The Arrays Lab

Introduction
In this laboratory, you will learn how to construct functions that operate on Java arrays of type int[] and double[]. Many of the techniques will generalize to arrays of objects but that issue will not be the focus of this laboratory.

The methods you will write fall into the general category of algorithms. These algorithms either create a new array with certain desired properties, modify an existing array to achieve some desired purpose, or extract information from an array. In a practical application of these algorithms, it is possible that several algorithms might execute at once or even that a single algorithm might execute on different arrays simultaneously. To prevent mutual interference, an algorithm must depend only on the data passed in as parameters plus any local data that is created by the algorithm during its execution. Under no circumstances should an algorithm access member or static data of the enclosing class since this could lead to two algorithms modifying the same data in a contradictory fashion. As a consequence of these considerations, the algorithms will be defined as public static methods of the enclosing class and the class will have no data whatsoever.

Another issue is that the functions that you write for int[] will need to be duplicated for double[]. At present, in Java, there is no tool such as templates to do this automatically. Thus, you will have to duplicate the code making small modifications by hand. We will discuss below how to do this reasonably efficiently.

Finally, the laboratory project will be based on a modification of the Problem Set Framework. You should read the tutorial on the framework to become familiar with how to create methods that automatically generate GUI buttons for testing purposes.

The Arrays Lab Structure
The Arrays Lab comes with three Java source files:

- ArrayTools.java // The file with the array algorithms
- ProblemSetClass.java // The file with the test code
- ProblemSetApplication.java // The file that builds the user interface

You will do most of your work in ArrayTools.java where you will create the algorithms. You will do some work in parallel in ProblemSetClass.java where you will create the test code. You will have no need to modify the file ProblemSetApplication.java in any way.

In order to give you a head start on this lab, we have provided some algorithms and some test code. We felt that these samples would be far easier to understand than a long explanation that attempted to say the information in English. Nevertheless, we are now going to briefly explain how the files are built from the beginning.
The Java language itself needed to face the problem of defining classes that are very closely related. For example, in defining the concept of 2-dimensional point, `Point2D`, Java needed to develop code in which the \( x,y \) coordinates are represented as float data and similar code in which the \( x,y \) coordinates are represented as double data. The solution invented by Java is called inner classes. The `Point2D` class contains within it two inner classes `Float` and `Double` that encapsulate the two variations. We refer to `Point2D.Float` or `Point2D.Double` when we need to specify the particular data representation.

We take a similar approach in defining array tools for \( int[] \) arrays and \( double[] \) arrays. Here is the conceptual structure of the `ArrayTools` class:

```java
public class ArrayTools implements JPTConstants, ConsoleAware {
    public static class Int {
        // definitions appropriate for \( int[] \)
    }
    public static class Double {
        // definitions appropriate for \( double[] \)
    }
    // any additional common helper methods
}
```

In fact, we have provided some code within the `Int` inner class but we have left the `Double` inner class empty for you to fill in at the end of the project. There is one useful helper method and we have already placed this method in the file.

### Array Creation and Array Input

Let us examine the first method that you will find within the `Int` inner class. This method will create and return an array initialized with random values.

```java
/**
 * Function to create a random array of the given length 
 * with the given minimum and maximum values.
 */
public static int[] randomArray(int length, int min, int max) {
    if (length < 0) {
        return null;
    }
    int[] data = new int[length];
    for (int i = 0; i < length; i++) {
        data[i] = MathUtilities.randomInt(min, max);
    }
    return data;
}
```
Before we discuss this code, let us see how it might be used in a test program. Consider the code:

```java
int[] a = ArrayTools.Int.randomArray(10, -100, 100);
```

This line introduces `a` as an array variable of type `int[]`. The variable `a` is immediately initialized to refer to the array constructed in and returned by the method `randomArray`. Notice that we must be explicit about where the method `randomArray` is to be found. We must write the full name `ArrayTools.Int.randomArray` to specify that the method is located in the inner class `Int` of the class `ArrayTools`. Notice that the general format for such calls is:

```java
outerclass.innerclass.method(parameters);
```

The call `ArrayTools.Int.randomArray(10, -100, 100)` says that within the method we will have `length = 10, min = -100,` and `max = 100`. Thus the array constructed will have length `10` and use random values between `-100` and `100`.

It is worth noting that the above line of code may also be written:

```java
int a[] = ArrayTools.Int.randomArray(10, -100, 100);
```

In other words, in the declaration of `a`, the `[]` can appear after `a` rather than after `int`. This is a concession to variations found in other programming languages. We prefer to use the first format since it emphasizes that `a` is a variable whose type is `int[]`.

