CAB301 Assignment 2: Empirical Comparison of Two Algorithms for Finding the Minimum Distance between Two Elements in an Array

Shane Havers N9477756

Andrew Grant N9167340

Date Submitted:

# Summary

The purpose of this report is to summarise and compare the results of experiments conducted on two algorithms that both find the smallest difference between two elements within a set of numbers. The tests were completed by implementing the algorithms and a test driver in C++ within the Code Blocks IDE, and outputting the results to Comma Separated Value files (.CSV). These final outputs were plotted in MATLAB, and the trends clearly reinforce the expectations that the second algorithm (MinDistance2) was indeed far more efficient than the first algorithm (MinDistance). As such, the end test results show a very high correlation with the estimated theoretical efficiency for both algorithms.

# Description of the Algorithms

The algorithms being compared in this report are algorithms that check the difference – or ‘distance’ – between each element of an input list of numbers, and seek to return the smallest difference. The first algorithm, MinDistance, was proposed by Levitin [1] and can be found in Appendix 1. MinDistance2, which can be found in Appendix 2, serves the same purpose as MinDistance but is proposed as a potentially more efficient version.

The MinDistance algorithm works by first setting the variable to an arbitrarily large value, ideally infinity. Then, it iterates over each element in the array with an index variable *,* and then again with an index variable *.* The algorithm then asks: first ensure the element is not being compared to itself (check ); and if not, is the difference between the two elements smaller than the current minimum (check ). If this difference is the smallest, set to this value. After checking every permutation of and , the algorithm returns the minimum value and exits.

The MinDistance2 starts the same way, by setting to a large value. Then, it iterates over every element of bar the final one (again with index ), and then iterates over every value with an index larger than the index (i.e. the second loop goes from index to the end of the array). Then, it stores the difference between the values at indices and in a temporary variable, and if that variable is smaller than the current minimum, change to that value. After both loops have exited, the algorithm returns the minimum value and exits.

# Theoretical Analysis of the Algorithms

## Choice of Basic Operation

In order to conduct a successful comparison, it makes logical sense to choose a common operation that is performed in both algorithms to serve as the basic operation. The common operation that we have identified as the basic operation in both algorithm is highlighted in both Appendix 1 and 2. It is the absolute value resulting from the subtraction of the ‘*jth’* element from the ‘*ith’* element of array A.

In ‘*MinDistance’* the operation can be potentially performed twice per iteration of the inner for-loop. The first opportunity for it to be performed is as part of the if-statement, conditional on whether index variables ‘i’ and ‘j’ are different values. The second performance of the basic operation is also conditional and occurs only when first performance above has been carried out, and the resultant absolute value is less than the current value of the variable *‘dmin’*.

In *‘MinDistance2’* the operation is only performed once per iteration of the inner for-loop where it’s absolute value is assigned to the variable *‘temp’*. However, it is performed every time the inner for-loop iterates without being subject to conditional statements.

The reason this operation has been selected (besides the fact that it is essentially the only operation commonly performed between the two algorithms) was because in both algorithms it was not only the most performed operation but the most expensive operation as well. While the majority of other operations are either comparisons or assignments, in the case of the identified operation; the subtraction performed, combined with the calculation of the absolute value is most strenuous part of each algorithm and therefore is the operation that would have the greatest influence on each algorithm’s execution time.

## Choice of Problem Size

The only logical choice of ‘problem size’ for these algorithms are the number of elements in each Array A, otherwise represented as the value ‘n’. This makes perfect sense, especially when considering that both algorithms are considered to take the same array A as the one and only input.

## Determining the Average-Case Efficiency

The estimated average-case efficiency of the ‘*MinDistance*’ algorithm is **Cavg(n) = n(n-1).** The inner part of that equation is derived from the fact that the algorithm is essentially being performed n -1 times with each iteration of the inner for-loop. 1 is subtracted from n to account for the fact that the basic operation is not performed when the *‘ith’* element equals the *‘jth’* element (in accordance with the conditional if-statement). ‘*i*’ will only ever equal ‘*j*’ once for each iteration of the inner for-loop, so that is why it is ‘n-1’. That value is then multiplied by n since the outer for-loop will be iterated n times, or, once for each element in the array A. **Therefore,** **Cavg(n) = n(n-1) OR n² - n.**

Note that this equation does not account for the small amount of iterations that the operation is carried out an extra time because the current value was less than *‘dmin’* and the assignment inside the if-statement was carried out. This is because it would have such a negligible effect on the overall average number of operations performed that it would not affect the expected trends.

