## 1 Submission Instructions

Create a folder named <*asuriteid*> where *asuriteid* is your ASURITE user id (for example, since my ASURITE user id is *kburger2* my folder would be named *kburger2*) and copy all of your *.java* source code files to this folder. Do not copy the *.class* files or any other files. Next, compress the <*asuriteid*> folder creating a **zip archive** file named <*asuriteid*>*.zip* (mine would be named *kburger2.zip*). Upload <*asuriteid*>*.zip* to the Project 1 link by the project deadline. The deadline is in course schedule. Consult the online syllabus for the late and academic integrity policies.

## 2 Learning Objectives

1. Use the Integer wrapper class.
2. Declare and use ArrayList clas
3. Write code to read from, and write to, text files.
4. Write an exception handler for an I/O exception.
5. Write Java classes and instantiate objects of those classes.

## 3 Background

Let *list* be a nonempty sequence of nonnegative random integers, each in the range [0, 32767] and let $n$ be the length of *list*, e.g.,

*list* = { 2, 8, 3, 2, 9, 8, 6, 3, 4, 6, 1, 9 }

where $n = 12$. List elements are numbered starting at 0. We define a **run up** to be a ($k$+1)-length subsequence $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$, that is **monotonically increasing** (i.e., $list_{i+j} \geq list_{i+j-1}$ for each $j = 1, 2, 3, ..., k$). Similarly, a **run down** is a ($k$+1)-length subsequence $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$, that is **monotonically decreasing** (i.e., $list_{i+j-1} \leq list_{i+j}$ for each $j = 1, 2, 3, ..., k$). For the above example *list* we have these runs up and runs down:

**Runs Up**

$list_0$ through $list_1$ = { 2, 8 }; $k = 1$
$list_2$ = { 3 }; $k = 0$
$list_3$ through $list_4$ = { 2, 9 }; $k = 1$
$list_5$ = { 8 }; $k = 0$
$list_6$ = { 6 }; $k = 0$
$list_7$ through $list_9$ = { 3, 4, 6 }; $k = 2$
$list_{10}$ through $list_{11}$ = { 1, 9 }; $k = 1$

**Runs Down**

$list_0$ = { 2 }; $k = 0$
$list_1$ through $list_3$ = { 8, 3, 2 }; $k = 2$
$list_4$ through $list_7$ = { 9, 8, 6, 3 }; $k = 3$
$list_8$ = { 4 }; $k = 0$
$list_9$ through $list_{10}$ = { 6, 1 }; $k = 1$
$list_{11}$ = { 9 }; $k = 0$

We are interested in the value of $k$ for each run up and run down and in particular we are interested in the total umber of runs for each nonzero $k$, which we shall denote by $runs_k$, $0 < k < n$ - 1. For the example *list* we have:

| ***k*** | ***runs*$_k$** | **runs** |
|---|---|---|
| 1 | 4 | { 2, 8 }, { 2, 9 }, { 1, 9 }, and { 6, 1 } |
| 2 | 2 | { 3, 4, 6, } and { 8, 3, 2 } |
| 3 | 1 | { 9, 8, 6, 3 } |
| 4-11 | 0 | |

Let $runs_{total}$ be the the sum from $k = 1$ to $n$ - 1 of $runs_k$. For the example *list*, $runs_{total} = 4 + 2 + 1 = 7$.

## 4 Software Requirements

Your program shall:

1. Open a file named *p01-in.txt* containing $n$ integers, $1 \leq n \leq 1000$, with each integer in [0, 32767]. There will be one or more integers per line. A sample input file:

   **Sample *p01-in.txt***
   ```
   2 8 3
   2 9
   8
   6
   3 4 6 1 9
   ```
2. The program shall compute $runs_k$ for $k = 1, 2, 3, ..., n - 1$.
3. The program shall compute $runs_{total}$.
4. The program shall produce an output file named *p01-runs.txt* containing $runs_{total}$ and $runs_k$ for $k = 1, 2, 3, ..., n - 1$. The file shall be formatted as shown in the example file below.

   **Sample *p01-runs.txt***
   ```
   runs_total, 7
   runs_1, 4
   runs_2, 2
   runs_3, 1
   runs_4, 0
   runs_5, 0
   runs_6, 0
   ```
5. If the input file *p01-in.txt* cannot be opened for reading (because it does not exist) then display an error message on the output window and immediately terminate the program, e.g.,

   *run program...*
   ```
   Sorry, could not open 'p01-in.txt' for reading. Stopping.
   ```

## 5 Software Design

Your program shall:

1. Contain a class named *Main*. This class shall contain the *main*() method. The *main*() method shall instantiate an object of the *Main* class and call *run*() on that object.
   ```
   // Main.java
   public class Main {
       public static void main(String[] pArgs) {
            Main mainObject = new Main();
            mainObject.run()
       }
       private void run() {
            // You will start writing code here to implement the software requirements.
       }
   }
   ```
2. One of the primary objectives of this programming project is to learn to use the *java.util.ArrayList* class. Therefore, you **are not permitted** to use 1D arrays. Besides, you will quickly discover that the ArrayList class is more convenient to use than 1D arrays.

3. ArrayList is a generic class meaning: (1) that it can store objects of any class; and (2) when an ArrayList object is declared and instantiated we must specify the class of the objects that will be stored in the ArrayList. For this project, you need to define an ArrayList that stores integers, but you cannot specify that your ArrayList stores **int**s because **int** is a primitive data type and not a class. Therefore, you will need to use the *java.lang.Integer* wrapper class:

```
ArrayList<Integer> list = new ArrayList<>():
int x = 1;
list.add(x); // Legal because of Java autoboxing.
```

4. You must write an **exception handler** that will catch the *FileNotFoundException* that gets thrown when the input file does not exist (make sure to test this). The exception handler will print the friendly error message and immediately terminate the Java program. To immediately terminate a Java program we call a static method named *exit*() which is in the *java.lang.System* class. The *exit*() method expects an **int** argument. For this project, it does not matter what **int** argument we send to *exit*(). Therefore, terminate the program this way:

```
try {
     // Try to open input file for reading
} catch (FileNotFoundException pExcept) {
     // Print friendly error message
     System.exit(-1);
}
```

5. Your programming skills should be sufficiently developed that you are beyond writing the entire code for a program in one method. Divide the program into multiple methods. Remember, a method should have one purpose, i.e., it should do one thing. If you find a method is becoming complicated because you are trying to make that method do more than one thing, then divided the method into 2, 3, 4, or more distinct methods, each of which does one thing.
6. Avoid making every variable or object an instance variable. For this project **you shall not declare any instance variables** in the class. That is, all variables should be declared as local variables in methods and passed as arguments to other methods when appropriate.
7. Format your code neatly. Use proper indentation and spacing. Study the examples in the book and the examples the instructor presents in the lectures and posts on the course website.
8. Put a comment header block at the top of each method formatted thusly:

```
/**
 * A brief description of what the method does.
 */
```

9. Put a comment header block at the top of each source code file—not just for this project, but for every project we write—formatted thusly:

```
//*************************************************************************************************
// CLASS: classname (classname.java)
//
// DESCRIPTION
// A description of the contents of this file.
//
// COURSE AND PROJECT INFO
// CSE205 Object Oriented Programming and Data Structures, semester and year
// Project Number: project-number
//
// AUTHOR
// your-name (your-email-addr)
//*************************************************************************************************
```