LOGBOOK

Algorithms – Processes and data

J.Pritchard U1661665
U1661665@unimail.hud.ac.uk

Practical 1

(Notes) Weeks 1 and 2:

- There are already plenty of well known, efficient algorithms, don't re-invent the wheel.
- Skills to be learnt:
  - Designing and implementing solutions to problems.
  - Analysing potent solutions.
  - Determining which fragments of a program to optimise.
- For each loops
  - Only use iterators in special circumstances (for instance making changes to the collection or stopping part way.)
  - Only use indices if that information is required.
  - If none of these are required, then use a for each loop.
- Testing (Black/White box)
  - Code should have clear unambiguous specification. (Cannot test without this)
  - Ideal testing done by someone else. (trying to break your code)
  - Black box = only knowledge about spec (no knowledge of implementation).
  - White box = knowledge of implementation (testing individual units of the implementation).
- Types of testing.
  - Unit : White : Do the methods/class work correctly?
  - Integration : Black/White : low level done, does it all work put together?
  - Functional : Black : Does system meet technical specifications?
  - Performance : Black : Is it efficient and effective?
  - Usability : Black : Easy to use? Logical? Good user experience?
  - Acceptance : Black : Real client testing, is it what they want?
  - System : Black : Does it work in other environments?
  - Stress : Black : Does it work at its absolute limit?
  - Regression : Black : Has the last modification broken everything?
- Junit testing
  - Test each method
    - Each test tests **one** facet of **each** method.
  - Sorter example.
    - Size? Contents? Order?
  - What properties should the result of the method have?
  - Boundary and beyond tests.
    - Does the method go wrong in the correct way?
- Assertions
  - assertEquals(type expected, type actual)
  - assertSame(Object expected, Object actual)
  - assertNotSame(Object expected, Object actual)
  - assertNull(Object object)
  - assertNotNull(Object object)
  - assertTrue(Boolean statement)
  - assertFalse(Boolean statement)
  - fail(String message)
- Exceptions

    - Write tests that expect errors to be thrown
    - Use a try catch with a fail()
- Set up and tear down
    - Methods ran @Before and @After can be used for timing the tests.
    - @BeforeClass and @AfterClass are ran for the entire testing suite, rather than for each method.

(Logbook) Question 1: Implement the searcher interface, using the more efficient approach (with a small helper array) outlined in the lecture. Call this class CleverSearcher.

Pseudo code for the ‘More efficient approach’

```
2.4.2 Clever Solution

Algorithm

_ Input: An array of ints and an array index
_ Output: kth largest element of the array
_ read the first k elements into an auxilliary array (these will be the k
  largest found so far)
_ sort the k-element array (in some way!)
_ then. . .

for each remaining element f
 if (it is smaller than the smallest element of the aux. array)
  throw it away;
 else
  remove the current smallest element of the aux. array;
  place the element into the correct position in the aux. array;
return the smallest element of the aux. array
```

Code listing.

```
package searcher;

import java.util.Arrays;

/**
 * @author JPritchardU1661665
 * @version October 2018
 */
public class CleverSearcher extends Searcher
{
    CleverSearcher(int[] array, int k)
    {
        super(array, k);
    }

    /**
     * Find the kth largest element in an array of ints using the "Clever"
     * solution from the lecture.
     *
     * <ul>
     *     <li> Create an auxilliary array of the first k elements in the original array</li>
     *     <li> For each of the remaining elements in the original array</li>
     *     <ul>
     *         <li> If it's bigger than the smallest element in the Aux array</li>
     *         <ul>
     *             <li> Replace the smallest element with the new element</li>
     *             <li> Sort the aux array</li>
     *         </ul>
     *     </ul>
     *     <li> When finished, return the smallest element in the Aux array (The kth largest in the original array)</li>
     * </ul>
     *
     * @return kth largest element of array.
     * @throws IndexingError
     */
    @Override
    public int findElement() throws IndexingError
    {
        int k = super.getIndex();
        int[] array = super.getArray();

        //Throw an exception if an invalid index has been provided.
```

```java
    if(k <= 0 || k > array.length)
        {
            throw new IndexingError();
        }

        //Create an array of size k
        //This will hold the 'k' largest elements at the end of the process.
        int[] auxArray = new int[k];

        //Populate it with the first k elements from the original array.
        for(int x = 0; x<k; x++)
        {
            auxArray[x] = array[x];
        }

        //Sort the 'k' array so that the smallest element is at the start.
        Arrays.sort(auxArray);

        //For each of the remaining elements in the original array.
        for(int x = k; x < array.length; x++)
        {
            //If it is larger than the smallest number in the 'current' 'k' largest elements...
            if(array[x] > auxArray[0])
            {
                //'bump' it out of the 'k' largest elements (replace it)
                auxArray[0] = array[x];
            }

            //Re-sort the 'k' array so the smallest is at the start again.
            Arrays.sort(auxArray);
        }
        //return the smallest element in the 'k' array (the kth largest element)
        return auxArray[0];
    }

}
```

(Logbook) Question 2: Create a test class to test the functionality of your implementation.

Code Listings.

```java
package searcher;

/**
 * @author JPritchardU1661665
 * @version October 2018
 */

class CleverSearcherTest extends SearcherTest{

    @Override
    protected Searcher createSearcher(int[] array, int index) throws IndexingError
    {
        return new CleverSearcher(array, index);
    }
}
```

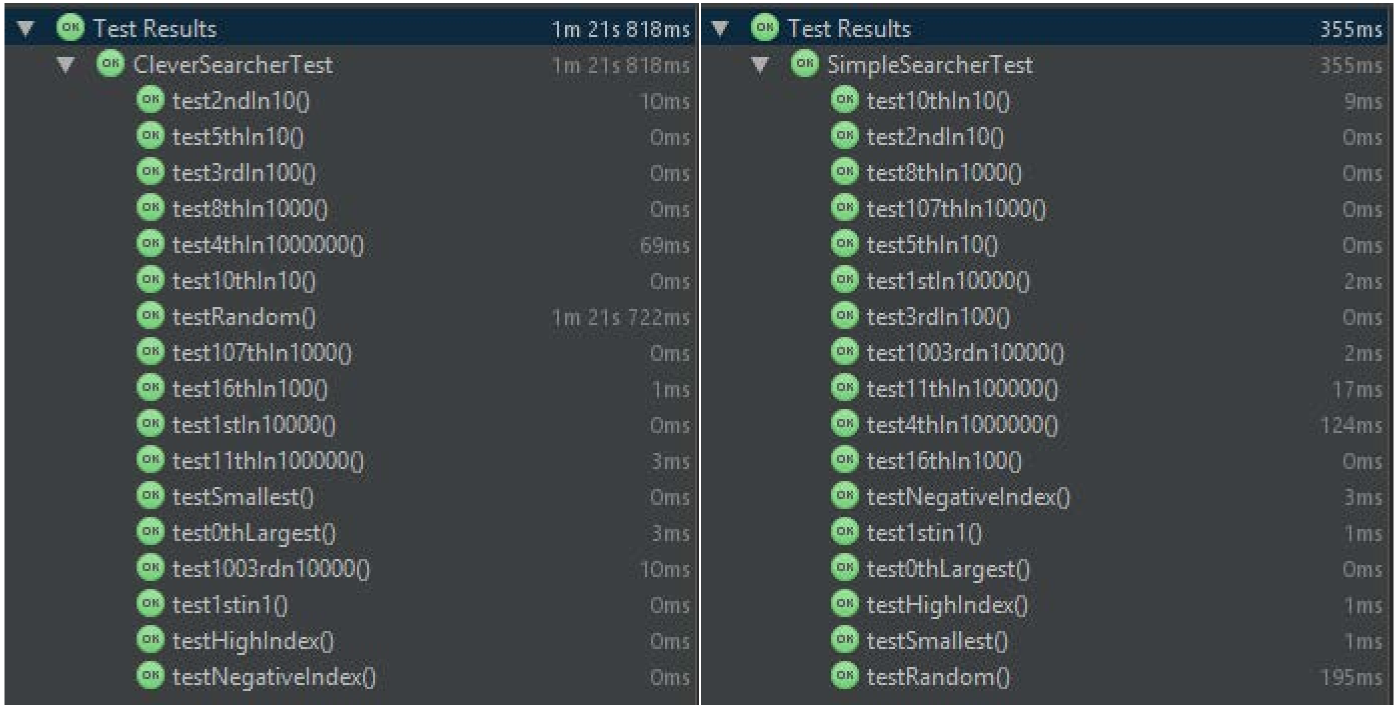
Additional tests added to SearcherTest

```java
//Used to control the number of randomly generated tests ran.
public static final int NUM_RANDOM_TESTS = 20;
...
...
/**
 * Test that a searcher throws an indexing error when an invalid index error is passed.
 * The test uses a random listing generator to create a random listing of the required size.
 *
 * @param arraySize the size of the random listing to be generated.
 * @param index the index (Should be
 * @throws IndexingError should be thrown when an invalid index value is passed in relative to arraySize
 */
protected void testIndexError(int arraySize, int index) throws IndexingError
{
    if(index < 1 || index > arraySize)
    {
        ArrayGenerator generator = new CleverRandomListingGenerator(arraySize);
        Searcher search = createSearcher(generator.getArray(), index);
        assertThrows(IndexingError.class, search::findElement);
    }
    else
```

```java
        fail("Expected an invalid index. re-evaluate test");
}
...
@org.junit.jupiter.api.Test
void test10thIn10() throws IndexingError
{
    testSearcher(10, 10);
}

@org.junit.jupiter.api.Test
void test1stin1() throws IndexingError
{
    testSearcher(1, 1);
}

/**
 * Create random test data between 1 and 200000
 * Run testSearcher with this generated data.
 *
 * Repeat this a number of times specified by NUM_RANDOM_TESTS
 *
 * @throws IndexingError If the index generated is invalid with respect to arraySize.
 */
@org.junit.jupiter.api.Test
void testRandom() throws IndexingError
{
    Random rand = new Random();
    for(int x = 0; x < NUM_RANDOM_TESTS; x++)
    {
        int size = rand.nextInt(199999) + 1;
        int index = rand.nextInt(size) + 1;
        testSearcher(size, index);
    }
}

@org.junit.jupiter.api.Test
void testSmallest() throws IndexingError
{
    testSearcher(10, 1);
}

@org.junit.jupiter.api.Test
void test0thLargest() throws IndexingError
{
```

```
    testIndexError(5, 0);
}

@org.junit.jupiter.api.Test
void testNegativeIndex() throws IndexingError
{
    testIndexError(5, -1);
}

@org.junit.jupiter.api.Test
void testHighIndex() throws IndexingError
{
    testIndexError(5, 6);
}
```

Test results



(Logbook) Question 3: Also create a timer class to time the execution of the findElement method in your CleverSearcher implementation. Compare this with the time taken by the SimpleSearcher implementation when performing searches of the same size.

CleverSearcherTimer code listing.

```
package searcher;

/**
   A timer implementation for Clever searchers that times the findElement method

   @author JPritchardU1661665
   @version October 2018
 */

import arrayGenerator.ArrayGenerator;
import arrayGenerator.CleverRandomListingGenerator;
import timer.Timer;

public class CleverSearcherTimer extends CleverSearcher implements Timer{
    // All timings will be done with an index of 5
    private final static int K = 5;

    private CleverSearcherTimer(int[] array) {
        super(array, K);
    }

    @Override
    public void timedMethod() {
        try {
            findElement();
        } catch (IndexingError indexingError) {
            // simply ignore indexing errors here
            // with K at 5, and a minimum task size (array size) of 10, indexing errors should not occur
            // duirng timing
        }
    }

    @Override
    public long getMaximumRuntime() {
        return 1;
    }
```

```java
    /**
     * Minimum task size (array size) is set to ten, to avoid indexing errors (index is always five)
     * @return minimum task size of ten
     */
    @Override
    public int getMinimumTaskSize() {
        return 10;
    }

    @Override
    public int getMaximumTaskSize() {
        return 100000000;
    }

    @Override
    public Timer getTimer(int size) {
        ArrayGenerator generator = new CleverRandomListingGenerator(size);
        return new CleverSearcherTimer(generator.getArray());
    }


    public static void main(String[] args) throws IndexingError {
        CleverSearcherTimer cleverTimer = new CleverSearcherTimer(null);
        cleverTimer.timingSequence();


    }
}
```

Simple searcher times.

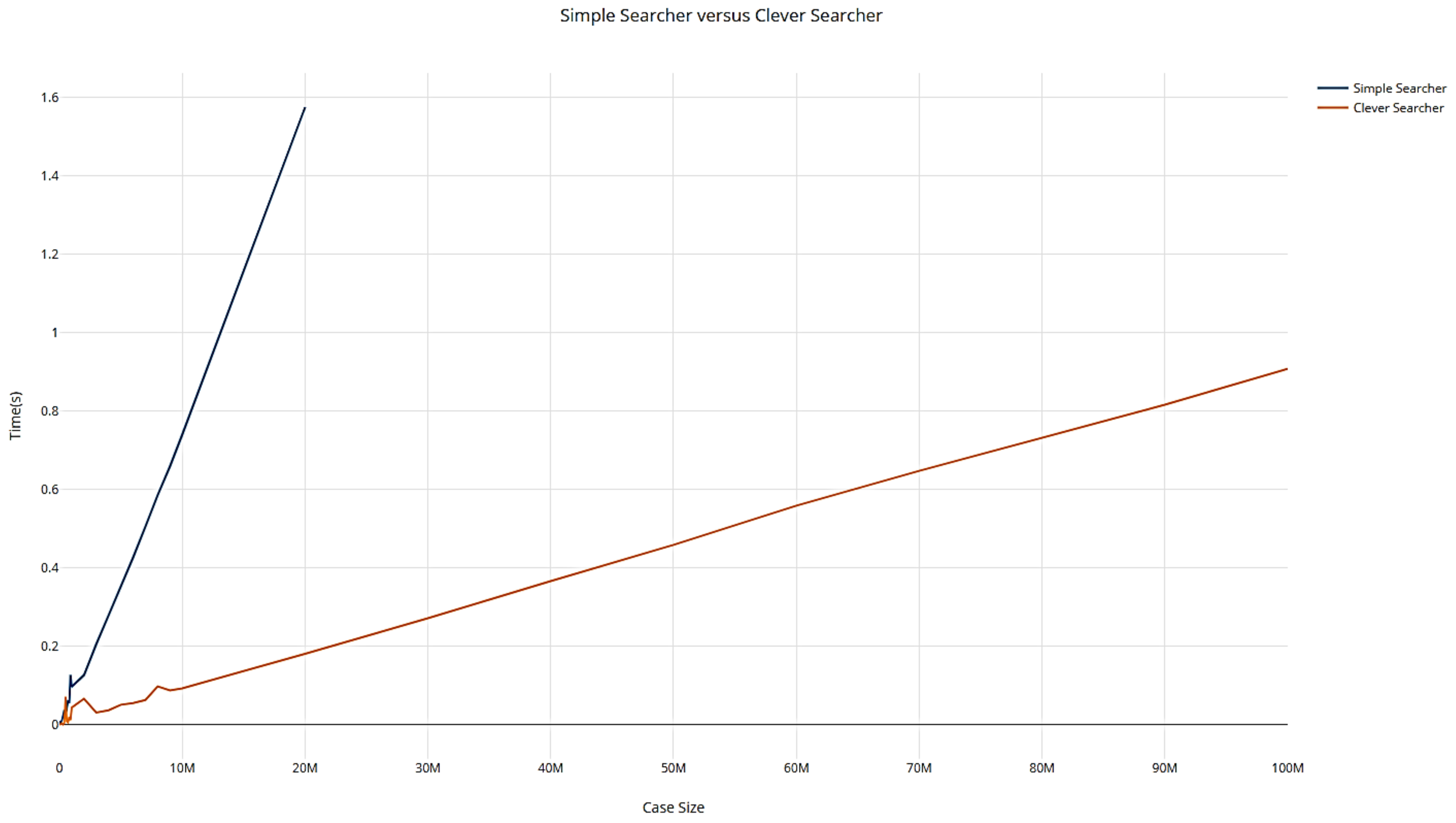
```
class searcher.SimpleSearcherTimer took 0.000284178 seconds for a task of size 10
class searcher.SimpleSearcherTimer took 0.000004381 seconds for a task of size 20
class searcher.SimpleSearcherTimer took 0.000006425 seconds for a task of size 30
class searcher.SimpleSearcherTimer took 0.000007886 seconds for a task of size 40
class searcher.SimpleSearcherTimer took 0.000010514 seconds for a task of size 50
class searcher.SimpleSearcherTimer took 0.000018692 seconds for a task of size 60
class searcher.SimpleSearcherTimer took 0.000013143 seconds for a task of size 70
class searcher.SimpleSearcherTimer took 0.000015772 seconds for a task of size 80
class searcher.SimpleSearcherTimer took 0.000020153 seconds for a task of size 90
class searcher.SimpleSearcherTimer took 0.000020737 seconds for a task of size 100
class searcher.SimpleSearcherTimer took 0.00003826 seconds for a task of size 200
class searcher.SimpleSearcherTimer took 0.000073308 seconds for a task of size 300
class searcher.SimpleSearcherTimer took 0.000092877 seconds for a task of size 400
class searcher.SimpleSearcherTimer took 0.000112445 seconds for a task of size 500
class searcher.SimpleSearcherTimer took 0.001391098 seconds for a task of size 600
class searcher.SimpleSearcherTimer took 0.000049943 seconds for a task of size 700
class searcher.SimpleSearcherTimer took 0.000046438 seconds for a task of size 800
class searcher.SimpleSearcherTimer took 0.000057244 seconds for a task of size 900
class searcher.SimpleSearcherTimer took 0.000062794 seconds for a task of size 1,000
class searcher.SimpleSearcherTimer took 0.000231899 seconds for a task of size 2,000
class searcher.SimpleSearcherTimer took 0.000340254 seconds for a task of size 3,000
class searcher.SimpleSearcherTimer took 0.000238616 seconds for a task of size 4,000
class searcher.SimpleSearcherTimer took 0.000294108 seconds for a task of size 5,000
class searcher.SimpleSearcherTimer took 0.000351353 seconds for a task of size 6,000
class searcher.SimpleSearcherTimer took 0.000419111 seconds for a task of size 7,000
class searcher.SimpleSearcherTimer took 0.000496215 seconds for a task of size 8,000
class searcher.SimpleSearcherTimer took 0.000601943 seconds for a task of size 9,000
class searcher.SimpleSearcherTimer took 0.000786818 seconds for a task of size 10,000
class searcher.SimpleSearcherTimer took 0.001587656 seconds for a task of size 20,000
class searcher.SimpleSearcherTimer took 0.005149078 seconds for a task of size 30,000
class searcher.SimpleSearcherTimer took 0.003828075 seconds for a task of size 40,000
class searcher.SimpleSearcherTimer took 0.005442309 seconds for a task of size 50,000
class searcher.SimpleSearcherTimer took 0.007034347 seconds for a task of size 60,000
class searcher.SimpleSearcherTimer took 0.007150588 seconds for a task of size 70,000
class searcher.SimpleSearcherTimer took 0.004036608 seconds for a task of size 80,000
class searcher.SimpleSearcherTimer took 0.007004849 seconds for a task of size 90,000
class searcher.SimpleSearcherTimer took 0.007524137 seconds for a task of size 100,000
class searcher.SimpleSearcherTimer took 0.010810143 seconds for a task of size 200,000
class searcher.SimpleSearcherTimer took 0.025035968 seconds for a task of size 300,000
class searcher.SimpleSearcherTimer took 0.036946314 seconds for a task of size 400,000
class searcher.SimpleSearcherTimer took 0.031489401 seconds for a task of size 500,000
class searcher.SimpleSearcherTimer took 0.036939888 seconds for a task of size 600,000
class searcher.SimpleSearcherTimer took 0.063135056 seconds for a task of size 700,000
class searcher.SimpleSearcherTimer took 0.054560366 seconds for a task of size 800,000
class searcher.SimpleSearcherTimer took 0.12695938 seconds for a task of size 900,000
class searcher.SimpleSearcherTimer took 0.096649332 seconds for a task of size 1,000,000
class searcher.SimpleSearcherTimer took 0.126786771 seconds for a task of size 2,000,000
class searcher.SimpleSearcherTimer took 0.205811536 seconds for a task of size 3,000,000
class searcher.SimpleSearcherTimer took 0.278739477 seconds for a task of size 4,000,000
class searcher.SimpleSearcherTimer took 0.352383851 seconds for a task of size 5,000,000
class searcher.SimpleSearcherTimer took 0.426500198 seconds for a task of size 6,000,000
class searcher.SimpleSearcherTimer took 0.504908418 seconds for a task of size 7,000,000
class searcher.SimpleSearcherTimer took 0.585993983 seconds for a task of size 8,000,000
class searcher.SimpleSearcherTimer took 0.658975956 seconds for a task of size 9,000,000
class searcher.SimpleSearcherTimer took 0.740822054 seconds for a task of size 10,000,000
class searcher.SimpleSearcherTimer took 1.575312639 seconds for a task of size 20,000,000
Time limit of 1 seconds reached.  Ending timing sequence.
```

