1 Submission Instructions

- In the submission instructions, we mention your ASURITE ID several times. Your ASURITE ID is the user name you use to log in to ASU computer systems such as MyASU and Canvas, e.g., the instructor's ASURITE ID is kburger2. It is not the same as your ASU ID number printed on your SunCard (that identifier is a 10-digit integer).
- Note that for each homework assignment, only some of the exercises will be graded. A list of the exercises that will be graded can be found in the Module 5: HW3 Due submission page in Canvas.
- There are two primary reasons we grade only some of the exercises: first, this course has a large enrollment so there are many assignments to be graded. Second, in the accelerated time frame for online courses, we want to return graded homework exercises as quickly as we can.
- Please understand that it is time-consuming to manually grade assignments, particularly programming exercises, so it may take as long as a week to return scores. We will always try to return them sooner.
- Not grading all of the exercises does not mean that the ungraded exercises are unimportant or that you will not be tested on the concepts in the exercise. They are equally important as the graded exercises and the concepts you learn may be on the exams. Consequently, we **strongly recommend** that you complete *all* of the exercises.
- Some of your solutions must be submitted in a PDF document. To start the assignment, create a word processing document named *asuriteid*-h03.*ext*, where *asurite* is your ASURITE ID and *ext* is docx if you are using Word or odt if you are using Libre Writer or Open Office Writer. For example, the instructor's file would be named kburger2-h03.odt because he uses Libre Writer.
- Please type your name, ASURITE ID, and the homework number (HW3) near the top of the document.
- For the **short-answer and description** exercises (e.g., see Exs. 4.3, 5.1, 5.5, 6.4), please neatly type your solution in the document. Clearly number each exercise so there is no confusion for the grader.
- If an exercise asks you to write Java code but **does not** ask you to submit your solution **in a separate Java source code file**, then copy-and-paste your Java code from your IDE or text editor into your word processing document. Make sure to neatly format your code, i.e., properly indent your code and use consistent indentation. It helps the grader understand it and you are less likely to lose points for unreadable code.
- For an exercise which asks you to write a Java method, please see §1.1 below for instructions on what to submit. We may employ an automated grading script to grade these exercises, so it is helps us when you follow the exercise instructions—especially in regard to naming of things such as file names, class names, etc. The reason is because we hardcode the names of these things into the script and when the script is looking for a file which *should* be named *Test.java* but is instead named *MyTest.java* (or something else) then the script will fail. Then we have to grade the exercise manually. This slows down the grading, and for this reason, there may be a point deduction for that exercise.
- When you are done with the document, please convert it **Adobe PDF format** and name the file *asuriteid*-h03.pdf, e.g., the instructors file would be named kburger2-h03.pdf.
- Next, create an empty folder named *asuriteid*-h03 and copy *asuriteid*-h03.pdf to that folder.
- For those exercises which asked you to submit the Java source code file, please copy the requested *.java* source code files to the *asuriteid*-h03 folder. You do not need to submit any other files, e.g., do not submit *.class* files or data files.
- Then compress the *asuriteid*-h03 folder creating a **zip archive** file named *asuriteid*-h03.zip. Upload *asuriteid*-h03.zip to Canvas using the submission link on the Module 5: HW3 Due submission page before the assignment deadline.
- Please see the Course Summary section on the Syllabus page in Canvas for the deadline. The deadline can also be found on the CSE205 course calendar, look for the events named Module 5: HW3 Due and Module 5: P3 Due.
- Consult the Syllabus for the late and academic integrity policies.

1.1 Submitting Java Source Code Files Containing Methods

- Some exercises ask you to submit a Java file containing just one method or a few methods which solve the problem, i.e., not a complete runnable program containing a *main*() method.
- We want you to write the requested method within a class declaration (i.e., in a *.java* file) which is named as requested in the exercise.