Now let us return to the code for the `randomArray` method. Notice that the code opens with an error check that asks if the `length` is less than `0`. If so, creation of the array is impossible and the method returns `null`. Checking for `null` data or `null` returns is critical if methods are to work properly. Next, since the `randomArray` method is going to construct a new array, there must be a call to the `new` operator of Java. This is the purpose of the line:

```java
int[] data = new int[length];
```

The local variable `data` will now be initialized to point to an `int[]` with `length` elements. Since `int` is a primitive numeric type, Java will initialize `data[i]` to `0` for all `i`. The next pair of lines then resets the values in the `data` array to the desired random values in the range `min` to `max` using a simple helper function from the JPT class `MathUtilities`. Finally, the reference to the new array that is being kept in the variable `data` must be returned to the caller and that is the purpose of the last line of the method.

The inner class `Int` is supplied with a method `makeArray` for array creation and input that permits a user to select from a random array, an array with linear data, or an array whose data is initialized explicitly by typing each data item. The form of the call is:

```java
int[] a = ArrayTools.Int.makeArray();
```

We will not discuss this or the other methods for array creation and input but it will be fruitful for you to read them in detail to improve your understanding of how arrays work.
Algorithms to Extract Information from an Array

As an example, we provide the code for one simple extraction algorithm, namely, a method to find the sum of the values in an array.

```java
/** Returns the sum of the values in the array. */
public static int sum(int[] data) {
    int s = 0;
    if (data == null)
        return s;
    int length = data.length;
    for (int i = 0; i < length; i++)
        s += data[i];
    return s;
}
```

The sum variable \(s\) is first initialized to 0 which is the proper value since no sum has yet taken place. Then we do the error check that asks if the parameter \(data\) is null. If so, we stop and return \(s\) immediately. Next we extract the length of the array \(data\). Notice that this line would cause a NullPointerException if \(data\) were null so the error check is quite important. Next, using a loop, we sum the values \(data[i]\) into the accumulator \(s\). Finally, we return \(s\).

Now let us examine the test code for this function in the ProblemSetClass.

```java
public synchronized void IntArraySum() {
    // make the data
    int[] data = ArrayTools.Int.makeArray();
    // compute and print sum
    console.out.println("Sum: " + ArrayTools.Int.sum(data) + \\
                        "\n");
}
```

Notice that we use ArrayTools.Int.makeArray() to permit the user to decide how to construct the test data: as a random array, as a linear array, or as user input entered item-by-item. Then we combine the computation of the sum with the code to print it to the console. Your task will be to write three additional extraction algorithms.

```java
/** Returns the average of the values in the array. */
public static int average(int[] data)
/** Returns the minimum of the values in the array. */
public static int minimum(int[] data)
/** Returns the maximum of the values in the array. */
public static int maximum(int[] data)
```
The average is the sum of the array values divided by the array length. However, you will get into trouble if you forget to check for errors. Therefore, the method `average` should first check if the parameter `data` is null or if `data.length` is 0. In these cases, the method should return 0 immediately. If all is OK, then `average` should use the method `sum` to compute the sum and then return the ratio of the sum by `data.length`.

Let us now discuss the method `minimum`. The basic idea is that you keep a copy of the minimum value found so far. Then, you loop through the data and each time you find a data value that is smaller than the minimum value found so far you replace the minimum value with this smaller data value.

Let us name the minimum value found so far as `minimumFound`. A critical question is how to initialize this variable. The initial value must be such that in the initial phase of the loop this value is automatically replaced by `data[0]`. The only initial value that will work is the largest integer value that may be represented in the system. Java provides this value in the class `Integer` so the first line of the `minimum` method should be:

```java
int minimumFound = Integer.MAX_VALUE;    // This value is +2147483647
```

The next step should be the required error check. If `data` is null or `data.length` is 0 then the variable `minimumFound` should be returned immediately. If all is OK, then the search for the minimum value can proceed as outlined above and `minimumFound` should be returned at the end.

The method `maximum` is programmed similarly except that you need the variable:

```java
int maximumFound = Integer.MIN_VALUE;    // This value is -2147483648
```

Notice that `Integer.MIN_VALUE` is not the same as the negative of `Integer.MAX_VALUE`. This is a fundamental property of how integer numbers are represented in the system.

