};
```

Clearly this tiny data set won't cause any algorithm to work very hard, so you really want to create a larger data set. It may take thousands of elements or more in order for your algorithm to take some measurable time to complete. You don't want to create all of those elements by hand! So your input data to a sorting test can be created within your computer code.

The method below will create an integer array of the specified number of elements. It will then pre-populate that array with either random or pre-sorted data.

```
public static int[] createData(int numElements)
{
   Random rand = new Random();

   int[] data = new int[numElements];
   for (int i=0; i<numElements; i++)
   {
      // uncomment the line below to create pre-sorted data
      //data[i] = i;

      // uncomment the line below to create random data
      data[i] = rand.nextInt(Integer.MAX_VALUE);
   }
   return data;
}
```

Now your initial data set can be created with a call to this function. Here we create 10,000 elements:

```
int[] myNumbers = createData(10000);
```

You can use this function or any similar technique to create your own data sets. We've shown two common types of data (pre-sorted and random). Is there some other special arrangement of initial data you might want to add to the function? What if the data was pre-sorted in reverse order? Could that have an impact on sorting performance?

Measuring Sorting Algorithms

You are familiar with four different sorting algorithms: bubble sort, selection sort, insertion sort, and merge sort. We also have Big-O formulas for each in the average case, either **O(n^2)** or **O(n * log(n))**. Once you capture some run-time measurements, you could in theory confirm those data points fit into the predicted Big-O function curves. However, curve-fitting to data points is beyond the scope of this course!

What we can do, though, is compare the function to each other and get a sense of the *relative* performance. For example, to compare the bubble and merge sorts, we might choose three different size data sets and record the resulting time from each run.

n (elements)	Bubble Sort	Merge Sort
1,000	0 ms	0 ms
10,000	234 ms	15 ms
100,000	22277 ms	31 ms

You can see that 1,000 isn't enough data to take any meaningful amount of time. However, the difference between the algorithms becomes obvious between 10,000 and 100,000 data points! Not only does the merge sort take less time, but the rate of increase as you add more data points is much slower.

Of course, these results are highly dependent on the particular computer and data set used to perform the measurements. Faster or slower computers may need to process different numbers of elements in order to give useful results.

Measurement Mistakes

There are a couple of things to keep in mind as you begin measuring your loops.

- Always remove all printing code (**System**.**out**.**println**() and similar functions) within your loops. Printing to the console takes some time, so if your loop is printing lines to the screen it will greatly slow down your algorithm, resulting in time measurements that are far too large.
- Make sure your measurement code only covers the actual algorithm execution. Avoid including the data generation (**createData**()) or any printing of sorted data and the end. Any set-up or reporting tasks can take a good amount of time and should not be included as part of your algorithm measurement.