The estimated average-case efficiency of the ‘*MinDistance2*’ algorithm is **Cavg(n) = (n(n - 1))/2.** It is essentially the same situation as with the first algorithm. However instead of operations being skipped in the event where the *‘ith’* element equals the *‘jth’*, in this case the *‘i’* will never be equal to *‘j’* since *‘j’* is never less than ‘*i+1’*. But the effect is the same such that the efficiency will include ‘*n(n-1)*’. It is then divided in half due to the length of each iteration of the inner for-loop decreasing by 1 as *‘i’* approaches *‘n’*. By the time the final iteration has been run and the outer for-loop has exited, half of the operations that would have been performed have not due to the inner for-loops assignment of *‘j’*. **Therefore,** **Cavg(n) = (n(n - 1))/2 OR (n² - n)/2.**

From these equations, we can clearly identify that both algorithm’s efficiency class and order of growth are quadratic [Θ(n²)]. This gives us a solid idea of what type of trend lines will appear when our test results are plotted.

Due to the number of operations performed being almost exclusively dependent on the size of the array in both algorithms, the best and worst case scenarios are essentially the same, bar some discrepancies in the *‘MinDistance’* mentioned above.

# Methodology, Tools and Techniques

## Programming Environment

The algorithm and testing suite used to run them were implemented in C++ using the Code Blocks IDE. C++ is a well-established, fast and flexible programming language [2], making it an appropriate choice for comparing the algorithms. The code was then compiled and run on a Microsoft Surface Pro 4 running Microsoft’s flagship operating system, Windows 10.

## Implementation of Algorithms

Both algorithms were implemented as C++ methods that were called within the Main function. The integer primitive type was used to represent the list of numbers, as whole numbers were adequate to demonstrate both algorithms’ effectiveness while also minimising the amount of memory used to store the array. The setup of the C++ program can be found in Appendix 3, and the Main method can be found in Appendix 4.

The translation of both algorithms from pseudocode to C++ was straightforward, with the only significant difference being that the initial value of was set to the maximum allowable integer value of rather than the impossible value of infinity – the risk of the distance between two values being larger than is completely negligible, therefore is a suitable substitute for infinity in this case. The MinDistance and MinDistance2 methods can be found in Appendix 5 and Appendix 6 respectively.

## Generating Test Data and Running Experiments

To properly compare the algorithms, the test array was populated with pseudo-random numbers before being run through both algorithms. The pseudo-random numbers were generated using C++’s *‘random’* library by using a Uniform Integer Distribution, seeded by a Mersenne Twister 19937 generator, between after being adapted from cppreference.com’s implementation [3].

The size of the test array varied from 25 to 10,000, and increased by 25 at each step. The operations counting was only carried out once per array size, because the number of operations carried out by both algorithms is constant for a given input size. There is however variation in the execution time, so for each input size 50 different arrays were tested on both algorithms to obtain an average.

Finally, the test results were written to .CSV files. This was so that the results could easily be output to and read by MATLAB, and accurate plots could be obtained. Allain’s use of the *‘fstream’* and *‘iostream’* libraries were consulted to achieve this [4].

## Implementation of Basic Operations Counters

The basic operations were counted once within the main loop for each distinct array size. To achieve this, the algorithms had to be modified to include a counter that is incremented each time the basic operation was executed. The counter for MinDistance and MinDistance2 were the global long integers and respectively.

The implementation of the modified algorithms was the separate methods MinDistance\_OpsCount and MinDistance2\_OpsCount, which are included in Appendix 7 and Appendix 8 respectively. In MinDistance\_OpsCount the counter is incremented immediately before the first array element comparison, which was separated from the initial condition to properly implement the counter, and immediately after the second comparison. In MinDistance2\_OpsCount the counter is only incremented after the assignment of the variable, which performs the same array element comparison the first algorithm.

After the basic operations had been counted, the program moved on to the execution timing. Once this was complete, the results of this part of the experiment were written to the Ops CSV file.