Clever searcher times.

```
class searcher.CleverSearcherTimer took 0.001265511 seconds for a task of size 10
class searcher.CleverSearcherTimer took 0.000006426 seconds for a task of size 20
class searcher.CleverSearcherTimer took 0.000007593 seconds for a task of size 30
class searcher.CleverSearcherTimer took 0.000009346 seconds for a task of size 40
class searcher.CleverSearcherTimer took 0.000021029 seconds for a task of size 50
class searcher.CleverSearcherTimer took 0.000013727 seconds for a task of size 60
class searcher.CleverSearcherTimer took 0.000024825 seconds for a task of size 70
class searcher.CleverSearcherTimer took 0.000026286 seconds for a task of size 80
class searcher.CleverSearcherTimer took 0.000024241 seconds for a task of size 90
class searcher.CleverSearcherTimer took 0.00002833 seconds for a task of size 100
class searcher.CleverSearcherTimer took 0.000136394 seconds for a task of size 200
class searcher.CleverSearcherTimer took 0.00006221 seconds for a task of size 300
class searcher.CleverSearcherTimer took 0.000070679 seconds for a task of size 400
class searcher.CleverSearcherTimer took 0.000086451 seconds for a task of size 500
class searcher.CleverSearcherTimer took 0.000176114 seconds for a task of size 600
class searcher.CleverSearcherTimer took 0.000126464 seconds for a task of size 700
class searcher.CleverSearcherTimer took 0.001062527 seconds for a task of size 800
class searcher.CleverSearcherTimer took 0.000049651 seconds for a task of size 900
class searcher.CleverSearcherTimer took 0.000060165 seconds for a task of size 1,000
class searcher.CleverSearcherTimer took 0.001922654 seconds for a task of size 2,000
class searcher.CleverSearcherTimer took 0.0001104 seconds for a task of size 3,000
class searcher.CleverSearcherTimer took 0.000139022 seconds for a task of size 4,000
class searcher.CleverSearcherTimer took 0.0001656 seconds for a task of size 5,000
class searcher.CleverSearcherTimer took 0.000224889 seconds for a task of size 6,000
class searcher.CleverSearcherTimer took 0.000365664 seconds for a task of size 7,000
class searcher.CleverSearcherTimer took 0.000262273 seconds for a task of size 8,000
class searcher.CleverSearcherTimer took 0.000297613 seconds for a task of size 9,000
class searcher.CleverSearcherTimer took 0.000329156 seconds for a task of size 10,000
class searcher.CleverSearcherTimer took 0.000490083 seconds for a task of size 20,000
class searcher.CleverSearcherTimer took 0.000599022 seconds for a task of size 30,000
class searcher.CleverSearcherTimer took 0.000452699 seconds for a task of size 40,000
class searcher.CleverSearcherTimer took 0.000615961 seconds for a task of size 50,000
class searcher.CleverSearcherTimer took 0.000634654 seconds for a task of size 60,000
class searcher.CleverSearcherTimer took 0.000663568 seconds for a task of size 70,000
class searcher.CleverSearcherTimer took 0.00069774 seconds for a task of size 80,000
class searcher.CleverSearcherTimer took 0.000867721 seconds for a task of size 90,000
class searcher.CleverSearcherTimer took 0.000895759 seconds for a task of size 100,000
class searcher.CleverSearcherTimer took 0.003799453 seconds for a task of size 200,000
class searcher.CleverSearcherTimer took 0.006918106 seconds for a task of size 300,000
class searcher.CleverSearcherTimer took 0.004956316 seconds for a task of size 400,000
class searcher.CleverSearcherTimer took 0.071783053 seconds for a task of size 500,000
class searcher.CleverSearcherTimer took 0.008708455 seconds for a task of size 600,000
class searcher.CleverSearcherTimer took 0.006275567 seconds for a task of size 700,000
class searcher.CleverSearcherTimer took 0.020362077 seconds for a task of size 800,000
class searcher.CleverSearcherTimer took 0.011277444 seconds for a task of size 900,000
class searcher.CleverSearcherTimer took 0.043705537 seconds for a task of size 1,000,000
class searcher.CleverSearcherTimer took 0.066527372 seconds for a task of size 2,000,000
class searcher.CleverSearcherTimer took 0.030960181 seconds for a task of size 3,000,000
class searcher.CleverSearcherTimer took 0.036818097 seconds for a task of size 4,000,000
class searcher.CleverSearcherTimer took 0.05107342 seconds for a task of size 5,000,000
class searcher.CleverSearcherTimer took 0.055553381 seconds for a task of size 6,000,000
class searcher.CleverSearcherTimer took 0.063107602 seconds for a task of size 7,000,000
class searcher.CleverSearcherTimer took 0.097865484 seconds for a task of size 8,000,000
class searcher.CleverSearcherTimer took 0.087484674 seconds for a task of size 9,000,000
class searcher.CleverSearcherTimer took 0.093011974 seconds for a task of size 10,000,000
class searcher.CleverSearcherTimer took 0.181060915 seconds for a task of size 20,000,000
class searcher.CleverSearcherTimer took 0.27074541 seconds for a task of size 30,000,000
class searcher.CleverSearcherTimer took 0.366562361 seconds for a task of size 40,000,000
class searcher.CleverSearcherTimer took 0.458872799 seconds for a task of size 50,000,000
class searcher.CleverSearcherTimer took 0.558946574 seconds for a task of size 60,000,000
class searcher.CleverSearcherTimer took 0.647832276 seconds for a task of size 70,000,000
class searcher.CleverSearcherTimer took 0.731417611 seconds for a task of size 80,000,000
class searcher.CleverSearcherTimer took 0.816461804 seconds for a task of size 90,000,000
class searcher.CleverSearcherTimer took 0.908251492 seconds for a task of size 100,000,000
Maximum task size, 100000000, reached. Ending timing sequence.
```

Comparison.

As can be seen in the test data, the simple searcher only managed to reach a test case of 20,000,000 before hitting the time limit set. The Clever searcher on the other hand hit the cap on task size, accomplishing a case of 100,000,000 in the same time it took the Simple searcher to accomplish a case of ~15,000,000. Plotting a graph of these results clearly shows the difference in efficiency between the two methods.



The Plotted results would suggest from their straightness that the two algorithms both have a time complexity of O(n). For the Clever searcher, this is also true for the resource complexity as each additional element in the case size triggers one more sorting operation. Despite being ignorant of the implementation, the sorting operation used is likely to be of an O(n^2) complexity, however the size of the array being sorted never changes with the clever searcher, therefore the performance scales linearly with the case size. It's a different story for the Simple searcher however, as the sorting operation sorts an array of one bigger for every new element introduced into the case size. Therefore the Simple searcher is going to inherit the complexity of the sorting algorithm, which is guaranteed to be >O(n).

(Logbook) Self evaluation.

I believe that my documentation is Javadoc compliant, with any methods/classes that require descriptions having them. Any complex methods also have comments and documentation within the code to explain what's happening to any outsider/"colleague" looking at the code for the first time. I believe the automated test suite I have is almost entirely complete, with randomised tests being used to simulate real-world data. All variables are named appropriately and with a consistent naming standard. The structure of my programming is in my eyes completely fine, with the use of white space being implemented to aid readability and decrease time spent re-reading code out of initial confusion.

I'd give myself a 5/5 for this logbook exercise due to the reasons stated above.

Practical 2

(Notes) Weeks 3 and 4:

- Generics
    - Define a class (etc.) with an abstract type
        - Public class Box <E>
            - Private E value
            - Public Box (E value) {this.value = value}
            - Public E getValue()
    - Generic types must be objects – Cannot be primitive data types.
        - Wrapper classes such as Integer exist.
    - Generic classes can be extended into specific classes.
        - Box<T>
        - IntegerBox extends Box<Integer>
    - Generic methods can be written inside non generic classes.
        - Public Class Boxes
            - Public <T> void method (Box<T> box){}
    - Developing generic methods
        - Develop a method for a single type, then adapt it to be generic.

(Logbook) Question 1: Write a generic method to exchange two elements of an array. The method should take an array, and two integer indices into the array, and swap the two entries in the array at those indices.

Code listing for 'Swap' class

```
package genericMethods;

/**
 * Created by u1661665 (Joshua Pritchard) on 24/10/2018.
 */


/**
 * This class defines a static method to swap two elements of the same type
 * in an array.
 */
public class Swap
{
    /**
     * Swaps two elements within an array
     *
     * @param array the array to be modified, of type T.
     * @param index1 the index of the first value to be swapped.
     * @param index2 the index of the second value to be swapped.
     * @param <T> the type of data the array contains.
     * @throws IndexOutOfBoundsException if any of (index1, index2) are invalid relative to param array.
     */
    public static <T> void swap(T[] array, int index1, int index2) throws IndexOutOfBoundsException
    {
        //This section builds a string of invalid indices, then if that string is not empty, throws an
        // indexOutOfBoundsException, passing through the indices that were found to be invalid and the size of the array.
        StringBuilder badIs = new StringBuilder("");
        boolean bad = false;
        if (index1 < 0 || index1 > array.length - 1 || index1 == index2)
        {
            badIs.append("1: " + index1 +", ");
            bad = true;
        }
        if (index2 < 0 || index2 > array.length - 1 || index1 == index2)
        {
            badIs.append("2: " + index2 +", ");
            bad = true;
        }
        if(bad)
```

```java
        {
            throw new IndexOutOfBoundsException("Invalid indexes: " + badIs.toString() + " relative to param array of size: " +
array.length);
        }

        //Use a temp T object to swap the two values.

        T temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;

    }
}
```

Code listing for ‘RandomString’ class

```java
package RandomMethods;

import java.security.SecureRandom;
import java.util.Locale;
import java.util.Objects;
import java.util.Random;

//Project: Code from StackOverflow page https://stackoverflow.com/questions/41107/how-to-generate-a-random-alpha-numeric-string
//Code Author: https://stackoverflow.com/users/3474
//Licensed under CC-Wiki  https://creativecommons.org/licenses/by-sa/3.0/

//This entire class has been taken from the above source.

public class RandomString {

    /**
     * Generate a random string.
     */
    public String nextString() {
        for (int idx = 0; idx < buf.length; ++idx)
            buf[idx] = symbols[random.nextInt(symbols.length)];
        return new String(buf);
    }

    public static final String upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public static final String lower = upper.toLowerCase(Locale.ROOT);

    public static final String digits = "0123456789";

    public static final String alphanum = upper + lower + digits;

    private final Random random;

    private final char[] symbols;

    private final char[] buf;

    public RandomString(int length, Random random, String symbols) {
        if (length < 1) throw new IllegalArgumentException();
        if (symbols.length() < 2) throw new IllegalArgumentException();
        this.random = Objects.requireNonNull(random);
```

```
        this.symbols = symbols.toCharArray();
        this.buf = new char[length];
    }

    /**
     * Create an alphanumeric string generator.
     */
    public RandomString(int length, Random random) {
        this(length, random, alphanum);
    }

    /**
     * Create an alphanumeric strings from a secure generator.
     */
    public RandomString(int length) {
        this(length, new SecureRandom());
    }

    /**
     * Create session identifiers.
     */
    public RandomString() {
        this(21);
    }

}
```

Code listing for 'RandomStringArray' class

```
package RandomMethods;

import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Created by u1661665 (Joshua Pritchard) on 24/10/2018.
 */

/**
 * Defines a static method for returning a random array of Strings.
 */
public class RandomStringArray
{
    /**
     * Generates a randomly sized and populated arrayList of Strings as per the parameters specified.
     *
     * @param arrayLowSize the lowest size the arrayList can be.
     * @param arrayHighSize the highest size the arrayList can be.
     * @param stringLowLen the lowest size a string within the arraylist can be.
     * @param stringHighLen the highest size a string within the arrayList can be.
     * @return the arrayList generated and populated with random strings.
     */
    public static ArrayList<String> getRandomStringArray(int arrayLowSize, int arrayHighSize, int stringLowLen, int
stringHighLen)
    {
        Random rand = new Random();

        //values used for random value generation.
        int randArrayBound = arrayHighSize - arrayLowSize;
        int randStringBound = stringHighLen - stringLowLen;
        int arraySize = rand.nextInt(randArrayBound) + arrayLowSize;

        //The array of strings to be created.
        ArrayList<String> testData = new ArrayList<>(arraySize);

        //Add random strings into 'testData' 'arraySize' times.
        for(int x = 0; x < arraySize; x++)
        {
            //Create a new random string generator, with a random length of string.
            RandomMethods.RandomString randString = new RandomMethods.RandomString
```

```
                    (rand.nextInt(randStringBound) + stringLowLen, ThreadLocalRandom.current());

            //Generate a random string and add it to the array.
            testData.add(randString.nextString());
        }

        return testData;
    }
}
```

Code listing for 'SwapTest' class

```java
package genericMethods;

/**
 * Created by u1661665 (Joshua Pritchard) on 24/10/2018.
 */

import RandomMethods.RandomStringArray;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.fail;

/**
 * Defines a set of testing methods for the class 'Swap'
 */
class SwapTest
{

    //RandomStringsArray values - Use these to modify the operation of the testRandomStringsArray test method.
    private final int HIGHEST_SIZE_ARRAY = 10;
    private final int LOWEST_SIZE_ARRAY = 3;
    private final int HIGHEST_SIZE_STRING = 10;
    private final int LOWEST_SIZE_STRING = 3;

    //Iterations of each random test method ran.
    private final int NUM_RANDOM_TESTS = 500000;

    /**
     * Defines a generic method to test that the swapped values have been swapped into
     * the correct positions.
     *
     * @param array the array to be modified.
     * @param index1 the first index of the array to be swapped.
     * @param index2 the second index of the array to be swapped.
     * @param <T> the type of data contained by the array.
     * @return true if after the swap, the elements have indeed, swapped place. false otherwise.
     */
    private <T> boolean testSwapContents(T[] array, int index1, int index2)
    {
        //Copy the initial values of the array values at the indices.
```

```java
        T valueI1 = array[index1];
        T valueI2 = array[index2];

        //perform the swap operation on the array.
        Swap.swap(array, index1, index2);

        //return the result of the comparison of initial values to 'post-swap' values.
        return array[index2] == valueI1 && array[index1] == valueI2;
    }

    private int getIndex(boolean badIndex, int size)
    {
        Random rand = new Random();

        //get a random index location inside of the array
        int ind = rand.nextInt(size - 1);

        //if a bad index is required, make sure it's not 0, then minus this index value.
        if(badIndex)
        {
            if(ind == 0) ind++;
            ind = -ind;
        }

        return ind;
    }

    /**
     * Defines a method for testing whether a bad index was passed in to a Swap operation or not.
     * This method expects that the indices passed to it are bad.
     *
     * @param array the array to be modified.
     * @param index1 the first index location to be swapped.
     * @param index2 the second index location to be swapped.
     * @param <T> the type of data contained within param 'array'
     * @return false if the indices were fine, true if the indices were bad, as expected.
     */
    private <T> boolean testBadIndex(T[] array, int index1, int index2)
    {
        try{
            Swap.swap(array, index1, index2);
            return false;
        }
        catch (IndexOutOfBoundsException e)
```

```java
    {
        return true;
    }
}

/**
 * A test method that creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values'
 * at the top of this class. Then tests this random array with the testSwapContents method.
 *
 * This is ran a number of times specified by NUM_RANDOM_TESTS
 */
@Test
void testRandomStringsArray()
{
    //Run the test a number of times specified by NUM_RANDOM_TESTS
    for(int x = 0; x < NUM_RANDOM_TESTS; x++)
    {
        //Get a random arrayList of strings as per the values defined under 'testRandomStringsArray values'
        ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
            LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

        //Create two valid indices, making sure they are inequal.
        int i1 = getIndex(false, testData.size());
        int i2;
        do{
            i2 = getIndex(false, testData.size());
        }while(i1 == i2);

        //Run the test method, failing the test if testSwapContents comes back false.
        if(!testSwapContents(testData.toArray(), i1, i2))
        {
            fail("contents not swapped correctly.");
        }
    }
}

/**
 * Creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values' at the top of
 * this class. Then creates a bad index1 value and runs the testBadIndex method with the generated data.
 *
 * This is ran a number of times specified by NUM_RANDOM_TESTS
 */
@Test
void testBadIndex1Strings()
```

