1 Submission Instructions

Create a folder named *asuriteid-p01* where *asuriteid* is your ASURITE user id (for example, if your ASURITE user id is *jsmith6* then your folder would be named *jsmith6-p02*) 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-p01* folder creating a **zip archive** file named *asuriteid-p01.zip* (e.g., *jsmith6-p01.zip*). Upload *asuriteid-p01.zip* on the Canvas *Module 2: P2* page by the project deadline. Please see the *Course Summary* section on the *Syllabus* page in Canvas for the deadline. Consult the Syllabus for the late and academic integrity policies.

2 Learning Objectives

1. Use the *Integer* wrapper class.
2. Declare and use *ArrayList<E>* class objects.
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[1]

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 starting at index i: $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$, 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 starting at index i: $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$, 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 }; $k = 1$, 2-length subseq
$list_2$ = { 3 }; $k = 0$, 1-length subseq
$list_3$ through $list_4$ = { 2, 9 }; $k = 1$, 2-length subseq
$list_5$ = { 8 }; $k = 0$, 1-length subseq
$list_6$ = { 6 }; $k = 0$, 1-length subseq
$list_7$ through $list_9$ = { 3, 4, 6 }; $k = 2$, 3-len subseq
$list_{10}$ through $list_{11}$ = { 1, 9 }; $k = 1$, 2-len subseq

Runs Down

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

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

k	$runs_k$	runs	
1	4	{ 2, 8 }, { 2, 9 }, { 1, 9 }, and { 6, 1 }	*Note: (1+1)-length subsequences*
2	2	{ 3, 4, 6, } and { 8, 3, 2 }	*Note: (2+1)-length subsequences*
3	1	{ 9, 8, 6, 3 }	*Note: (3+1)-length subsequence*
4-11	0		

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

1 The runs up and runs down test is used in statistics. When I wrote my master's thesis on random number generation algorithms, I had to write these tests to determine the indepence between successive random numbers generated by my algorithms.

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. Please make sure your output file exactly matches the format shown below, e.g., all text is in lowercase, an underscore separates "runs" from the text that follows it, there is a space following each colon, and each line of text ends with a newline/endline character. During grading, we will compare your output file to our correct output file using an automated comparison program and if there is *any* mismatch then your program may be considered incorrect and you may lose substantial points.

   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
   runs_7: 0
   runs_8: 0
   runs_9: 0
   runs_10: 0
   runs_11: 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., something like this,

   run program...
   ```
   Oops, could not open 'p01-in.txt' for reading. The program is ending.
   ```

6. If the output file "p01-runs.txt" cannot be opened for writing (e.g., because the write access to the file is disabled) then display an error message on the output window and immediately terminate the program, e.g.,

   run program...
   ```
   Oops, could not open 'p01-runs.txt' for writing. The program is ending.
   ```

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, see template code below. Other than *main*() there shall not be any static or class methods.

```
// Main.java
public class Main {
    public static void main(String[] pArgs) {
         Main mainObject = new Main();   // Or you can just write: new Main().run();
         mainObject.run()                // in place of these two lines.
    }
    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<E>* class. Therefore, you **are not permitted** to use primitive 1D arrays. Besides, you will quickly discover that the ArrayList class is more convenient to use than 1D arrays.

3. *ArrayList<E>* is a generic class meaning: (1) that it can store objects of any reference type, e.g., *E* could be the classes *Integer* or *String*; 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 as shown in Software Requirement 5 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 by sending -100 to *exit*().

```
try {
     // Try to open input file for reading
} catch (FileNotFoundException pExcept) {
     // Print friendly error message
     System.exit(-100);
}
```

5. Similar to Item 4, you must write an exception handler that will catch the *FileNotFoundException* that gets thrown when the output file cannot be opened for writing. The exception handler will print the message as shown in Software Requirement 6 and then terminate the program by sending -200 to *exit*().

6. 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.

7. 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.

8. Neatly format your code. 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.

9. Put a comment header block at the top of each method formatted thusly:

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

10. 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 (or you may use /** ... */ Javadoc comment style if you wish).