- Remember that Java is case-sensitive so your filenames, method names, variable names, etc. must be named **exactly** as requested in the homework document. See the tenth bullet item in §1 *Submission Instructions* for why this may cause you to lose points on an exercise.
- For testing, we will use a testing driver in a class we write, which will call your method—the method under test or MUT. Since Java cannot compile a method which is not in a class, this is the reason we want you to declare a class which contains your method. It is so we can build this testing program and run it.
- For example, the class template we want you to submit for Ex. 3.1 is shown below. The templates for other related exercises will be similar but the class name will contain the exercise number, e.g., H03_**34** for Ex. 3.4, and so on.
- See Ex. 3.1, where you will write the method *int sum1toN*(*int n*) to solve the problem:

```
// CLASS:  H03_31 (Source Code File: H03_31.java)
// AUTHOR: your name, your ASURITE username, your email address

Import any required classes using import statements...

// Remember that the class and source code filename must be the same (excluding the .java filename extension
// in the source code filename). Letter case also matters. Therefore, a class named Class must be declared in
// a source code file named Class.java. Remember that it is also a Java convention to capitalize the first
// letter of class names, so class would be an unconventional class name (although technically, it would
// compile just fine). Note that per §1 Submission Instructions, if our automated grading script cannot com-
// pile the class you submit because you did not correctly name the class per the exercise requirements, then
// we treat your class as if it does not compile due to syntax errors, and grade accordingly.
public class H03_31 {
    // This is the method you are asked to write for Ex. 3.1. Name it exactly as requested so our testing
    // driver can call it. We refer to this method when our driver is testing it as the method under test
    // (MUT). If our automated grading script cannot call the MUT because you did not not correctly name it
    // per the exercise requirements, then we treat your method as if it does not compile due to syntax errors
    // and grade accordingly. Remember it is a Java convention to write the first letter of variable and
    // method names using a lowercase letter, so although Sum1toN() would compile just fine, it is an
    // unconventional method name and would confuse other Java programmers who are reading your code (and
    // you do not want to confuse your grader).
    public int sum1toN(int n) {
        // 1. Check for the base case of n = 1 and return 1 when it is detected (the sum of 1 to 1 is 1).
        // 2. Otherwise, call the method recursively to calculate sum1toN(n - 1). Then add n to the return
        // value from the recursive method call and return the sum.
    }
}
```

- The CLASS: and AUTHOR: comment lines must be included. It is good programming style to always write a header comment block at the top of each source code file. You may, and really should, add other information to your header comment blocks, but please include the CLASS: and AUTHOR: lines.
- All other comment lines are optional but we strongly encourage you to comment your code. In some situations, well-written comments may help us assign partial credit.
- Note that your instructor indents using 4 spaces. It does not matter to us how many spaces you indent but please configure your editor to insert **spaces** and not **hard tabs** when you hit the Tab key.
- Be sure for each exercise that you import any required classes so that your class will build. It is preferable to import just the one class that may be needed from a package, e.g., `import java.util.ArrayList;` rather than writing `import java.util.*;` which would import every class in the java.util package.

- Do not provide a *main*() method or driver routine in your class: for testing on your end, we suggest that you write your own test driver classes. For example, a simple test driver for Ex. 3.1 could be as shown below (note: the instructor typed this code in the assignment document but has not actually tried to compile it, so there could be some syntax errors in it; treat this as pseudocode).

```
// CLASS: H03_31_Test (Source Code File: H03_31_Test.java)
Import any required classes using import statements...
public class H03_31_Test {
    // Declare an instance of the class under test (CUT).
    private H03_31 mCut;

    public static void main(String[] pArgs) {
        new H03_31_Test().run();
    }

    private void run() {
        // Instantiate an object of the class under test (CUT), i.e., an instance of H03_31.
        mCut = new H03_31();

        // Write statements to perform a test case for a few different values of n. Before
        // you even attempt to compile this code and test your class, use Wolfram Alpha to
        // determine the correct sum for each value of n. To compute the sum of the integers
        // from 1 to n in Wolfram Alpha, enter the command: sum 1 to n
        performTestCase(1, 1, 1);
        performTestCase(2, 2, 3);
        performTestCase(3, 10, 55);
        ... And so on for a few other values of n. Make sure to try n = 65535.
    }

    private void performTestCase(int pTestCaseNum, int pN, int pExpectedSum) {
        printTestCaseInfo(pTestCaseNum, pN, pExpectedSum);
        int actualSum = mCut.sum1toN(pN);
        if (actualSum == pExpectedSum) System.out.println("passed\n");
        else System.out.println("failed\n");
    }

    private void printTestCaseInfo(int pTestCaseNum, int pN, int pExpectedSum) {
        System.out.println("Test Case Number " + pTestCaseNum);
        System.out.print("n = " + n + ", expected sum = " + pExpectedSum + " ==> ");
    }
}
```