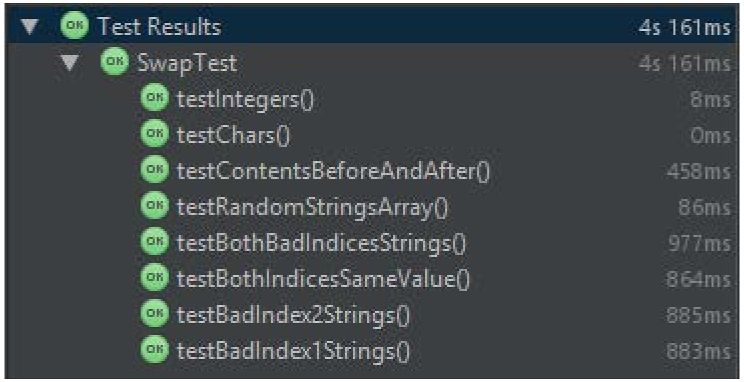
```java
    {
        ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
                LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

        for(int x = 0; x < NUM_RANDOM_TESTS; x++)
        {
            if(!testBadIndex(testData.toArray(), getIndex(true, testData.size()), getIndex(false, testData.size())))
            {
                fail("bad index was not caught.");
            }
        }
    }

    /**
     * Creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values' at the top of
     * this class. Then creates a bad index2 value and runs the testBadIndex method with the generated data.
     *
     * This is ran a number of times specified by NUM_RANDOM_TESTS
     */
    @Test
    void testBadIndex2Strings()
    {
        for(int x = 0; x < NUM_RANDOM_TESTS; x++)
        {
            ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
                LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

            if(!testBadIndex(testData.toArray(), getIndex(false, testData.size()), getIndex(true, testData.size())))
            {
                fail("bad index was not caught.");
            }
        }
    }

    /**
     * Creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values' at the top of
     * this class. Then creates a bad index1 and index2 value and runs the testBadIndex method with the generated data.
     *
     * This is ran a number of times specified by NUM_RANDOM_TESTS
     */
    @Test
    void testBothBadIndicesStrings()
    {
        ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
```

```java
                LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

        for(int x = 0; x < NUM_RANDOM_TESTS; x++)
        {
            if(!testBadIndex(testData.toArray(), getIndex(true, testData.size()), getIndex(true, testData.size())))
            {
                fail("bad index was not caught.");
            }
        }
    }

    /**
     * Creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values' at the top of
     * this class. Then creates one index value and runs the testBadIndex method with the generated data, using this index
     * value for both of the index parameters.
     *
     * This is ran a number of times specified by NUM_RANDOM_TESTS
     */
    @Test
    void testBothIndicesSameValue()
    {
        ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
                LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

        for(int x = 0; x < NUM_RANDOM_TESTS; x++)
        {
            int ind = getIndex(false, testData.size());
            if(!testBadIndex(testData.toArray(), ind, ind))
            {
                fail("Identical indices not caught.");
            }
        }
    }

    /**
     * Creates a random array of strings per the bounds of the values labelled 'RandomStringsArray values' at the top of
     * this class. Then creates 2 valid indices to swap and runs the swap.
     *
     * Then checks that each element in the test data before the swap is contained within the test data after the swap
     * failing if it is not found.
     *
     * This is ran a number of times specified by NUM_RANDOM_TESTS
     */
    @Test
```

```java
    void testContentsBeforeAndAfter()
    {

        for(int x  = 0; x < NUM_RANDOM_TESTS; x++)
        {
            ArrayList<String> testData = RandomStringArray.getRandomStringArray(LOWEST_SIZE_ARRAY, HIGHEST_SIZE_ARRAY,
                    LOWEST_SIZE_STRING, HIGHEST_SIZE_STRING);

            ArrayList<String> testDataBefore = new ArrayList<>(testData);

            int i1 = getIndex(false, testData.size());
            int i2;
            do
            {
                i2 = getIndex(false, testData.size());
            } while (i1 == i2);

            Swap.swap(testData.toArray(), i1, i2);
            for (String s : testDataBefore)
            {
                if (!testData.contains(s))
                {
                    fail("String: " + s + " not present after swap occurred.");
                }
            }
        }
    }

    /**
     * A test method to test an array of integers using random index values to make sure the contents are swapped correctly.
     */
    @Test
    void testIntegers()
    {
        Integer[] ints = {5, 23, 456, 2, 3463, 123, 83456};

        int i1 = getIndex(false, ints.length);
        int i2;
        do
        {
            i2 = getIndex(false, ints.length);
        } while (i1 == i2);

        if(!testSwapContents(ints, i1, i2))
```

```java
        {
            fail("Contents not swapped properly with integers.");
        }
    }

    /**
     * A test method to test an array of chars using random index values to make sure the contents are swapped correctly.
     */
    @Test
    void testChars()
    {
        Character[] chars = {'a', 'b', 'y', '2', 'g', 'A', '='};

        int i1 = getIndex(false, chars.length);
        int i2;
        do
        {
            i2 = getIndex(false, chars.length);
        } while (i1 == i2);

        if(!testSwapContents(chars, i1, i2))
        {
            fail("Contents not swapped properly with characters.");
        }
    }


}
```

Result of testing



(Additional) Question 3: Write a generic method to count the number of elements in an array that have a specific property.

Code Listings.

CountingUnaryPredicate

```java
package unaryPredicate;
/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * A partial implementation of the UnaryPredicateCount interface, implementing the numberSatisfying method.
 */
abstract class CountingUnaryPredicate<T> implements UnaryPredicateCount<T>
{
    /**
     * Returns the number of objects within the array that satisfy the predicate's test method.
     *
     * @param array An array of objects of the type tested by this predicate's test
     * @return the number of objects within param array that satify test(). Return -1 if not all elements in the array
     * are of type T.
     */
    @Override
    public int numberSatisfying(T[] array)
    {
        for (Object a:array)
        {
            try
            {
                T b = (T)a;
            }catch (ClassCastException e)
            {
                return -1;
            }
        }

        int num = 0;
        for (Object a : array)
        {
            if(test((T) a))
            {
                num++;
            }
        }
        return num;
    }
}
```

IsOdd

```java
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * An implementation of UnaryPredicate over integers.
 *
 * the overridden test method returns true if the passed in integer is odd.
 */
public class IsOdd extends CountingUnaryPredicate<Integer>
{

    /**
     * Returns a boolean specifying whether or not the integer n is odd or not.
     *
     * @param n an Integer to be checked.
     * @return true if n is odd, false if otherwise.
     */
    @Override
    public boolean test(Integer n)
    {
        return (Math.abs(n % 2) == 1);
    }


}
```

IsPrime

```java
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * an Implementation of a UnaryPredicate over integers.
 *
 * The test method returns whether or not the passed in Integer is prime or not.
 */
public class IsPrime extends CountingUnaryPredicate<Integer>
{
    /**
     * Returns whether or not the integer passed in is a prime number or not.
     *
     * @param n the integer to be checked for prime-ness.
     * @return true if n is prime, false otherwise.
     */
    @Override
    public boolean test(Integer n)
    {
        n = Math.abs(n);
        if(n == 0 || n == 1 || n == 2)
        {
            return false;
        }

        //Go through each number from 2 to n-1, checking if a modulus comes back as 0. If it does for any value
        //of 'x', then return false, as n is not prime.
        for(int x = 2; x < n; x++)
        {
            if(n % x == 0)
            {
                return false;
            }
        }
        return true;
    }
}
```

IsPalindrome

```java
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * An implementation of UnaryPredicate over Strings.
 *
 * The test method returns whether or not the passed in string is a palindrome or not.
 */
public class IsPalindrome extends CountingUnaryPredicate<String>
{

    /**
     * Returns whether or not the string s is a palindrome or not.
     *
     * @param s the string to be checked for palindrome-ness.
     * @return true if the string s is a palindrome, false otherwise.
     */
    @Override
    public boolean test(String s)
    {
        //Check each pair of characters from the start/end to the middle pair of the string.
        //If any pair are not identical, the string is not a palindrome, so return false.
        String a = s.replaceAll("[^a-zA-Z]", "");
        a = a.toLowerCase();
        int end = a.length() - 1;
        for(int x = 0; x < end/2; x++)
        {
            if (a.charAt(x) != a.charAt(end - x))
            {
                return false;
            }
        }
        return true;
    }
}
```

IsMonotonic

```java
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * An implementation of UnaryPredicate over an array of Comparables.
 *
 * the test method returns whether or not the array is monotonic. (constantly increasing or constantly decreasing.
 */
public class IsMonotonic extends CountingUnaryPredicate<Comparable[]>
{
    /**
     * Returns whether or not the specified array comparables is monotonic.
     *
     * @param comparables an Array of Comparable to be checking for monotonicity.
     * @return true if the array is monotonic, false otherwise.
     */
    @Override
    public boolean test(Comparable[] comparables)
    {
        boolean increasing = comparables[0].compareTo(comparables[1]) < 0;
        for(int x = 1; x < (comparables.length - 1); x++)
        {
            if(increasing && comparables[x].compareTo(comparables[x+1]) > 0)
            {
                return false;
            }
            else if(!increasing && comparables[x].compareTo(comparables[x+1]) < 0)
            {
                return false;
            }
            else if(comparables[x].compareTo(comparables[x+1]) == 0)
            {
                return false;
            }
        }
        return true;
    }
}
```

Test Class code listings.

IsOddTest

```
package unaryPredicate;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.fail;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

/**
 * A selection of tests testing IsOdd and its methods.
 */
class IsOddTest
{
    private IsOdd predicate = new IsOdd();

    @Test
    void testZero() {
        assertFalse(predicate.test(0));
    }

    @Test
    void testOne() {
        assertTrue(predicate.test(1));
    }

    @Test
    void testTwo() {
        assertFalse(predicate.test(2));
    }

    @Test
    void testThree() {
        assertTrue(predicate.test(3));
    }

    @Test
    void testBigEven() {
```

```java
        assertFalse(predicate.test(2*((Integer.MAX_VALUE-1)/2)));
    }

    @Test
    void testBigOdd() {
        assertTrue(predicate.test(2*((Integer.MAX_VALUE-1)/2)-1));
    }

    @Test
    void testMinusOne() {
        assertTrue(predicate.test(-1));
    }

    @Test
    void testMinusTwo() {
        assertFalse(predicate.test(-2));
    }

    @Test
    void testMinusThree() {
        assertTrue(predicate.test(-3));
    }

    @Test
    void testMinusBigEven() {
        assertFalse(predicate.test(2*((Integer.MIN_VALUE+1)/2)));
    }

    @Test
    void testMinusBigOdd() {
        assertTrue(predicate.test(2*((Integer.MIN_VALUE+1)/2)+1));
    }

    @Test
    void testNumberOddInArray()
    {
        Integer[] ints = {1, 2, 3, 4, 5, 6};
        if(predicate.numberSatisfying(ints) != 3)
            fail("Should be 3");
    }
}
```

IsPrimeTest

```java
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.fail;
import org.junit.jupiter.api.Test;

/**
 * A selection of tests testing the IsPrime Class and its methods.
 */
class IsPrimeTest
{
    private IsPrime predicate = new IsPrime();

    @Test
    void testZero()
    {
        assertFalse(predicate.test(0));
    }

    @Test
    void testOne()
    {
        assertFalse(predicate.test(1));
    }

    @Test
    void testTwo()
    {
        assertFalse(predicate.test(2));
    }

    @Test
    void testThree()
    {
        assertTrue(predicate.test(3));
    }
```

```java
    @Test
    void testFour()
    {
        assertFalse(predicate.test(4));
    }

    @Test
    void testFive(){
        assertTrue(predicate.test(5));
    }

    @Test
    void testSix(){
        assertFalse(predicate.test(6));
    }

    @Test
    void testSeven(){
        assertTrue(predicate.test(7));
    }

    @Test
    void testNine(){
        assertFalse(predicate.test(9));
    }

    @Test
    void testEleven(){
        assertTrue(predicate.test(11));
    }

    @Test
    void testTwelve(){
        assertFalse(predicate.test(12));
    }

    @Test
    void testFifteen(){
        assertFalse(predicate.test(15));
    }

    @Test
    void testMinusOne(){
        assertFalse(predicate.test(-1));
```

```java
    }

    @Test
    void testMinusTwo(){
        assertFalse(predicate.test(-2));
    }

    @Test
    void testMinusEleven(){
        assertTrue(predicate.test(-11));
    }

    @Test
    void testNumberPrimeInArray()
    {
        Integer[] ints = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        if(predicate.numberSatisfying(ints) != 3)
            fail("Should be 3");
    }

}
```

IsPalindromeTest

```
package unaryPredicate;

/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.fail;

/**
 * A selection of tests testing the IsPalindrome class and its methods.
 */
class IsPalindromeTest
{
    IsPalindrome predicate = new IsPalindrome();

    @Test
    void testBasicPalindromes(){
        if(!predicate.test("mum"))
            fail("mum");
        if(!predicate.test("naan"))
            fail("naan");
        if(!predicate.test("radar"))
            fail("radar");
        if(!predicate.test("rotator"))
            fail("rotator");
    }

    @Test
    void testBasicNonPalindromes()
    {
        if(predicate.test("hvds"))
            fail("hvds");
        if(predicate.test("palindrome"))
            fail("palindrome");
        if(predicate.test("boi"))
            fail("boi");
        if(predicate.test("yes"))
            fail("yes");
    }

    @Test
```

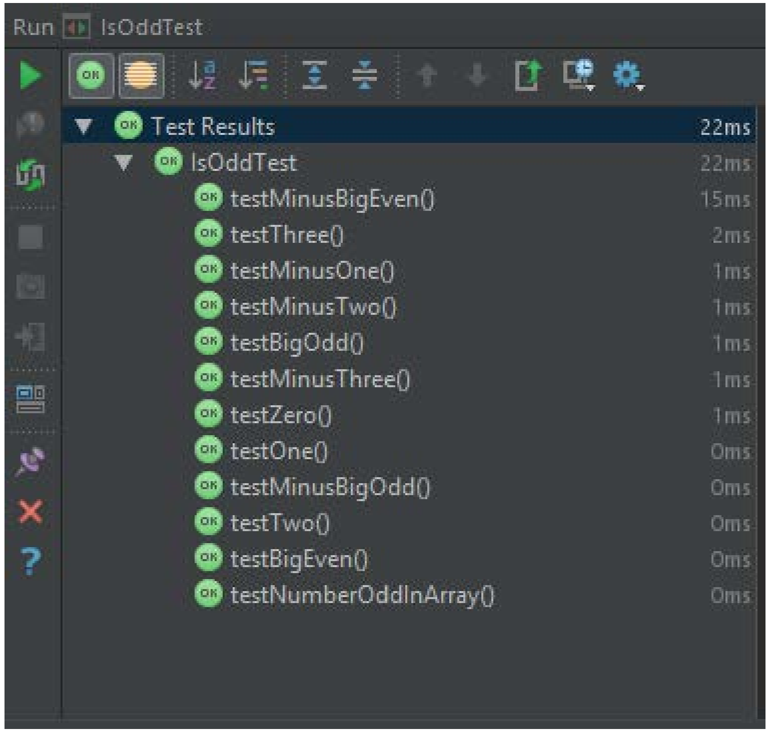
```java
    void testCasePalindromes()
    {
        if(!predicate.test("Malayalam"))
            fail("Malayalam");
        if(!predicate.test("RaCEcAr"))
            fail("RaCEcAr");
    }

    @Test
    void testPunctuation()
    {
        if(!predicate.test("Gods's dog"))
            fail("God's dog");
        if(!predicate.test("Able was I ere I saw Elba"))
            fail("Able was I ere I saw Elba");
        if(!predicate.test("A man, A plan, A canal - Panama!"))
            fail("A man, A plan, A canal, - Panama!");
    }

    @Test
    void testNumberPalindromesInArray()
    {
        String[] strings = {"naan", "Malayalam", "God's Dog", "nope"};
        if(predicate.numberSatisfying(strings) != 3)
            fail("Should be 3");
    }
}
```

