J.Pritchard U1661665

U1661665@unimail.hud.ac.uk

Logbook

Algorithms – Processes and data

# Practical 1 (Week 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.  
 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.

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

//Used to control the number of randomly generated tests ran.

public static final int *NUM\_RANDOM\_TESTS* = 20;

*...*

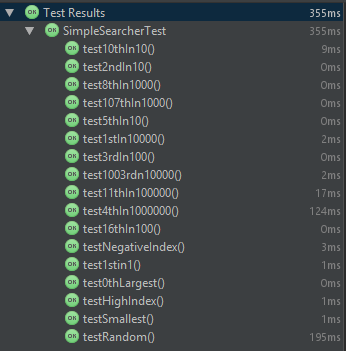
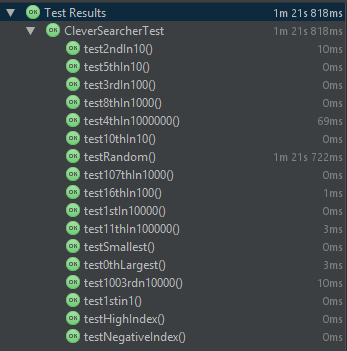
*...*

*/\*\*  
 \* 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  
 *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

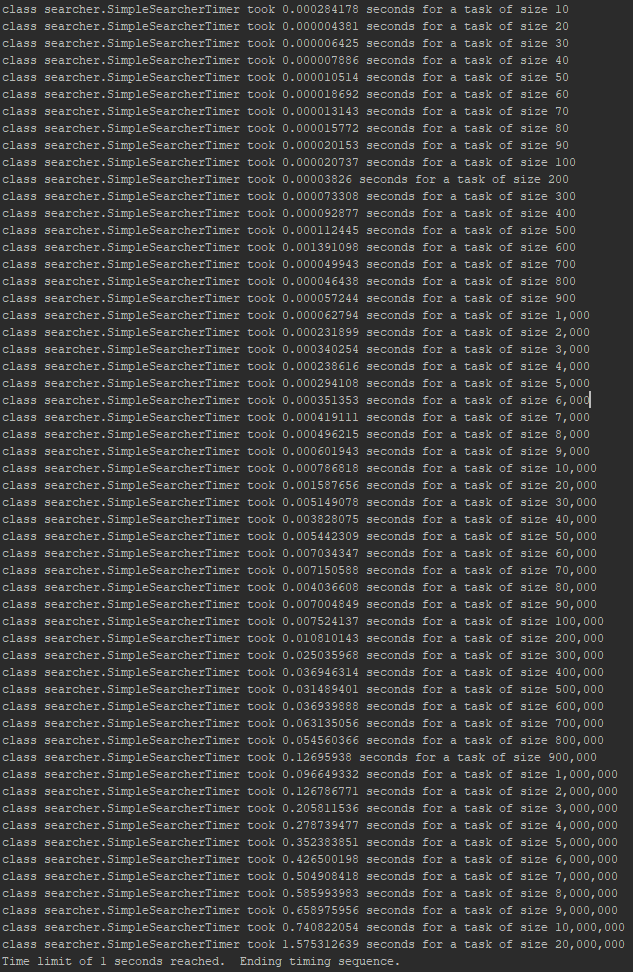


## (Logbook) Question 3: Also create a timer class to time the execution of the findElement method in your CleverSearcher implementation. Compare this with the time taken by the SimpleSearcher implementation when performing searches of the same size.

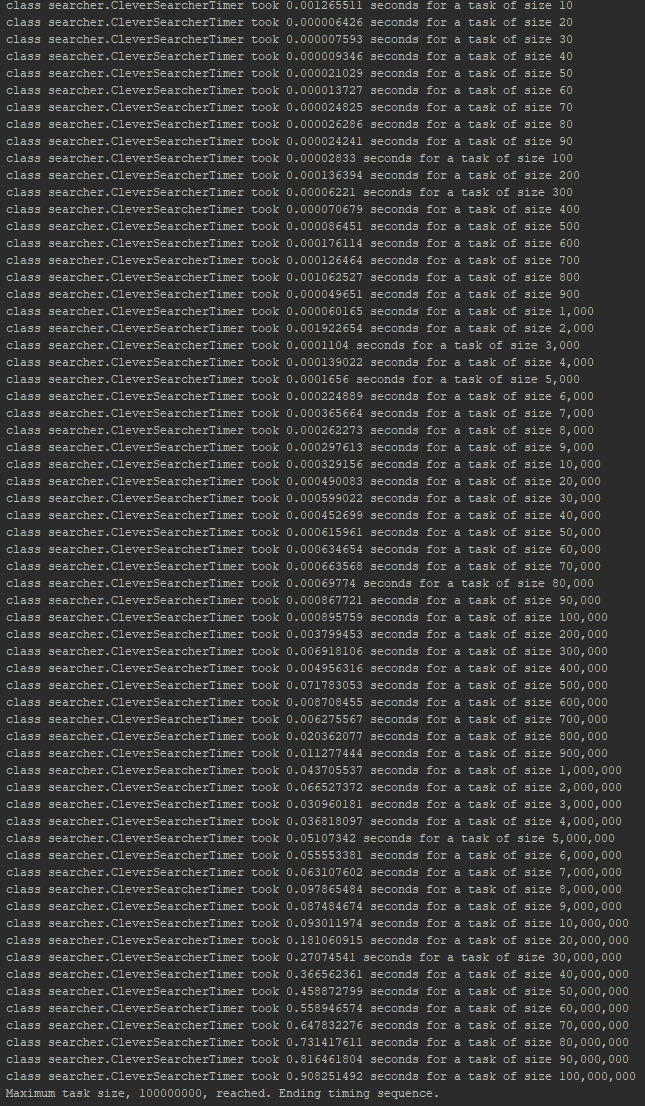
### CleverSearcherTimer code listing.

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;  
 }  
  
 */\*\*  
 \* 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.

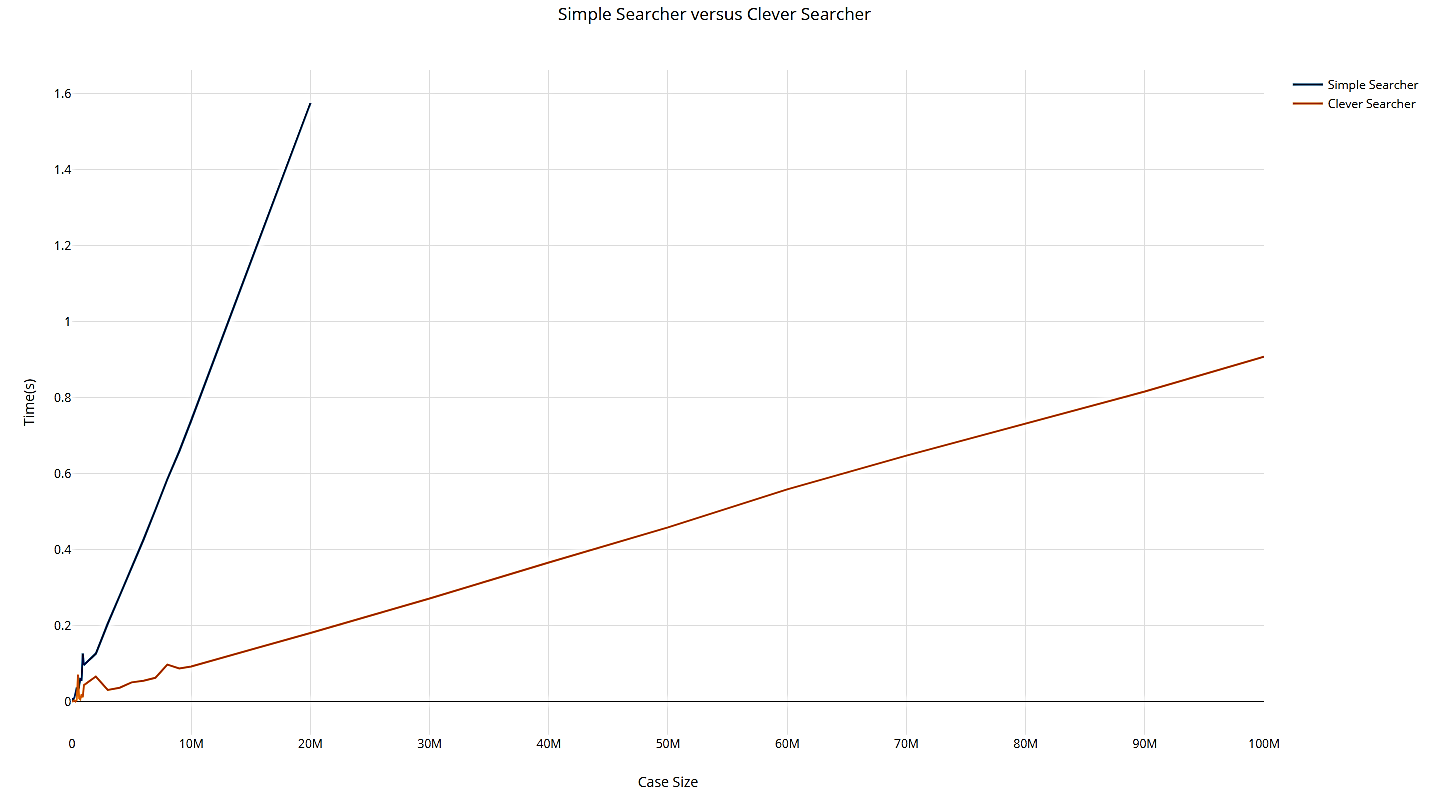


### Clever searcher times.



### 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 (Week 3)

## (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)  
 {  
 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

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

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.  
 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)  
 {  
 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++)  
 {

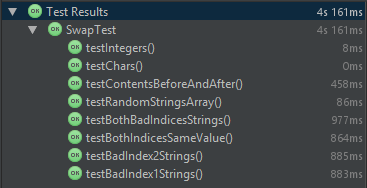
//Get a random arrayList of strings as per the values defined under 'testRandomStringsArray values'  
 ArrayList<String> testData = RandomStringArray.*getRandomStringArray*(LOWEST\_SIZE\_ARRAY, HIGHEST\_SIZE\_ARRAY,  
 LOWEST\_SIZE\_STRING, HIGHEST\_SIZE\_STRING);

//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()  
 {  
 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,  
 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  
 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))  
 {  
 *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

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

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

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

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

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() {  
 *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

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));  
 }  
  
 @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));  
 }  
  
 @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");  
 }  
  
}

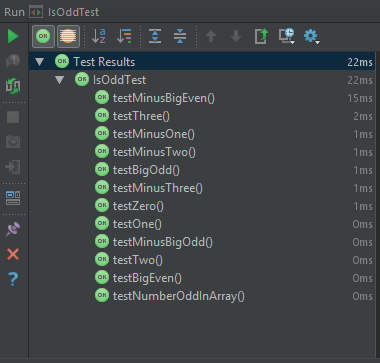
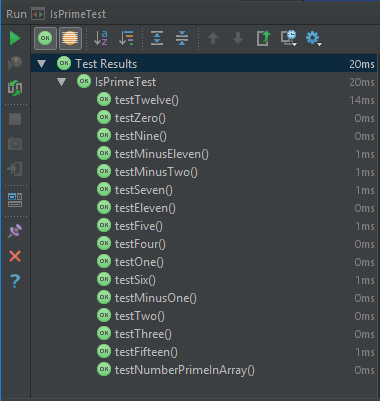
#### IsPalindromeTest

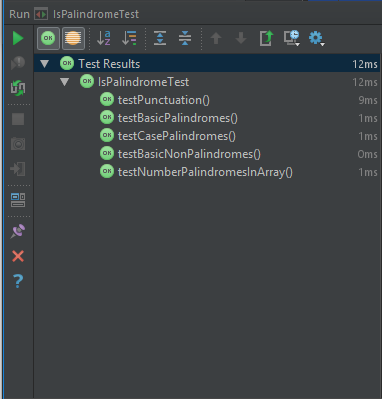
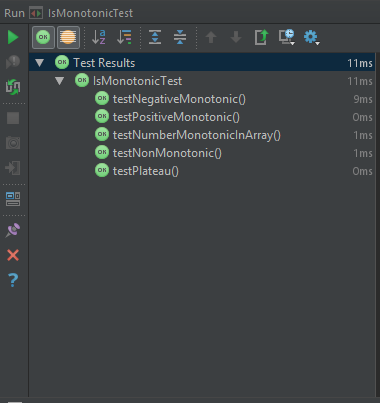
package unaryPredicate;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 11/11/2018.  
 \*/*import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*fail*;  
  
*/\*\*  
 \* A selection of tests testing the IsPalindrome class and its methods.  
 \*/*class IsPalindromeTest  
{  
 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  
 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

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");  
 }  
  
 @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.



# Practical 3 (Week 5)

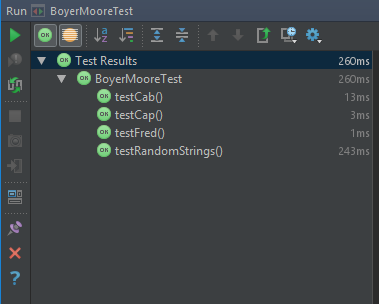
## (Additional) Question 1

### – 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  
 {  
 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;  
 }  
}

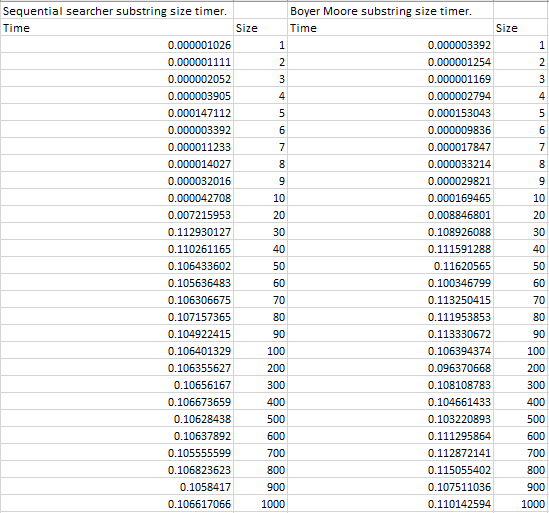
#### Result of testing.



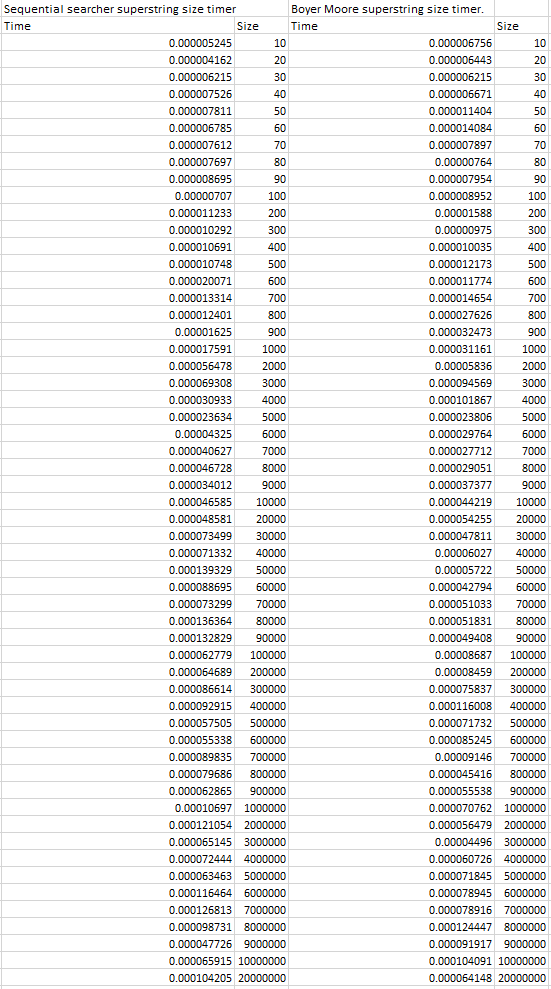
### – Use your algorithm, and the implementation of the sequential substring search algorithm provided to perform an empirical comparison of the efficiency of the two algorithms.