- For these exercises, you do not need to copy-and-paste your code into the word processing document but **the *.java* source code files must be included in your zip archive** or otherwise we will be unable to compile your code and you may be assigned a score of zero.

2 Learning Objectives

1. To apply recursion solutions to solve problems.
2. To use the linear search and binary search algorithms to search a list for a key.
3. To analyze an algorithm to determine the asymptotic time complexity.
4. To determine the order of growth of a function and express it in Big O notation.
5. To implement sorting algorithms to sort a list of elements.
6. To analyze the time complexity of sorting algorithms.

3 Recursion

3.1 Learning Objective: To write a recursive method.

Instructions: See the instructions in §1.1 for what to submit for grading. This is not a complete program. Name your class H03_31 and save it in a file named H03_31.java. When you are done, copy H03_31.java to your *asuriteid*-h03 folder, i.e., to the same folder as the PDF.

Problem: The sum of the first n positive integers, $n \geq 1$, is:

$$sum(n) = \sum_{i=1}^{n} i$$

and can easily be computed using a for loop:

```
public int sum1toN(int n) { // You may assume n ≥ 1
     int sum = 0;
     for (int i = 1; i <= n; ++i) {
          sum += i;
     }
     return sum;
}
```

It is also possible to recursively compute the sum by recognizing that $sum(n) = n + sum(n - 1)$. For this exercise, write a recursive version of `public int sum1toN(int n)`.

Testing: We will be testing your method using our driver routine. For testing on your end, write your own driver routine in a class named H03_31_Test. See the example test driver source code file in §1.1.

3.2 Learning Objective: To write a recursive method.

Instructions: See the instructions in §1.1 for what to submit for grading. This is not a complete program. Name your class H03_32 and save it in a file named H03_32.java. When you are done, copy H03_32.java to your *asuriteid*-h03 folder, i.e., to the same folder as the PDF.

Problem: Write a recursive method `public double power(double x, int n)` that computes and returns x^n where $n \geq 0$. Hint: remember from algebra that $x^n = x \cdot x^{n-1}$, and $x^0 = 1$.

Testing: We will be testing your method using our driver routine. For testing on your end, write your own driver routine in a class named H03_32_Test. See the example test driver in §1.1 for Ex. 3.1; the test driver for H03_32_Test would be similar. Note: we will not call your method with a negative value for n.

3.3 Learning Objective: To write a recursive method.

Instructions: Name your class H03_33 and save it in a file named H03_33.java. Write your own test driver in H03_33_Test.java. Write H03_33_Test.run() so it will call and test your method for $x = 1, 2, 3$ and for each value of x, $n = 0, 1, 2, ..., 10$. This exercise is not graded so you do not need to submit H03_33.java nor H03_33_Test.java.

Problem: Write a recursive method `public double powerFaster(double x, int n)` that computes and returns x^n. In this method, when n is odd, use the same technique you used in Ex. 3.2 to compute x^n. However, when n is even, have the method return $(x^{n/2})^2$.

3.4 Learning Objective: To compare the time for *power*() and *powerFaster*() to calculate x^n. To determine why in some cases *powerFaster*() is faster than *power*(). To understand that the time complexity of a solution to a problem can vary depending on the algorithm that is employed.