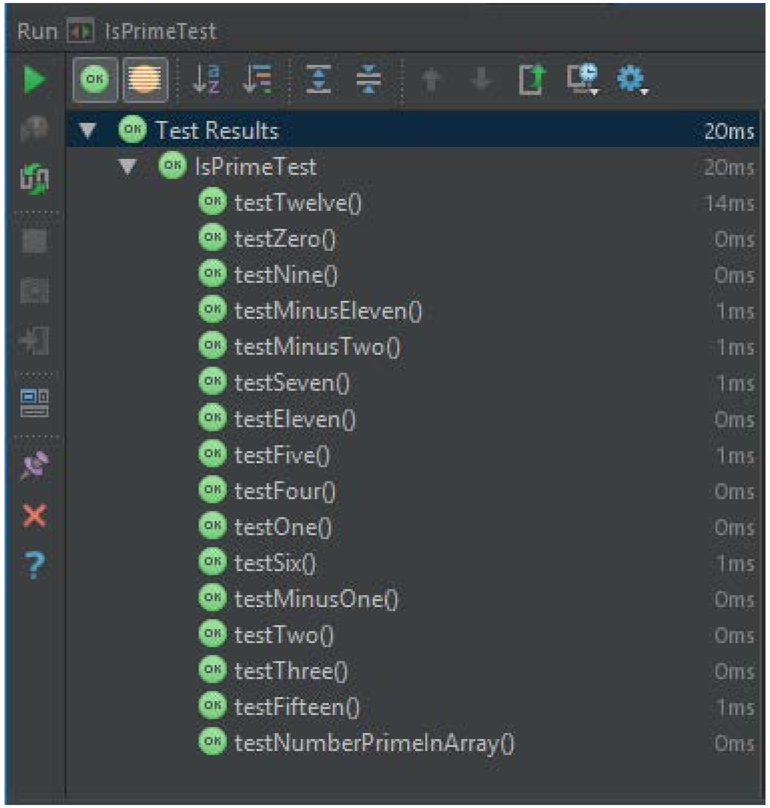
IsMonotonicTest

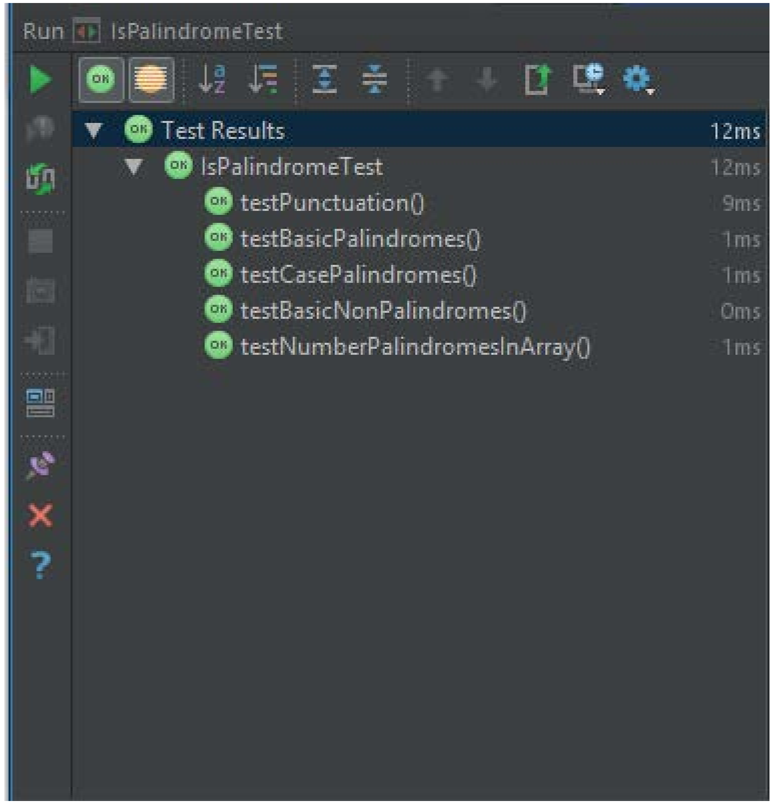
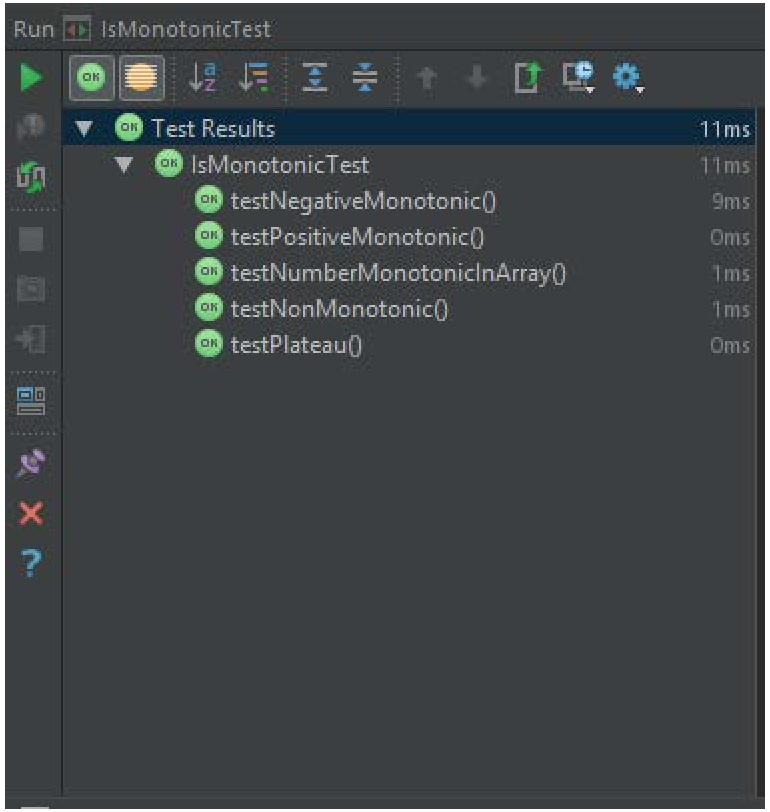
```java
package unaryPredicate;
/**
 * Created by u1661665(Joshua Pritchard) on 11/11/2018.
 */
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.fail;
/**
 * A selection of tests testing IsMonotonic and its methods.
 */
class IsMonotonicTest
{
    IsMonotonic predicate = new IsMonotonic();

    @Test
    void testPositiveMonotonic(){
        Integer[] ints = {1, 2, 3, 4, 5};
        if(!predicate.test(ints))
            fail("Ints array 1, 2, 3, 4, 5");

        String[] strings = {"aaa", "bbb", "ccc", "ddd", "eee"};
        if(!predicate.test(strings))
            fail("String array aaa, bbb, ccc, ddd, eee");

        Character[] chars = {'a', 'b', 'c', 'd', 'e'};
        if(!predicate.test(chars))
            fail("Chars array a, b, c, d, e");
    }

    @Test
    void testNegativeMonotonic(){
        Integer[] ints = {5, 4, 3, 2, 1};
        if(!predicate.test(ints))
            fail("Ints array 5, 4, 3, 2, 1");

        String[] strings = {"eee", "ddd", "ccc", "bbb", "aaa"};
        if(!predicate.test(strings))
            fail("String array eee, ddd, ccc, bbb, aaa");

        Character[] chars = {'e', 'd', 'c', 'b', 'a'};
        if(!predicate.test(chars))
            fail("Chars array e, d, c, b, a");
    }
```

```java
    @Test
    void testNonMonotonic(){
        Integer[] ints = {5, 4, 5, 2, 1};
        if(predicate.test(ints))
            fail("Ints array 5, 4, 5, 2, 1");

        String[] strings = {"eee", "ddd", "eee", "bbb", "aaa"};
        if(predicate.test(strings))
            fail("String array eee, ddd, eee, bbb, aaa");

        Character[] chars = {'a', 'b', 'a', 'd', 'e'};
        if(predicate.test(chars))
            fail("Chars array a, b, a, d, e");
    }

    @Test
    void testPlateau(){
        Integer[] ints = {5, 4, 4, 2, 1};
        if(predicate.test(ints))
            fail("Ints array 5, 4, 4, 2, 1");

        String[] strings = {"eee", "ddd", "ddd", "bbb", "aaa"};
        if(predicate.test(strings))
            fail("String array eee, ddd, ddd, bbb, aaa");

        Character[] chars = {'a', 'b', 'b', 'd', 'e'};
        if(predicate.test(chars))
            fail("Chars array a, b, b, d, e");
    }

    @Test
    void testNumberMonotonicInArray()
    {
        Integer[][] ints = new Integer[3][5];
        ints[0] = new Integer[]{1, 2, 3, 4, 5};
        ints[1] = new Integer[]{1, 1, 1, 1, 1};
        ints[2] = new Integer[]{1, 2, 3, 4, 5};

        if(predicate.numberSatisfying(ints) != 2)
            fail("Should be 2");
    }
}
```

Test Results.






Run IsPalindromeTest
Test Results 12ms
IsPalindromeTest 12ms
testPunctuation() 9ms
testBasicPalindromes() 1ms
testCasePalindromes() 1ms
testBasicNonPalindromes() 0ms
testNumberPalindromesInArray() 1ms

Run IsMonotonicTest
Test Results 11ms
IsMonotonicTest 11ms
testNegativeMonotonic() 9ms
testPositiveMonotonic() 0ms
testNumberMonotonicInArray() 1ms
testNonMonotonic() 1ms
testPlateau() 0ms

Practical 3

(Additional) Question 1

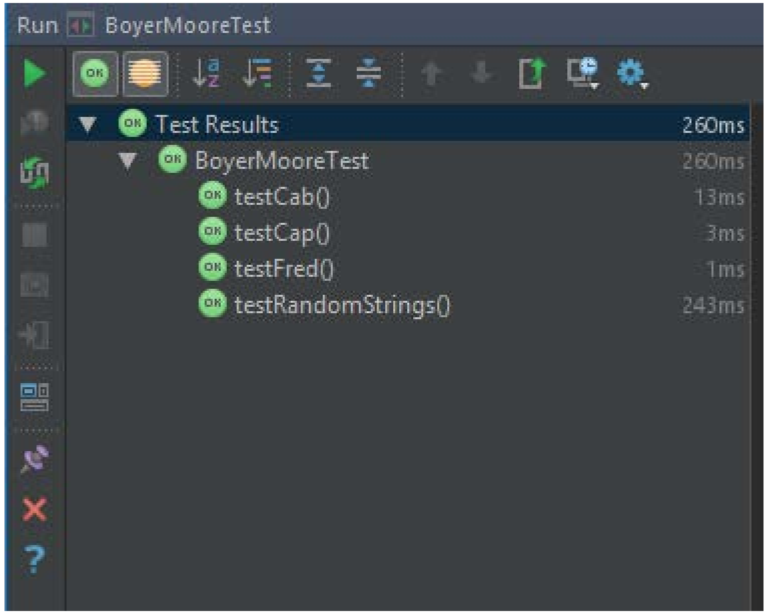
1.1 – Implement the Boyer Moore algorithm. Your implementation should implement the StringSearch interface.

Code listing.

```
package stringSearcher;
/**
 * A class defining an implementation of a String searcher, using the BoyerMoore string searching algorithm.
 *
 * @author Joshua Pritchard.
 * @version November 2018
 */
public class BoyerMoore extends StringSearcher
{
    /**
     * Creates a BoyerMoore instance
     * @param string the string to be used as the substring that this BoyerMoore searches for.
     */
    public BoyerMoore(char[] string)
    {
        super(string);
    }

    /**
     * Creates a BoyerMoore instance.
     * @param string the string to be used as the substring that this BoyerMoore searches for.
     */
    public BoyerMoore(String string)
    {
        super(string);
    }

    /**
     * Find the first occurrence of the substring member variable 'string' in param superstring.
     * Implementation is the BoyerMoore algorithm.
     *
     * @param superstring the superstring to be searched
     * @return the index value of the first occurrence of the substring 'string'
     * @throws NotFound if the substring is not found within param superstring.
     */
    @Override
    public int occursIn(char[] superstring) throws NotFound
    {
        int i = getString().length - 1;
        int j = getString().length - 1;

        do
```

```java
        {
            if(getString()[j] == superstring[i])
            {
                if(j == 0)
                {
                    return i;
                }
                else
                {
                    i--;
                    j--;
                }
            }
            else
            {
                i = i + getString().length - Math.min(j, 1 + last(superstring[i]));
                j = getString().length - 1;
            }
        } while(i < superstring.length);

        throw new NotFound();
    }

    /**
     * Return the last occurrence's index of a given character in the subString.
     *
     * @param c The character to be searched for in the substring.
     * @return the index value of the rightmost occurrence of c in subString.\nReturn -1 if c does not occur.
     */
    private int last(char c)
    {
        for (int x = getString().length - 1; x > -1; x--)
        {
            if (c == getString()[x])
                return x;
        }

        return -1;
    }
}
```

Result of testing.



1.2 – Use your algorithm, and the implementation of the sequential substring search algorithm provided to perform an empirical comparison of the efficiency of the two algorithms.

Test data acquired.

SubString Size Timer.

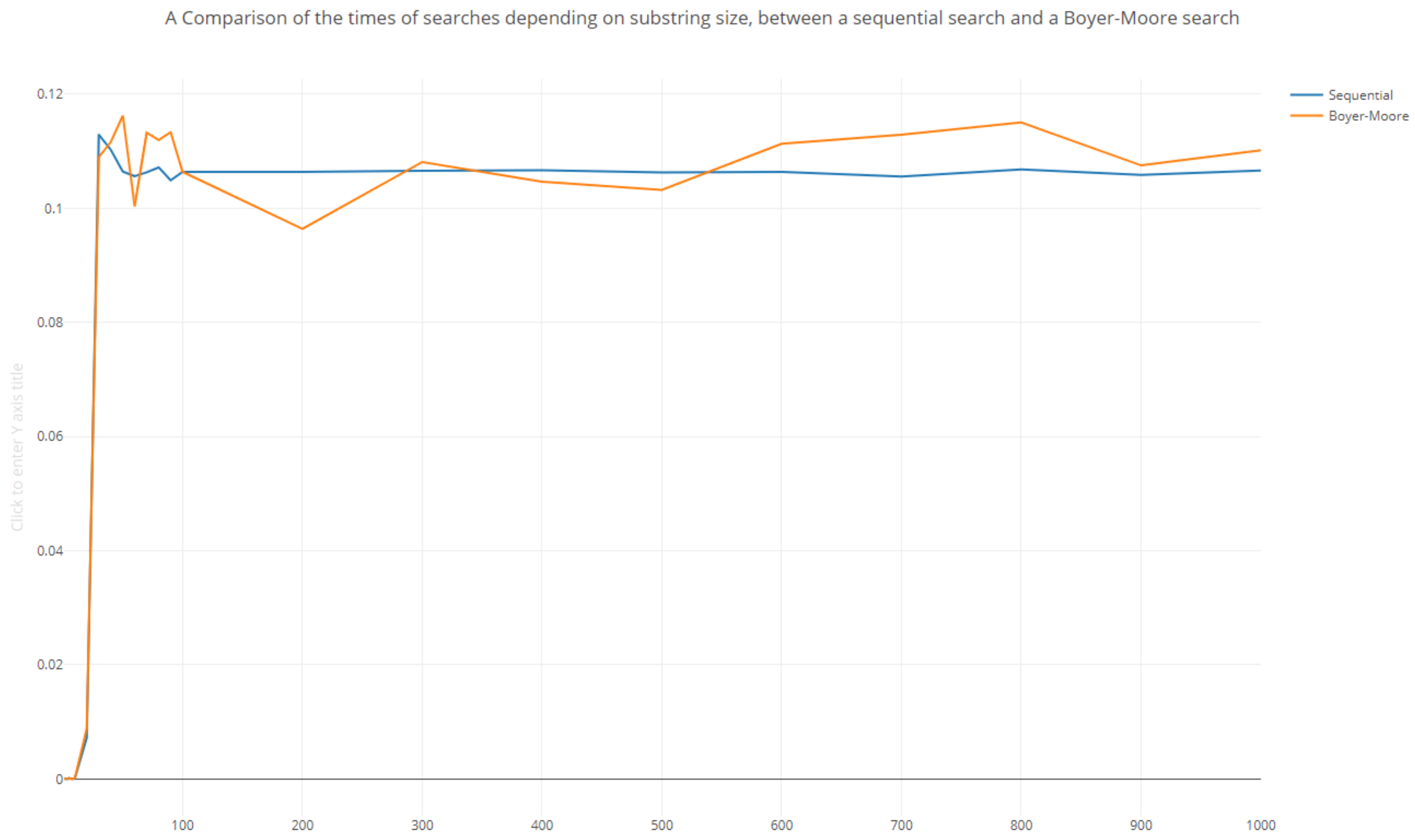
Sequential searcher substring size timer.		Boyer Moore substring size timer.	
Time	Size	Time	Size
0.000001026	1	0.000003392	1
0.000001111	2	0.000001254	2
0.000002052	3	0.000001169	3
0.000003905	4	0.000002794	4
0.000147112	5	0.000153043	5
0.000003392	6	0.000009836	6
0.000011233	7	0.000017847	7
0.000014027	8	0.000033214	8
0.000032016	9	0.000029821	9
0.000042708	10	0.000169465	10
0.007215953	20	0.008846801	20
0.112930127	30	0.108926088	30
0.110261165	40	0.111591288	40
0.106433602	50	0.11620565	50
0.105636483	60	0.100346799	60
0.106306675	70	0.113250415	70
0.107157365	80	0.111953853	80
0.104922415	90	0.113330672	90
0.106401329	100	0.106394374	100
0.106355627	200	0.096370668	200
0.10656167	300	0.108108783	300
0.106673659	400	0.104661433	400
0.10628438	500	0.103220893	500
0.10637892	600	0.111295864	600
0.105555599	700	0.112872141	700
0.106823623	800	0.115055402	800
0.1058417	900	0.107511036	900
0.106617066	1000	0.110142594	1000

SuperString Size Timer.