#### Test data acquired.

##### SubString Size Timer.



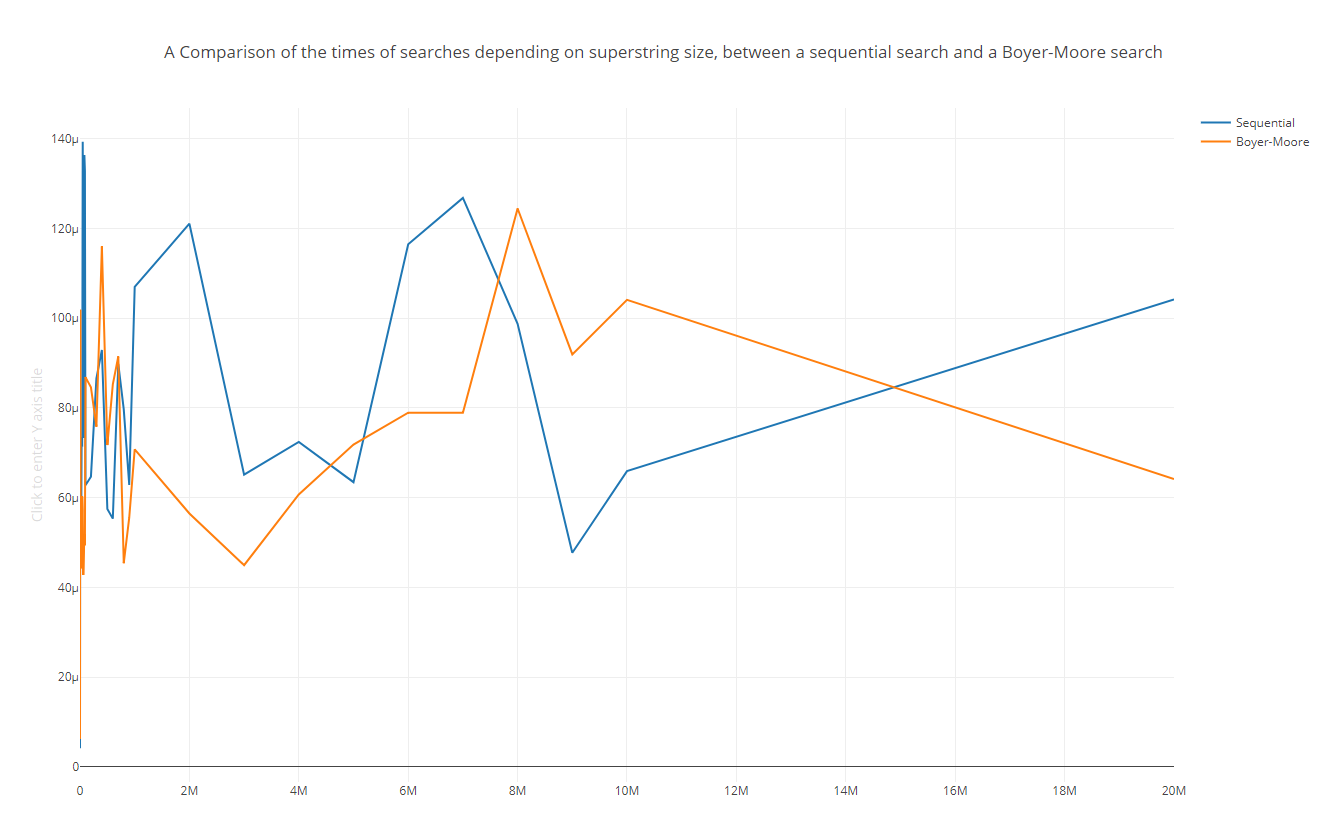
##### SuperString Size Timer.



#### 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.

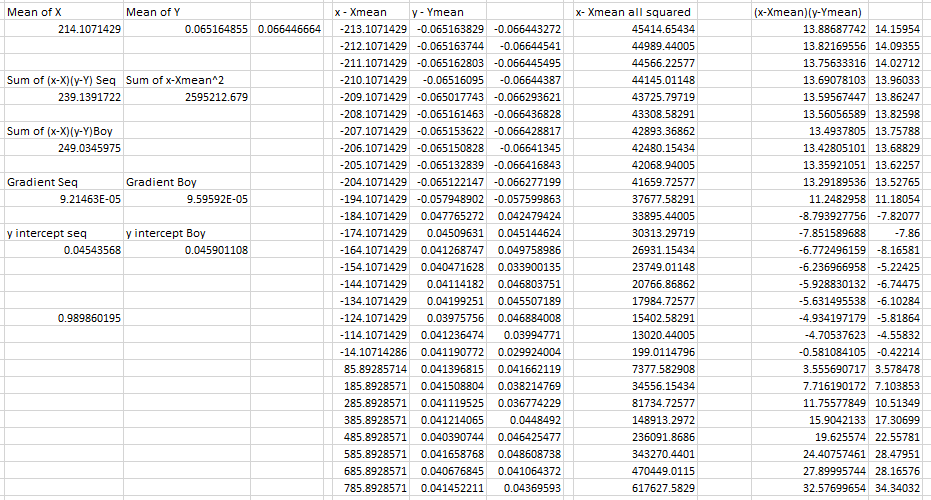
When small SubStrings are concerned, both implementations rapidly increase in time taken to their averages. Both implementations however, are fairly ignorant to increasing values of SubString size, showing very comparable times for searches of small and large SubStrings.

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.

### – 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



##### For SuperStrings

##### 

##### 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.

### – 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));  
 }  
}

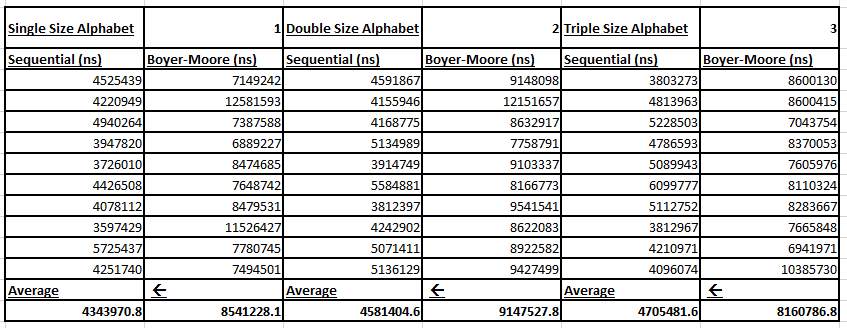
#### CharacterScope Code Listing

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.

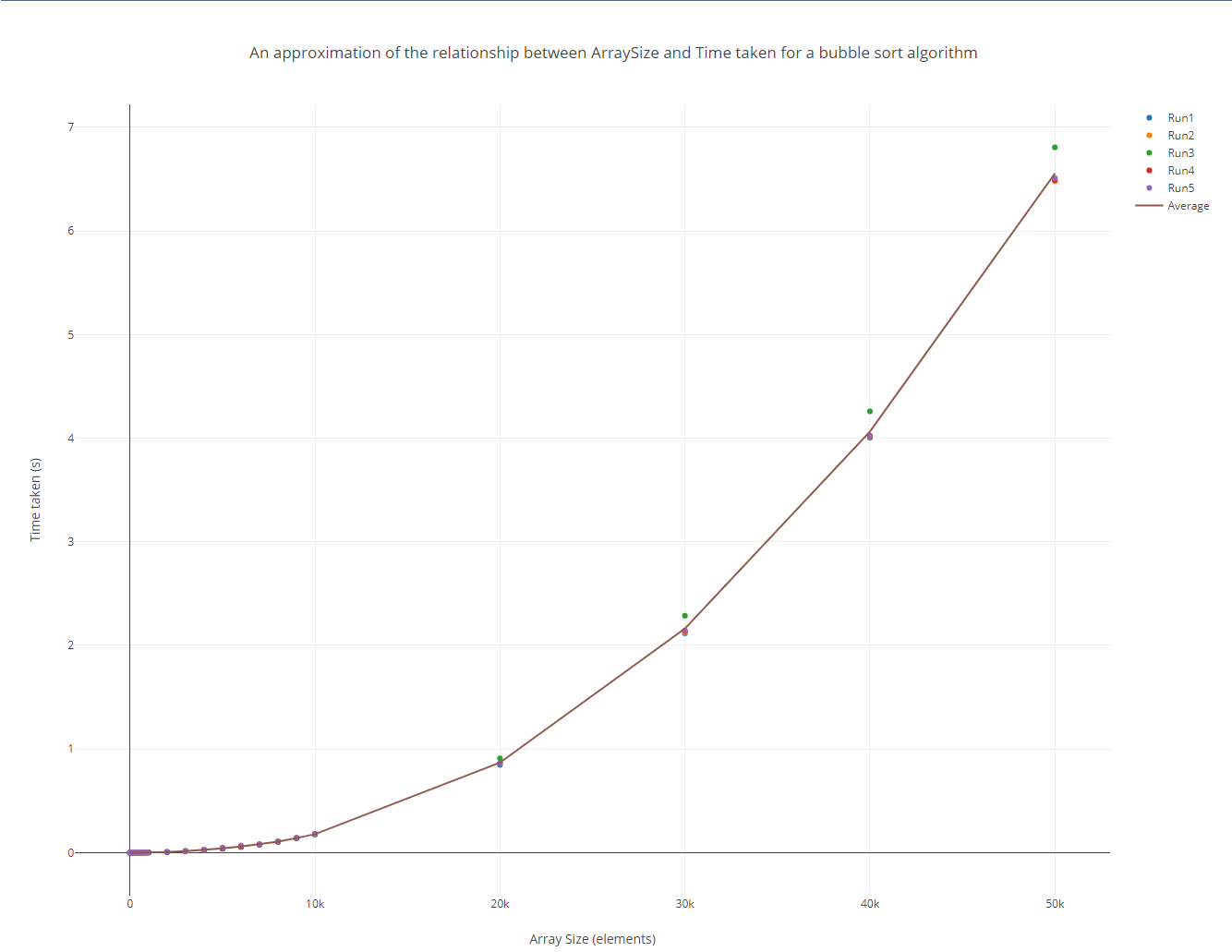


As can be seen from the test data, the Sequential searcher takes slightly longer each time the alphabet gains another 26 characters (26, 52, 78). The partial Boyer-Moore however stays relatively the same. Despite my implementation of Boyer-Moore being ineffective due to its partial implementation, the 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

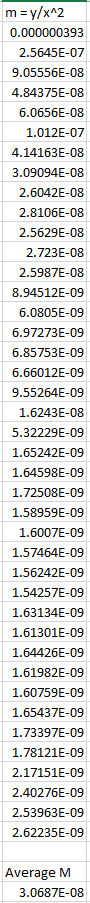
### Graph



### 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.

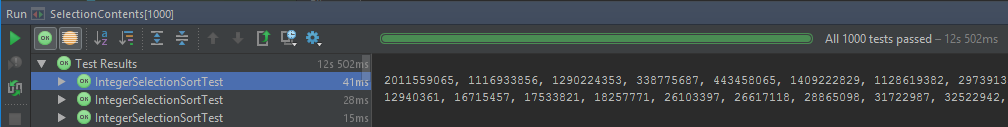
However where Big O notation is concerned, the function falls under the O(n2) classification, as the time taken is directly proportional to the size of the array squared.

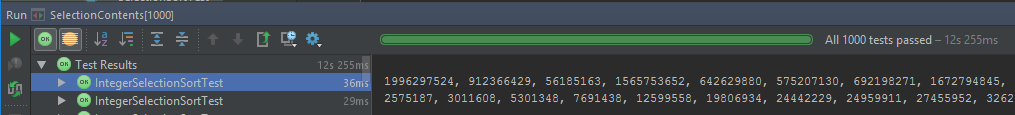
## (Logbook) Sorting Question 2: Implement the selection sort algorithm. Your implementation should implement the ArraySort Interface.

### Code Listing.

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;  
 }  
}

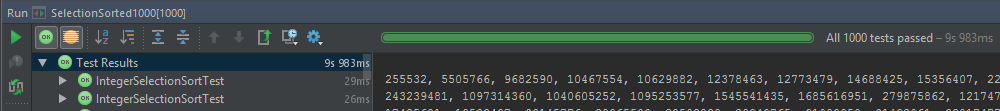
### Testing

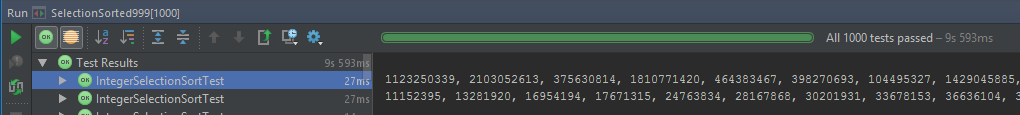


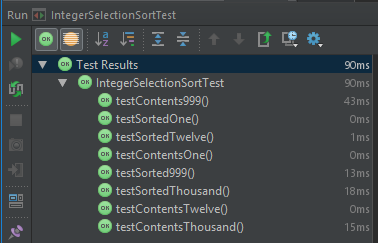


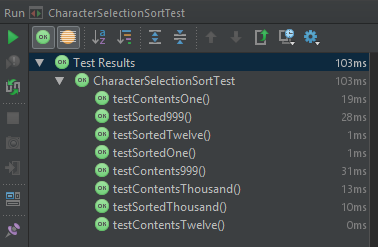
To test the selection sort’s contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

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.







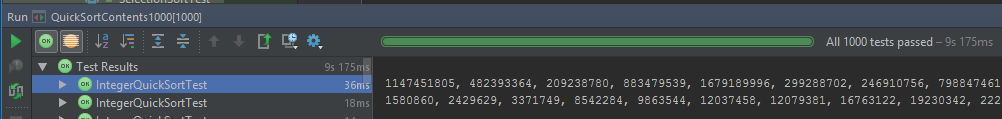


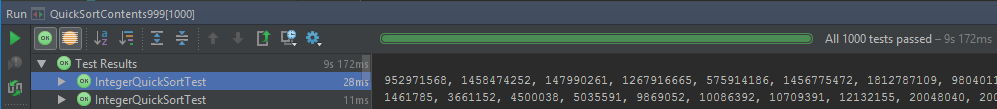
## (Logbook) Sorting Question 3: Implement the quicksort algorithm. Your Implementation should Implement the ArraySort Interface.

### Code Listing

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);  
 }  
 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;  
 }  
}

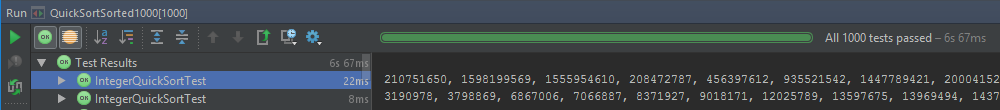
### Testing

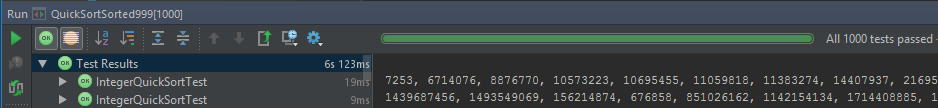


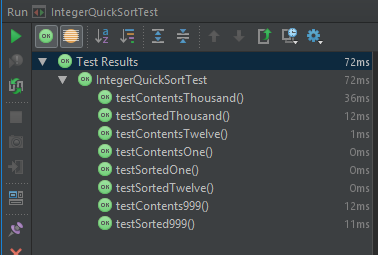


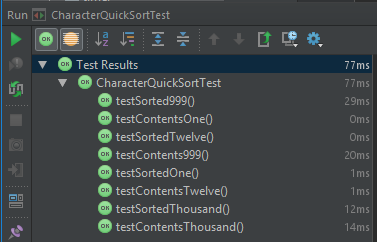
To test the quick sort’s contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

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.









## (Additional) Additional algorithms

### Insertion sort

#### Code Listing

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;  
 }  
}

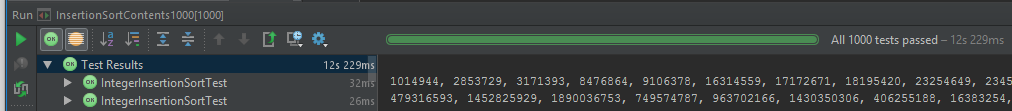
### Merge sort

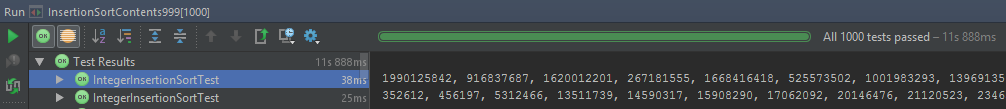
#### Code Listing

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);  
 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

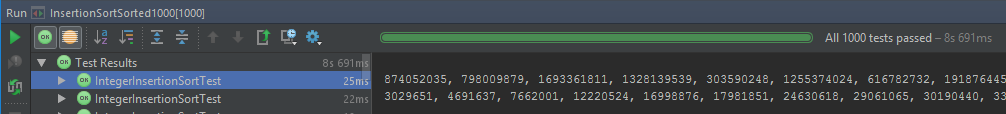
#### Insertion Sort.