Instructions: Modify H03_33.java by copying the *power*() method from class H03_32 to the H03_33 class. This question is not graded so you do not need to submit H03_33.java nor H03_33_Test.java.

Problem: Within the H03_33_Test class, declare two private int instance variables named *calls* and *callsFaster*. Modify the test driver in H03_33_Test that your wrote for Ex. 3.3 to call both *power*() and *powerFaster*() to calculate x^n for $x = 1, 2, 3$, and for each value of x, $n = 0, 1, 2, ..., 10$.

Next, within *power*(), write a statement as the first statement in the method which increments *calls*, before you check the base case. Within *powerFaster*(), write a statement as the first statement in the method which increments *callsFaster*, again, before you check the base case. After each method returns the computed power x^n back to H03_33_Test.run(), print the values of *calls* and *callsFaster* (I hope you have figured out that *calls* and *callsFaster* are counting the number of times each method is called during the computation of x^n). You should notice a difference between these two values for some values of n.

Explain why the number of calls to *powerFaster*() is fewer than the number of calls to *power*() in some cases.

3.5 Learning Objective: To write a recursive method.

Instructions: See the instructions in §1.1 for what to submit for grading. This is not a complete program. Name your class H03_35 and save it in a file named H03_35.java. When you are done, copy H03_35.java to your *asuriteid*-h03 folder, i.e., to the same folder as the PDF.

Problem: Write a recursive method `public String reverse(String s)` that returns the reverse of *s*. For example, if *s* is "Hello world" then the method shall return "dlrow olleH".

Hint: The base case occurs when the length of *s* is either 0 or 1. When the length is 0, it means *s* is the empty string "" and the reverse of the empty string is the empty string. When the length is 1, it means that you have a string such as "A" and the reverse of "A" is "A" (or in general, the reverse of a string of length ≤ 1 is the string itself).

Otherwise, retrieve the first character of *s* at index 0, call the character *c* (hint: read about the *String.charAt*() method). Extract the substring *t* at indices 1..*s.length*() - 1 (read the Java API documentation for the *String.substring*() method). Then, call *reverse*(*t*) which will return a string, let's call the string *revT*. Concatenate *c* onto the end of *revT* and then return the newly formed string.

Testing: We will be testing your method using our driver routine. For testing on your end, write your own test driver in a class named H03_35_Test. Within H03_35_Test.run() you should call *reverse*() on many different strings, including the empty string, to verify your solution is correct.

4 Linear and Binary Search

4.1 Learning Objective: To write a recursive method which searches a list for a key element.

Instructions: See the instructions in §1.1 for what to submit for grading. This is not a complete program. Name your class H03_41 and save it in a file named H03_41.java. When you are done, copy H03_41.java to your *asuriteid*-h03 folder, i.e., to the same folder as the PDF.

Problem: We discussed in the lectures how to write a linear search algorithm using a for loop which iterates over each element of a list. Linear search can also be implemented recursively. For this exercise, write a recursive method `public int recLinearSearch(ArrayList<String> pList, String pKey, int pBeginIdx, int pEndIdx)` that searches *pList* elements *pBeginIdx* up to and including *pEndIdx* for *pKey* and returns the index of *pKey* in *pList* if found or -1 if not found.

Hint: The base case is reached when *pBeginIdx* is greater than *pEndIdx* (what does this mean?). Otherwise, check to see if the element at *pBeginIdx* is equal to *pKey*. If it is, then return *pBeginIdx*. If it is not, then make a recursive method call which will search *pList* at elements *pBeginIdx* + 1 to *pEndIdx*.

Testing: We will be testing your method using our driver routine. For testing on your end, write your own testing driver in a class named H03_41_Test. For testing, your method will be called in this manner to search the entire list:

```
ArrayList<String> list = new ArrayList<>();
// we will populate list with several Strings...
int idx = recLinearSearch(list, "the key", 0, list.size() - 1);
```

Note that if *list* is empty, the method should return -1.