Although you will work with `double[]` arrays at the end of the laboratory, let us mention here how you must deal with initialization in that situation. First of all, you must be aware that the following conventions are used for `MAX_VALUE` and `MIN_VALUE` in Java class `Double`:

- `Double.MAX_VALUE` is the maximum positive double value (1.7976931348623157E308)
- `Double.MIN_VALUE` is the minimum positive double value (4.9E-324)

This means that `Double.MIN_VALUE` is irrelevant to the problems in this laboratory and that you must use `-Double.MAX_VALUE` to obtain the most negative `double` value. Secondly, there is a name conflict between the Java class `Double` and our use of `Double` as an inner class of `ArrayTools`. This means that when refer to the constants you must give what is called a fully qualified name. Thus, when defining the methods `minimum` and `maximum` for `double[]` arrays, you will need the following definitions:

```java
double minimumFound = java.lang.Double.MAX_VALUE;    // use in minimum

double maximumFound = -java.lang.Double.MAX_VALUE;    // use in maximum
```
The test method in ProblemSetClass for the average, minimum, and maximum methods is not in the source code provided with the laboratory but let us give it here so you can see that it is a simple amplification of the test code for sum.

```java
public synchronized void IntArrayStatistics() {
    // make the data
    int[] data = ArrayTools.Int.makeArray();
    
    // compute and print statistics
    console.out.println("Sum: " + ArrayTools.Int.sum(data));
    console.out.println("Average: " + ArrayTools.Int.average(data));
    console.out.println("Minimum: " + ArrayTools.Int.minimum(data));
    console.out.println("Maximum: " + ArrayTools.Int.maximum(data));
}
```

A Digression on Duplication or Cloning of an Array

To **duplicate** or **clone** an array is the make a new array structure that is initialized with exactly the same data as the original array. Java provides the foundation for this activity by defining the following method in class Object:

```java
protected Object clone() {
}
```

Normally, to actually use the `clone()` method, a class that extends `Object` must declare that it implements the `Cloneable` interface and must define and make public the `clone()` method:

```java
public Object clone() { ... detailed implementation ... }
```

Since this definition process is impossible for array objects because you may not add methods to array objects, Java declares that all array objects are automatically `Cloneable` and provides the necessary `clone()` method itself.

So, for example, suppose that you have an `int[]` array object `data` that is non-null and you wish to clone this array into another `int[]` array object `copy`. Here is what to do:

```java
int[] copy = (int[]) data.clone();
```

Notice that you must add a **type cast**, namely, `(int[])` in front of `data.clone()`. The reason is that the generic `clone()` method returns a reference of type `Object` and Java must be told with the type cast exactly what type the returned object really is.

**Algorithms to Move Data Within an Array**

The next task of the laboratory exercise is to write methods for moving data within an array. The three generic methods are to rotate the data one cell to the left, rotate the data once cell to the right, and to reverse the data by swapping items at small indices with the corresponding items at large indices. The method headers are:
/** Function to rotate the array values left. */
public static void rotateLeft(int[] data)

/** Function to rotate the array values right. */
public static void rotateRight(int[] data)

/** Function to reverse the array values. */
public static void reverse(int[] data)

Here are some pictures that illustrate the behavior of these functions on an array of size 10.
We assume, for simplicity, that the initial data is 0, 10, 20, 30, 40, 50, 60, 70, 80, 90

Original Data:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<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>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>

After RotateLeft:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<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>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

After RotateRight:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<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>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
</tr>
</tbody>
</table>

After Reverse:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<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>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>90</td>
<td>80</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

We will leave the details of these algorithms to you but we caution you to keep in mind that it is impossible to move data without temporarily saving some information outside of the array. For example, in RotateLeft, you must save the value in cell 0, move the remaining data to the left, and then copy the saved value into the highest cell.

In order to test these algorithms in ProblemSetClass, you will need to ask the user for an array (using ArrayTools.Int.makeArray() as usual) and then you will need to clone this array three times to test the three algorithms. The use of clone guarantees that each algorithm starts with the same initial data. Here is a sample of our test output on an array of size 5.

Make random data? [Y] n
Make linear data? [Y]
Linear Array
Data array length: 5
Linear start: 1
Linear delta: 1

Show the data? [Y]
Data array length: 5
data[0] = 1
data[1] = 2
data[2] = 3
data[3] = 4
data[4] = 5
### Algorithms to Sort an Array

To sort an array means to rearrange the array values so that they proceed in some desired order. The most common order is *increasing numerical order* and that is what we will focus on in this laboratory. In this segment of the laboratory, you will examine one algorithm, `insertionSort`, and will write one algorithm, `selectionSort`.