## Implementation of Execution Timers

The execution time of each algorithm was measured within the main loop, and was measured 50 times for each distinct array size to find a reasonable average. Both algorithms were timed by using the *“chrono”* library by adapting Smistad’s method [5]. For the first algorithm, the timer was started, then the MinDistance method was run, and then the stop time was added to the global long integer – the same process was then performed for the second algorithm, except using the MinDistance2 method and the variable.

The process of timing the algorithm’s execution time was performed 50 times on each algorithm. Once this was completed, the and variables were divided by 50 to obtain an average, and these averages were written to the Time CSV file.

# Experimental Results

## Functional Testing

Prior to the experiments, the functionality of the code was tested to ensure the algorithms behaved in the expected way. This was performed by calling the FunctionalTesting method, which is included in Appendix 9, at the start of the Main method. If any of the tests failed the Main method would return a value of 1 and exit, indicating an error. Five tests were performed on both the MinDistance and MinDistance2 functions, to check:

1. The general case
2. When the smallest distance is between the first and last terms of the array
3. When the smallest distance is between the last two elements of the array
4. The extreme case that there are two identical elements (i.e. )
5. Behaviour when negative numbers are introduced

Figure 1 below shows the first twelve lines of console output of the program, which confirms that the methods follow the algorithms as expected.

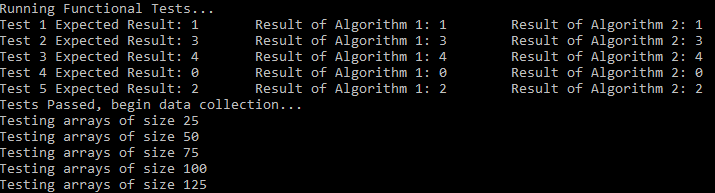


Figure 1: Console output after successful functional testing

## Number of Basic Operations

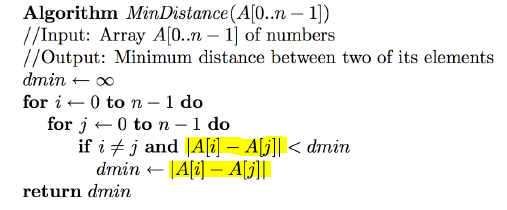
## Execution Time

# References

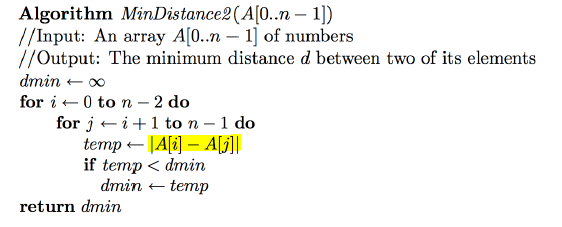
|  |  |
| --- | --- |
| [1] | A. Levitin, Introduction to the design and analysis of algorithms, 2nd ed., Pearson Addison-Wesley, 2007. |
| [2] | "Albatross", “C++: A Brief Description,” cplusplus.com, 2017. [Online]. Available: http://www.cplusplus.com/info/description/. [Accessed 17 May 2017]. |
| [3] | cppreference.com, “cppreference.com,” 2017. [Online]. Available: httpp://en.cppreference.com/w/cpp/numeric/random/uniform\_int\_distribution. [Accessed May 2017]. |
| [4] | A. Allain, “C++ File I/O,” Cprogramming.com, 2011. [Online]. Available: http://www.cprogramming.com/tutorial/lesson10.html. [Accessed May 2017]. |
| [5] | E. Smistad, “Measuring runtime in milliseconds using the C++ 11 chrono library,” Erik Smistad, 2012. [Online]. Available: https://www.eriksmistad.no/measuring-runtime-in-milliseconds-using-the-c-11-chrono-library/ . [Accessed May 2017]. |