4.2 Learning Objective: For the student to demonstrate that he or she understands how the recursive binary search algorithm works.

Instructions: This question is not graded.

Problem: Suppose *list* is an *ArrayList* of *Integer*s and contains these elements (note that *list* is sorted in ascending order);

```
list = { 2, 3, 5, 10, 16, 24, 32, 48, 96, 120, 240, 360, 800, 1600 }
```

and we call the recursive binary search method, discussed in the lecture notes:

```
int index = recursiveBinarySearch(list, 10, 0, list.size() - 1);
```

where 10 is the key, 0 is the index of the first element in the range of *list* that we are searching (this becomes *pLow*), and *list.size*() - 1 is the index of the last element in the range of *list* that we are searching (this becomes *pHigh*).

Trace the method by hand and show the following: (1) The values of *pLow* and *pHigh* on entry to each method call; (2) The value of *middle* that is computed; (3) State which clause of the if-elseif-elseif statement will be executed, i.e., specify if `return middle;` will be executed, or if `return recursiveBinarySearch(pList, pKey, pLow, middle - 1);` will be executed, or if `return recursiveBinarySearch(pList, pKey, middle + 1, pHigh);` will be executed; (4) After the method returns, specify the value assigned to *index* and the total number of times that *recursiveBinarySearch*() was called, including the original call shown above.

4.3 Repeat Exercise 4.2 but this time let *pKey* be 150. This exercise is graded, so include your trace in the word processing document.

5 Analysis of Algorithms and Big O Notation

5.1 Learning Objective: To demonstrate that the student understands the definition of Big O. To demonstrate that the student knows how to use the definition of Big O to formally determine the asymptotic time complexity of a specific function.

Instructions: This exercise is graded, so include your solution in your word processing document.

Problem: Using the formal definition of Big O, prove mathematically that $f(n) = 2.5n + 4$ is $O(n)$. Hint: you must choose values for C and n_0 and a function $g(n)$, such that the criteria in definition are satisfied.

5.2 Learning Objective: To demonstrate that the student understands the definition of Big O. To demonstrate that the student knows how to use the definition of Big O to formally determine the asymptotic time complexity of a specific function. To demonstrate that the student understands that when $f(n)$ is a constant, no matter how large, the time complexity is $O(1)$.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: Using the formal definition of Big O, prove mathematically that $f(n) = -4 \times 10^{600{,}000}$ is $O(1)$. Hint: you must determine values for C and n_0 and a function $g(n)$, such that the criteria in definition are satisfied.

5.3 Learning Objective: To demonstrate that the student understands the definition of Big O. To demonstrate that the student understands that if a function $f(n)$ is $O(log_a\ n)$ then that same function is also $O(log_b\ n)$, where a and b are constants. That is, different bases for the logarithm function do not change the time complexity.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: An important concept to know regarding Big O notation is that for logarithmic complexity, the base is irrelevant. In other words, if $f(n)$ is a function that counts the number of times the key operation is performed as a function of n and $f(n)$ is $O(log_a\ n)$ then it it also true that $f(n)$ is $O(log_b\ n)$. For example, binary search—which is usually stated as being $O(lg\ n)$—is also $O(ln\ n)$, $O(log_{10}\ n)$, and $O(log_{3.14159265}\ n)$. Using the formal definition of Big O, prove mathematically that if $f(n)$ is $O(log_a\ n)$ then $f(n)$ is also $O(log_b\ n)$.

Hint: First, show that $log_a\ n = (log_a\ \mathrm{b})(log_b\ n)$. Note that $(log_a\ \mathrm{b})$ is a constant and that in the expression $C \cdot g(n)$, C is a constant. Next, since we are trying to show that $f(n) = log_a\ n$ is $log_b\ n$, note that you already showed that $log_a\ n = (log_a\ \mathrm{b})(log_b\ n)$.