```
//************************************************************************************************************
// 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-asuriteid, your-email-addr
//************************************************************************************************************
```

5.1 Software Design: Pseudocode

To help you complete the program, you may implement this pseudocode if you wish.

```
-- Note: In the Java implementation, main() would call run(), so in essence, run() becomes the starting point
-- of execution.
Method run() Returns Nothing
     -- Make sure to catch and handle the FileNotFoundExceptions that may get thrown in readInputFile() and
     -- writeOutputFile() when the input and output files cannot be opened for reading and writing.
     Declare ArrayList of Integers list ← readInputFile("p01-in.txt")  -- Reads the integers from the input file
     Declare and create an ArrayList of Integers named listRunsUpCount
     Declare and create an ArrayList of Integers named listRunsDnCount
     listRunsUpCount ← findRuns(list, RUNS_UP)  -- RUNS_UP and RUNS_DN are named constants, it does not matter what
     listRunsDnCount ← findRuns(list, RUNS_DN)  -- value you assign to them as long as the values are different
     Declare ArrayList of Integers listRunsCount ← mergeLists(listRunsUpCount, listRunsDnCount)
     writeOutputFile("p01-runs.txt", listRunsCount)
End Method run

-- pList is the ArrayList of Integers that were read from "p01-in.txt". pDir is an int and is either RUNS_UP or RUNS_DN
-- which specifies in this method whether we are counting the number of runs up or runs down.
Method findRuns(pList : ArrayList of Integers, pDir : int) Returns ArrayList of Integers
     listRunsCount ← arrayListCreate(pList.size(), 0)  -- size is the same as pList and each element is init'd to 0
     Declare int variables initialized to 0: i ← 0, k ← 0  -- the left arrow represents the assignment operator
     While i < pList.size() - 1 Do
          If pDir is RUNS_UP and pList element at i is ≤ pList element at i + 1 Then
               Increment k
          ElseIf pDir is RUNS_DN and pList element at i is ≥ pList element at i + 1 Then
               Increment k
          Else
               If k does not equal 0 Then
                    Increment the element at index k of listRunsCount
                    k ← 0
               End if
          End If
          Increment i
     End While
     If k does not equal 0 Then
          Increment the element at index k of listRunsCount
     End If
```

```
     Return listRunsCount
End Method findRuns

Method mergeLists(pListRunsUpCount is ArrayList of Integers, pListRunsDnCount is ArrayList of Integers)
Returns ArrayList of Integers
     listRunsCount ← arrayListCreate(pListRunsUpCount.size(), 0)
     For i ← 0 to pListRunsUpCount.size() - 1 Do
          Set element i of listRunsCount to the sum of the elements at i in pListRunsUpCount and pListRunsDnCount
     End For
     Return listRunsCount
End Method mergeLists

Method arrayListCreate(int pSize; int pInitValue) Returns ArrayList of Integers
     Declare and create an ArrayList of Integers named list
     Write a for loop that iterates pSize times and each time call add(pInitValue) to list
     Return list
End Method arrayListCreate

Method writeOutputFile(String pFilename; pListRuns is ArrayList of Integers) Returns Nothing
     -- Make sure to throw the FileNotFoundException that is raised when the output file cannot be opened for writing.
     out ← open pFilename for writing
     out.println("runs_total: ", the sum of pListRuns)
     For k ← 1 to pListRuns.size() - 1 Do
          out.println("runs_k: ", the element at index k of pListRuns)
     End For
     Close out
End Method output

Method readInputFile(String pFilename; pListRuns is ArrayList of Integers) Returns ArrayList of Integers
     -- Make sure to throw the FileNotFoundException that is raised when the input file cannot be opened for reading.
     in ← open pFilename for reading
     Declare and create an ArrayList of Integers named list
     While there is more data to be read from in Do
          Read the next integer and add it to list
     End While
     Close in
     Return list
End Method readInputFile
```