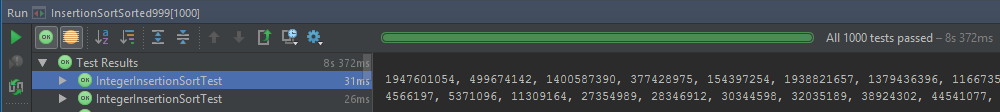


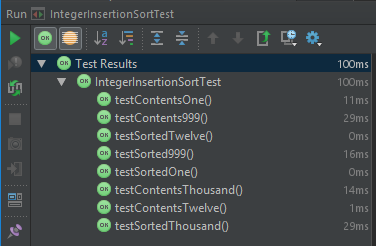


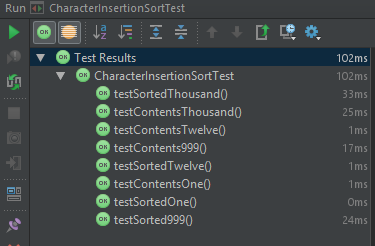
To test the Insertion sort’s contents before and after, I ran the contents test on a 1000 size array 1000 times. I also did the same with arrays of size 999 to test that arrays of an odd size also work. The first image is on 1000 size arrays, the second on 999 size arrays.

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.



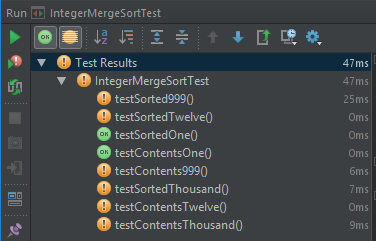




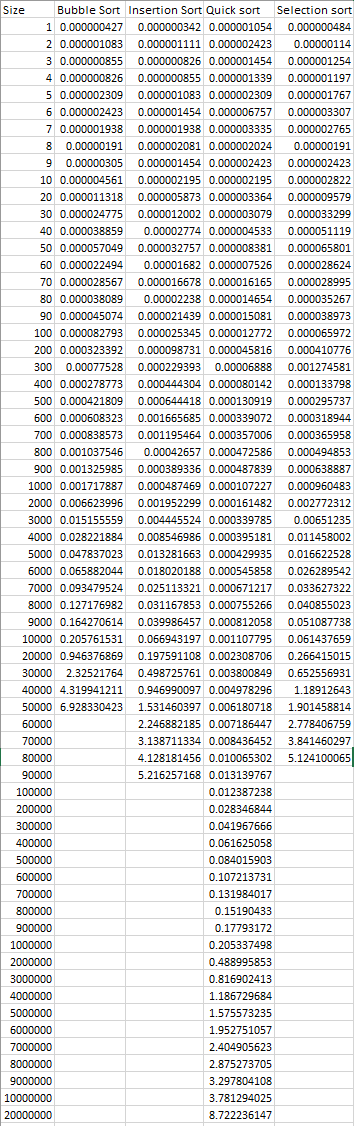


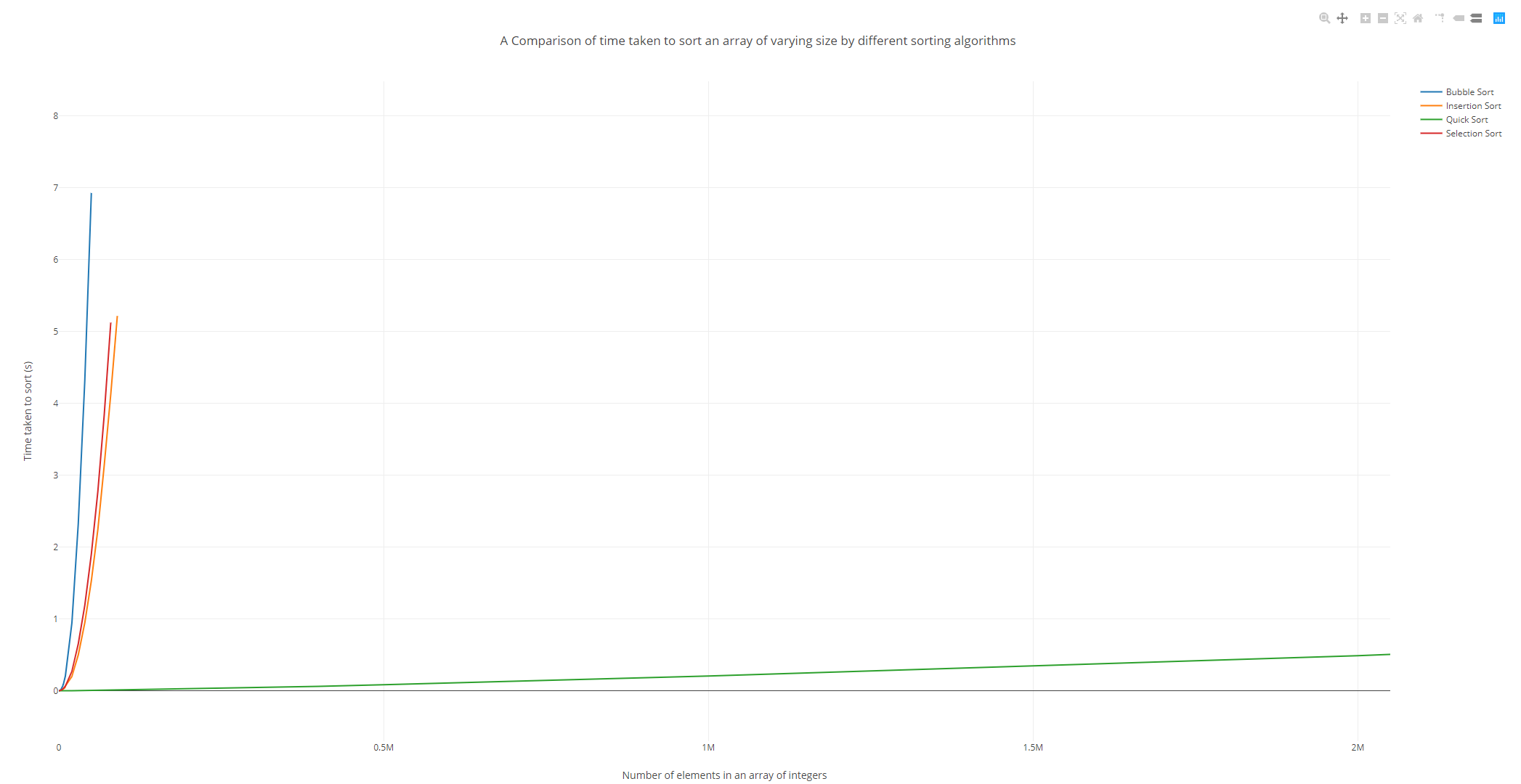
#### Merge Sort

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.



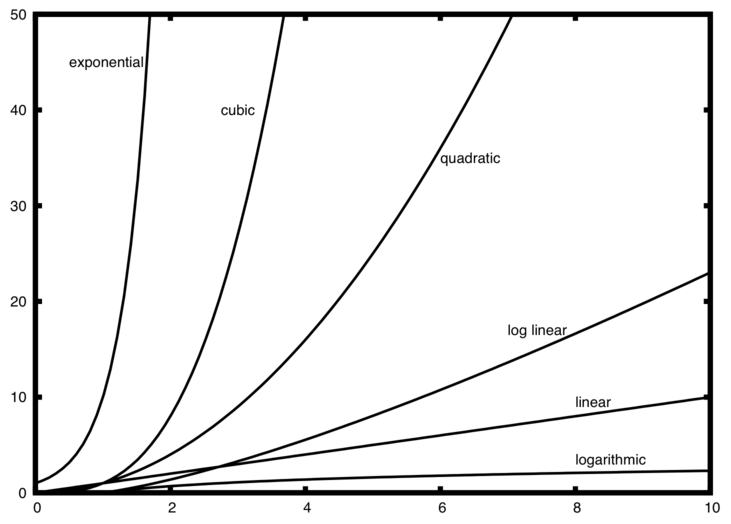
## (Logbook) Sorting Question 4: Use your implementations to time the execution of (at least) these two sorting algorithms for various sizes of array, and plot the results on a graph. Can you arrive at (approximate) formulae for how to execution times vary in relation to the data size?





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(n2) 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(n2) 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(n2) 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(n2) comparisons made, O(n2) swaps made.

Selection: O(n2) comparisons made, O(n) swaps made.

Insertion: O(n2) 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).

# Practical 4 (Week 7)

## (Logbook) Question 1: Implement the List<T> interface, using singly linked lists.

### Code Listing.

package linkedList.list;  
  
import linkedList.node.SingleLinkNode;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 14/11/2018.  
 \* Version: 19/11/2018  
 \*/*public class SingleLinkList<T> extends BasicList<SingleLinkNode<T>, T> implements List<T>  
{  
 */\*\*  
 \* Used to record the size of the list.  
 \*/* private int size;  
  
 */\*\*  
 \* Create an empty SingleLinkList.  
 \*/* public SingleLinkList()  
 {  
 root = null;  
 size = 0;  
 }  
  
 */\*\*  
 \* Create a SingleLinkList with one node (the value passed in).  
 \*  
 \** ***@param*** *value the value with which to create the root node of the SingleLinkList.  
 \*/* public SingleLinkList(T value)  
 {  
 root = new SingleLinkNode<T>(value);  
 size = 1;  
 }

*/\*\*  
 \* Returns the size of the list.  
 \*  
 \** ***@return*** *the size of the list.  
 \*/* public int getSize()  
 {  
 return this.size;  
 }

*/\*\*  
 \* Adds a new Value to the SingleLinkList at the index specified.  
 \*  
 \** ***@param*** *index the index at which the new entry should be added.  
 \** ***@param*** *value the value to be added.  
 \** ***@throws*** *ListAccessError if the index is invalid in respect to the size of the list.  
 \*/* @Override  
 public void add(int index, T value) throws ListAccessError  
 {  
 //Make sure the index trying to be accessed is not invalid.  
 if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }  
  
 //Increase the recorded size of the list.  
 size++;  
  
 //Special case for adding to the root.  
 if(index == 0)  
 {  
 setRoot(new SingleLinkNode<T>(value, getRoot()));  
 return;  
 }  
  
 //Initial states for finding the previous and next elements (the case for adding at index 1)  
 SingleLinkNode<T> previous = getRoot();  
  
 /\*  
 If not adding at index 2:  
 . Go through the list until the correct element prior to the element to be added is found.  
 . Then find the element after this.  
 . These two nodes will be used as the previous and next element for the element to be added.  
 \*/  
 for(int x = 2; x <= index; x++)  
 {  
 previous = previous.getNext();  
 }  
 SingleLinkNode<T> nextAfterNew = previous.getNext();  
  
 //Create the new listNode and set up its next element.  
 SingleLinkNode<T> newNode = new SingleLinkNode<>(value, nextAfterNew);  
  
 //Correct the next element of the previousNode.  
 previous.setNext(newNode);  
 }  
 */\*\*  
 \* Remove the element at the specified index and return the value.  
 \*  
 \** ***@param*** *index the index of the entry to be removed.  
 \** ***@return*** *the value of the element removed.  
 \** ***@throws*** *ListAccessError if the index is invalid in respect to the size of the list.  
 \*/* @Override  
 public T remove(int index) throws ListAccessError {  
 //Make sure the index trying to be accessed is not invalid.  
 if(index < 0 || index >= size) { throw new ListAccessError("Invalid index location " + index); }  
  
 //Initial states for finding the previous and the new index for the previous to point to  
 //(the case for removing the root node).  
 SingleLinkNode<T> previous = null;  
 SingleLinkNode<T> previousNewNext = getRoot().getNext();  
  
 //Get the value being removed before it becomes inaccessible via list modification.  
 T removed = get(index);  
  
 //The special case for removing the root node.  
 if(index == 0)  
 {  
 setRoot(previousNewNext);  
 size--;  
 return removed;  
 }

//If the root node is not the one to be removed, set up basic conditions for removing an internal element.  
 previous = getRoot();  
 previousNewNext = previousNewNext.getNext();  
  
 //Cycle through the list to find the correct values for previous and previousNewNext.  
 for(int x = 1; x < index; x++)  
 {  
 previous = previous.getNext();  
 previousNewNext = previousNewNext.getNext();  
 }  
  
 //Re-arrange the pointer for the previous element, and return the removed element.  
 previous.setNext(previousNewNext);  
 size--;  
 return removed;  
 }  
  
 */\*\*  
 \* Return the value of the element in the list at the index specified.  
 \*  
 \** ***@param*** *index the index of the entry to be accessed.  
 \** ***@return*** *the value of the element at the index specified.  
 \** ***@throws*** *ListAccessError if the index is invalid in respect to the size of the list.  
 \*/* @Override  
 public T get(int index) throws ListAccessError  
 {  
 //Make sure the index trying to be accessed is not invalid.  
 if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }  
  
 //Set up a storage variable for the node being accessed.  
 SingleLinkNode<T> get = getRoot();  
  
 //Cycle through the list until the correct element has been found.  
 for(int x = 1; x <= index; x++)  
 {  
 get = get.getNext();  
 }  
  
 //Return the element's value.  
 return get.getValue();  
 }  
}

### Test Class Code Listing.

package list;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 14/11/2018.  
 \* Version: 14/11/2018  
 \*/*import linkedList.list.ListAccessError;  
import linkedList.list.SingleLinkList;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
*/\*\*  
 \* A selection of test methods to test the SingleLinkList class and its methods.  
 \*/*public class SingleLinkListTest  
{  
 @Test  
 void testCreateSize0()  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>();  
  
 if(intList.getRoot() != null)  
 {  
 *fail*("Root is not null");  
 }  
 if(intList.getSize() != 0)  
 {  
 *fail*("Size is not 0");  
 }  
 }

@Test  
 void testCreateSize1()  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(2);  
  
 if(intList.getRoot() == null)  
 {  
 *fail*("Root is null for some reason");  
 }  
 if(intList.getSize() != 1)  
 {  
 *fail*("Size of list is not 1.");  
 }  
 }  
  
 @Test  
 void testInitialNodeValue()  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
  
 //List = {1}  
  
 if(intList.getRoot().getValue() != 1)  
 {  
 *fail*("Root value is not 1 for some reason");  
 }  
 }

@Test  
 void testSizeWorks() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 intList.add(3, 4);  
  
 //List = {1, 2, 3, 4}  
  
 if(intList.getSize() != 4)  
 {  
 *fail*("Size is not 4");  
 }  
  
 intList.remove(0);  
 //List = {2, 3, 4}  
 if(intList.getSize() != 3)  
 {  
 *fail*("Size not correctly updated to 3");  
 }  
  
 intList.remove(1);  
 //List = {2, 4}  
 if(intList.getSize() != 2)  
 {  
 *fail*("Size not correctly updated to 2");  
 }  
  
 intList.remove(1);  
 //List = {2}  
 if(intList.getSize() != 1)  
 {  
 *fail*("Size not correctly updated to 1");  
 }  
  
 intList.remove(0);  
 //List = {}  
 if (intList.getSize() != 0)  
 {  
 *fail*("Size not correctly updated to 0");  
 }  
 }  
  
 @Test  
 void testAddIndexNegativeIndex() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 try  
 {  
 intList.add(-1, 1);  
 *fail*("Index of -1 not caught.");  
 }  
 catch (ListAccessError e) {}  
 }  
  
 @Test  
 void testAddIndexGreaterThanSize() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 try  
 {  
 intList.add(2, 1);  
 *fail*("Index of 2 not caught");  
 }  
 catch(ListAccessError e) {}  
 }  
  
 @Test  
 void testAddToEmptyListRootValue() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>();  
 //List = {}  
 intList.add(0, 1);  
 //List = {1}  
 if(intList.getRoot().getValue() != 1)  
 {  
 *fail*("Root value is not 1 for some reason.");  
 }  
 }

@Test  
 void testAddToListSize1AtRoot() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 //List = {1}  
 intList.add(0, 2);  
 //List = {2, 1}  
 if(intList.get(0) != 2)  
 {  
 *fail*("Root value is not 2");  
 }  
 if(intList.get(1) != 1)  
 {  
 *fail*("Root value of 1 was not moved up to index 1");  
 }  
 }  
  
 @Test  
 void testAddToListSize1AfterRoot() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 //List = {1}  
 intList.add(1, 2);  
 //List = {1, 2}  
 if(intList.get(0) != 1)  
 {  
 *fail*("Root value is no longer 1");  
 }  
 if(intList.get(1) != 2)  
 {  
 *fail*("Value of 2 not correctly added to index 1");  
 }  
 }