5.4 Learning Objective: To demonstrate the student can identify the key operation when finding the asymptotic time complexity of an algorithm.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: Consider this *split*() method where: *pList* is an *ArrayList* of *Integer*s containing zero or more elements; *pEvenList* is an empty *ArrayList* of *Integer*s; and *pOddList* is an empty *ArrayList* of *Integer*s. On return, *pEvenList* will contain the even *Integer*s of *pList* and *pOddList* will contain the odd *Integer*s.

```
public void split(ArrayList<Integer> pList, ArrayList<Integer> pEvenList, ArrayList<Integer> pOddList) {
    for (int n : pList) {
        if (n % 2 == 0) pEvenList.add(n);
        else pOddList.add(n);
    }
}
```

To analyze the worst case time complexity of an algorithm we first identify the "key operation" and then derive a function which counts how many times the key operation is performed as a function of the size of the input. What is the key operation in this algorithm? Explain.

Hint: One way to determine the key operation is to identify the operation which will be performed the most number of times during the execution of algorithm. Also, the key operation is generally an operation that is at the heart of the algorithm. For example, in the iterative linear search algorithm, the key operation is the comparison of *pKey* to the element in *pList* that is currently being examined because in order to find *pKey* we compare each element in *pList* to *pKey*. That comparison will be performed the most number of times during the execution of the algorithm and the entire point of the algorithm is to find the key, so comparing the current element to *pKey* is at the heart of the algorithm.

5.5 Learning Objective: To demonstrate that the student understands the definition of Big O. To demonstrate that the student understands how to derive the function $f(n)$ which computes how many times the key operation of an algorithm is performed as a function of the size of the problem.

Instructions: This exercise is graded, so include your solution in your word processing document.

Problem: Continuing with the previous exercise, derive a function $f(n)$ which equates to the number of times the key operation is performed as a function of n, where n is the size of *pList*. State the worst case time complexity of *split*() in Big O notation. You do not need to formally prove it but explain your answer.

5.6 Learning Objective: To demonstrate that the student understands the definition of Big O. To demonstrate the student can analyze the time complexity of an algorithm.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: Would the time complexity of *split*() change if the elements of *pList* were sorted into ascending order? Explain.

5.7 Learning Objective: To demonstrate that the student understands how the binary search algorithm works and to implement an algorithm that is similar to binary search.

Instructions: See the instructions in §1.1 for what to submit for grading. This is not a complete program. Name your class H03_57 and save it in a file named H03_57.java. When you are done, copy H03_57.java to your *asuriteid*-h03 folder, i.e., to the same folder as the PDF.

Problem: Binary search is such an efficient searching algorithm because during each pass of the loop (in the iterative version) or in each method call (in the recursive version) the size of the list is essentially halved. An algorithm which repeatedly divides the problem size by two each time will have $O(lg\ n)$ performance. If reducing the size of the list to be searched by two is so efficient, it may seem that dividing the problem size by three each time would be even faster. To that end, consider this iterative ternary (ternary is base three) search method. Rewrite it is as a recursive ternary search method named `public int recTernarySearch(ArrayList<Integer> pList, Integer pKey, int pLow, int pHigh)`.

```
int ternarySearch(ArrayList<Integer> pList, Integer pKey) {
     int low = 0, high = pList.size() - 1;
     while (low <= high) {
          int range = high - low;
          int oneThirdIdx = (int)Math.round(low + range / 3.0);
          int twoThirdIdx = (int)Math.round(low + range / 1.3333333333333333);
          if (pKey.equals(pList.get(oneThirdIdx))) {
               return oneThirdIdx;
          } else if (pKey.equals(pList.get(twoThirdIdx))) {
               return twoThirdIdx;
          } else if (pKey < pList.get(oneThirdIdx)) {
               high = oneThirdIdx - 1;
          } else if (pKey > pList.get(twoThirdIdx)) {
               low = twoThirdIdx + 1;
          } else {
               low = oneThirdIdx + 1;
               high = twoThirdIdx - 1;
          }
     }
     return -1;
}
```

Testing: We will be testing your method using our testing driver. For testing on your end, write your own driver routine in a class H03_57_Test. Note that if *pList* is empty, the method should return -1. You do not need to be concerned with *pList* being **null**.