# Appendix

1. MinDistance Algorithm



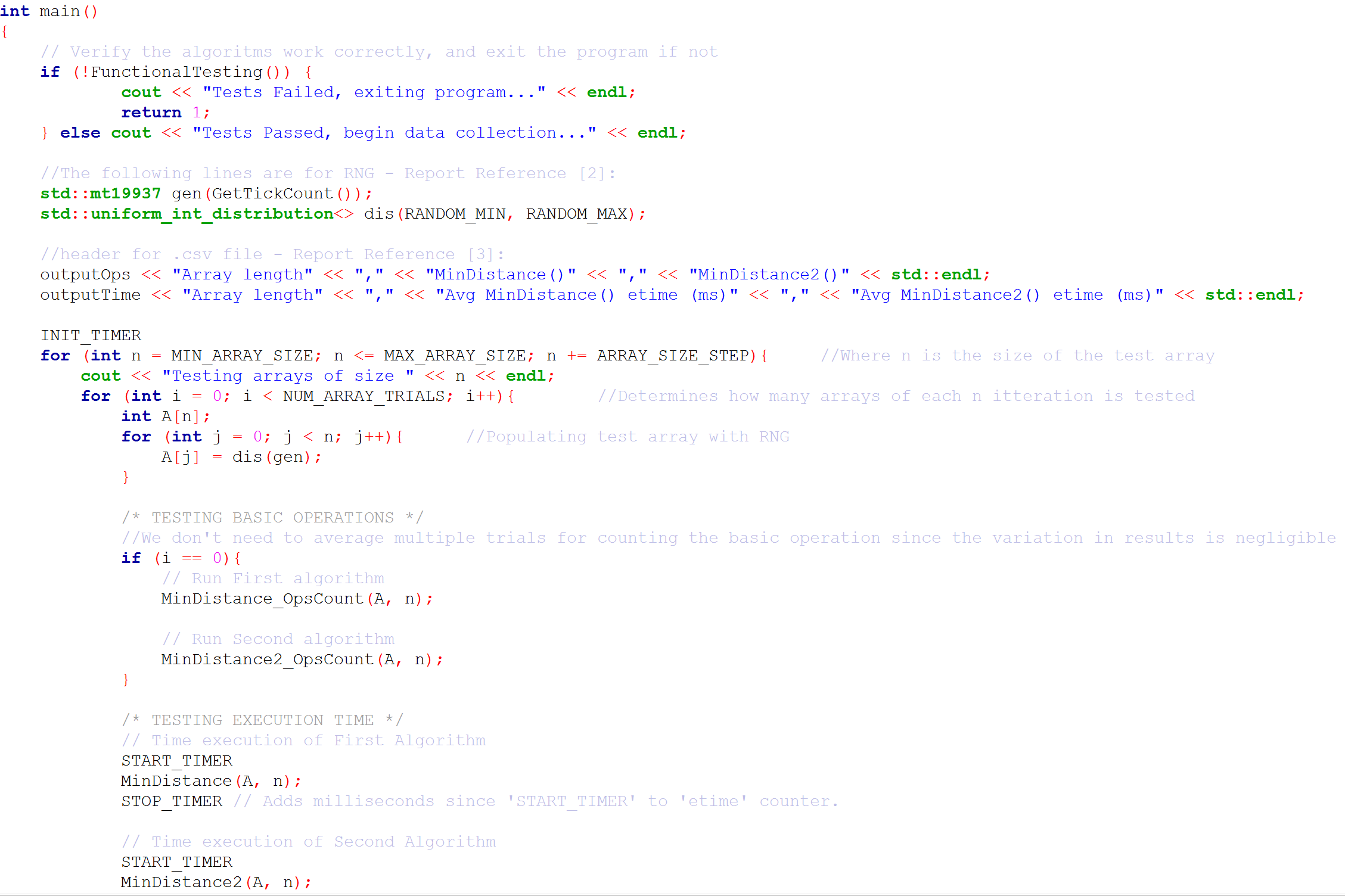
1. MinDistance2 Algorithm

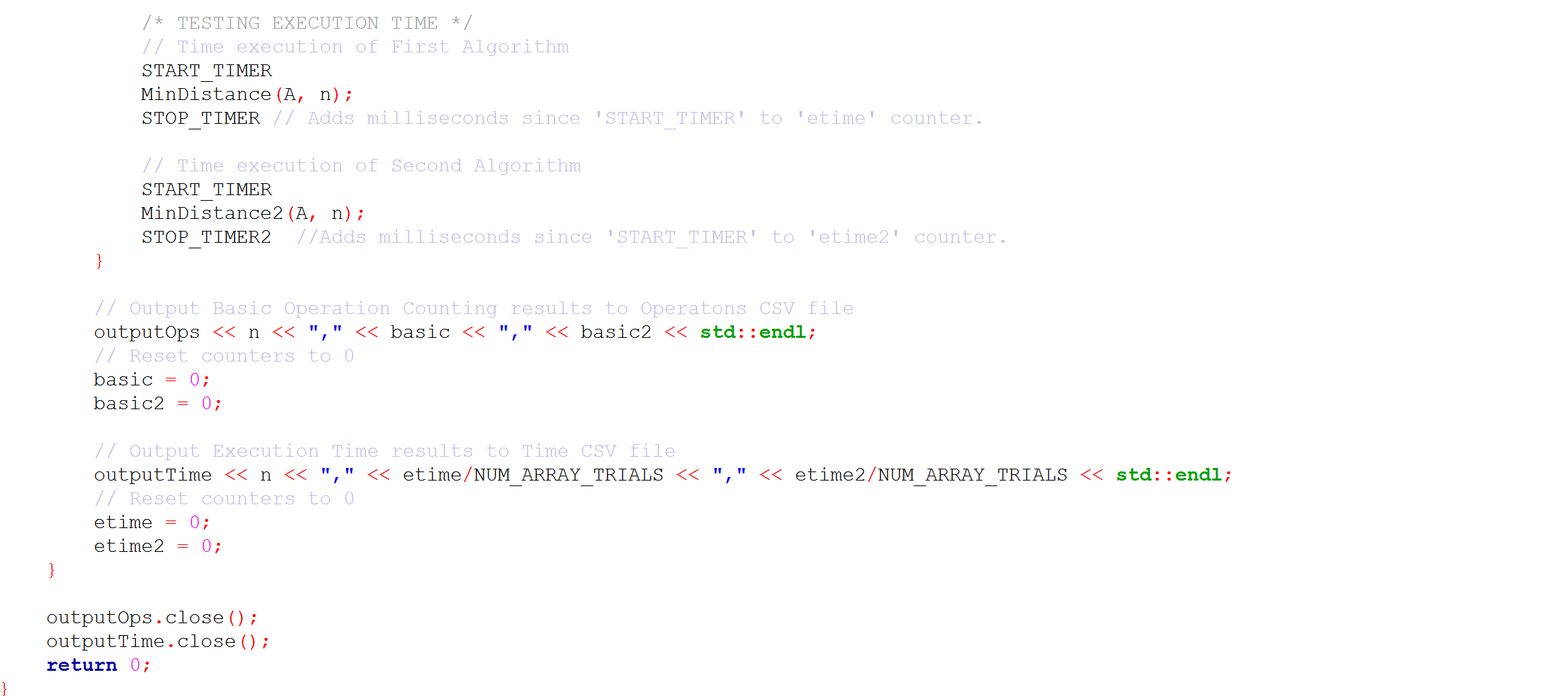