@Test  
 void testAddToGenericInternalIndex() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
 if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)  
 {  
 *fail*("List not set up correctly for test.");  
 }  
  
 intList.add(1, 10);  
 //List = {1, 10, 2, 3}  
 if(intList.get(0) != 1)  
 {  
 *fail*("Root node is no longer 1");  
 }  
  
 if(intList.get(1) != 10)  
 {  
 *fail*("10 not added to index 1");  
 }  
 if(intList.get(2) != 2)  
 {  
 *fail*("2 not correctly moved to index 2");  
 }  
 if(intList.get(3) != 3)  
 {  
 *fail*("3 not correctly moved to index 3");  
 }  
  
 if(intList.getSize() != 4)  
 {  
 *fail*("Size not correctly updated to 4.");  
 }  
 }

@Test  
 void testAddToEndOfList() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
 if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)  
 {  
 *fail*("List not set up correctly for test.");  
 }  
  
 intList.add(3, 10);  
 //List = {1, 2, 3, 10}  
  
 if(intList.get(2) != 3)  
 {  
 *fail*("3 no longer at correct index of 2");  
 }  
 if(intList.get(3) != 10)  
 {  
 *fail*("10 not added to end of list correctly.");  
 }  
  
 if(intList.getSize() != 4)  
 {  
 *fail*("Size not correctly updated to 4.");  
 }  
 }  
  
 @Test  
 void testRemoveIndexNegativeIndex() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 try  
 {  
 intList.remove(-1);  
 *fail*("Index of -1 not caught.");  
 }  
 catch (ListAccessError e) {}  
 }

@Test  
 void testRemoveIndexEqualToSize() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 try  
 {  
 intList.remove(1);  
 *fail*("Index of 1 not caught");  
 }  
 catch(ListAccessError e) {}  
 }  
  
 @Test  
 void testRemoveIndexGreaterThanSize() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 try  
 {  
 intList.remove(2);  
 *fail*("Index of 2 not caught");  
 }  
 catch(ListAccessError e) {}  
 }  
  
 @Test  
 void testRemoveFromListSize1AtRoot() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 //List = {1}  
 int removed = intList.remove(0);  
 //List = {}  
 if(removed != 1)  
 {  
 *fail*("Removed element value 1 not correctly returned.");  
 }  
 if(intList.getRoot() != null)  
 {  
 *fail*("root not removed correctly");  
 }  
 }

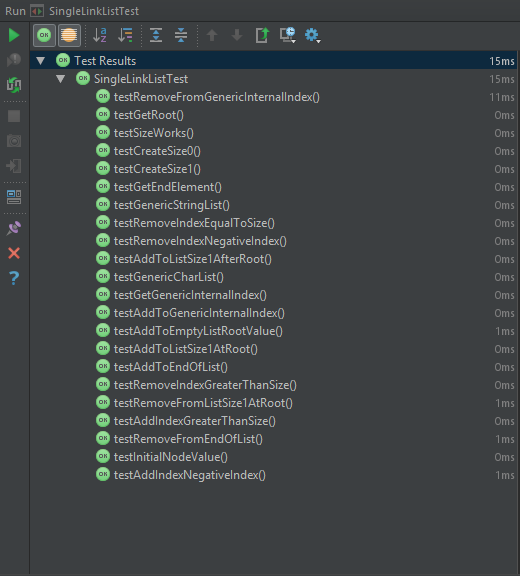
@Test  
 void testRemoveFromGenericInternalIndex() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
 if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)  
 {  
 *fail*("List not set up correctly for test.");  
 }  
  
 int removed = intList.remove(1);  
 //List = {1, 3}  
 if(removed != 2)  
 {  
 *fail*("Removed element value 2 not correctly returned");  
 }  
 if(intList.get(0) != 1)  
 {  
 *fail*("Root node is no longer 1");  
 }  
  
 if(intList.get(1) != 3)  
 {  
 *fail*("3 not correctly moved down to index 1");  
 }  
  
 if(intList.getSize() != 2)  
 {  
 *fail*("Size not correctly updated to 2.");  
 }  
 }

@Test  
 void testRemoveFromEndOfList() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
 if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)  
 {  
 *fail*("List not set up correctly for test.");  
 }  
  
 int removed = intList.remove(2);  
 //List = {1, 2}  
  
 if(removed != 3)  
 {  
 *fail*("Removed element value 3 not correctly returned.");  
 }  
  
 if(intList.get(1) != 2)  
 {  
 *fail*("2 no longer at correct index of 1");  
 }  
  
 if(intList.getSize() != 2)  
 {  
 *fail*("Size not correctly updated to 2.");  
 }  
 }

@Test  
 void testGetRoot() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
  
 if(intList.get(0) != 1)  
 {  
 *fail*("Root element value 1 not correctly returned.");  
 }  
 }  
  
 @Test  
 void testGetGenericInternalIndex() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
  
 if(intList.get(1) != 2)  
 {  
 *fail*("Element value 2 not correctly returned");  
 }  
 }  
  
 @Test  
 void testGetEndElement() throws ListAccessError  
 {  
 SingleLinkList<Integer> intList = new SingleLinkList<>(1);  
 intList.add(1, 2);  
 intList.add(2, 3);  
 //List = {1, 2, 3}  
  
 if(intList.get(2) != 3)  
 {  
 *fail*("End element value 3 not correctly returned.");  
 }  
 }

@Test  
 void testGenericCharList() throws ListAccessError {  
 SingleLinkList<Character> charList = new SingleLinkList<>('a');  
 charList.add(1, 'b');  
 charList.add(2, 'c');  
 //List = {a, b, c}  
 if(charList.getSize() != 3)  
 {  
 *fail*("Size of 3 not correctly returned.");  
 }  
  
 charList.add(0, 'z');  
 //List = {z, a, b, c}  
 if(charList.getSize() != 4)  
 {  
 *fail*("list size not correctly updated to 4");  
 }  
 if(charList.get(0) != 'z')  
 {  
 *fail*("Root element not correctly changed to z");  
 }  
 if(charList.get(1) != 'a')  
 {  
 *fail*("root element a not correctly moved to index 1");  
 }  
 char removed = charList.remove(2);  
 //List = {z, a, c}  
 if(charList.getSize() != 3)  
 {  
 *fail*("List size not correctly updated to 3");  
 }  
 if(charList.get(1) != 'a')  
 {  
 *fail*("Previous element no longer a");  
 }  
 if(charList.get(2) != 'c')  
 {  
 *fail*("Next element no longer c");  
 }  
 if(removed != 'b')  
 {  
 *fail*("Removed element value b not correctly returned.");  
 }  
 }  
 @Test  
 void testGenericStringList() throws ListAccessError {  
 SingleLinkList<String> stringList = new SingleLinkList<>("aaa");  
 stringList.add(1, "bbb");  
 stringList.add(2, "ccc");  
 //List = {aaa, bbb, ccc}  
 if(stringList.getSize() != 3)  
 {  
 *fail*("Size of 3 not correctly returned.");  
 }  
  
 stringList.add(0, "zzz");  
 //List = {zzz, aaa, bbb, ccc}  
 if(stringList.getSize() != 4)  
 {  
 *fail*("list size not correctly updated to 4");  
 }  
 if(!(stringList.get(0).equals("zzz")))  
 {  
 *fail*("Root element not correctly changed to zzz");  
 }  
 if(!(stringList.get(1).equals("aaa")))  
 {  
 *fail*("root element aaa not correctly moved to index 1");  
 }  
 String removed = stringList.remove(2);  
 //List = {zzz, aaa, ccc}  
 if(stringList.getSize() != 3)  
 {  
 *fail*("List size not correctly updated to 3");  
 }  
 if(!(stringList.get(1).equals("aaa")))  
 {  
 *fail*("Previous element no longer aaa");  
 }  
 if(!(stringList.get(2).equals("ccc")))  
 {  
 *fail*("Next element no longer ccc");  
 }  
 if(!(removed.equals("bbb"))) {  
 *fail*("Removed element value bbb not correctly returned.");  
 }  
 }  
}

### Result of testing.



### Self Evaluation.

The marking scheme lists 5 marks for a solution having Boundary checking and exceptions with the inclusion of a test suite.

My implementation implements boundary checking with exceptions at the start of all methods and includes a comprehensive test suite. Evidence for both can be found in the above three headings. My implementation further more includes boundary checking as part of its optimisation, reducing the number of checks and re-allocations required. This improves the efficiency of the implementation and allows it to scale up much further than other implementations.

## (Additional) Question 1.5: Model answer comparison.

### Model answer for Get method.

#### Code Listing.

package linkedList.list;  
  
import linkedList.node.ListNode;  
import linkedList.node.SingleLinkNode;  
  
*/\*\*  
 \* A partial implementation of the List interface.  
 \* This implementation only implements the T get(int index) method, and the class must, therefore  
 \* be declared abstract.  
 \*  
 \** ***@param*** <*T*> *the type of object stored in the list.  
 \*/*public abstract class SingleLinkListModel<T> extends BasicList<SingleLinkNode<T>,T> implements List<T> {  
  
 */\*\*  
 \* A helper method to access a node at a specified index.  
 \*  
 \** ***@param*** *index the index of the node to be accessed.  
 \** ***@throws*** *ListAccessError if there is no node with the given index.  
 \*/* ListNode<T> getNode(int index) throws ListAccessError {  
 // Is the list empty? If so, cannot access the node.  
 if (isEmpty()) {  
 throw new ListAccessError("Cannot get node. List is empty.");  
 }  
 // Is the given index negative? If so, this is an error.  
 if (index < 0) {  
 throw new ListAccessError("Cannot get node. Negative index.");  
 }  
 /\*  
 \* Try to find the specified node by "walking" through the list, following links to successor  
 \* nodes. The index tells us how many links need to be followed to reach the required node,  
 \* so reduce the index by one each time a link is followed. When the index reaches zero, the  
 \* required node has been found. If the end of the list is reached (next node is null), before  
 \* the index reaches zero, there were not enough nodes in the list (the index was too high).  
 \*/  
 ListNode<T> currentNode = getRoot(); // start at the root  
 while (index != 0 && currentNode != null) { // walk along the list (if haven't reached the end by hitting null node)  
 currentNode = currentNode.getNext(); // by gettting next node in the list  
 index--; // and reducing index by one  
 }  
 // Reached the end of the list (by hitting null node)? If so, cannot access the required node.  
 if (currentNode == null) {  
 throw new ListAccessError("Cannot get node. Not enough nodes in the list.");  
 }  
 // Successfully found node by walking through until index was zero.  
 return currentNode;  
 }  
  
 */\*\*  
 \* Access the value at a given index.  
 \*  
 \** ***@param*** *index the index of the value to be accessed.  
 \** ***@throws*** *ListAccessError if there is no value with the given index.  
 \*/* public T get(int index) throws ListAccessError {  
 return getNode(index).getValue();  
 }  
}

### My Implementation for Get method.

#### Code Listing.

*/\*\*  
 \* Return the value of the element in the list at the index specified.  
 \*  
 \** ***@param*** *index the index of the entry to be accessed.  
 \** ***@return*** *the value of the element at the index specified.  
 \** ***@throws*** *ListAccessError if the index is invalid in respect to the size of the list.  
 \*/*@Override  
public T get(int index) throws ListAccessError  
{  
 //Make sure the index trying to be accessed is not invalid.  
 if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }  
  
 //Set up a storage variable for the node being accessed.  
 SingleLinkNode<T> get = getRoot();  
  
 //Cycle through the list until the correct element has been found.  
 for(int x = 1; x <= index; x++)  
 {  
 get = get.getNext();  
 }  
  
 //Return the element's value.  
 return get.getValue();  
}

### Comparison.

One thing that is noticeable straight off the bat is the larger exception handling that the model answer includes at the top of the function. Despite the extra lines, the model answer handles initial exceptions in a slightly more user friendly way, giving the user/programmer information of the exact problem they’ve encountered, rather than my method which simply returns the invalid index.

The model answer also does exception handling during the ‘walk-through’ to find the node requested, whereas as mine does not. This is because my implementation for the SingleLinkList adds a recorded size of the list which is used during the initial error checking to make sure the programmer/user is not trying to access a data location past the end of the list. The model answer does not have this ‘size’ recording, therefore cannot do this check at the start of the method, bulking out the code and making it slightly less readable.

The perfect combination of error checking would be the inclusion of my size recording and initial error checking handling everything along with the model implementations user friendly exception reporting.

The exclusion of this error handling at the start has impacts upon the efficiency of the model solution, as when finding the correct node, additional checks have to be done each time a new element is attempted to be accessed. This check makes sure that the next element is not null before attempting to access it. My implementation however, can simply walk through without any possibility of reaching a null node. This is objectively better than the model implementation’s approach as it improves code readability & maintainability and improves efficiency via reducing the amount of brute force checks that have to be made during the walk-through.

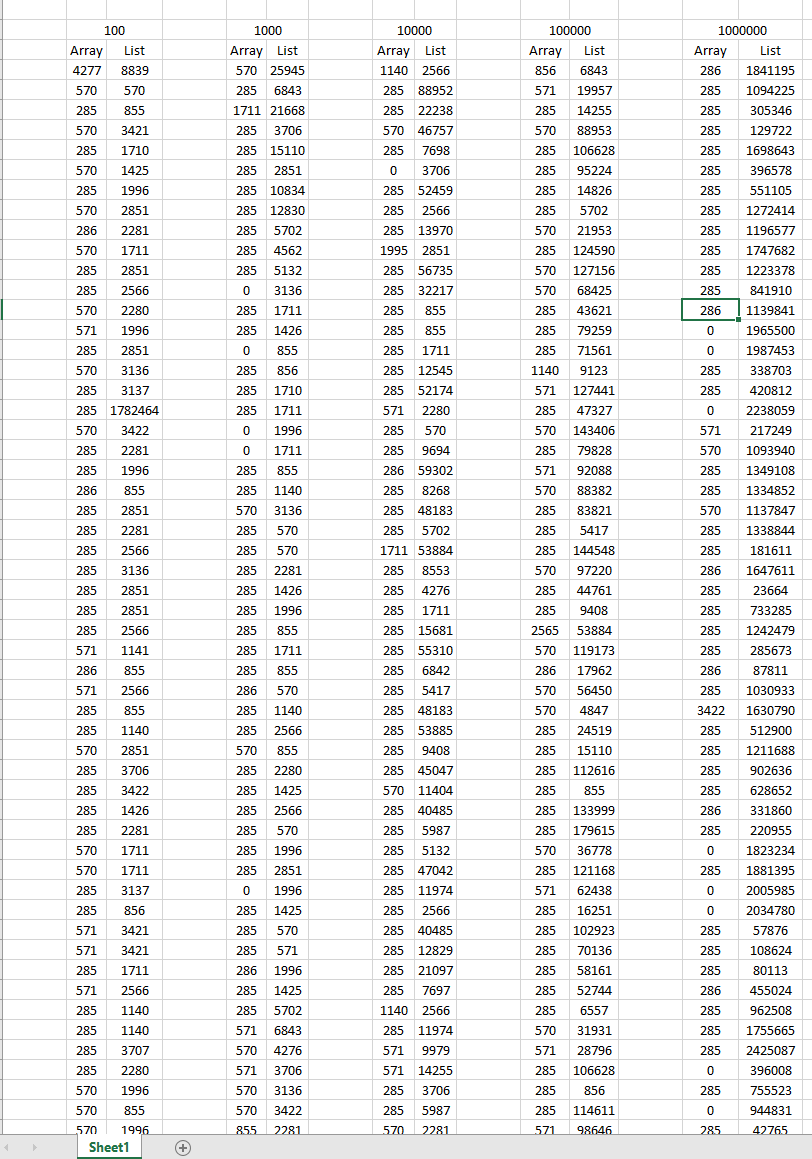
In conclusion, I’d state that the model answer outweighs my solution in terms of user experience, allowing easier bugfixing and smoother usage. This could arguably be a key factor given the abstract implementation nature of these exercises. However, my implementation takes the lead where efficiency is concerned. Despite this efficiency lead being very small given the computational power of systems today, my implantation would scale far better into a system the likes of which are developed and used today.

## (Additional) Question 2: Write some test code that uses array generators to create large random arrays. Use the values in these arrays to populate instances of your implementation of linked lists. Now attempt multiple accesses of the data both in the arrays and in the lists.