5.8 Learning Objective: To demonstrate the student understands the definition of Big O. To informally prove the time complexity of an algorithm.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: For *recTernarySearch*() the key operations are the four comparisons of *pKey* to the the elements of *pList*. Treat these four comparisons as one comparison and provide an informal proof of the worst case time complexity of ternary search (doing so will not change the time complexity of the algorithm because it only multiplies $g(n)$ by the constant 4). To simplify the analysis, assume that the size of the list on entry to *recTernarySearch*() is always a power of 3, e.g., on the first call assume the size of the list is 3^p, on the second call the size of the list is 3^{p-1}, on the third call 3^{p-2}, and so on.

6 Sorting

6.1 Learning Objective: To demonstrate that the student can implement the *java.lang.Comparable*<*T*> interface.

Instructions: For this exercise you will be modifying the *Point* class declaration which was discussed in the Module 1 lecture notes in *Objects and Classes : Section 1* (in *cse205-note-objs-classes-01.pdf*). The *Point.java* source code file can be found in the *Module 1 Source Code* zip archive). Include *Point.java* in your assignment zip archive.

Problem: Consider the *Point* class mentioned in the Instructions. Modify this class so it implements the *java.lang.Comparable*<*Point*> interface. We define *Point p1* to be less than *Point p2* if the distance from the origin to *p1* is less than the distance from the origin to *p2*; *p1* is greater than *p2* if the distance from the origin to *p1* is greater than the distance from the origin to *p2*; otherwise, if the distances are equal then *p1* is equal to *p2*.

Testing: We will be testing your method using our test drive. For testing on your end, write your own driver routine in a class named H03_61_Test.

6.2 Learning Objective: To demonstrate an understanding of how the insertion sort algorithm works.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: Consider this *ArrayList* of *Integer*s, which is to be sorted into ascending order:

```
list = { 13, 75, 12, 4, 18, 6, 9, 10, 7, 14, 15 }.
```

Trace the *insertionSort*() method and show the contents of *list*: (1) on entry to *insertionSort*(); (2) after the for j loop terminates each time but before the for i loop is repeated; and (3) after the for i loop terminates.

6.3 Learning Objective: To demonstrate that the student can estimate the actual time an algorithm will take to execute. For the student to understand that even on a spectacularly fast computer system, a $O(n^2)$ algorithm is still too slow for large problem sizes.

Instructions: This exercise is not graded, so you do not need to include your solution in your word processing document.

Problem: Your after-hours research finally pays off: you discover a way to build a computer system that is one billion (10^9) times faster than the fastest system currently in existence, where we are assuming that the fastest system in existence is the one we discussed in the Week 5 notes *Sorting Algorithms : Section 6* (the file is named *cse205-note-sort-06.pdf*) which performs a comparison in 25 nanoseconds.

On your new computer system, each comparison in *findMinIndex*() and *findMaxIndex*() of selection sort requires only 25×10^{-18} seconds, which is 25 attoseconds[1]. Create a table similar to the one in the lecture notes showing how long selection sort will take to sort lists of sizes 10^p, for $p = 2, 3, 4, ..., 10$ (Excel is an ideal tool for doing this).

6.4 Learning Objective: To demonstrate that the student can use mathematics to solve a time-complexity related problem.

Instructions: This exercise is graded, so write your solution in your word processing document.

Problem: If we require *insertionSort*() to sort a list of 10-billion elements in no more than one minute, how many times faster would your new computer system need to be than the fastest system currently in existence, i.e., the one that performs a comparison in 25 ns?

1 That's fast. The smallest amount of time ever measured is on the order of zeptoseconds, 10^{-21} seconds which is only 1,000 times shorter than an atto-second.