The code for `insertionSort` that is provided in the `ArrayTools` class is:

```java
/** Insertion Sort. */
public static void insertionSort(int[] data) {
    if ((data == null) || (data.length <= 1))
        return;

    int last = data.length - 1;
    for (int i = 1; i <= last; i++) {
        // move element at i into proper spot
        int temp = data[i];
        int spot = i;
        while (spot > 0) {
            if (data[spot - 1] > temp) {
                data[spot] = data[spot - 1];
                spot--;
            } else
                break;
        }
        data[spot] = temp;
    }
}
```
The philosophy of *insertionSort* is that you sort the array from the left side. At each stage, you copy the *next unsorted element* to a spare location, then shift larger elements that occur to the left one cell to the right, and then copy the value from the spare location into its proper location in the array. This process is of course repeated in an outer loop until the entire array is sorted.

In the implementation above, the first step as usual is to do error checks. There is nothing to do if the `data` parameter is `null` or if the array has 0 or 1 element. Next, for convenience, the last valid array index is stored in `last` (this index is not `array.length`!). Then the outer loop begins that will control each major stage. You may think of the index `i` as the index of the *next unsorted element*. The first step in the outer loop is to copy the next unsorted element to a spare location named `temp`. The variable `spot` will denote the index we will eventually use to reinsert the value `temp` into the array. Of course, `spot` is initialized to `i` since that is where the work of the inner `while` loop begins. Within the `while` loop, we move elements that are larger than `temp` to the right and then move the “open spot” to the left. If we encounter an element that is less than or equal to `temp`, then we leave the loop via a `break`. Finally, when we are out of the `while` loop, we insert `temp` back into the array at `spot` as planned.

It is critical that you understand a simple algorithm such as *insertionSort* perfectly in order that you have the foundation for learning more complex algorithms later. Read the code and compare it to the prose description until you understand every single line. Try the algorithm out by hand with examples created with pencil and paper. Do whatever it takes to know this algorithm thoroughly.

The algorithm that you will write is *selectionSort* which has the header:

```java
/** Selection Sort. */
public static void selectionSort(int[] data)
```

The philosophy of *selectionSort* is in some ways the opposite to *insertionSort*. The algorithm begins by scanning the entire array for the location of the element with minimum value. Once this location is found, then the minimum element is swapped with the element in cell 0. Once this is done, cell 0 does not need to be looked at again. The process repeats by searching for the minimum value in the cells from 1 to `last` (which is `data.length - 1`). Then it repeats again with the cells from 2 to `last` and so on until the array is sorted.

Once you have coded *selectionSort*, go to the *ProblemSetClass* and remove some comments in the method *IntArraySorts* and you will be ready to test your algorithm.

**Algorithms to Search an Array**

It is often necessary to search an array to determine if a given value (the *key*) occurs in the array and if it does to return its *index location*. The difficulty of this problem depends on whether the array is sorted or unsorted. If it is sorted, then a fast strategy such as binary search may be used. If not, there is no choice but to use linear search which examines each cell in turn to see if its value is equal to the key.
Whatever the search algorithm, the following convention is used for the value returned by the search method. If the key did occur in the array, its index location is returned. If the key did not occur in the array, then the value \(-1\) is returned to signal that the key is missing. Thus, it is always critical to test the return value to see if it is positive or is \(-1\).

You will program `linearSearch` whose header is as follows:

```java
/**        /**        /**        /**
* Return the index of the key value in the data array.         * Return the index of the key value in the data array.         * Return the index of the key value in the data array.         * Return the index of the key value in the data array.
* If the key value is not in the array then return         * If the key value is not in the array then return         * If the key value is not in the array then return         * If the key value is not in the array then return -1.         * -1.
*/      */      */      */
public static int linearSearch(int[] data, int key)      public static int linearSearch(int[] data, int key)      public static int linearSearch(int[] data, int key)      public static int linearSearch(int[] data, int key)
```

As usual, you must do an error check and return \(-1\) if the `data` is `null` or if `data.length` is `0`. If all is OK, then you must loop through the array and return the index location if the value at the index is equal to the key. If you get to the end of the loop without finding the key then you must again return \(-1\).

The more difficult search algorithm, `binarySearch`, and its test, `IntArrayBinarySearch`, are provided to you in the code. We will discuss `binarySearch` in lecture since it introduces the use of recursion. If you look at its test code, `IntArrayBinarySearch`, you will see that you can adapt this code to test `linearSearch`. Be careful, however. The binary search algorithm requires that its array be sorted so this is done in the test code inside `IntArrayBinarySearch`. In contrast, the linear search algorithm can search an arbitrary array so it would be an error to sort the array in its test code.

**Simple Bar Chart Graphics**

The final task that you will do for `int[]` arrays is to create a simple method to draw a bar chart from the data in the array. We will intentionally make things easy rather than general. If you wish to enhance the bar chart method for extra credit then you may do so.