### Code Listing.

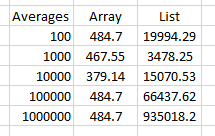
package Comparison;  
  
import arrayGenerator.generator.IntegerArrayGenerator;  
import arrayGenerator.scope.IntegerScope;  
import linkedList.list.ListAccessError;  
import linkedList.list.SingleLinkList;  
  
import java.util.Random;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 19/11/2018.  
 \* Version : 20/11/2018  
 \*/*public class SingleLinkListComparison  
{  
 private static final int *NUM\_TESTS* = 100;  
 private static final int *SIZE\_OF\_ARRAY* = 100;  
  
 private static void timeComparison() throws ListAccessError  
 {  
 //Create a new random instance to obtain random indices.  
 Random rand = new Random();  
  
 //Set up an iterator to gradually increase the size of the array.  
 for(int arraySize = *SIZE\_OF\_ARRAY*; arraySize <= 1000000; arraySize \*= 10 )  
 {  
 //Create a new integer array based on the specified size of array.  
 Integer[] ints = new IntegerArrayGenerator(new IntegerScope()).getArray(arraySize);  
  
 //Use this array to populate a SingleLinkList.  
 SingleLinkList<Integer> list = new SingleLinkList<>();  
 for(int x = 0; x < arraySize; x++)  
 {  
 list.add(x, ints[x]);  
 }  
  
 System.*out*.println("Testing array of size: " + arraySize);  
  
 //Repeat this however many times specified by num\_tests.  
 for (int x = 0; x < *NUM\_TESTS*; x++)  
 {  
 //Obtain a random index within the array/list.  
 int i = rand.nextInt(arraySize);  
  
 //Time how long it takes to access the array and print this value.  
 double before = System.*nanoTime*();  
 int arrayInt = ints[i];  
 System.*out*.println("System took " + ((System.*nanoTime*()) - before) + " ns. to access the array");  
  
 //Time how long it takes to access the list and print this value.  
 before = System.*nanoTime*();  
 int listInt = list.get(i);  
 System.*out*.println("System took " + ((System.*nanoTime*()) - before) + " ns. to access the list.");  
 }  
  
 System.*out*.println();  
 }  
 }  
  
 public static void main(String[] args)  
 {  
 try  
 {  
 SingleLinkListComparison.*timeComparison*();  
 }  
 catch (ListAccessError listAccessError)  
 {  
 listAccessError.printStackTrace();  
 }  
 }  
}

### Test data.

This is an example of the test data I acquired from running the code listing above. There are 100 rows of recordings for each array size, however I have abbreviated the table size in the interests of brevity.

### Averages.

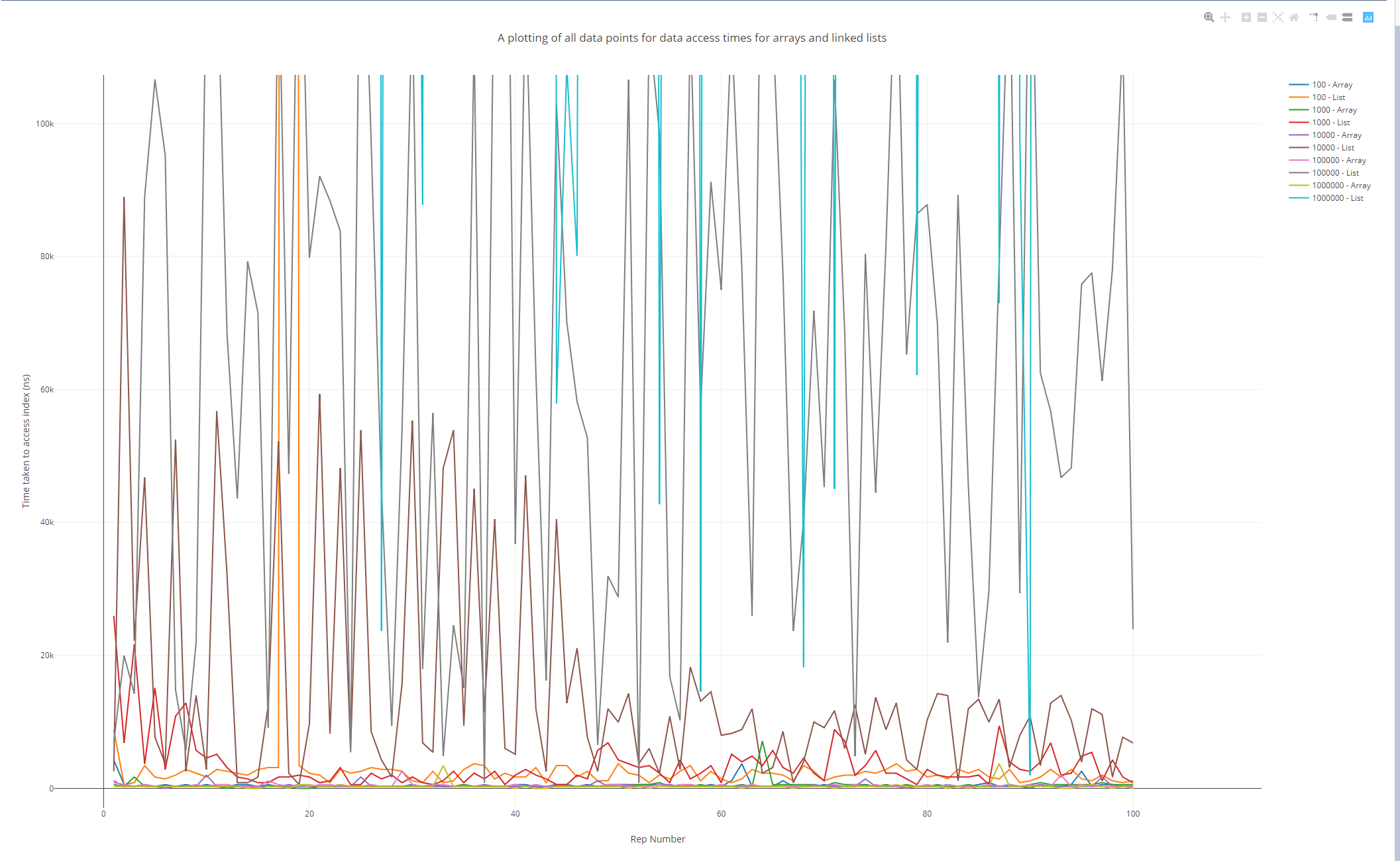
From this data I calculated the average access time for each data structure for each size of array tested.

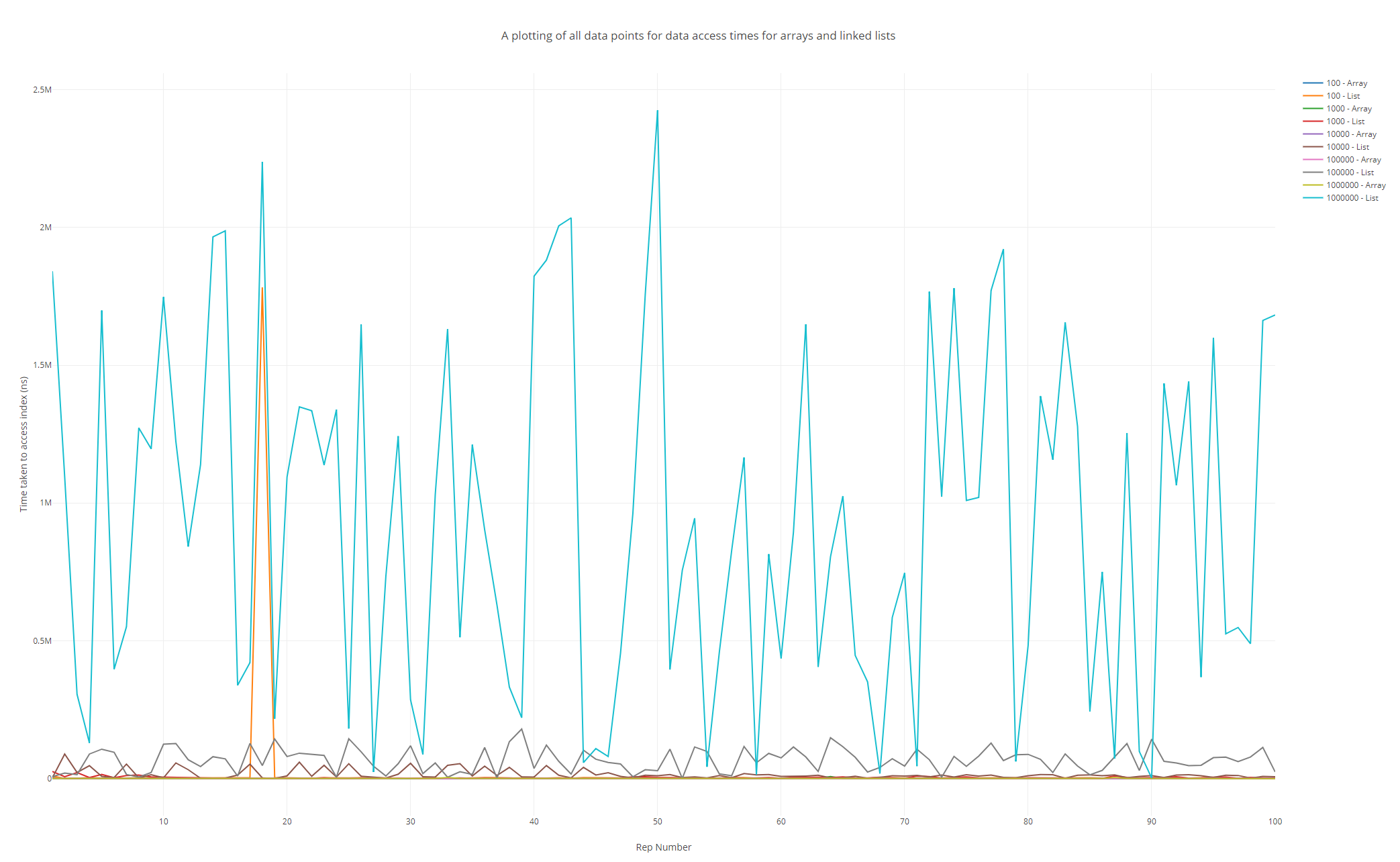


### Graph plotting.

#### All data points

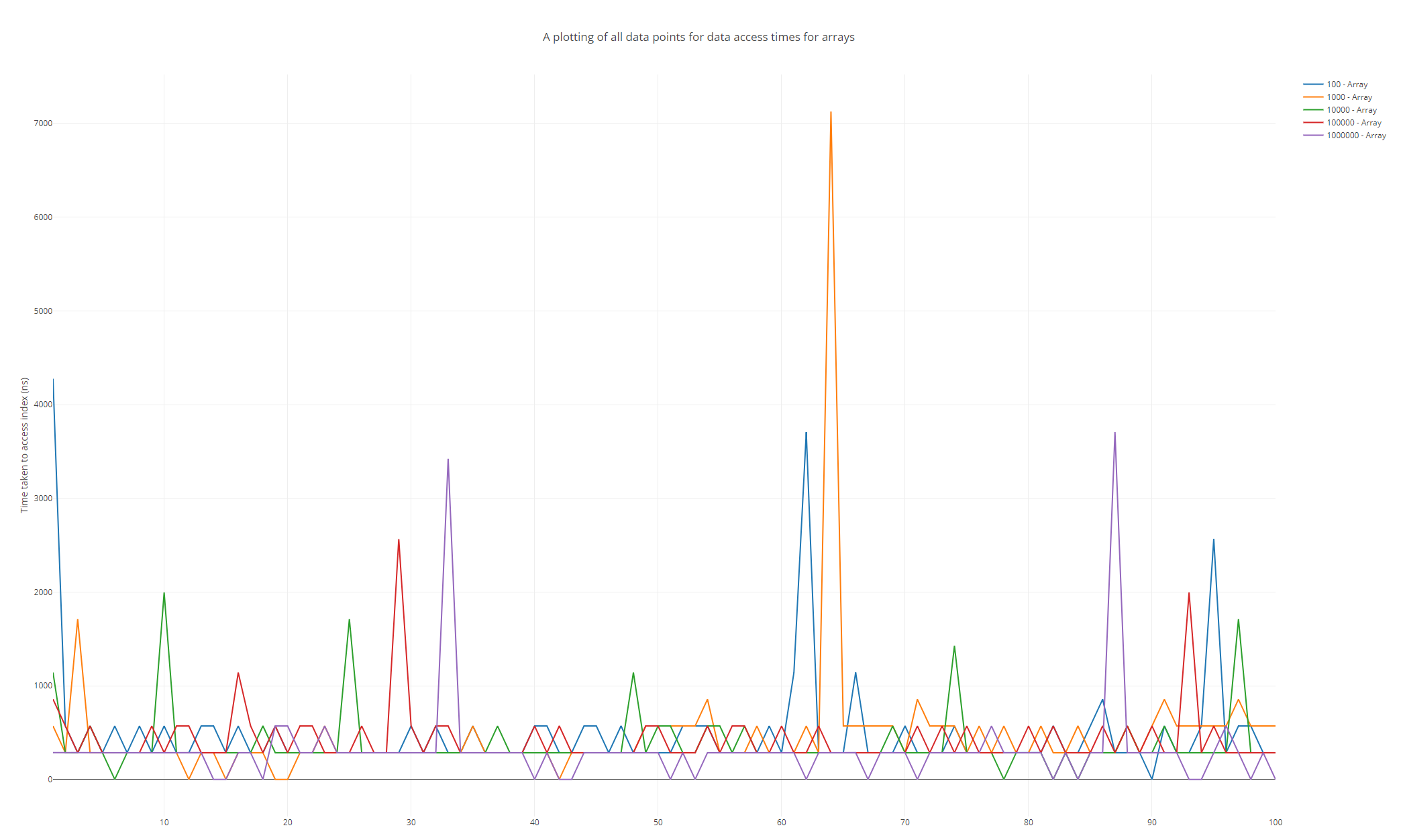
This graph shows every data point plotted. From this scale it is easy to see the constant access time of arrays from the traces near the bottom of the graph.





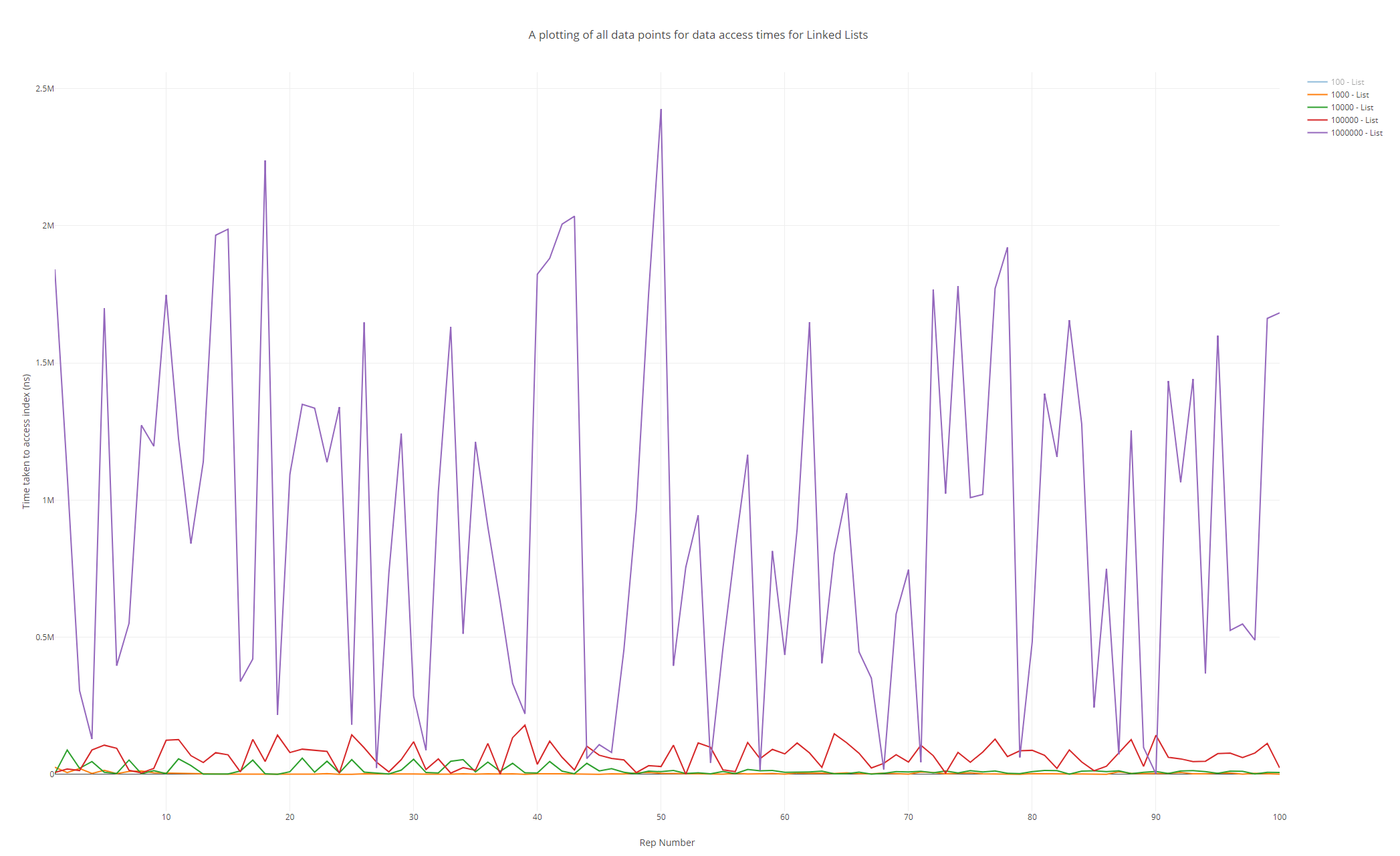
When the graph is scaled like this however, it becomes much easier to grasp just how badly the Linked List performs at higher element counts. The blue line on the graph plots the times for a Linked List of 1,000,000 elements and absolutely dwarfs everything beneath it. In comparison, the array trace for the same number of elements retains its constant access time.

