**Summary**

**1 Description of the alogirthms**

Both of the algorithms used in this experimental process have the same basic functionality, with both determining the minimum distance between two integers in an array.

The first array is the simpler of the two, by having a nested loop, with both loops being of size n where n is the size of the array. It then checks whether or not it is being checked against itself (i is not equal to j), as well as checking if the two elements in each of the positions in the arrays being looped through, subtracted from each other is less than the current minimum stored in dmin (which starts of as an infiinite value).

The second array is mostly similar, with the exception of the number of iterations in each of the loops. The outer loop is looped through n-1 times, and the inner array being looped through n – i times where I is the index of the outer loop. This also mean that the check against iteslf (i not equal to j) is redundant and can be removed. This also avoids ever recomputing the same expression and should theortically make it more efficient

**2 Theoretical Analysis of Algorithms**

**2.1 Choice of Basic Operation**

**2.2 Choice of Problem Size**

**2.3 Average Case Efficiency**

**3 Methodology, Tools and Techniques**

**3.1 Programming Environment**

To test this algorithm, it was implemented in c++ in the Code::Blocks environment. The tests were run on two different computers, given the algorithms, tests and experimental results were executed by two different people.

The first computer is a laptop with an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz and 4GB of ram. It is running ubuntu with the KDE desktop environment.

**3.2 Implementation of Algorithm**

The implementation of the algorithms was done by creating a class for each, both with the same basic functions, just with slightly different implementation for each algorithm.

Each class had a function that simply implements the algorithm, a function that runs in and return the number of basic operations and finally a function that runs the algorithm while recording the time it took to execute it.

These function were then called in output.cpp to generate the experimental results as outlined below

**3.3 Generating Test Data and Running the Experiments**

To test the functionality of the algorithms, a variety of test arrays were used to cover most bases. This was done using a custom test function instead of a c++ testing framework due to lack of time and ease of use.

In total, there were 6 test cases used to test corectness, each with differing array sizes and distances between seperated integers. The test cases included a small sequential array, the same array reversed, an onordered array and several arrays testing varying distances between each integer to test it could handle large cases.

The code base for these test can be found in tests.cpp in the src folder, and can be executed by uncommenting the tests section in main.cpp

As mentioned above, to generate to experimental results, output.cpp was used to execute the function found in each of the algorithms classes. For both the basic number of operations and time taken for each algorithm, an array was created of n length using random\_device, random\_number\_generator, uniform\_int\_ditribution and generate, which was then converted to a vector to be used in the algorithm testing.

For each of the experiments, a loop was created to iterate through n amount of times (in this case 50), with each jump in that specific array going up 100, making the array size increase by 100 every time. For each of these array sizes, a nested loop was created to run through m times (10) and find the average of each of these runs. From here the average is recorded and later outputted in a csv to be analysed and graphed.

**4 Experimental Results**

**4.1 Average Case Number of Basic Items**

**4.2 Average Case Execution Time**