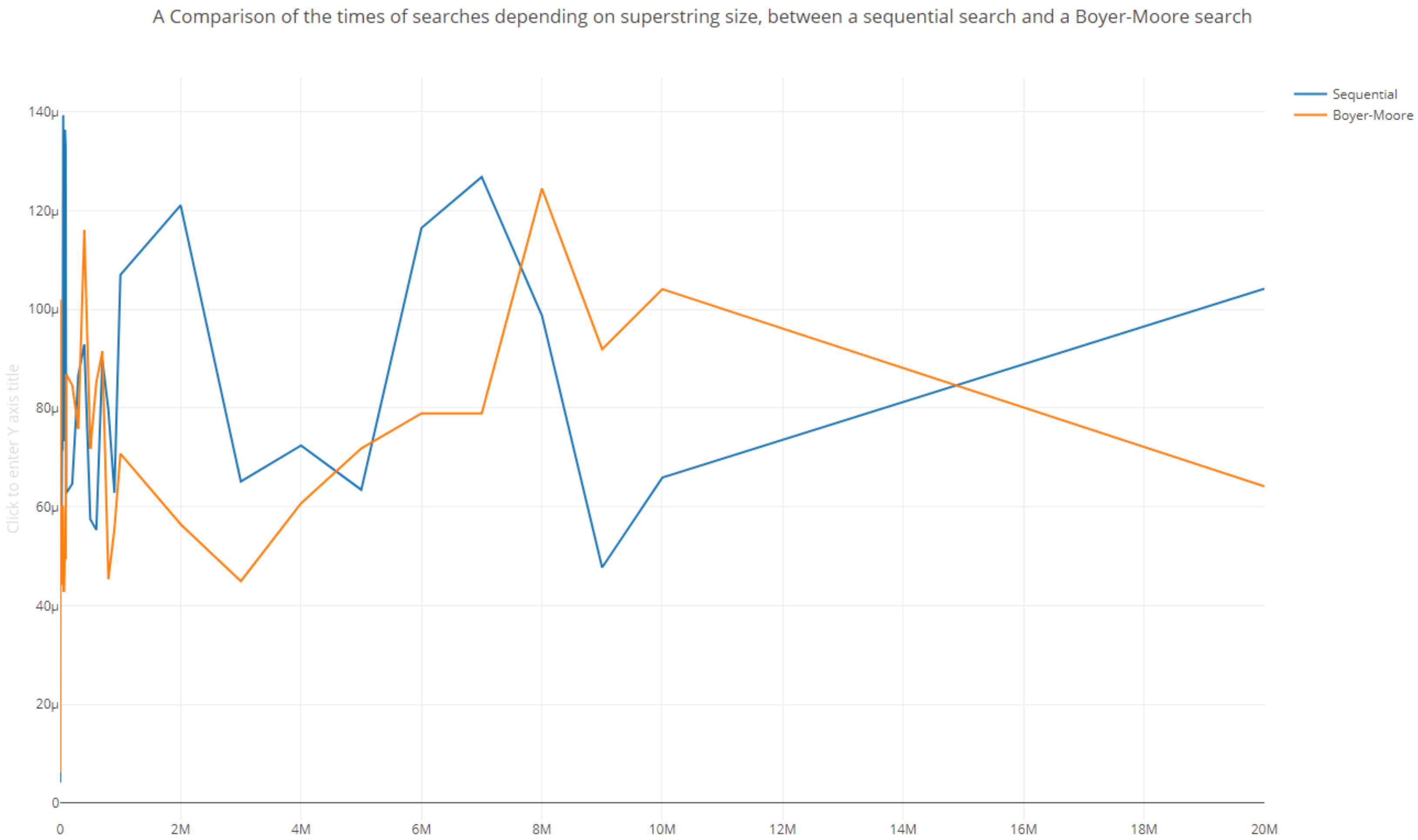
Sequential searcher superstring size timer		Boyer Moore superstring size timer.	
Time	Size	Time	Size
0.000005245	10	0.000006756	10
0.000004162	20	0.000006443	20
0.000006215	30	0.000006215	30
0.000007526	40	0.000006671	40
0.000007811	50	0.000011404	50
0.000006785	60	0.000014084	60
0.000007612	70	0.000007897	70
0.000007697	80	0.00000764	80
0.000008695	90	0.000007954	90
0.00000707	100	0.000008952	100
0.000011233	200	0.00001588	200
0.000010292	300	0.00000975	300
0.000010691	400	0.000010035	400
0.000010748	500	0.000012173	500
0.000020071	600	0.000011774	600
0.000013314	700	0.000014654	700
0.000012401	800	0.000027626	800
0.00001625	900	0.000032473	900
0.000017591	1000	0.000031161	1000
0.000056478	2000	0.00005836	2000
0.000069308	3000	0.000094569	3000
0.000030933	4000	0.000101867	4000
0.000023634	5000	0.000023806	5000
0.00004325	6000	0.000029764	6000
0.000040627	7000	0.000027712	7000
0.000046728	8000	0.000029051	8000
0.000034012	9000	0.000037377	9000
0.000046585	10000	0.000044219	10000
0.000048581	20000	0.000054255	20000
0.000073499	30000	0.000047811	30000
0.000071332	40000	0.00006027	40000
0.000139329	50000	0.00005722	50000
0.000088695	60000	0.000042794	60000
0.000073299	70000	0.000051033	70000
0.000136364	80000	0.000051831	80000
0.000132829	90000	0.000049408	90000
0.000062779	100000	0.00008687	100000
0.000064689	200000	0.00008459	200000
0.000086614	300000	0.000075837	300000
0.000092915	400000	0.000116008	400000
0.000057505	500000	0.000071732	500000
0.000055338	600000	0.000085245	600000
0.000089835	700000	0.00009146	700000
0.000079686	800000	0.000045416	800000
0.000062865	900000	0.000055538	900000
0.00010697	1000000	0.000070762	1000000
0.000121054	2000000	0.000056479	2000000
0.000065145	3000000	0.00004496	3000000
0.000072444	4000000	0.000060726	4000000
0.000063463	5000000	0.000071845	5000000
0.000116464	6000000	0.000078945	6000000
0.000126813	7000000	0.000078916	7000000
0.000098731	8000000	0.000124447	8000000
0.000047726	9000000	0.000091917	9000000
0.000065915	10000000	0.000104091	10000000
0.000104205	20000000	0.000064148	20000000

Graphing results

SubString Size Timer



SuperString Size Timer.



Empirical Comparison.

As can be evidently seen from both graphs, there is no significant difference between either algorithm. I place this unexpected result down to the fact that I was unable to implement the lookup tables for character shifts for the Boyer-Moore method.

When small SubStrings are concerned, both implementations rapidly increase in time taken to their averages. Both implementations however, are fairly ignorant to increasing values of SubString size, showing very comparable times for searches of small and large SubStrings.

Moving onto the changing size of the SuperString, the same trends hold true. The data recorded shows a wild unpredictability with linearity within bounds in the time taken by both implementations throughout all the test data.

1.3 – Express the efficiency of the two algorithms, expressed in terms of the sizes of both the SuperString and the SubString. Justify your answers.

Least Squares method for line of best fit data.

For SubStrings

Mean of X	Mean of Y		x - Xmean	y - Ymean		x- Xmean all squared	(x-Xmean)(y-Ymean)	
214.1071429	0.065164855	0.066446664	-213.1071429	-0.065163829	-0.066443272	45414.65434	13.88687742	14.15954
			-212.1071429	-0.065163744	-0.06644541	44989.44005	13.82169556	14.09355
			-211.1071429	-0.065162803	-0.066445495	44566.22577	13.75633316	14.02712
Sum of (x-X)(y-Y) Seq	Sum of x-Xmean^2		-210.1071429	-0.06516095	-0.06644387	44145.01148	13.69078103	13.96033
239.1391722	2595212.679		-209.1071429	-0.065017743	-0.066293621	43725.79719	13.59567447	13.86247
			-208.1071429	-0.065161463	-0.066436828	43308.58291	13.56056589	13.82598
Sum of (x-X)(y-Y)Boy			-207.1071429	-0.065153622	-0.066428817	42893.36862	13.4937805	13.75788
249.0345975			-206.1071429	-0.065150828	-0.06641345	42480.15434	13.42805101	13.68829
			-205.1071429	-0.065132839	-0.066416843	42068.94005	13.35921051	13.62257
Gradient Seq	Gradient Boy		-204.1071429	-0.065122147	-0.066277199	41659.72577	13.29189536	13.52765
9.21463E-05	9.59592E-05		-194.1071429	-0.057948902	-0.057599863	37677.58291	11.2482958	11.18054
			-184.1071429	0.047765272	0.042479424	33895.44005	-8.793927756	-7.82077
y intercept seq	y intercept Boy		-174.1071429	0.04509631	0.045144624	30313.29719	-7.851589688	-7.86
0.04543568	0.045901108		-164.1071429	0.041268747	0.049758986	26931.15434	-6.772496159	-8.16581
			-154.1071429	0.040471628	0.033900135	23749.01148	-6.236966958	-5.22425
			-144.1071429	0.04114182	0.046803751	20766.86862	-5.928830132	-6.74475
			-134.1071429	0.04199251	0.045507189	17984.72577	-5.631495538	-6.10284
0.989860195			-124.1071429	0.03975756	0.046884008	15402.58291	-4.934197179	-5.81864
			-114.1071429	0.041236474	0.03994771	13020.44005	-4.70537623	-4.55832
			-14.10714286	0.041190772	0.029924004	199.0114796	-0.581084105	-0.42214
			85.89285714	0.041396815	0.041662119	7377.582908	3.555690717	3.578478
			185.8928571	0.041508804	0.038214769	34556.15434	7.716190172	7.103853
			285.8928571	0.041119525	0.036774229	81734.72577	11.75577849	10.51349
			385.8928571	0.041214065	0.0448492	148913.2972	15.9042133	17.30699
			485.8928571	0.040390744	0.046425477	236091.8686	19.625574	22.55781
			585.8928571	0.041658768	0.048608738	343270.4401	24.40757461	28.47951
			685.8928571	0.040676845	0.041064372	470449.0115	27.89995744	28.16576
			785.8928571	0.041452211	0.04369593	617627.5829	32.57699654	34.34032

For SuperStrings

Mean of X	Mean of Y		x - Xmean	y - Ymean		x- Xmean all squared	(x-Xmean)(y-Ymean)	
1428570.536	0.000053358	4.76932E-05	-1428560.536	-0.000048113	-4.09372E-05	2.04079E+12	68.73233305	58.48131
			-1428550.536	-0.000049196	-4.12502E-05	2.04076E+12	70.27897216	58.92804
			-1428540.536	-0.000047143	-4.14782E-05	2.04073E+12	67.34568648	59.25334
Sum of (x-X)(y-Y) Seq	Sum of x-Xmean^2		-1428530.536	-0.000045832	-4.10222E-05	2.0407E+12	65.47241151	58.60151
2808.254344	6.73593E+14		-1428520.536	-0.000045547	-3.62892E-05	2.04067E+12	65.06482484	51.83991
			-1428510.536	-0.000046573	-3.36092E-05	2.04064E+12	66.53002118	48.01114
Sum of (x-X)(y-Y)Boy			-1428500.536	-0.000045746	-3.97962E-05	2.04061E+12	65.34818551	56.84894
2635.120828			-1428490.536	-0.000045661	-4.00532E-05	2.04059E+12	65.22630635	57.21566
			-1428480.536	-0.000044663	-3.97392E-05	2.04056E+12	63.80022617	56.76672
Gradient Seq	Gradient Boy		-1428470.536	-0.000046288	-3.87412E-05	2.04053E+12	66.12104416	55.34071
4.16907E-12	3.91204E-12		-1428370.536	-0.000042125	-3.18132E-05	2.04024E+12	60.17010882	45.44108
			-1428270.536	-0.000043066	-3.79432E-05	2.03996E+12	61.50989889	54.1932
y intercept seq	y intercept Boy		-1428170.536	-0.000042667	-3.76582E-05	2.03967E+12	60.93575225	53.78238
4.74022E-05	4.21046E-05		-1428070.536	-0.00004261	-3.55202E-05	2.03939E+12	60.85008553	50.7254
			-1427970.536	-0.000033287	-3.59192E-05	2.0391E+12	47.53285522	51.29161
			-1427870.536	-0.000040044	-3.30392E-05	2.03881E+12	57.17764773	47.17575
			-1427770.536	-0.000040957	-2.00672E-05	2.03853E+12	58.47719783	28.6514
			-1427670.536	-0.000037108	-1.52202E-05	2.03824E+12	52.97799824	21.72948
			-1427570.536	-0.000035767	-1.65322E-05	2.03796E+12	51.05991535	23.60093
			-1426570.536	3.12E-06	1.06668E-05	2.0351E+12	-4.450900071	-15.2169
			-1425570.536	0.00001595	4.68758E-05	2.03225E+12	-22.73785004	-66.8247
			-1424570.536	-0.000022425	5.41738E-05	2.0294E+12	31.94599426	-77.1744
			-1423570.536	-0.000029724	-2.38872E-05	2.02655E+12	42.3142106	34.00516
			-1422570.536	-0.000010108	-1.79292E-05	2.02371E+12	14.37934298	25.5056
			-1421570.536	-0.000012731	-1.99812E-05	2.02086E+12	18.09801449	28.40473
			-1420570.536	-6.63E-06	-1.86422E-05	2.01802E+12	9.418382652	26.48261
			-1419570.536	-0.000019346	-1.03162E-05	2.01518E+12	27.46301158	14.64462
			-1418570.536	-6.773E-06	-3.47423E-06	2.01234E+12	9.607978238	4.928443
			-1408570.536	-4.777E-06	6.56177E-06	1.98407E+12	6.728741449	-9.24271
			-1398570.536	0.000020141	1.17768E-07	1.956E+12	-28.16860916	-0.16471
			-1388570.536	0.000017974	1.25768E-05	1.92813E+12	-24.95816681	-17.4637
			-1378570.536	0.000085971	9.52677E-06	1.90046E+12	-118.5170875	-13.1333
			-1368570.536	0.000035337	-4.89923E-06	1.87299E+12	-48.36117702	6.704945
			-1358570.536	0.000019941	3.33977E-06	1.84571E+12	-27.09125505	-4.53731
			-1348570.536	0.000083006	4.13777E-06	1.81864E+12	-111.9394459	-5.58007
			-1338570.536	0.000079471	1.71477E-06	1.79177E+12	-106.377539	-2.29534
			-1328570.536	9.421E-06	3.91768E-05	1.7651E+12	-12.51646302	-52.0491
			-1228570.536	0.000011331	3.68968E-05	1.50939E+12	-13.92093274	-45.3303
			-1128570.536	0.000033256	2.81438E-05	1.27367E+12	-37.53174174	-31.7622
			-1028570.536	0.000039557	6.83148E-05	1.05796E+12	-40.68716468	-70.2666
			-928570.5357	4.147E-06	2.40388E-05	8.62243E+11	-3.850782012	-22.3217
			-828570.5357	1.98E-06	3.75518E-05	6.86529E+11	-1.640569661	-31.1143
			-728570.5357	0.000036477	4.37668E-05	5.30815E+11	-26.57606743	-31.8872
			-628570.5357	0.000026328	-2.27723E-06	3.95101E+11	-16.54900506	1.431401
			-528570.5357	9.507E-06	7.84477E-06	2.79387E+11	-5.025120083	-4.14651
			-428570.5357	0.000053612	2.30688E-05	1.83673E+11	-22.97652356	-9.88659
			571429.4643	0.000067696	8.78577E-06	3.26532E+11	38.68348901	5.020447
			1571429.464	0.000011787	-2.73323E-06	2.46939E+12	18.5224391	-4.29508
			2571429.464	0.000019086	1.30328E-05	6.61225E+12	49.07830276	33.51284
			3571429.464	0.000010105	2.41518E-05	1.27551E+13	36.08929474	86.25634
			4571429.464	0.000063106	3.12518E-05	2.0898E+13	288.4846278	142.8653
			5571429.464	0.000073455	3.12228E-05	3.10408E+13	409.2493513	173.9554
			6571429.464	0.000045373	7.67538E-05	4.31837E+13	298.1654691	504.382
			7571429.464	-5.632E-06	4.42238E-05	5.73265E+13	-42.64229074	334.8371
			8571429.464	0.000012557	5.63978E-05	7.34694E+13	107.6314398	483.4095
			18571429.46	0.000050847	1.64548E-05	3.44898E+14	944.301474	305.5886

Derived equations.

Time taken in terms of SubString size:

Sequential searcher : $t = 9.2E^{-5}s + 0.045$

BoyerMoore : $t = 9.6E^{-5} + 0.046$

Time taken in terms of SuperString size:

Sequential searcher : $t = 4.2E^{-12}s + 4.7E^{-5}$

BoyerMoore : $t = 3.9E^{-12}s + 4.2E^{-5}$

As can be seen from the gradients of both SuperString size equations, the size of the SuperString and SubString has negligible impact on either of the two equations.

It is safe to assume that the SuperString size has essentially zero impact on the time taken, especially considering the wild variation found in the values.

1.4 – Does the size of the alphabet the strings are generated from affect the efficiency of the algorithm?

StringSearcherTest Code Listing

Modified to include a method to generate and test random strings.

```
package stringSearcher;

...

abstract class StringSearcherTest {

    abstract StringSearcher getSearcher(String string);

    int SIZE_OF_STRINGS = 20;
    Random rand = new Random();
    CharacterArrayGenerator stringGenerator = new CharacterArrayGenerator();

    Character[] stringGen()
    {
        return stringGenerator.getArray(SIZE_OF_STRINGS);
    }

    ...
    /**
     * Creates a random string using the CharacterArrayGenerator,
     * finds a substring within it,
     * then challenges the Searcher to find the substring.
     *
     * @throws NotFound if the calculated substring is not found within the generated string (impossible)
     */
    @Test
    void testRandomStrings() throws NotFound
    {
        String string = stringGen().toString();
        int index = rand.nextInt(14);
        String substring = string.substring(index, index + 5);

        assertEquals(index, test(substring, string));
    }
}
```

CharacterScope Code Listing

Modified to include and use alphabets of single, double and triple size.

```
package scope;

import java.util.HashSet;
import java.util.Set;

/**
 * An implementation of an {@link scope.AlphabetScope} for characters.
 *
 * @author Hugh Osborne
 * @version October 2018
 */

public class CharacterScope extends AlphabetScope<Character> {

    // Default alphabet.  This uses the standard lower case English alphabet.
    // It would be better to redefine this to take its character set from the {@link Locale}.
    private static final String _1_DEFAULT_ALPHABET = "abcdefghijklmnopqrstuvwxyz";
    private static final String _2_DEFAULT_ALPHABET = _1_DEFAULT_ALPHABET + "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private static final String _3_DEFAULT_ALPHABET = _2_DEFAULT_ALPHABET + "1234567890!£$%^&*()-=_+#~@";

...

    /**
     * If no scope is specified, use the default alphabet.
     */
    public CharacterScope() {
        super(toCharacterSet(_1_DEFAULT_ALPHABET));
...
```

Test Data

1000 sequential tests of random strings of size 1000 with a randomly chosen substring of random length.

