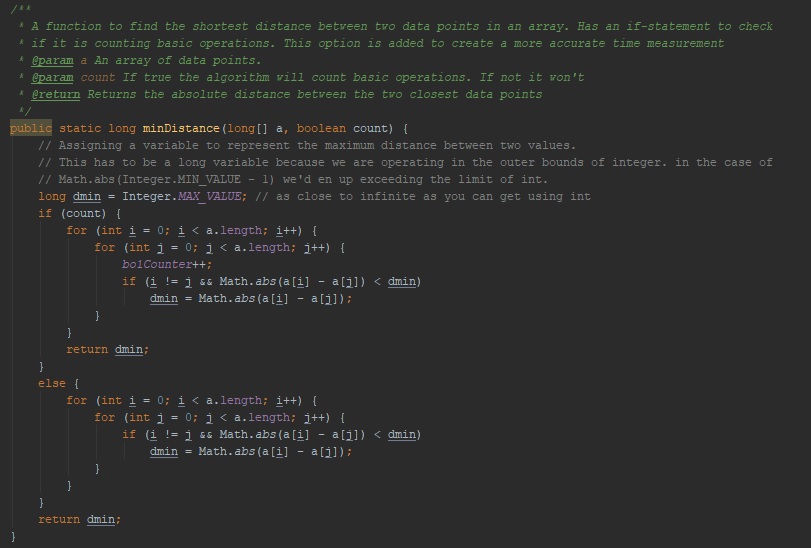
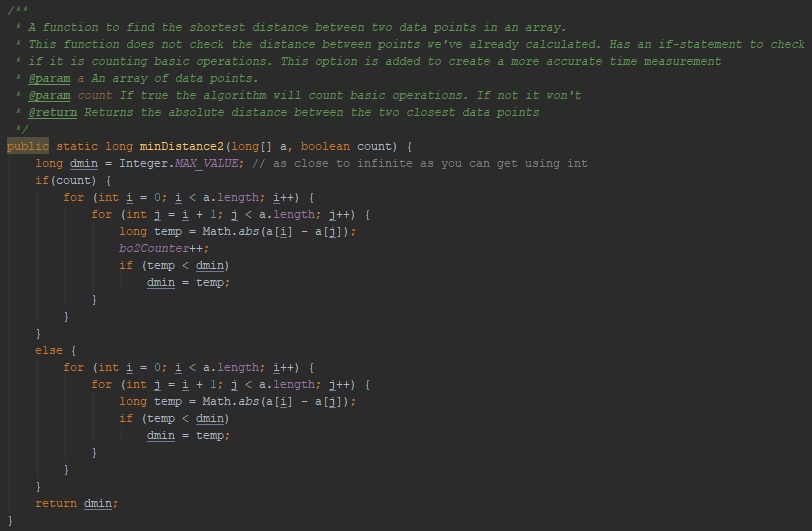
# Important bullet points

* 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

Java was chosen as the programming language for data collection throughout this project as it can implement more accurate timing modules and take advantage of multithreading applications much more efficiently than many of its competitors. 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). Its vast popularity will also provide an accurate representation of what environment these algorithms will run in, as Java is an extremely diverse language that runs on all systems from micro-controllers to supercomputers.

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. Java incorporates an extremely accurate timing system that was utilized for this assignment called the inbuilt nanoTime[[3]](#endnote-3) function. This will return the time taken in the most accurate format that is available to consumer grade hardware.

For producing the graphs the written code produces a commas separated values file which includes the 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 the 2D line plot tool in Microsoft® Office Excel 2016.

## Producing test data

For producing the test data, a function was written to create an array containing pseudo random integers. The integers could be any valid number in Java’s representation of an integer[[4]](#endnote-4). Since the algorithm is comparing integer-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

### Levitin’s algorithm

Using the options for basic operations described and enumerated in “Choice of basic operations” we can describe each of the operation’s complexity using their representative numbers.

Number 6 and 5 are not affected by the inner loop and therefore have O(n) complexity. This is because the loop starts at 0, is incremented by 1 and ends at N-1.

Both number 3 and 4 we found to have complexity. The inner and outer loop has the same iteration size (start at 0, increments by 1 and ending at N-1) and because they do the inner loop has a complexity of O(n) and is ran n times. This give number 3 a complexity of and number 4 a complexity of .