#### Array time comparisons.



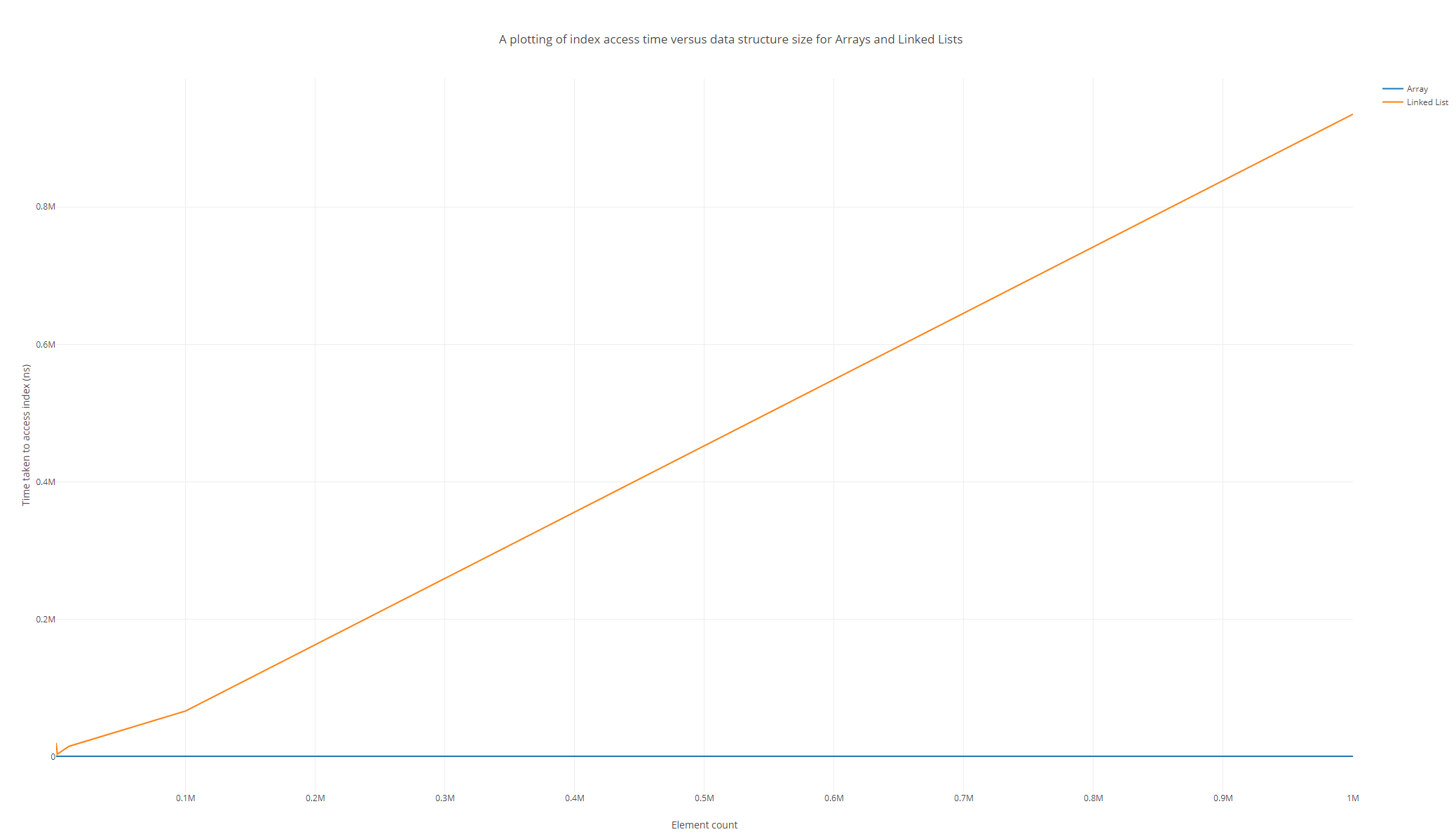
This graph shows the times taken by the array to access the same data as the Linked List. As can be seen, the highest time recorded to access an index was taken by an array of size 1000. This proves that arrays are not affected by size in the same way that Linked Lists are when it comes to access time. Interestingly, the graph’s visibly discrete Y values display the constant access time quite well, as there are no analogue values.

#### Linked List time comparisons.



This graph shows the times taken by Linked Lists to access the same data as the arrays. Each trace visibly increases its average up the Y axis, proving the Linked List dependence on element count. This graph, in stark contrast to the graph plotted for the array, shows the continuous/analogue performance of the Linked List. This also suggests a dependence on computer performance that is much higher than the arrays. This has an upper limit and Linked Lists will always be slower than arrays however, as Linked Lists still have to do operations before accessing a memory address whereas arrays can almost directly access a memory address. This means that however fast a CPU gets at processing these instructions, a Linked List will **always** be slower than an array for the same tasks.

#### Average Plots.

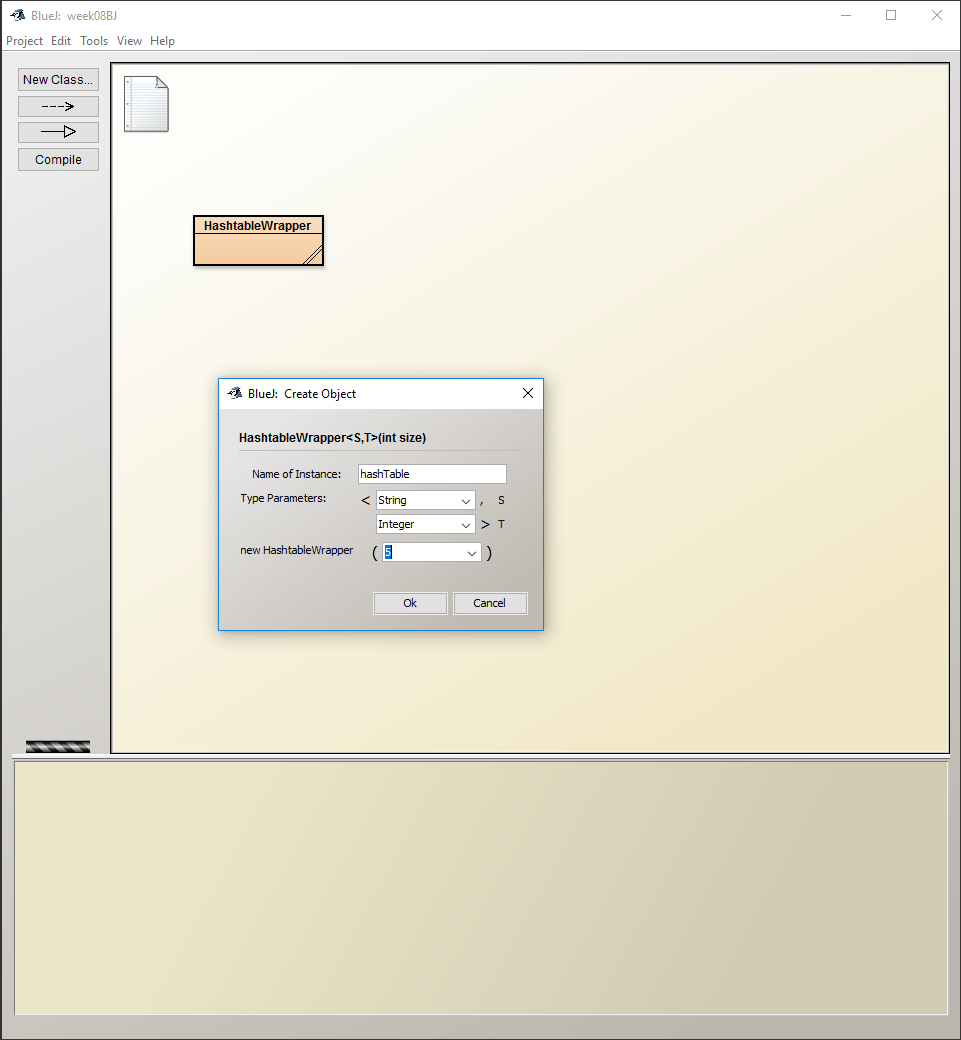


Finally, this graph (plotting the average access time per element count) shows the constant access time for Arrays very well. The graph may also suggest a linear relationship between data structure size and access time, however I am reluctant to outright state this as there are only 5 data point plotted on this graph, and the majority of the graph is taken up by the difference between 100,000 and 1,000,000 points.

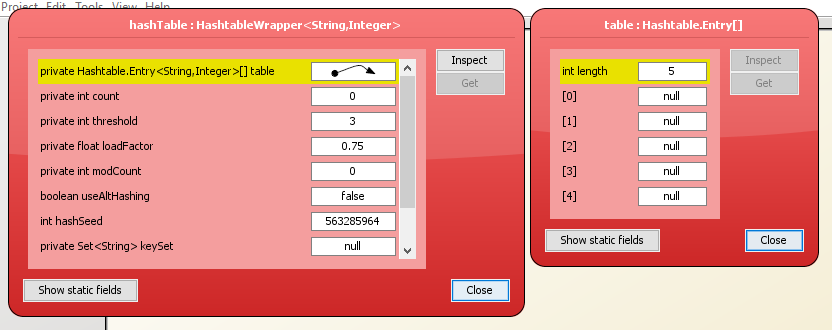
# Practical 5 (Week 8)

## (Logbook) Question1: Create an object instance of the HashtableWrapper(String, Integer) class.

### Ensure this hash table uses Strings as keys, Integers as values, and has an initial size of 5.

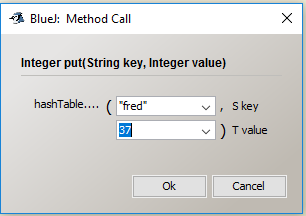


### Inspect the object you have just created, paying particular attention to the object’s internal array.



Some fields of note in this inspection might be the count of 0, (Stating the number of elements in the array) and each field in the internal array, showing that each entry is null and has nothing contained within.

### Now, using the void put(String key, Integer data) method, add the key/value pair (“fred”, 37) to the hashtable.



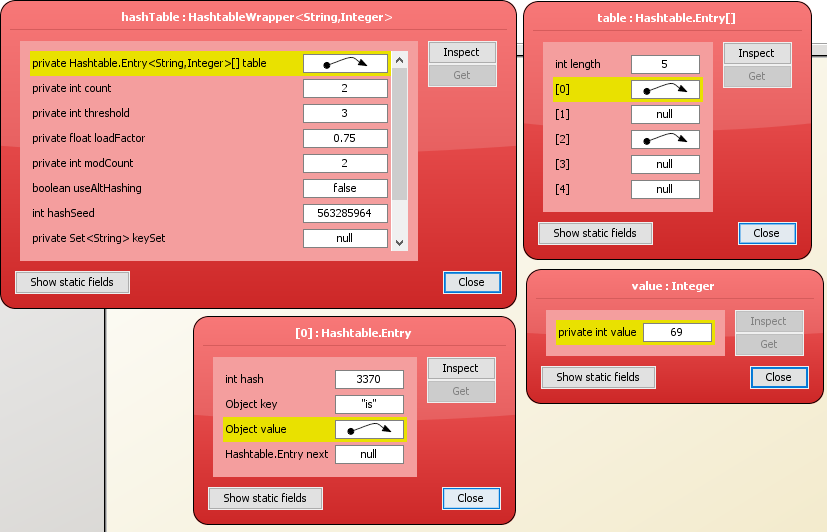
### Inspect the object again.



Fields of note here: The count field has changed to 1 to signify there is one element within the internal array. The internal array at index 2 contains a pointer to a ‘Hashtable Entry’ which contains the hash, key, next entry (for collisions) and a pointer to the actual value held. The modcount field has increased to 1.

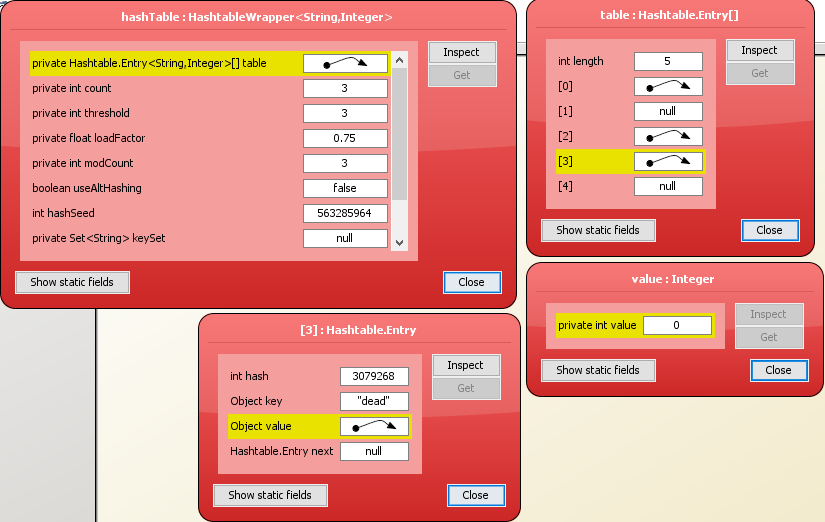
### Now add the following key/value pairs, again inspecting the hashtable object after each new pair is entered.

#### (“is”, 69)



The count field has changed to 2. The modcount field has increased to 2. Index 0 in the internal array contains a hashtable entry for (“is”, 69).

#### (“dead”, 0)



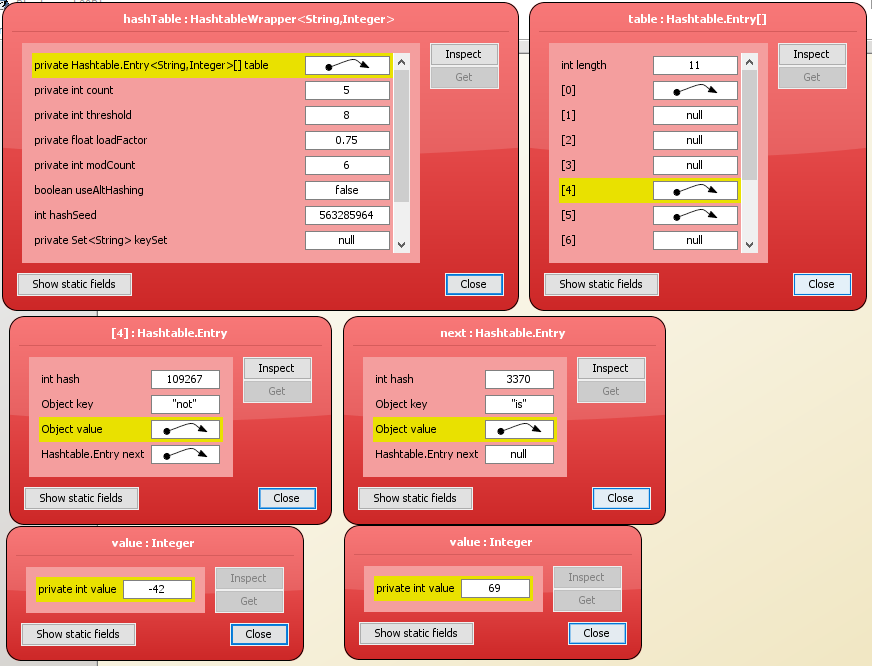
The count field has increased to 3, the modcount field has increased to 3. Index 3 contains a pointer to a hashtable entry for (“dead”, 0).

|  |  |  |
| --- | --- | --- |
| Value | Old Index | New Index |
| (“fred”, 37) | 2 | 0 |
| (“is”, 69) | 0 | 4 |
| (“dead”, 0) | 3 | 5 |

#### (“but”, 999)

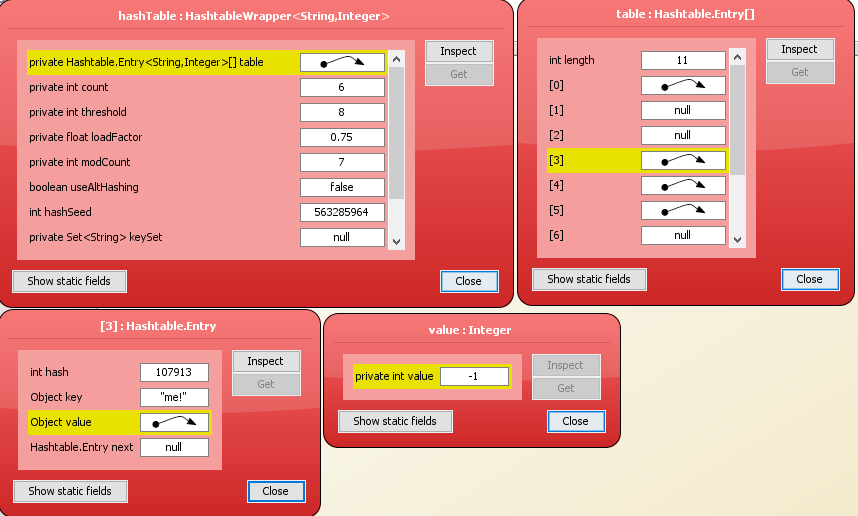
The count field has increased to 4. The threshold field has increased to 8. This is perhaps due to count reaching the value of threshold, upon which the hashtable realises it needs to increase the size of its internal array. The value of modCount has increased to 5. The hashtable entries have also been moved around, however I’m unsure exactly as to how the Hashtable class has determined this.