Single Size Alphabet	1	**Double Size Alphabet**	2	**Triple Size Alphabet**	3
Sequential (ns)	**Boyer-Moore (ns)**	**Sequential (ns)**	**Boyer-Moore (ns)**	**Sequential (ns)**	**Boyer-Moore (ns)**
4525439	7149242	4591867	9148098	3803273	8600130
4220949	12581593	4155946	12151657	4813963	8600415
4940264	7387588	4168775	8632917	5228503	7043754
3947820	6889227	5134989	7758791	4786593	8370053
3726010	8474685	3914749	9103337	5089943	7605976
4426508	7648742	5584881	8166773	6099777	8110324
4078112	8479531	3812397	9541541	5112752	8283667
3597429	11526427	4242902	8622083	3812967	7665848
5725437	7780745	5071411	8922582	4210971	6941971
4251740	7494501	5136129	9427499	4096074	10385730
Average	**←**	**Average**	**←**	**Average**	**←**
4343970.8	**8541228.1**	**4581404.6**	**9147527.8**	**4705481.6**	**8160786.8**

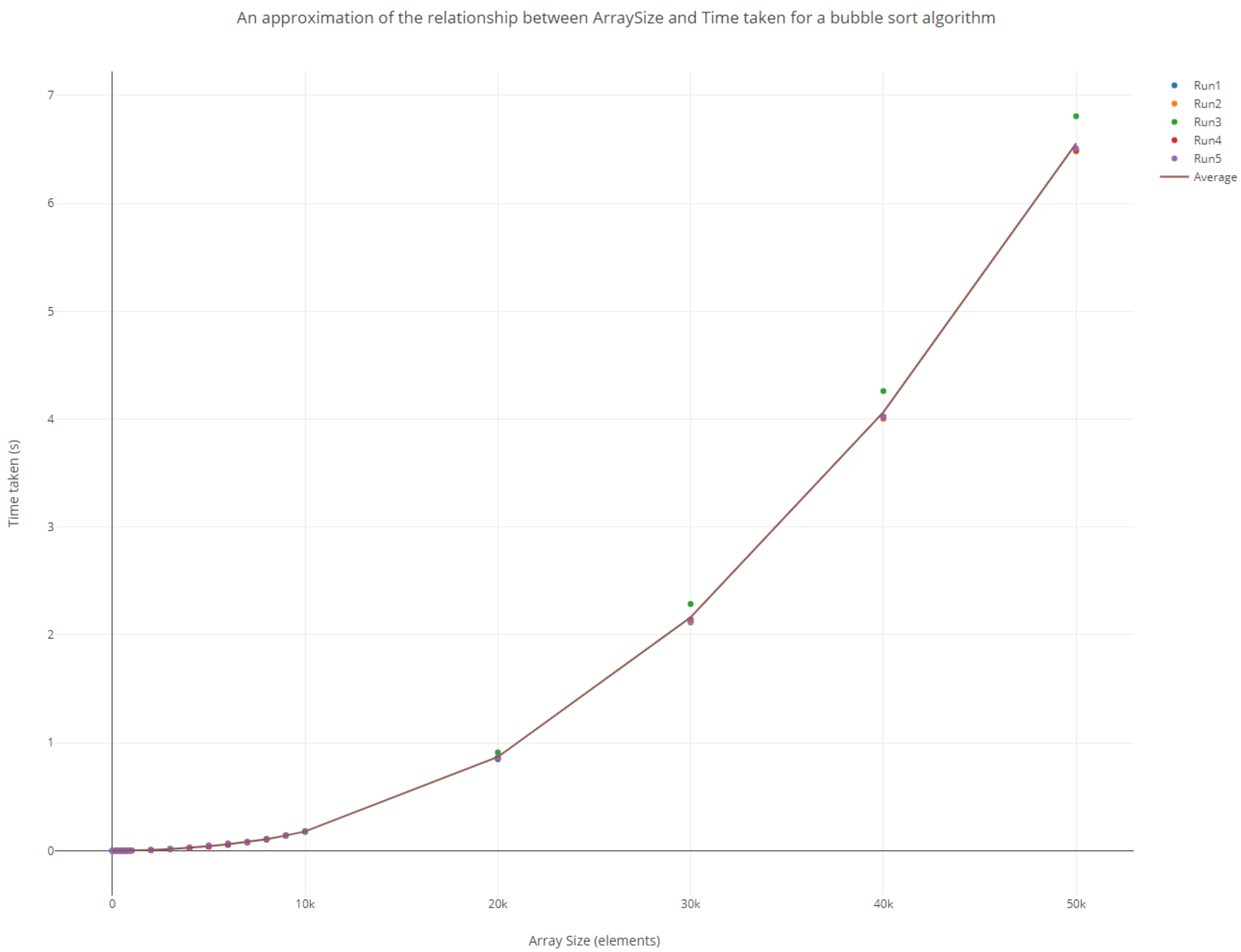
As can be seen from the test data, the Sequential searcher takes slightly longer each time the alphabet gains another 26 characters (26, 52, 78). The partial Boyer-Moore however stays relatively the same. Despite my implementation of Boyer-Moore being ineffective due to its partial implementation, the linearity seems to hold true where the alphabet size is concerned, whereas the Sequential searcher shows this gradual increase.

(Model) Sorting Question 1: Run and time the bubble sort algorithm a number of times for arrays of different sizes. Plot the timing results on a graph, try to arrive at an (approximate) formula relating the time taken to the size of the array.

Test data acquired

Size	Run1	Run2	Run3	Run4	Run5	Average
1	0.00000037	0.000000399	0.000000285	0.000000541	0.00000037	0.000000393
2	0.000000969	0.000001026	0.00000094	0.000001111	0.000001083	1.0258E-06
3	0.000000855	0.000000798	0.000000741	0.000000855	0.000000826	0.000000815
4	0.000000798	0.000000855	0.000000798	0.000000712	0.000000712	0.000000775
5	0.000001425	0.00000134	0.000002195	0.000001254	0.000001368	1.5164E-06
6	0.00000191	0.000008752	0.00000191	0.000001938	0.000003706	3.6432E-06
7	0.00000228	0.00000191	0.000001653	0.000002622	0.000001682	2.0294E-06
8	0.000002081	0.000001938	0.000001967	0.000001995	0.00000191	1.9782E-06
9	0.000002081	0.000002138	0.000002081	0.000002195	0.000002052	2.1094E-06
10	0.000002765	0.000002822	0.000002794	0.00000285	0.000002822	2.8106E-06
20	0.000010206	0.000009921	0.000010092	0.000010833	0.000010206	1.02516E-05
30	0.000029764	0.000023207	0.000023235	0.000023407	0.000022922	0.000024507
40	0.000048781	0.000040456	0.000038118	0.000038802	0.000041739	4.15792E-05
50	0.000019643	0.000015338	0.00001491	0.000037975	0.000023948	2.23628E-05
60	0.000020784	0.000023834	0.000020869	0.000021468	0.000022494	2.18898E-05
70	0.000029793	0.000035552	0.000028481	0.00004325	0.000033756	3.41664E-05
80	0.000035067	0.000039829	0.000037604	0.00007042	0.000036521	4.38882E-05
90	0.00004593	0.000047811	0.000045645	0.00008593	0.000044419	0.000053947
100	0.000058218	0.00011866	0.000104518	0.000131831	0.000064405	9.55264E-05
200	0.000658302	0.000805643	0.000724931	0.00066218	0.000397547	0.000649721
300	0.000537675	0.000364161	0.000365587	0.000356321	0.000771288	0.000479006
400	0.000266172	0.000260384	0.000254938	0.000256193	0.000284247	0.000264387
500	0.000408352	0.000407697	0.000408324	0.000380583	0.000452515	0.000411494
600	0.000577333	0.00059404	0.000622436	0.000575822	0.000735508	0.000621028
700	0.000777247	0.000738901	0.00078657	0.000760882	0.000830904	0.000778901
800	0.001040141	0.000977418	0.000954353	0.001008038	0.001142293	0.001024449
900	0.001314809	0.001162735	0.001213654	0.001209976	0.00147612	0.001275459
1000	0.001628679	0.001440169	0.00151612	0.001483504	0.001743632	0.001562421
2000	0.006278479	0.005856356	0.005865679	0.00568672	0.007164095	0.006170266
3000	0.01537409	0.013583187	0.014567278	0.013551684	0.016334174	0.014682083
4000	0.027623852	0.024749187	0.023602988	0.02338993	0.029674512	0.025808094
5000	0.039363992	0.039491034	0.039971774	0.037951449	0.04875457	0.041106564
6000	0.054535545	0.0557029	0.056058793	0.056643939	0.068625903	0.058313416
7000	0.077976909	0.078922823	0.07771413	0.077026832	0.082218671	0.078771873
8000	0.102611571	0.107694871	0.105835487	0.104766721	0.108490992	0.105879928
9000	0.139072389	0.141089321	0.138921028	0.141456106	0.141719056	0.14045158
10000	0.175247561	0.178879631	0.176470796	0.180256763	0.179751104	0.178121171
20000	0.847069099	0.860740954	0.910497575	0.861443504	0.863262946	0.868602816
30000	2.117008899	2.126667303	2.286498265	2.139520742	2.142731685	2.162485379
40000	4.003887863	4.009231319	4.259782666	4.019337164	4.024785709	4.063404944
50000	6.489019317	6.48139977	6.807469041	6.492450075	6.509091562	6.555885953

Graph



An approximation of the relationship between ArraySize and Time taken for a bubble sort algorithm
Time taken (s)
Array Size (elements)
0
1
2
3
4
5
6
7
0
10k
20k
30k
40k
50k
Run1
Run2
Run3
Run4
Run5
Average

Approximation of formula.

It is obvious from the shape of the graph that the graph's highest power will be x^2.

An array of size 0 would take 0 seconds to solve, which lies true with the graph's y intercept of 0, therefore there is no additional constant added to the line. There is 1 unknown about this formula:

The coefficient of x^2.

As I have a set of x and y co-ordinates, with the x^2 part of the function being known, I can rearrange y = mx^2 and solve for an approximation of m.

As can be observed from the data on the left, a good approximation of the function's x^2 coefficient would be $3x10^{-8}$.

However where Big O notation is concerned, the function falls under the $O(n^2)$ classification, as the time taken is directly proportional to the size of the array squared.

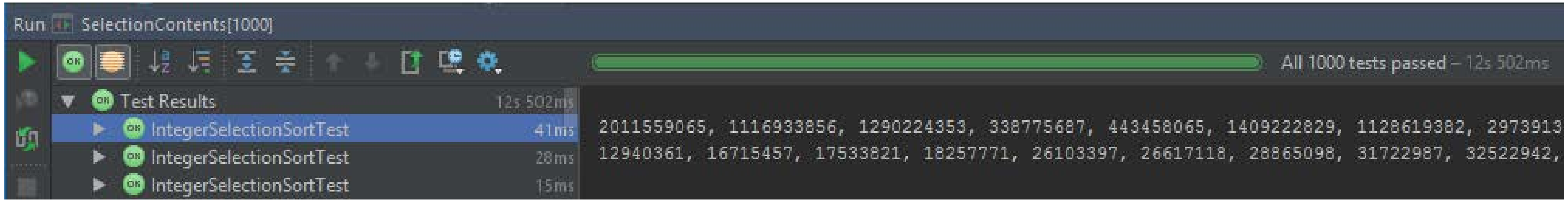
m = y/x^2
0.000000393
2.5645E-07
9.05556E-08
4.84375E-08
6.0656E-08
1.012E-07
4.14163E-08
3.09094E-08
2.6042E-08
2.8106E-08
2.5629E-08
2.723E-08
2.5987E-08
8.94512E-09
6.0805E-09
6.97273E-09
6.85753E-09
6.66012E-09
9.55264E-09
1.6243E-08
5.32229E-09
1.65242E-09
1.64598E-09
1.72508E-09
1.58959E-09
1.6007E-09
1.57464E-09
1.56242E-09
1.54257E-09
1.63134E-09
1.61301E-09
1.64426E-09
1.61982E-09
1.60759E-09
1.65437E-09
1.73397E-09
1.78121E-09
2.17151E-09
2.40276E-09
2.53963E-09
2.62235E-09
Average M
3.0687E-08

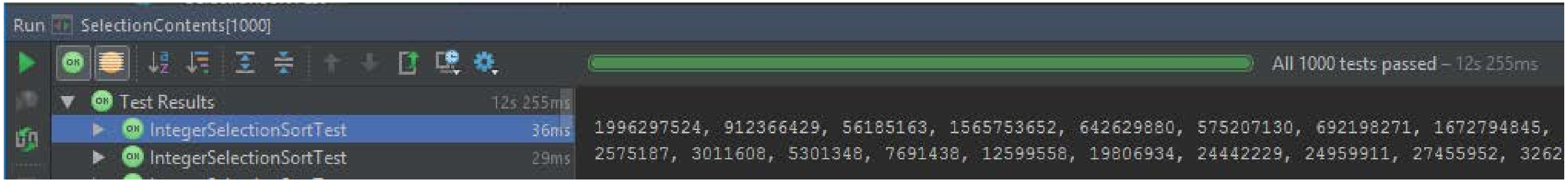
(Logbook) Sorting Question 2: Implement the selection sort algorithm. Your implementation should implement the ArraySort Interface.

Code Listing.

```java
package arraySorter;
/**
 * Created by u1661665(Joshua Pritchard) on 12/11/2018.
 */
/**
 * An implementation of a Selection Sort
 *
 * @param <T> The type of data held by the Array to be sorted.
 */
public class SelectionSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    @Override
    public T[] sort(T[] array)
    {
        //For(The whole list is unsorted, there is still part of the list unsorted, another element has been sorted.)
        for (int sortedElements = 0; sortedElements < array.length; sortedElements++)
        {
            //Find the index of the largest unsorted element.
            int indexOfLargestElement = -1;
            T largestElement = null;
            for(int x = 0; x < array.length - sortedElements; x++)
            {
                if(largestElement == null)
                {
                    largestElement = array[x];
                    indexOfLargestElement = x;
                    continue;
                }
                if (array[x].compareTo(largestElement) > 0)
                {
                    largestElement = array[x];
                    indexOfLargestElement = x;
                }
            }
            //Swap it with the last unsorted element.
            T temp = array[array.length - sortedElements - 1];
            array[array.length - sortedElements - 1] = array[indexOfLargestElement];
            array[indexOfLargestElement] = temp;
        }
        return array;
    }
}
```

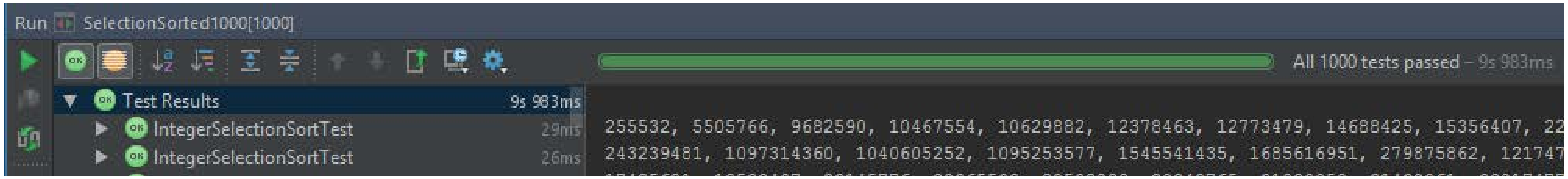
Testing

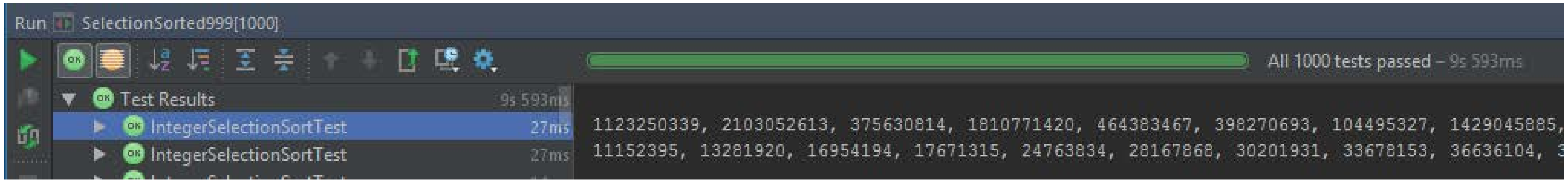


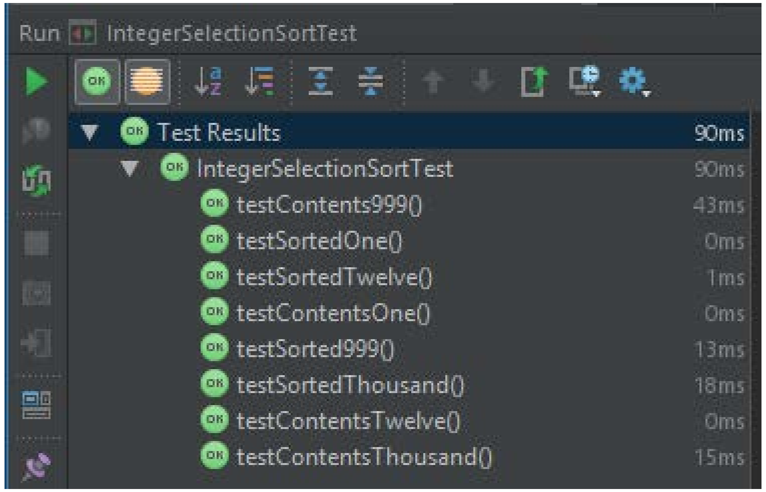


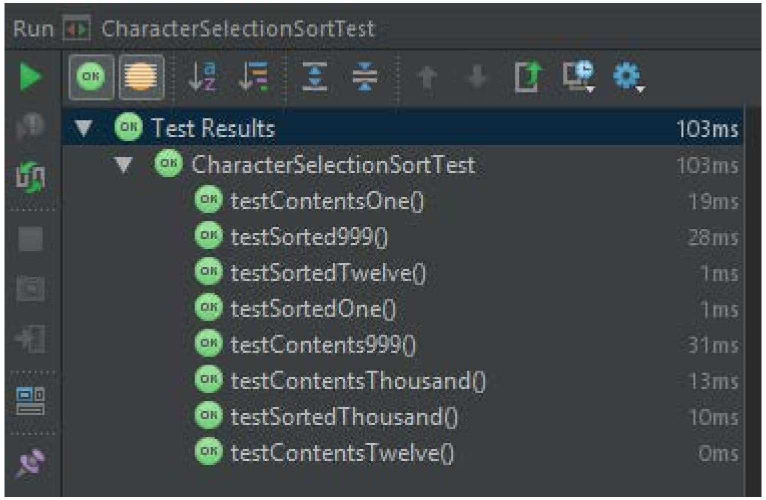
To test the selection sort's contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

I did the same but to test that the array was sorted, with 1000 tests each on arrays of 1000 and 999, the ordering of images is the same.






