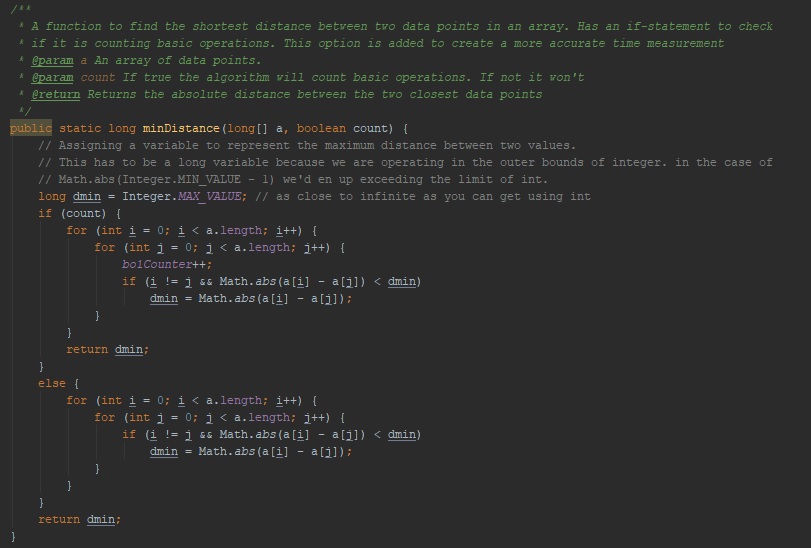
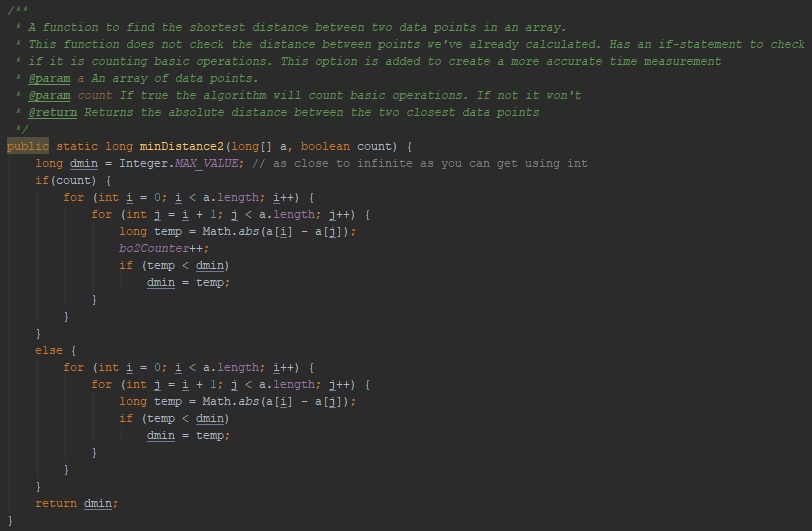
# Important bullet points

* Description of the algorithm
* ~~Choice of basic operation~~
* Choice of problem size
* ~~Implementation of the algorithm~~
* ~~Proof of correctness~~
* ~~Methodology, tools and techniques~~
* ~~Choice of computing env.~~
* ~~Show how we produced test data~~
* Count number of basic operations / justify it compared to theory
* State how our experiment compares to results
* Show how we measured BOs
* Explain how we produced test data / output
* Enough to show a clear trend
* Show growth compared to input size
* State if result of the experiment matched our predictions
* Clearly show how many data points contribute to a line
* Measure time on a range of different inputs
* Explain how we measured time
* Enough output to show a clear trend
* Show growth compared to input size
* State if result match our predictions

# Implementation of the algorithm

## Our code implementation

****



We made both algorithms with counting basic operations separated to the actual algorithm so that when we were timing the algorithm we didn’t spend time counting basic operations.

## Proof of correctness

To test that our algorithms were correctly implemented we created our own test class. Using JUnit’s testing utility[[1]](#endnote-1) we created a test to check for possible errors in the algorithms. We tested that:

* The algorithms gave us the correct value of the shortest distance between values.
* Negative numbers were correctly handled.
* Equal values would give 0 as distance.
* Equal distances would still provide correct answers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test case** | **Test instance** | **Expected result** | **Actual output** | **Test result** |
| Shortest distance between values found | A = [10, 4, 8, 1, 2] | 1 | 1 | Passed |
| Negative numbers handled correctly | A =  [0, 500, 1000, -501, 10] | 10 | 10 | Passed |
| Multiple occurrences of shortest distance | A = [0, -1, -2, -3, 1, 2, 3] | 1 | 1 | Passed |
| Equal values correctly handled | A = [10, 0, 3, 1, 0, 6, 0] | 0 | 0 | Passed |

# Design of experiment

## Methodology, tools and techniques

We chose Java as our coding language because it’s the language we’ve both had the most experience with and the language where we could represent the algorithm as accurate as we could. Java is an object oriented programing language provided by Oracle and is one of the most popular programming languages as of early 2018[[2]](#endnote-2).