#### (“not”, -42)



Count has increased to 5. Modcount has increased to 6. The new hashtable entry has fallen on the index of (“is”, 69), so the new hashtable entry has taken its place, and kicked (“is”, 69) down a place so that (“not”, -42) is the first visible entry in the internal array and (“is”, 69) resides at the second layer.

#### (“me!”, -1)



Count has increased to 6. Modcount has increased to 7. Otherwise, (“me!”, -1) has been treated very much like a standard hashtable entry, taking the until now empty index of 3.

### Additional explanation.

For each key-value pair, the key is passed through the hashing function of the hashtable, to gain an integer hash value. This is used as an index (Wrapped around using a modulus) to find the position of the hashtable entry.

The java Hashtable class is an implementation of an *open* hashtable, which means that when collisions occur, the hashtable uses *buckets* to store multiple entries (searched seququentially) under one index value. An example of this behaviour is when (“not”, -42) is added.

The loadFactor variable is a measure of how full the hash table is allowed to get before its capacity is automatically increased. 0.75 (used here) is the default and offers ‘a good tradeoff between time and space costs’. It’s obvious to see that a lower load factor means that the array will be expanded more often, therefore there is less chance of a collision. However, the array is larger and takes up more space in memory. On the flipside, having a larger loadFactor means that the array will be expanded less often, resulting in less space in memory used, but larger chance for collisions to occur. This results in more buckets and sequential searches, diminishing the advantages the hashtable provides.

When the size of the array is dynamically increased, the entire array is rehashed. This explains the behaviour of the hashtable entries moving around when (“but”, 999) is added. Rehashing is a relatively time consuming operation however. If a lot of entries are going to be made into the hashtable, then it can sometimes be better to create the hashtable with a larger initial capacity so that rehashing is less likely to occur.

The Java Hashtable documentation states that **no** rehashing will ever occur **if** the **initial capacity** is **greater than** the (**maximum entries to contain** / **load factor**)

The only danger of this is setting the initial capacity too high, which can waste space if a lot of duplicate entries are added.

## Self Evaluation

I believe I provide more than just a sequence of screenshots, providing a step by step description then an extended analysis at the end of the exercise. I believe my analysis explains how internal array slots are allocated and provides a look at the differences in choice where loadFactor and initialCapacity are concerned. I believe my analysis is full and takes everything into consideration. Therefore, I would self-evaluate this week’s exercises as a 5/5.

# Practical 6 (Week 9)

## (Logbook) Question 1: Complete the implementation of the binary tree class.

### Code Listing

package binaryTree;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class BinaryTree<T extends Comparable<? super T>> implements BTree<T> {  
  
 private TreeNode<T> root; // the root node  
  
 */\*\*  
 \* Construct an empty tree.  
 \*/* public BinaryTree() {  
 root = null;  
 }  
  
 */\*\*  
 \* Construct a singleton tree.  
 \* A singleton tree contains a value in the root, but the left and right subtrees are  
 \* empty.  
 \** ***@param*** *value the value to be stored in the tree.  
 \*/* public BinaryTree(T value) {  
 root = new TreeNode(value);  
 }  
  
 */\*\*  
 \* Construct a tree with a root value, and left and right subtrees.  
 \** ***@param*** *value the value to be stored in the root of the tree.  
 \** ***@param*** *left the tree's left subtree.  
 \** ***@param*** *right the tree's right subtree.  
 \*/* public BinaryTree(T value,BinaryTree<T> left,BinaryTree<T> right) {  
 root = new TreeNode(value,left,right);  
 }

*/\*\*  
 \* Check if the tree is empty.  
 \** ***@return*** *true iff the tree is empty.  
 \*/* @Override  
 public boolean isEmpty() {  
 return root == null;  
 }  
  
 */\*\*  
 \* Insert a new value in the binary tree at a position determined by the current contents  
 \* of the tree, and by the ordering on the type T.  
 \** ***@param*** *value the value to be inserted into the tree.  
 \*/* @Override  
 public void insert(T value) {  
 if(isEmpty())  
 {  
 root = new TreeNode<T>(value, new BinaryTree<>(), new BinaryTree<>());  
 return;  
 }  
  
 if(value.compareTo(this.getValue()) < 0)  
 {  
 root.getLeft().insert(value);  
 }  
 else  
 {  
 root.getRight().insert(value);  
 }  
 // implement insert(T value) here  
 }

*/\*\*  
 \* Get the value stored at the root of the tree.  
 \** ***@return*** *the value stored at the root of the tree.  
 \*/* @Override  
 public T getValue() throws NullPointerException {  
 // Note: it might make sense to define getValue() to throw a (custom) exception if an attempt  
 // is made to access a value from an empty tree.  
 // However, since a tree is empty iff it's root node is null, it is also acceptable to rely  
 // on Java's NullPointerException.  
 // This comment also applies to the other get and set methods defined in this interface.  
  
 // placeholder return value below - replace with implementation of getValue()  
  
 if(isEmpty())  
 {  
 throw new NullPointerException("Tree at current node is empty.");  
 }  
 else  
 {  
 return root.value;  
 }  
 }  
  
 */\*\*  
 \* Change the value stored at the root of the tree.  
 \** ***@param*** *value the new value to be stored at the root of the tree.  
 \*/* @Override  
 public void setValue(T value) {  
 // implement setValue(T value) here  
 if(isEmpty())  
 root = new TreeNode<T>(value, new BinaryTree<>(), new BinaryTree<>());  
 else  
 root.value = value;  
 }

*/\*\*  
 \* Get the left subtree of this tree.  
 \** ***@return*** *This tree's left subtree.  
 \*/* @Override  
 public BTree<T> getLeft() throws NullPointerException {  
 // placeholder return value below - replace with implementation of getLeft()  
 if(isEmpty())  
 {  
 throw new NullPointerException("Current node is empty.");  
 }  
 return root.left;  
 }  
  
 */\*\*  
 \* Change the left subtree of this tree.  
 \** ***@param*** *tree the new left subtree.  
 \*/* @Override  
 public void setLeft(BTree<T> tree) {  
 // implement setLeft(BTree<T> tree) here  
 root.left = tree;  
 }  
  
 */\*\*  
 \* Get the right subtree of this tree.  
 \** ***@return*** *this tree's right subtree.  
 \*/* @Override  
 public BTree<T> getRight() throws NullPointerException {  
 // placeholder return value below - replace with implementation of getRight()  
 if(isEmpty())  
 {  
 throw new NullPointerException("Current node is empty.");  
 }  
 return root.right;  
 }

*/\*\*  
 \* Change the right subtree of this tree.  
 \** ***@param*** *tree the new right subtree.  
 \*/* @Override  
 public void setRight(BTree<T> tree) {  
 // implement setRight(BTree<T> tree) here  
 root.right = tree;  
 }  
  
 */\*\*  
 \* Check if the tree contains a given value.  
 \** ***@param*** *value the value to be checked.  
 \** ***@return*** *true iff the value is in the tree.  
 \*/* @Override  
 public boolean contains(T value) {  
 // placeholder return value below - replace with implementation of contains(T value)  
 //Terminate this branch of the recursion if the current node is empty.  
 if(!isEmpty())  
 {  
 //If the value is found, return it.  
 if (value.equals(getValue()))  
 {  
 return true;  
 }  
 //Else search the left subtree if less than the current node.  
 else if (value.compareTo(root.getValue()) < 0)  
 {  
 return root.getLeft().contains(value);  
 }  
 //Or search the right subtree if greater than the current node.  
 else if (value.compareTo(root.getValue()) > 0)  
 {  
 return root.getRight().contains(value);  
 }  
 }  
 return false;  
 }

*/\*\*  
 \* Traverse the tree, producing a list of all the values contained in the tree.  
 \* This is an implementation of an inOrder traversal.  
 \** ***@return*** *a list of all the values in the tree.  
 \*/* @Override  
 public List<T> traverse() {  
 // placeholder return value below - replace with implementation of traverse()  
  
 //Create a new arrayList to store the values.  
 ArrayList<T> list = new ArrayList<>();  
  
 //Call inorder traverse with this tree and the created list.  
 inOrderTraverse(this, list);  
  
 //Return the populated list.  
 return list;  
 }  
  
 private void inOrderTraverse(BTree<T> tree, List<T> list)  
 {  
 //Terminate this branch of the recursion if the subtree is empty.  
 if(tree.isEmpty())  
 {  
 return;  
 }  
  
 //recursively search the left subtree  
 inOrderTraverse(tree.getLeft(), list);  
  
 //add the current value.  
 list.add(tree.getValue());  
  
 //And recursively search the right subtree.  
 inOrderTraverse(tree.getRight(), list);  
 }  
}

### Testing class.

package binaryTreeTest;  
  
import ...;  
  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 30/11/2018.  
 \* Version: 30/11/2018  
 \*/*public class BinaryTreeTest  
{  
 */\*\*  
 \* Test that a null tree is created correctly by using the isEmpty() and getValue() methods.  
 \*/* @Test  
 void testNullTreeConstructor()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>();  
  
 if(!intTree.isEmpty())  
 *fail*("Binary tree has not been created empty.");  
 try  
 {  
 int x = intTree.getValue();  
 *fail*("Exception for root being null not caught.");  
 }  
 catch(NullPointerException e){}  
 try  
 {  
 BTree<Integer> leftTree = intTree.getLeft();  
 *fail*("Exception for trying to access left tree from null node not caught.");  
 }  
 catch(NullPointerException e){}  
 try  
 {  
 BTree<Integer> rightTree = intTree.getRight();  
 *fail*("Exception for trying to access right tree from null node not caught.");  
 }  
 catch(NullPointerException e){}  
 }  
  
 */\*\*  
 \* Test that a single node tree is created correctly by using the isEmpty() and getValue() methods.  
 \*/* @Test  
 void testRootTreeConstructor()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(1);  
  
 if(intTree.isEmpty())  
 *fail*("Binary tree created empty.");  
  
 try  
 {  
 BTree<Integer> leftTree = intTree.getLeft();  
 }  
 catch (NullPointerException e)  
 {  
 *fail*("Exception thrown trying to access available left null subtree.");  
 }  
  
 try  
 {  
 BTree<Integer> rightTree = intTree.getRight();  
 }  
 catch(NullPointerException e)  
 {  
 *fail*("Exception thrown trying to access available right null subtree.");  
 }  
  
 try  
 {  
 *assertTrue*(1 == intTree.getValue(), "Root value not 1 as expected.");  
 }  
 catch (NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown against node holding a value.");  
 }  
 }

*/\*\*  
 \* Test that a tree created with a left subtree and a null right subtree is created correctly using the  
 \* isEmpty(), getValue(), getLeft() and getRight() methods.  
 \*/* @Test  
 void testRootAndLeftTreeConstructor()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>());  
 /\*  
 \* 2  
 \* / \  
 \* 1 null  
 \*/  
  
 if(intTree.isEmpty())  
 *fail*("Binary tree created empty.");  
  
 try  
 {  
 int x = intTree.getValue();  
 if(x != 2)  
 *fail*("Root value not 2 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown against node holding a value");  
 }  
  
 try  
 {  
 BTree<Integer> lTree = intTree.getLeft();  
 int x = lTree.getValue();  
 if(x != 1)  
 *fail*("left subtree value not 1 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available subtree/its value.");  
 }  
  
 try  
 {  
 BTree<Integer> rTree = intTree.getRight();  
 try  
 {  
 int x = rTree.getValue();  
 *fail*("Exception for right root being null not caught.");  
 }  
 catch(NullPointerException e) {}  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available right null subtree.");  
 }  
 }

*/\*\*  
 \* Test that a tree created with a right subtree and a null left subtree is created correctly using the  
 \* isEmpty(), getValue(), getLeft() and getRight() methods.  
 \*/* @Test  
 void testRootAndRightTreeConstructor()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(), new BinaryTree<>(3));  
 /\*  
 \* 2  
 \* / \  
 \* null 3  
 \*/  
  
 if(intTree.isEmpty())  
 *fail*("Binary tree created empty.");  
  
 try  
 {  
 int x = intTree.getValue();  
 if(x != 2)  
 *fail*("Root value not 2 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown against node holding a value");  
 }  
  
 try  
 {  
 BTree<Integer> rTree = intTree.getRight();  
 int x = rTree.getValue();  
 if(x != 3)  
 *fail*("Right subtree value not 3 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available subtree/its value.");  
 }  
  
 try  
 {  
 BTree<Integer> lTree = intTree.getLeft();  
 try  
 {  
 int x = lTree.getValue();  
 *fail*("Exception for left root being null not caught.");  
 }  
 catch(NullPointerException e) {}  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available left null subtree.");  
 }  
 }

*/\*\*  
 \* Test that a tree created with a left subtree and a right subtree is created correctly using the  
 \* isEmpty(), getValue(), getLeft() and getRight() methods.  
 \*/* @Test  
 void testRootAndBothTreeConstructor()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>(3));  
 /\*  
 \* 2  
 \* / \  
 \* 1 3  
 \*/  
  
 if(intTree.isEmpty())  
 *fail*("Binary tree created empty.");  
  
 try  
 {  
 int x = intTree.getValue();  
 if(x != 2)  
 *fail*("Root value not 2 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown against node holding a value");  
 }  
  
 try  
 {  
 BTree<Integer> rTree = intTree.getRight();  
 int x = rTree.getValue();  
 if(x != 3)  
 *fail*("Right subtree value not 3 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available right subtree/its value.");  
 }  
  
 try  
 {  
 BTree<Integer> lTree = intTree.getLeft();  
 int x = lTree.getValue();  
 if(x != 1)  
 *fail*("Left subtree value not 1 as expected.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown trying to access available left subtree/its value.");  
 }  
 }

@Test  
 void testInsertFullLeft()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(5, new BinaryTree<>(4), new BinaryTree<>(6));  
 /\*  
 \* 5  
 \* / \  
 \* 4 6  
 \*/  
  
 intTree.insert(3);  
 /\*  
 \* 5  
 \* / \  
 \* 4 6  
 \* /  
 \* 3  
 \*/  
  
 try  
 {  
 if (intTree.getLeft().getLeft().getValue() != 3)  
 *fail*("Left Left subtree of intTree not correctly set as 3.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for inserted accessible value.");  
 }  
 }

@Test  
 void testInsertLeftThenRight()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(10, new BinaryTree<>(5), new BinaryTree<>(15));  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \*/  
  
 intTree.insert(6);  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \* \  
 \* 6  
 \*/  
  
 try  
 {  
 if (intTree.getLeft().getRight().getValue() != 6)  
 *fail*("Left Right subtree of intTree not correctly set as 6.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for inserted accessible value.");  
 }  
 }