Run IntegerSelectionSortTest
Test Results 90ms
IntegerSelectionSortTest 90ms
testContents999() 43ms
testSortedOne() 0ms
testSortedTwelve() 1ms
testContentsOne() 0ms
testSorted999() 13ms
testSortedThousand() 18ms
testContentsTwelve() 0ms
testContentsThousand() 15ms


Run CharacterSelectionSortTest
Test Results 103ms
CharacterSelectionSortTest 103ms
testContentsOne() 19ms
testSorted999() 28ms
testSortedTwelve() 1ms
testSortedOne() 1ms
testContents999() 31ms
testContentsThousand() 13ms
testSortedThousand() 10ms
testContentsTwelve() 0ms

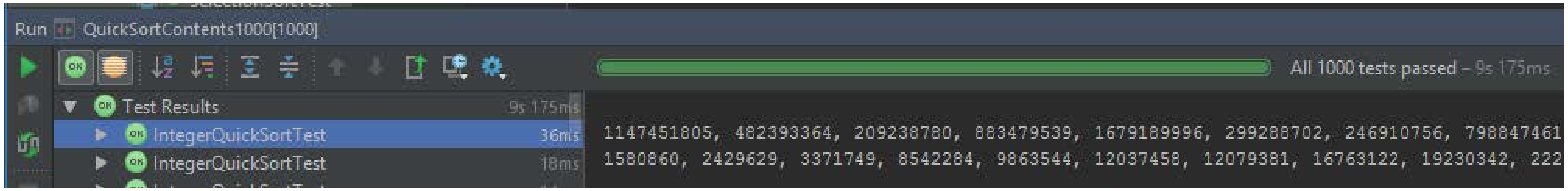
(Logbook) Sorting Question 3: Implement the quicksort algorithm. Your Implementation should Implement the ArraySort Interface.

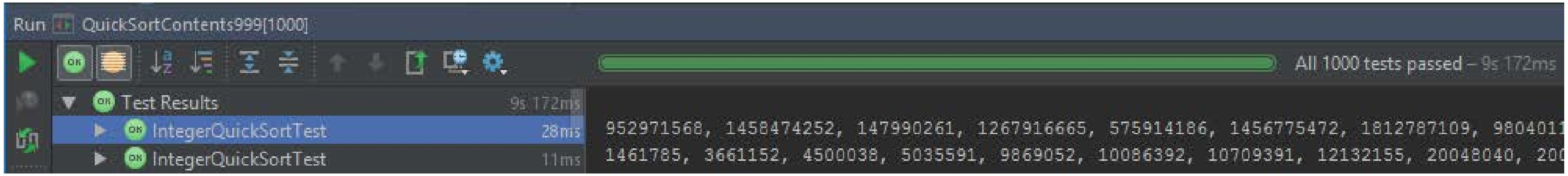
Code Listing

```java
package arraySorter;
/**
 * Created by u1661665(Joshua Pritchard) on 12/11/2018.
 */
/**
 * An Implementation of a QuickSort.
 *
 * @param <T> the type of data held by the array to be sorted.
 */
public class QuickSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    /**
     * Sorts the passed in array using a QuickSort implementation, then returns the sorted array.
     *
     * @param array the array to be sorted.
     * @return array (sorted)
     */
    @Override
    public T[] sort(T[] array)
    {
        return quickSort(array, 0, array.length);
    }
    /**
     * A recursive method to arrange an array around a pivot point, then repeat this process to the left and right sides
     * of this pivot point until the array ends sorted.
     *
     * @param array the array on which the quicksort is being performed.
     * @param low the smallest index location to sort.
     * @param high the largest index location to sort + 1.
     * @return the sorted array.
     */
    private T[] quickSort(T[] array, int low, int high)
    {
        if(low < high)
        {
            //Arrange the array around an initial pivot point.
            int pivotNewLocation = partition(array, low, high);
            //QuickSort the array to the left of the original pivot point.
            quickSort(array, low, pivotNewLocation);
            //QuickSort the array to the right of the original pivot point.
            quickSort(array, pivotNewLocation + 1, high);
        }
```

```java
        return array;
    }
    /**
     * On the array, set the pivot point as the left most element.
     * sort all smaller elements to the left of the pivot point.
     * sort all larget elements to the right of the pivot point.
     * swap the still leftmost pivot point into its correct position.
     * return the index location the pivot point was swapped into.
     *
     * @param array the array on which the QuickSort is operating.
     * @param low the bottom element to arrange around the pivot.
     * @param high the last element to arrange around the pivot
     * @return the index location in the array the pivot was placed into.
     */
    private int partition(T[] array, int low, int high)
    {
        T pivot = array[low];
        int leftWall = low;
        for(int i = low + 1; i < high; i++)
        {
            if(array[i].compareTo(pivot) < 0)
            {
                swap(array, i, leftWall + 1);
                leftWall++;
            }
        }
        swap(array, low, leftWall);
        return leftWall;
    }
    /**
     * Swap two elements within the array the QuickSort is operating on
     *
     * @param array the array the QuickSort is operating on.
     * @param i1 the first index location to swap.
     * @param i2 the second index location to swap.
     */
    private void swap(T[] array, int i1, int i2)
    {
        T temp = array[i1];
        array[i1] = array[i2];
        array[i2] = temp;
    }
}
```

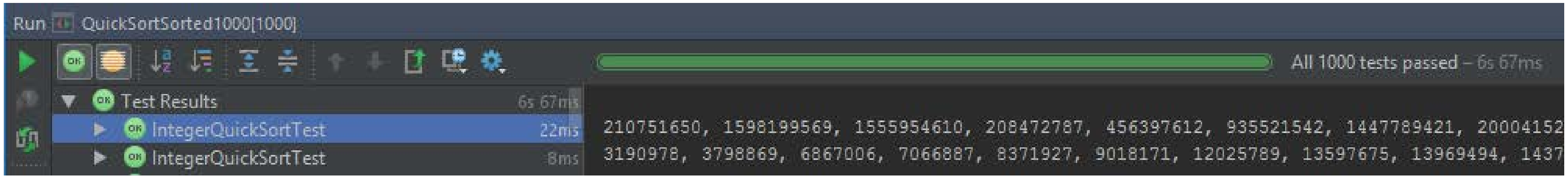
Testing

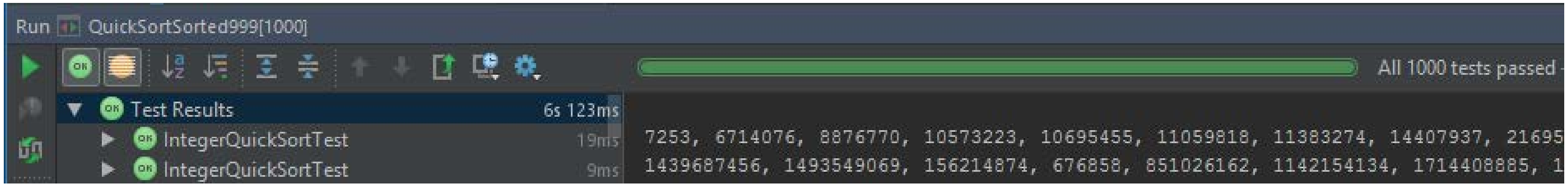


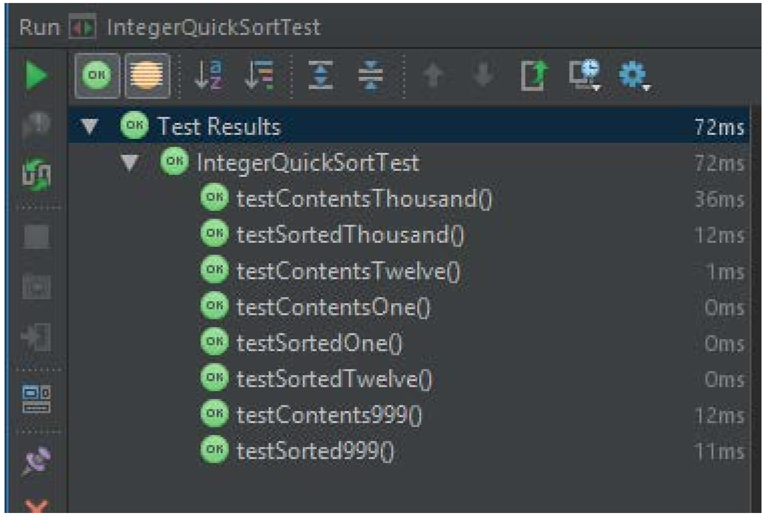


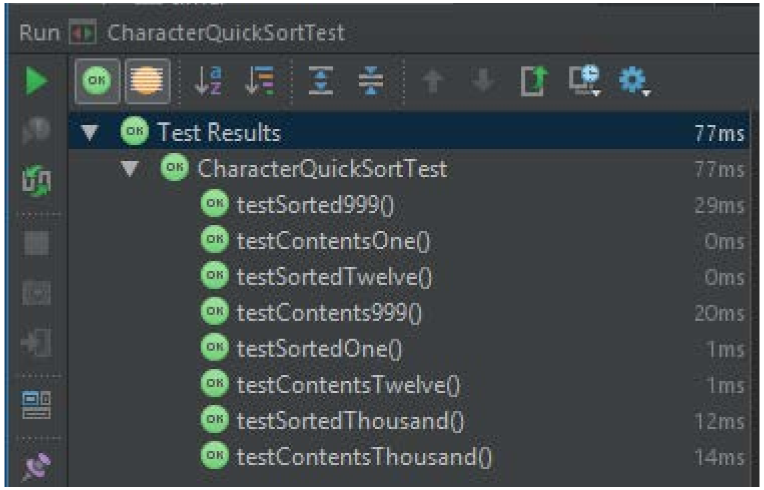
To test the quick sort's contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

I did the same but to test that the array was sorted, with 1000 tests each on arrays of 1000 and 999, the ordering of images is the same.






Run IntegerQuickSortTest
Test Results 72ms
IntegerQuickSortTest 72ms
testContentsThousand() 36ms
testSortedThousand() 12ms
testContentsTwelve() 1ms
testContentsOne() 0ms
testSortedOne() 0ms
testSortedTwelve() 0ms
testContents999() 12ms
testSorted999() 11ms


Run CharacterQuickSortTest
Test Results 77ms
CharacterQuickSortTest 77ms
testSorted999() 29ms
testContentsOne() 0ms
testSortedTwelve() 0ms
testContents999() 20ms
testSortedOne() 1ms
testContentsTwelve() 1ms
testSortedThousand() 12ms
testContentsThousand() 14ms

(Additional) Additional algorithms

Insertion sort

Code Listing

```java
package arraySorter;

/**
 * Created by u1661665(Joshua Pritchard) on 12/11/2018.
 */

/**
 * An implementation of an Insertion Sort.
 *
 * @param <T> the type of data held by the array.
 */
public class InsertionSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    /**
     * Sorts the array using an insertion sort.
     *
     * @param array the array to be sorted.
     * @return the sorted array.
     */
    @Override
    public T[] sort(T[] array)
    {
        //sorted = the largest element index that has been sorted
        //    (Start out with the first element in the list being the sorted list).
        for(int sorted = 0; sorted < array.length - 1; sorted++)
        {
            //The element to be 'inserted' into the sorted part of the array.
            T newElement = array[sorted + 1];
            //Compare to the largest element in the sorted part of the array first.
            int sortedListCompareIndex = sorted;
            //Move larger elements up in their place in the sorted list. (making space for the newElement
            while (sortedListCompareIndex >= 0 && newElement.compareTo(array[sortedListCompareIndex]) < 0)
            {
                array[sortedListCompareIndex + 1] = array [sortedListCompareIndex];
                sortedListCompareIndex--;
            }
            //Place the newElement in its correct place within the array.
            array[sortedListCompareIndex + 1] = newElement;
        }
        return array;
    }
}
```

Merge sort

Code Listing

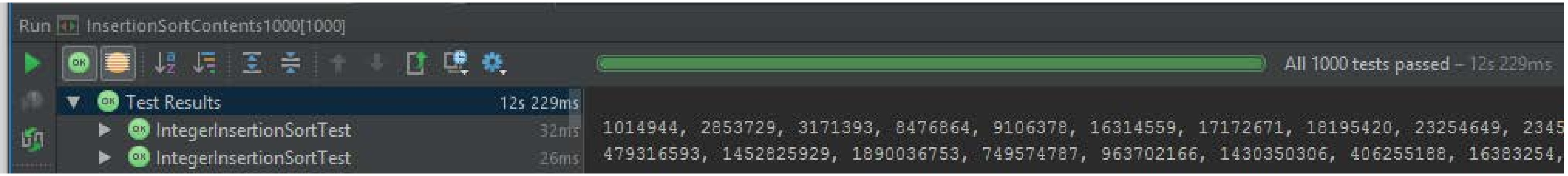
```java
package arraySorter;

/**
 * Created by u1661665(Joshua Pritchard) on 13/11/2018.
 */

import java.util.Arrays;

/**
 * An implementation of MergeSort
 *
 * @param <T> the type of data held by the array.
 */
public class MergeSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    /**
     * Sort the array using a MergeSort.
     *
     * (recursively breaks down the array into sub arrays until arrays of size 1 are reached.)
     * (then works backwards merging these arrays back into each other until a sorted array state is reached containing
     *  all the elements of the initial unsorted array, but now sorted).
     *
     * @param array the array to be sorted.
     * @return array (sorted)
     */
    @Override
    public T[] sort(T[] array)
    {
        //If the array being sorted is less than 2 elements, it is already sorted and cannot be split into two separate
        //     arrays, so return the initial array to prevent an exception.
        if(array.length < 2)
        {
            return array;
        }

        //Create two sub arrays from the initial array passed in.
        T[] temp1 = copy(array, 0, (array.length/2) + (array.length % 2));
        T[] temp2 = copy(array, (array.length/2) + (array.length % 2), array.length);

        //Sort these two arrays.
        temp1 = sort(temp1);
```

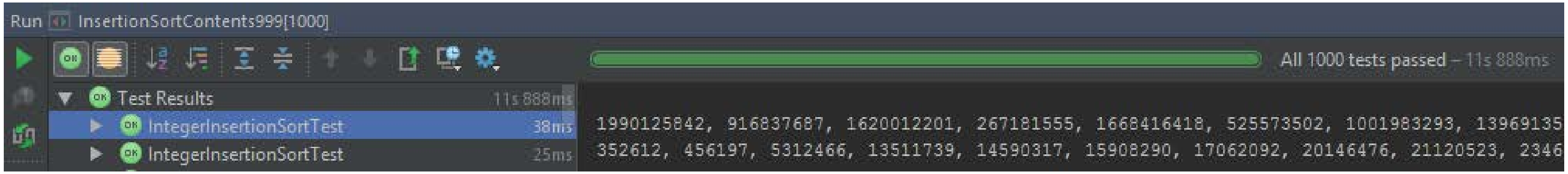
```java
        temp2 = sort(temp2);

        //Merge the two subarrays back into the initial array.
        merge(array, temp1, temp2);

        //return the now sorted array.
        return array;
    }

    /**
     * Create and return a new array, comprised of a sublist of a passed in array.
     *
     * @param list the array to create the subarray from.
     * @param from the lower bound within list to copy (inclusive).
     * @param to the upper bound within list to copy (exclusive).
     * @return a new list comprised of the elements from 'from' to 'to' in 'list'
     */
    private <T> T[] copy(T[] list, int from, int to)
    {
        return Arrays.copyOfRange(list, from, to);
    }

    /**
     * Merge two sorted arrays into one larger array, keeping the sorted.
     *
     * @param target the array to merge source1 and source2 into.
     * @param source1 a sorted array
     * @param source2 a second sorted array.
     */
    private void merge(T[] target, T[] source1, T[] source2)
    {
        //Variables to keep track of the smallest unused element in each source array.
        int source1Index = 0;
        int source2Index = 0;

        int lowestMaxIndex = (source1.length < source2.length ? source1.length : source2.length);

        //targetIndex keeps track of the next index to place the next smallest element into.
        for(int targetIndex = 0; targetIndex < lowestMaxIndex * 2; targetIndex++)
        {
            //If the end of source1 has been reached, add the rest of source2 to target, then break the merge.
            if(source1Index == source1.length)
            {
                int iterator = 0;
```

```
                for(int x = source2Index; x < source2.length; x++)
                {
                    target[targetIndex + iterator] = source2[x + iterator];
                }
                break;
            }
            //if the end of source2 has been reached, add the rest of source1 to target, then break the merge.
            if(source2Index == source2.length)
            {
                int iterator = 0;
                for(int x = source1Index; x < source1.length; x++)
                {
                    target[targetIndex + iterator] = source1[x + iterator];
                }
                break;
            }

            //Choose the smallest element from the two source arrays to place into the target(sorted) array
            if(source1[source1Index].compareTo(source2[source2Index]) < 0)
            {
                target[targetIndex] = source1[source1Index];
                source1Index++;
            }
            else
            {
                target[targetIndex] = source2[source2Index];
                source2Index++;
            }
        }


    }
}
```

Testing

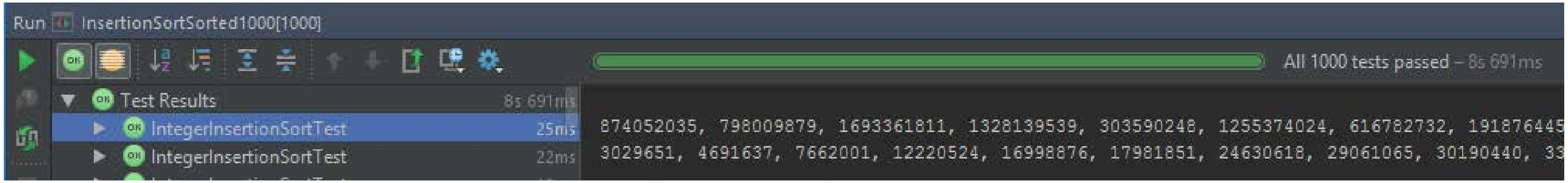
Insertion Sort.