The header for the bar chart method is:

```java
/**        /**        /**        /**
* Draws a simple bar chart of the given data in the given window         * Draws a simple bar chart of the given data in the given window         * Draws a simple bar chart of the given data in the given window         * Draws a simple bar chart of the given data in the given window.
* using bars that are 8 pixels wide with 10 pixel spacing         * using bars that are 8 pixels wide with 10 pixel spacing         * using bars that are 8 pixels wide with 10 pixel spacing         * using bars that are 8 pixels wide with 10 pixel spacing.
* but with no vertical scaling.         * but with no vertical scaling.         * but with no vertical scaling.         * but with no vertical scaling.
*/         */         */         */
public static void SimpleBarChart(int[] data, BufferedPanel window)         public static void SimpleBarChart(int[] data, BufferedPanel window)         public static void SimpleBarChart(int[] data, BufferedPanel window)         public static void SimpleBarChart(int[] data, BufferedPanel window)
```

The test code in the `ProblemSetClass` for this method will look like:

```java
public synchronized void IntArrayBarChart() {
    int[] data = ArrayTools.Int.makeArray();
    window.clearPanel();
    ArrayTools.Int.SimpleBarChart(data, window);
}
```
The window object in the test code is a 400 x 400 BufferedPanel that is supplied automatically by the ProblemSetApplication class to the ProblemSetClass class. Below is a snapshot that shows the bar chart produced by a random array with 40 elements.

We have used the convention that the width of each bar is 8 pixels, the gap between bars is 2 pixels, and therefore amount you must jump from bar to bar is 8+2 = 10 pixels. These simple rules determine the horizontal spacing: bar 0 is at x = 0, bar 1 is at x = 10, bar 2 is at x = 20, etc.

To understand how to position a bar vertically, you must understand that in computer graphics y increases downwards, that is, y = 0 corresponds to the top of the window (rather than the bottom which is the tradition in mathematics). Since the window is 400 x 400, the vertical pixels are numbered downwards 0, 1, 2, ..., 397, 398, 399. Just as in array indexing, the last valid pixel is one less than the size. Furthermore, to draw a bar you must give its upper left corner (x, y) and its width and height:

\[(x, y)\]

Let’s see what this tells us about plotting the i-th bar in an array data.
The $i$-th bar is $10 \times i$ pixels from the left side, so $x = 10 \times i$.

The bar width is a constant 8, so $\text{width} = 8$.

The bar height is the data value since we are not scaling, so $\text{height} = \text{data}[i]$.

The $y$ coordinate must count upward from the bottom, so $y = 400 - \text{height}$.

You may ask why the last formula uses 400 rather than 399. To understand this, think of the limiting case when the height is 1 pixel. Then we should draw on vertical pixel 399. If we compute $y$ in this case as $400 - 1$ we get 399 as the top of the bar as desired.

To understand how to do the rest of the graphics, examine the sample code for PaintCircles in the Problem Set Framework. Instead of drawing circles (using Ellipse2D.Double), you need to draw bars so you will use Rectangle2D.Double. Instead of using random coordinates for the shape, you will use $x$, $y$, width, height as discussed above. Instead of a random color, you will use the color constant Color.black. Finally, you will use the size of the array to determine the number of bars.

This simple bar chart method can draw at most 40 bars. For extra credit, you can make the width of the bars and the gap between computed quantities so that the window can hold up to 400 bars. We leave this to you to think about.

**Finishing the Lab: From int[] Arrays to double[] Arrays**

When you are 100% certain that the algorithms and test code for int[] arrays is working, then you will create the corresponding code for double[] arrays. Save a backup copy of your file before you do the next steps!

In ArrayTools, you should select and copy all of the code in the Int inner class and then paste a duplicate of this code back into the file. Do a similar action for the Int test code in the ProblemSetClass.

Now use the “replace” command in the editor to search for and replace the following in the new text blocks you have just pasted: replace Int by Double where appropriate and int by double where appropriate. You should use case sensitive search since you do not want to change case during the replace operation. You need to be very careful since some int variables correspond to lengths or loop indices and should not be converted to double.

Finally, go back and read the discussion on the minimum and maximum functions above since you will need to manually fix the definitions of maximumFound and minimumFound.

When this is done, thoroughly test the double[] functions.

**A Word to the Wise**

There is a method in software engineering called the Big Bang approach in which everything is coded before anything is tested. The Big Bang method is considered horribly stupid by all of the leading developers on the planet. We have carefully set up this laboratory so that you can build and test in small incremental stages. If you do this, you can be successful. If you choose, despite all advice, to try Big Bang, then you may be caught in the explosion.