Number 1, our choice of basic operation, can be described like this:

The first comparison of the if-statement is always executed because of how java’s if-statements work[[5]](#endnote-5), meaning . The second comparison done in the statement is therefore: . Minus n because for each iteration of the inner loop the i will be equal to j once per iterations of the outer loop. The total complexity is then:

Operation number 2 is dependent on the encapsulating if-statement described above which means we have to divide this into best, worst and average case.

### Best case

Best case scenario when we’re looking for the smallest distance between two values is when the first and second value in the array are the smallest distance. This means that we’re doing operation number 2 only once. We get the following complexity when adding the numbers of all the operations together:

### Average case

In order to find the average operation count of this algorithm we need to consider all the possible outcomes from best to worst. The nature of Levetin’s algorithm dictates that we have several constants that will not change based on worst or best case scenario. These constants are operations number 1, 3, 4, 5 and 6, leaving only operation 2 as the inconsistent operation.

### Worst case

The worst case scenario is a bit complicated for this algorithm. Given an array of size N, to find the worst case we need to assume that every comparison we do in operation 1 gives true (distance is shorter than the previous comparison). That means that

We can already see that this is not true because we already made the comparison between . Moving forward there is a pattern emerging with already compared distances:

etc. The pattern shows that after comparing to we have already made that comparison for all and therefore worst case of operation 2 cannot be . Instead we see that the worst case is the sum of all comparisons between the current smallest distance minus the comparisons we haven’t already done: .

This means worst case is:

### Improved algorithm

To compare with Levetin’s algorithm we were given an algorithm that we needed to show was an improved version. As mentioned in the worst case for Levetin’s algorithm we found that a lot of comparisons were done twice. This algorithm proposes an answer to this

## Counting basic operations

A simple integer variable was used to count each nested for loop for both of the tested algorithms. This variable is simply increased by 1 for each iteration. This variable is then written to a .CSV file along with the size of the array that was tested.

## Measuring time

The time taken for each test was recorded using the nanoTime() function included in the Java system function set. Two times were taken, once before the test begins, and once after. The difference between these two measurements is then calculated to find how long that each test took. This time was then written to a .CSV file along with the corresponding array size. It is important to note that the basic operation count and the time measurement results were taken at separate times to further reduce the potential increase in processing time that the counting operation would add to the result.

## Test results

Each algorithm was run 10 times for both the basic operation count and the time taken for each test. This generated 20 datasets for each of the algorithms. These were then imported into Excel, and line graphs were generated for each of these. The outputs of the program are listed in the appendix.

Each of the 2 algorithms returned results in line with the theoretical function found in the pre-experimental analysis. While testing results were similar, it was found that for large datasets, algorithm 2 was significantly slower, with a consistent 20% increase in processing time in comparison to the first algorithm tested for the largest dataset provided.

For this reason, it would be recommended that for very large datasets, algorithm 2 should be used as it tends to a greater efficiency which becomes increasingly accurate when dealing with large scale datasets.

# Analysis of experiment results

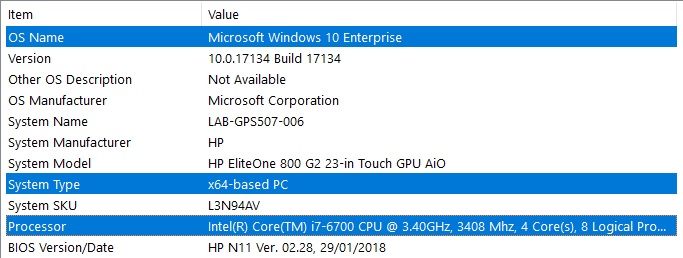
## Basic operations

## Time estimate

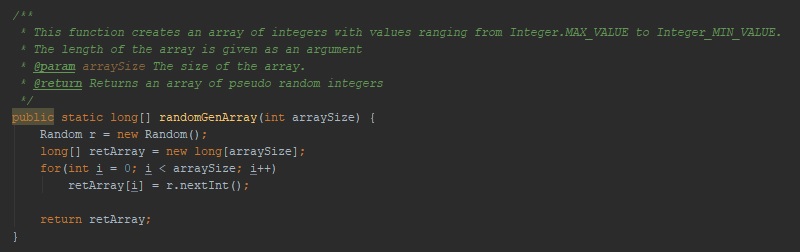
## Conclusion ?

# Appendix

#1



#2



#3

#4

#5

Algorithm 1 Results

Algorithm 2 Results

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/7/docs/api/java/lang/System.html [↑](#endnote-ref-3)
4. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html> [↑](#endnote-ref-4)
5. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html [↑](#endnote-ref-5)