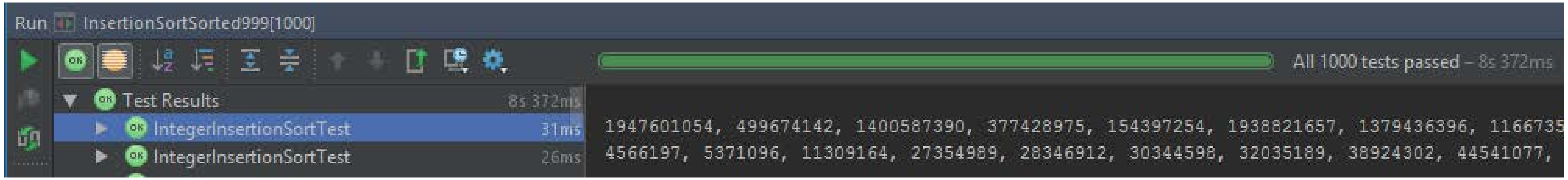


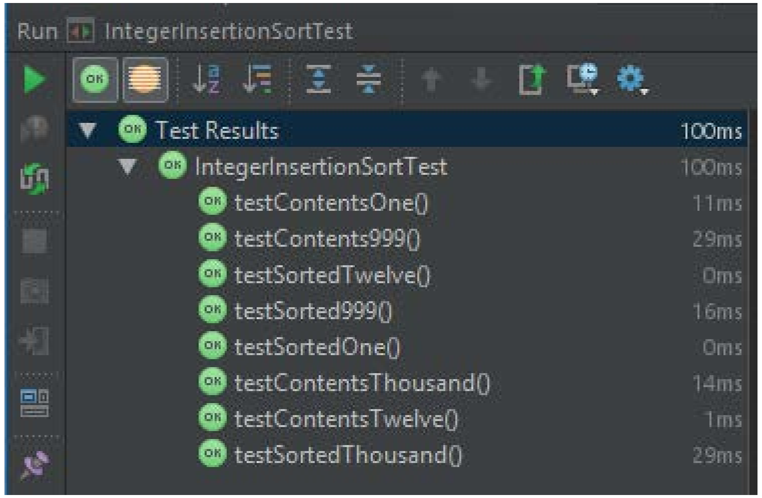


To test the Insertion sort’s contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

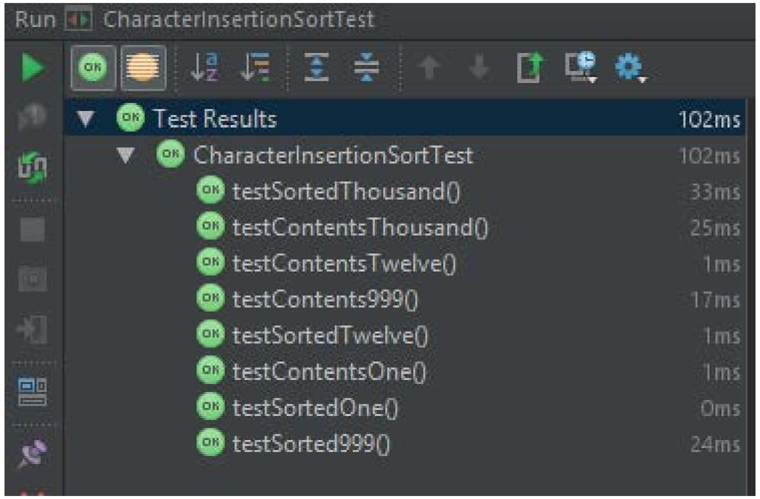
I did the same but to test that the array was sorted, with 1000 tests each on arrays of 1000 and 999, the ordering of images is the same.







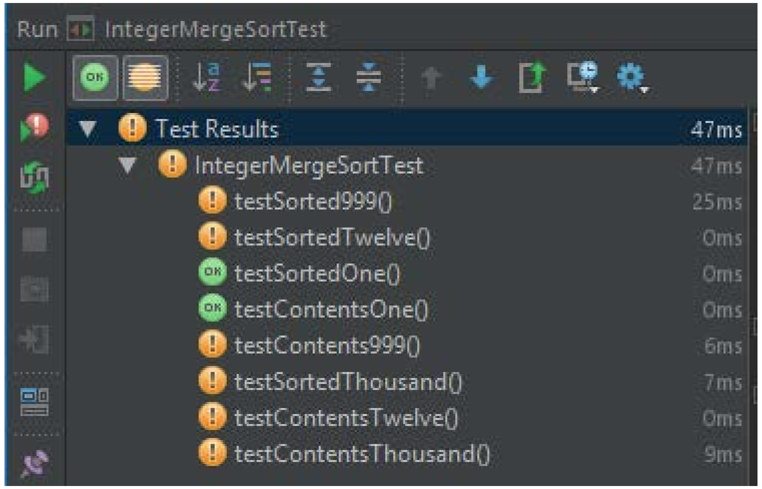
Run IntegerInsertionSortTest
Test Results 100ms
IntegerInsertionSortTest 100ms
testContentsOne() 11ms
testContents999() 29ms
testSortedTwelve() 0ms
testSorted999() 16ms
testSortedOne() 0ms
testContentsThousand() 14ms
testContentsTwelve() 1ms
testSortedThousand() 29ms



Run CharacterInsertionSortTest
Test Results 102ms
CharacterInsertionSortTest 102ms
testSortedThousand() 33ms
testContentsThousand() 25ms
testContentsTwelve() 1ms
testContents999() 17ms
testSortedTwelve() 1ms
testContentsOne() 1ms
testSortedOne() 0ms
testSorted999() 24ms

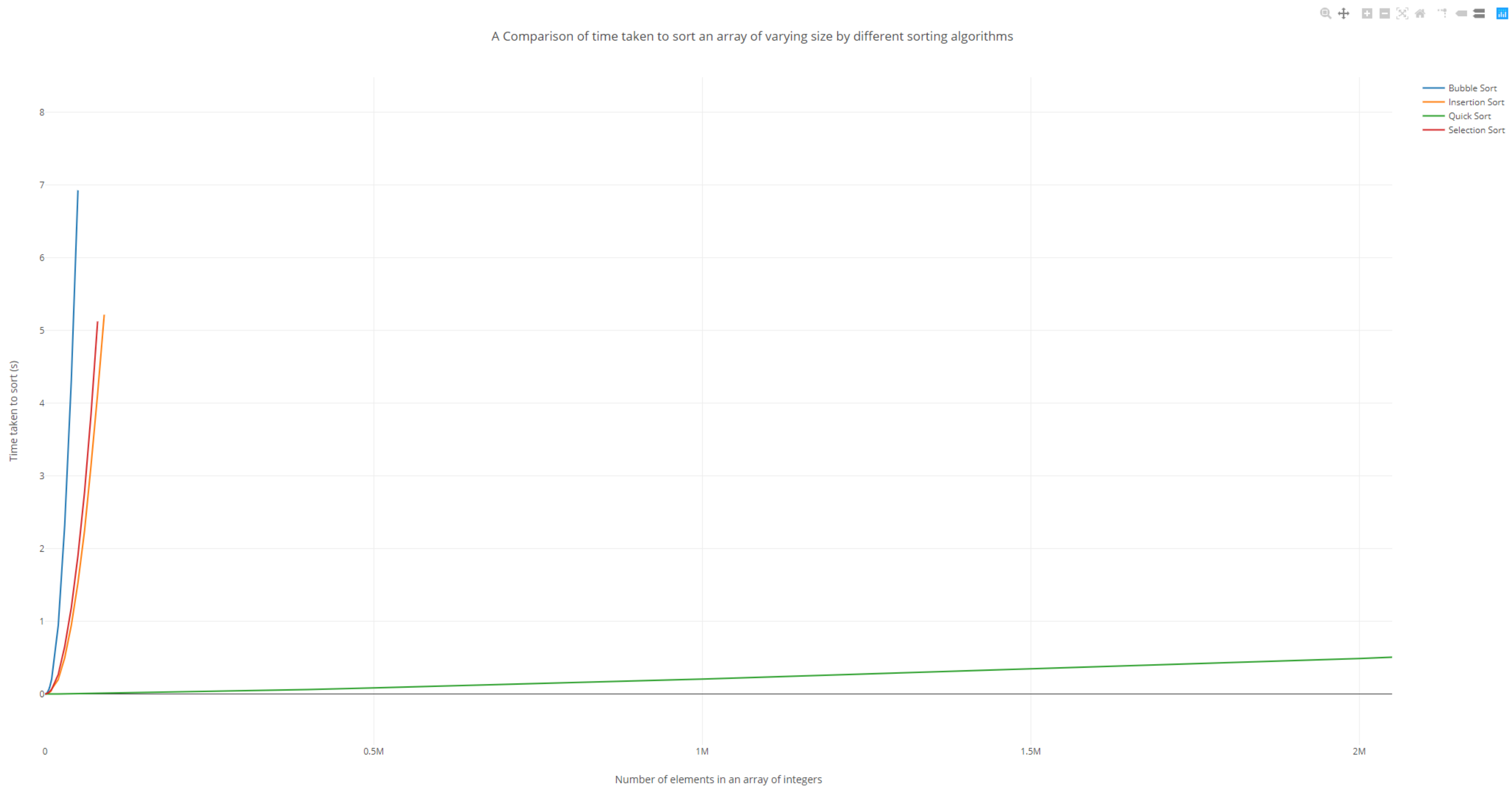
Merge Sort

I couldn’t get my implementation of a merge sort to correctly function. I believe I was close, and with more time would reach a correct implementation, however as of this point, my implementation is non-functional. I have included it here merely as proof of attempt.



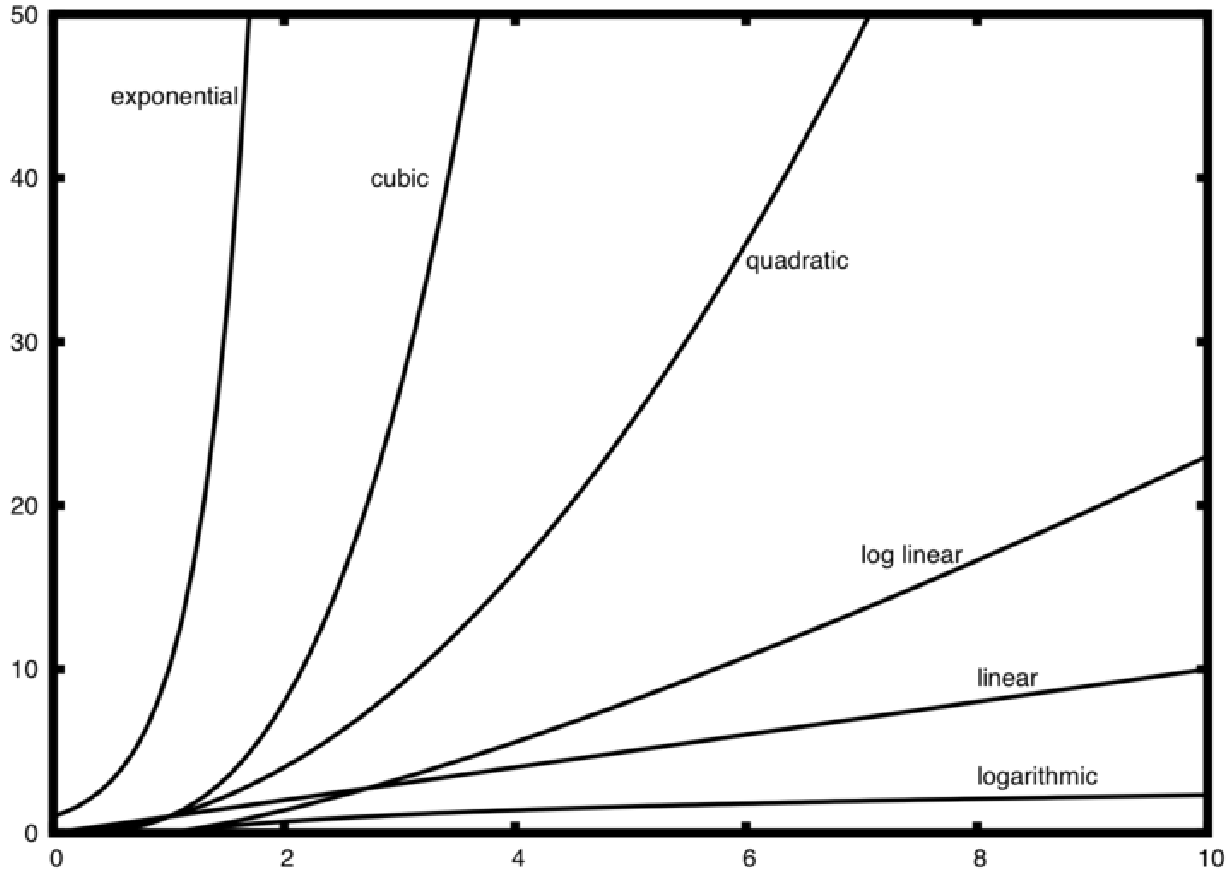
(Logbook) Sorting Question 4: Use your implementations to time the execution of (at least) these two sorting algorithms for various sizes of array, and plot the results on a graph. Can you arrive at (approximate) formulae for how to execution times vary in relation to the data size?

Size	Bubble Sort	Insertion Sort	Quick sort	Selection sort
1	0.000000427	0.000000342	0.000001054	0.000000484
2	0.000001083	0.000001111	0.000002423	0.00000114
3	0.000000855	0.000000826	0.000001454	0.000001254
4	0.000000826	0.000000855	0.000001339	0.000001197
5	0.000002309	0.000001083	0.000002309	0.000001767
6	0.000002423	0.000001454	0.000006757	0.000003307
7	0.000001938	0.000001938	0.000003335	0.000002765
8	0.00000191	0.000002081	0.000002024	0.00000191
9	0.00000305	0.000001454	0.000002423	0.000002423
10	0.000004561	0.000002195	0.000002195	0.000002822
20	0.000011318	0.000005873	0.000003364	0.000009579
30	0.000024775	0.000012002	0.000003079	0.000033299
40	0.000038859	0.00002774	0.000004533	0.000051119
50	0.000057049	0.000032757	0.000008381	0.000065801
60	0.000022494	0.00001682	0.000007526	0.000028624
70	0.000028567	0.000016678	0.000016165	0.000028995
80	0.000038089	0.00002238	0.000014654	0.000035267
90	0.000045074	0.000021439	0.000015081	0.000038973
100	0.000082793	0.000025345	0.000012772	0.000065972
200	0.000323392	0.000098731	0.000045816	0.000410776
300	0.00077528	0.000229393	0.00006888	0.001274581
400	0.000278773	0.000444304	0.000080142	0.000133798
500	0.000421809	0.000644418	0.000130919	0.000295737
600	0.000608323	0.001665685	0.000339072	0.000318944
700	0.000838573	0.001195464	0.000357006	0.000365958
800	0.001037546	0.00042657	0.000472586	0.000494853
900	0.001325985	0.000389336	0.000487839	0.000638887
1000	0.001717887	0.000487469	0.000107227	0.000960483
2000	0.006623996	0.001952299	0.000161482	0.002772312
3000	0.015155559	0.004445524	0.000339785	0.00651235
4000	0.028221884	0.008546986	0.000395181	0.011458002
5000	0.047837023	0.013281663	0.000429935	0.016622528
6000	0.065882044	0.018020188	0.000545858	0.026289542
7000	0.093479524	0.025113321	0.000671217	0.033627322
8000	0.127176982	0.031167853	0.000755266	0.040855023
9000	0.164270614	0.039986457	0.000812058	0.051087738
10000	0.205761531	0.066943197	0.001107795	0.061437659
20000	0.946376869	0.197591108	0.002308706	0.266415015
30000	2.32521764	0.498725761	0.003800849	0.652556931
40000	4.319941211	0.946990097	0.004978296	1.18912643
50000	6.928330423	1.531460397	0.006180718	1.901458814
60000		2.246882185	0.007186447	2.778406759
70000		3.138711334	0.008436452	3.841460297
80000		4.128181456	0.010065302	5.124100065
90000		5.216257168	0.013139767	
100000			0.012387238	
200000			0.028346844	
300000			0.041967666	
400000			0.061625058	
500000			0.084015903	
600000			0.107213731	
700000			0.131984017	
800000			0.15190433	
900000			0.17793172	
1000000			0.205337498	
2000000			0.488995853	
3000000			0.816902413	
4000000			1.186729684	
5000000			1.575573235	
6000000			1.952751057	
7000000			2.404905623	
8000000			2.875273705	
9000000			3.297804108	
10000000			3.781294025	
20000000			8.722236147	


A Comparison of time taken to sort an array of varying size by different sorting algorithms
Bubble Sort
Insertion Sort
Quick Sort
Selection Sort
Time taken to sort (s)
8
7
6
5
4
3
2
1
0
0
0.5M
1M
1.5M
2M
Number of elements in an array of integers

As can be seen from the graph, Bubble sort has the worst time complexity out of the four algorithms tested. Selection sort and insertion sort have very comparable time complexities. Quick sort dwarfs all of the other algorithms though, with a very shallow increase in time complexity as the size of the array increases.

The most noticeable increase in time complexity in the three 'inefficient' algorithms is observed near the start of the increase in number of elements, with the time complexity increases getting severer, but not matching the severity of the initial rate of gradient change.



above image sourced from (http://interactivepython.org/runestone/static/pythonds/AlgorithmAnalysis/BigONotation.html)

Using the above image, I can compare the timing data generated by the four algorithms and decide on the most accurate Big O classification for them.

It seems that the Bubble sort should be classed as exponential, as it's rate of increase becomes almost asymptotic almost instantly. However, the algorithm for bubble sort does not support this, as it adds another pass of most of the array for each element added. Bubble sort is classed as $O(n^2)$ or quadratic.

QuickSort is absolutely a logarithmic best case algorithm and is given the big O classification O(log n).

Looking at the graph, log linear appears to be around half way between the values of logarithmic and quadratic. Looking at our produced graph, the Selection and Insertion sorts definitely do not follow this trend, leaving them somewhere between the Quadratic $O(n^2)$ and logarithmic O(log n) classifications.

Both of the algorithms are very close to bubble sort however, so I would class them as worst case $O(n^2)$ as well. However it is clear from the graph that this worst case scenario is less common than in bubble sort, which still makes them more efficient.

Aside from my analysis of these results the actual average performances for all 4 algorithms are shown below.

Bubble: $O(n^2)$ comparisons made, $O(n^2)$ swaps made.

Selection: $O(n^2)$ comparisons made, O(n) swaps made.

Insertion: $O(n^2)$ comparisons made, O(1) swaps made.

Quick: O(n log n)

Self Evaluation

For 1 mark: Basic timing results for method provided. No analysis.

I believe I have achieved this easily with my results shown in the excel table.

For 2 marks: Graphic presentation. Some comparison.

My graph achieves the graphic presentation and I have at least ‘some’ comparison.

For 3 marks: Additional algorithms, larger data set.

I have included additional algorithms, however have not tested this on any character or string arrays, only integer arrays.

For 4 marks: Extensive testing, good analysis.

I believe that my testing was good, but not extensive, and that my analysis is worthy of a ‘good’ description, as I attempted to classify the algorithms even if I didn’t get them completely correct.

For 5 marks: Big O analysis or near equivalent.

My analysis includes a Big O classification, however I was not able to correctly classify the algorithms perfectly.

I believe for the reasons against the marks stated above, my work for practical 3 warrants a 4/5, as I missed out some of the potential extensions to this work (such as different data types), but made up for it in other areas (such as the attempted big O classification).