@Test  
 void testInsertRightThenLeft()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(10, new BinaryTree<>(5), new BinaryTree<>(15));  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \*/  
  
 intTree.insert(14);  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \* /  
 \* 14  
 \*/  
 try  
 {  
 if (intTree.getRight().getLeft().getValue() != 14)  
 *fail*("Right Left subtree of intTree not correctly set as 14.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for inserted accessible value.");  
 }  
 }

@Test  
 void testInsertFullRight()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(5, new BinaryTree<>(4), new BinaryTree<>(6));  
 /\*  
 \* 5  
 \* / \  
 \* 4 6  
 \*/  
  
 intTree.insert(7);  
 /\*  
 \* 5  
 \* / \  
 \* 4 6  
 \* \  
 \* 7  
 \*/  
 try  
 {  
 if (intTree.getRight().getRight().getValue() != 7)  
 *fail*("Right Right subtree of intTree not correctly set as 7.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for inserted accessible value.");  
 }  
 }

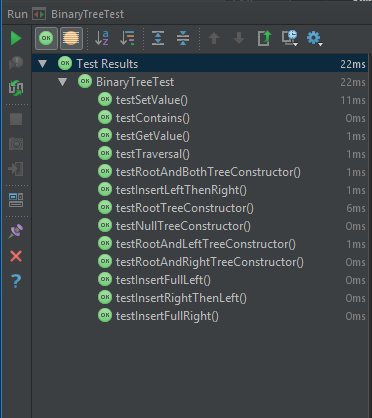
@Test  
 void testGetValue()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>());  
 /\*  
 \* 2  
 \* / \  
 \* 1 null  
 \*/  
 try  
 {  
 if (intTree.getValue() != 2)  
 *fail*("getValue() not correctly returned 2.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for accessible value.");  
 }  
  
 try  
 {  
 if (intTree.getLeft().getValue() != 1)  
 *fail*("getValue() not correctly returned 1.");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for accessible value.");  
 }  
 }

@Test  
 void testSetValue()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>();  
 /\*  
 \* null  
 \*/  
  
 intTree.setValue(2);  
 /\*  
 \* 2  
 \* / \  
 \* null null  
 \*/  
  
 try  
 {  
 if (intTree.getValue() != 2)  
 *fail*("2 not correctly set as intTree value");  
 }  
 catch(NullPointerException e)  
 {  
 e.printStackTrace();  
 *fail*("Exception thrown for accessible value.");  
 }  
 }

@Test  
 void testContains()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(10);  
 intTree.insert(5);  
 intTree.insert(2);  
 intTree.insert(7);  
 intTree.insert(15);  
 intTree.insert(12);  
 intTree.insert(17);  
  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \* / \ / \  
 \* 2 7 12 17  
 \*/  
  
 if(!intTree.contains(2))  
 *fail*("2 not found within tree.");  
 if(!intTree.contains(7))  
 *fail*("7 not found within tree.");  
 if(!intTree.contains(5))  
 *fail*("5 not found within tree.");  
 if(!intTree.contains(10))  
 *fail*("10 not found within tree.");  
 if(!intTree.contains(15))  
 *fail*("15 not found within tree.");  
 if(!intTree.contains(12))  
 *fail*("12 not found within tree.");  
 if(!intTree.contains(17))  
 *fail*("17 not found within tree.");  
  
 if(intTree.contains(1))  
 *fail*("1 found within tree");  
 }

*/\*\*  
 \* Output is known seeing as a predefined traversal method is being used.  
 \*/* @Test  
 void testTraversal()  
 {  
 BinaryTree<Integer> intTree = new BinaryTree<>(10);  
 intTree.insert(5);  
 intTree.insert(2);  
 intTree.insert(7);  
 intTree.insert(15);  
 intTree.insert(12);  
 intTree.insert(17);  
  
 /\*  
 \* 10  
 \* / \  
 \* 5 15  
 \* / \ / \  
 \* 2 7 12 17  
 \*/  
  
 ArrayList<Integer> expectedList = new ArrayList<>();  
 expectedList.add(2);  
 expectedList.add(5);  
 expectedList.add(7);  
 expectedList.add(10);  
 expectedList.add(12);  
 expectedList.add(15);  
 expectedList.add(17);  
  
 ArrayList<Integer> returnedList = (ArrayList<Integer>)intTree.traverse();  
  
 for (int x = 0; x < 7; x++)  
 {  
 if(!Objects.*equals*(expectedList.get(x), returnedList.get(x)))  
 *fail*("Discrepancy between expected and returned results at index: " + x);  
 }  
 }  
}

### Test results.



## Self Evaluation.

For 3 marks, I was asked to implement all methods, which I have done so. This qualifies me for 3 marks at the least. For 4 and 5 marks I was asked for good documentation and good testing. I don’t think my documentation or testing are absolutely perfect, but given that the descriptor used is good, I would say my inclusion of testing and documentation is enough to qualify my work this week being 5/5

# Practical 7 (Week 10)

## (Logbook) Question 1: Implement the Traversal interface using depth-first traversal.

### Code listing.

package graph;  
  
import java.util.\*;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 30/11/2018.  
 \* Version: 30/11/2018  
 \*/*public class DepthFirstTraversal<T> extends AdjacencyGraph<T> implements Traversal<T>  
{  
 */\*\*  
 \* Used to hold the list in which the nodes of the graph are traversed.  
 \*/* ArrayList<T> traversal;  
  
 */\*\*  
 \* A constructor for a depth first traversal that initializes the data structure used for the traversal list.  
 \*/* public DepthFirstTraversal()  
 {  
 traversal = new ArrayList<>();  
 }

*/\*\*  
 \* An implementation of the traverse() method defined in Traversal.  
 \*  
 \** ***@return*** *Returns a list of nodes in the order of which they were found during traversal.  
 \** ***@throws*** *GraphError if the graph given to the traversal is empty.  
 \*/* @Override  
 public List<T> traverse() throws GraphError  
 {  
 //This is used to ensure the existence of at least one node.  
 if(getNoOfNodes() == 0)  
 throw new GraphError("Graph is empty.");  
  
  
 //At least one node exists so this can be used safely.  
 T node = getUnvisitedNode();  
 do  
 {  
 //Begin the population of the traversal list from the chosen node.  
 populateTraversal(getUnvisitedNode());  
  
 //Check for any more unvisited nodes.  
 node = getUnvisitedNode();  
  
 //Repeat if another unvisited node was found after the first pass.  
 }while(node != null);  
  
 //Return the now populated traversal list.  
 return traversal;  
 }

*/\*\*  
 \* A recursive method to find and populate the traversal list for graph based on the initial T node.  
 \*  
 \** ***@param*** *node the node to begin the traversal and search from.  
 \** ***@throws*** *GraphError  
 \*/* private void populateTraversal(T node) throws GraphError  
 {  
 //If the node has not already been visited.  
 if(!traversal.contains(node))  
 {  
 //Record the node as visited and add it to the traversal list.  
 traversal.add(node);  
  
 //For each of its neighbours, run this method again.  
 for(T neighbour:getNeighbours(node))  
 {  
 populateTraversal(neighbour);  
 }  
 }  
 }

*/\*\*  
 \* Get an unvisited node from the graphs set of nodes.  
 \*  
 \** ***@return*** *a T node within the graph that has been unvisited according to the traversal list in this class. OR null  
 \* if no unvisited node has been found./  
 \* if no unvisited node has been found.  
 \*/* private T getUnvisitedNode()  
 {  
 //For each node in the set of nodes.  
 for(T node : getNodes())  
 {  
 //If it is unvisited, return this node.  
 if(!traversal.contains(node))  
 return node;  
 }  
  
 //Default case for when no node is found.  
 return null;  
 }  
}

### Testing class.

import graph.AdjacencyGraph;  
import graph.DepthFirstTraversal;  
import graph.GraphError;  
import org.junit.Test;  
  
import static org.junit.Assert.*fail*;  
  
  
import java.util.ArrayList;  
  
*/\*\*  
 \* Created by u1661665(Joshua Pritchard) on 30/11/2018.  
 \* Version: 30/11/2018  
 \*/*public class DepthFirstTest<T> extends AdjacencyGraph<T>  
{  
  
 //Specifies the number of repetitions of circularGraphTest()  
 private final int NUMBER\_CIRCULAR\_GRAPH\_TESTS = 10;

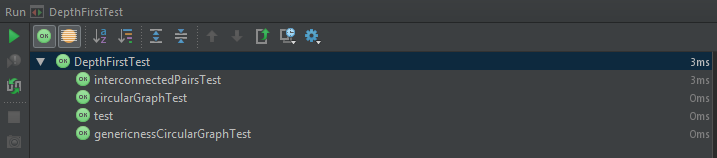
*/\*\*  
 \* Creates a determined circular integer graph with 3 elements and tests a traversal of this graph for correctness.  
 \*  
 \** ***@throws*** *GraphError if any attempt to access an invalid element is made.  
 \*/* @Test  
 public void test() throws GraphError  
 {  
 DepthFirstTraversal<Integer> intGraph = new DepthFirstTraversal<>();  
  
 intGraph.add(1);  
 intGraph.add(2);  
 intGraph.add(3);  
  
 intGraph.add(1, 2);  
 intGraph.add(2, 3);  
 intGraph.add(3, 1);  
  
 ArrayList<Integer> returnedList = (ArrayList<Integer>) intGraph.traverse();  
  
 if (returnedList.get(0) == 1 && returnedList.get(1) == 2 && returnedList.get(2) == 3)  
 {  
 return;  
 }  
 else if (returnedList.get(1) == 1 && returnedList.get(2) == 2 && returnedList.get(0) == 3)  
 {  
 return;  
 }  
 else if(returnedList.get(2) == 1 && returnedList.get(0) == 2 && returnedList.get(1) == 3)  
 {  
 return;  
 }  
 else  
 {  
 *fail*("Correct path not returned.");  
 }  
 }

*/\*\*  
 \* Creates a determined circular graph out of 5 elements and tests a traversal of this graph for correctness.  
 \*  
 \** ***@throws*** *GraphError if any attempt to access an invalid element of the graph is made.  
 \*/* @Test  
 public void circularGraphTest() throws GraphError  
 {  
 //Create a circular graphh  
 DepthFirstTraversal<Integer> circularIntGraph = new DepthFirstTraversal<>();  
  
 //Add nodes.  
 circularIntGraph.add(1);  
 circularIntGraph.add(2);  
 circularIntGraph.add(3);  
 circularIntGraph.add(4);  
 circularIntGraph.add(5);  
  
 /\*  
 1 2 3 4 5  
 \*/  
  
 //Add edges.  
 circularIntGraph.add(1, 2);  
 circularIntGraph.add(2, 3);  
 circularIntGraph.add(3, 4);  
 circularIntGraph.add(4, 5);  
 circularIntGraph.add(5, 1);  
  
 /\*  
 1 -> 2 -> 3 -> 4 -> 5  
 ^ |  
 |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|  
 \*/  
  
 //---All test cases---  
 //A search from 1 should return {1, 2, 3, 4, 5}  
 //A search from 2 should return {2, 3, 4, 5, 1}  
 //A search from 3 should return {3, 4, 5, 1, 2}  
 //A search from 4 should return {4, 5, 1, 2, 3}  
 //A search from 5 should return {5, 1, 2, 3, 4}  
  
 //Do a number of tests to make sure all test cases are covered.  
 for(int x = 0; x < NUMBER\_CIRCULAR\_GRAPH\_TESTS; x++)  
 {  
 //Cast the returned set to an array list.  
 ArrayList<Integer> returnedList = (ArrayList<Integer>) circularIntGraph.traverse();  
  
 if(returnedList.get(0) == 1 && returnedList.get(1) == 2 && returnedList.get(2) == 3 && returnedList.get(3) == 4 && returnedList.get(4) == 5)  
 {break;}  
 else if(returnedList.get(0) == 2 && returnedList.get(1) == 3 && returnedList.get(2) == 4 && returnedList.get(3) == 5 && returnedList.get(4) == 1)  
 {break;}  
 else if(returnedList.get(0) == 3 && returnedList.get(1) == 4 && returnedList.get(2) == 5 && returnedList.get(3) == 1 && returnedList.get(4) == 2)  
 {break;}  
 else if(returnedList.get(0) == 4 && returnedList.get(1) == 5 && returnedList.get(2) == 1 && returnedList.get(3) == 2 && returnedList.get(4) == 3)  
 {break;}  
 else if(returnedList.get(0) == 5 && returnedList.get(1) == 1 && returnedList.get(2) == 2 && returnedList.get(3) == 3 && returnedList.get(4) == 4)  
 {break;}  
 else  
 {  
 *fail*("Returned list did not match any possible test case.");  
 }  
 }  
 }

*/\*\*  
 \* Creates a determined graph with 6 elements, then interconnects these to make 3 pairs. Then tests a traversal  
 \* of this graph for correctness based on the guaranteed difference between each pair.  
 \*  
 \** ***@throws*** *GraphError if any attempt to access and invalid element of the graph is made.  
 \*/* @Test  
 public void interconnectedPairsTest() throws GraphError  
 {  
 //Create a graph.  
 DepthFirstTraversal<Integer> interconnectedPairsGraph = new DepthFirstTraversal<>();  
  
 //Create 6 nodes.  
 interconnectedPairsGraph.add(1);  
 interconnectedPairsGraph.add(2);  
 interconnectedPairsGraph.add(3);  
 interconnectedPairsGraph.add(4);  
 interconnectedPairsGraph.add(5);  
 interconnectedPairsGraph.add(6);  
  
 //Link 1&2, 3&4 and 5&6  
 interconnectedPairsGraph.add(1, 2);  
 interconnectedPairsGraph.add(2, 1);  
 interconnectedPairsGraph.add(3, 4);  
 interconnectedPairsGraph.add(4, 3);  
 interconnectedPairsGraph.add(5, 6);  
 interconnectedPairsGraph.add(6, 5);  
  
 /\*  
 1<->2 3<->4 5<->6  
 \*/  
  
  
 //Traverse the graph.  
 ArrayList<Integer> returnedList = (ArrayList<Integer>) interconnectedPairsGraph.traverse();  
  
 //each pair will be adjacent in the list.  
 if(Math.*abs*(returnedList.get(0) - returnedList.get(1)) != 1)  
 {  
 *fail*("first pair not adjacent");  
 }  
 else if(Math.*abs*((returnedList.get(2) - returnedList.get(3))) != 1)  
 {  
 *fail*("Second pair not adjacent.");  
 }  
 else if(Math.*abs*(returnedList.get(4) - returnedList.get(5)) != 1)  
 {  
 *fail*("Third pair not adjacent.");  
 }  
 }

*/\*\*  
 \* Creates a determined circular graph with 3 string elements and tests a traversal of this graph for correctness.  
 \*  
 \** ***@throws*** *GraphError if any attempt to access an invalid element of the graph is made.  
 \*/* @Test  
 public void genericnessCircularGraphTest() throws GraphError  
 {  
 DepthFirstTraversal<String> stringGraph = new DepthFirstTraversal<>();  
  
 stringGraph.add("first");  
 stringGraph.add("second");  
 stringGraph.add("third");  
  
 stringGraph.add("first", "second");  
 stringGraph.add("second", "third");  
 stringGraph.add("third", "first");  
  
 ArrayList<String> returnedList = (ArrayList<String>) stringGraph.traverse();  
  
 if (returnedList.get(0).equals("first") && returnedList.get(1).equals("second") && returnedList.get(2).equals("third"))  
 {  
 return;  
 }  
 else if (returnedList.get(1).equals("first") && returnedList.get(2).equals("second") && returnedList.get(0).equals("third"))  
 {  
 return;  
 }  
 else if(returnedList.get(2).equals("first") && returnedList.get(0).equals("second") && returnedList.get(1).equals("third"))  
 {  
 return;  
 }  
 else  
 {  
 *fail*("Correct path not returned with a string graph");  
 }  
 }  
}

### Test results.



## Self Evaluation.

The self evaluation criteria for 3 marks asks for a full implementation of the Depth-First traversal algorithm. I believe my implementation is full and works correctly returning a traversal list with no duplicated elements. Furthermore I believe that my re-use of the traversal list as a visited list removes space complexity and makes my implementation better.

For 4 and 5 marks I must have Javadoc documentation, which I have included, and full testing. Testing was difficult to develop for this due to the inherent difficulty in testing of graphs. This is because of their use of sets, which are not guaranteed to return the same element every time one is requested. This means complex graphs are very hard to test unless they have testable *properties* consistent across the entire graph. If none of these are present, testing a large, complex graph, would require hardcoding a test against every single possible result of the traversal.

This is long winded, repetitive and pointless considering that smaller graphs with intrinsic properties can be tested in the same way, meaning the more complex graphs will test fine as well.

I believe that my testing is full, adequate and covers all bases of the traversal method that may come into question.