The experiment was performed on one of QUT’s S-block computers. It runs 64x Windows 10 Enterprise with an Intel® Core™ i7-6700 CPU @ 3.40GHz processor*[Appendix #1]*. Java’s pseudorandom Random class was used to create the test data. For estimating time we used Java’s innate System class. It has a function to return the current system’s time in millisecond. In order to assure a minimum of interference from other processes we closed as many other processes as we could while the experiment was running.

For producing the graphs we ran our java code that writes array sizes, basic operations counted and time spent for each run through of the algorithm to two separate files. One for Levitin’s algorithm and one for the second provided algorithm. Then we plotted our test results using Microsoft® Office Excel 2016.

## Producing test data

For producing the test data we wrote a function to create an array containing pseudo random integers. The integers could be any valid number in Java’s representation of int[[3]](#endnote-3). And since the algorithm is comparing int-variables we used long as the data type in the algorithm. The array generating random function we created takes the array size as an argument and returns a pseudo random array of the given size [appendix #2].

## Problem size

# Experiment results

## Choice of basic operation

### Levetin’s algorithm

For Levetin’s algorithm we decided that the if-statement in the inner loop was the basic operation. The operations that we considered for the basic operation in the algorithm were:

1. If-statement in inner loop
   * Comparison of i == j
   * If i != j, comparison and calculation of |a[j] - a[i]| < dmin.
2. The arithmetic calculation of a[i] – a[j] if if-statement is true
3. Comparison of j < array size in the inner loop
4. Incrementing j in the inner loop
5. Comparison of i < array size in the outer loop
6. Incrementing I in the outer loop

Number 6 and 5 were immediately dismissed as they are not affected by the inner loop. Both number 3 and 4 we consider because they would both happen for every iteration of both the inner and outer loop because the algorithm doesn’t have any choice but to complete without completing both for-loops.

For number 2 we found that this is dependent on the encapsulating if-statement which means this is not a basic operation.

Our choice of basic operation fell on number 1, the if-statement in the inner loop checking i == j, and if that is true it compares a[j] to dmin. This will be the most time consuming and most defining operation of the algorithm and therefore a good indicator of number of operations.

### Improved algorithm

For the given improved version of Levetin’s algorithm we decided to again choose the if-statement in the inner for-loop using the exact same logic as we did for choosing Levetin’s algorithm’s basic operation.

## Complexity

Number 6 and 5 were immediately dismissed as they are not affected by the inner loop and have O(n) complexity. Both number 3 and 4 we consider because they would both happen times because the algorithm doesn’t have any choice but to complete the both for-loops. This give number 3 a complexity of and number 4 a complexity of .

For number 2 we found that this is dependent on the encapsulating if-statement which means this is not the basic operation.

Our choice of basic operation was therefore number 1, the if-statement in the inner loop checking i == j, and if that is true it compares a[j] to dmin. The complexity for this is dependent on earlier comparisons so we focused on the worst case order which gives the following complexity which gives us

### Best case

### Average case

### Worst case

## Counting basic operations

## Measuring time

## Test results

# Analysis of experiment results

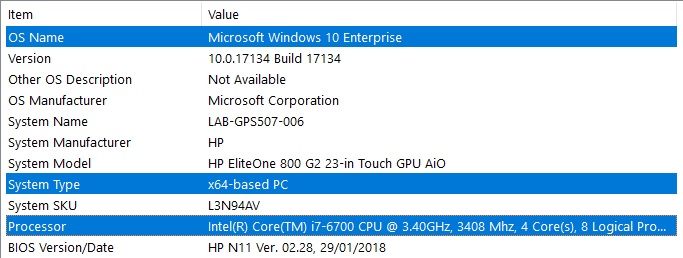
## Basic operations

## Time estimate

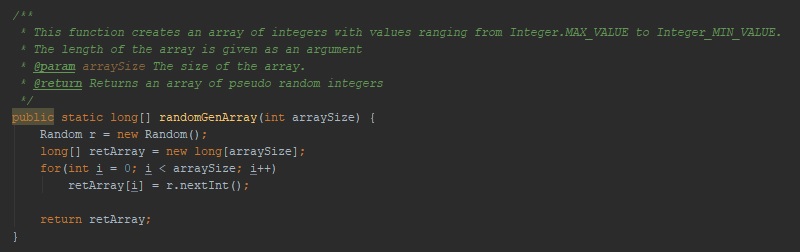
## Conclusion ?

# Appendix

#1



#2



#3

#4

#5

1. Endnotes:

   http://junit.sourceforge.net/javadoc/org/junit/Assert.html [↑](#endnote-ref-1)
2. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ [↑](#endnote-ref-2)
3. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html> [↑](#endnote-ref-3)