1. Setup of C++ Program

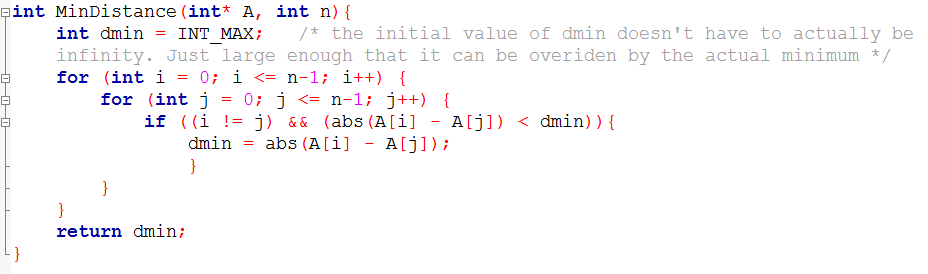


1. Main Method

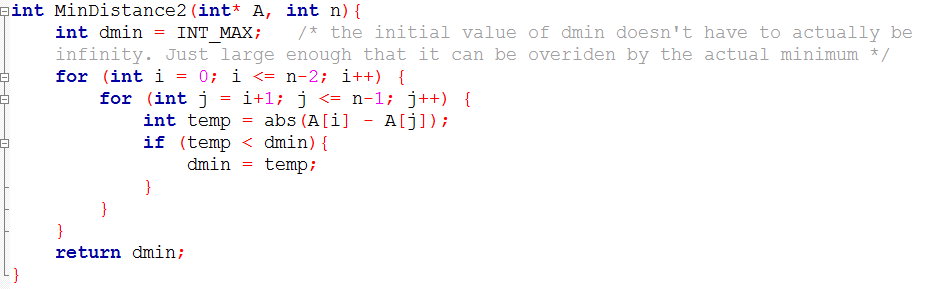




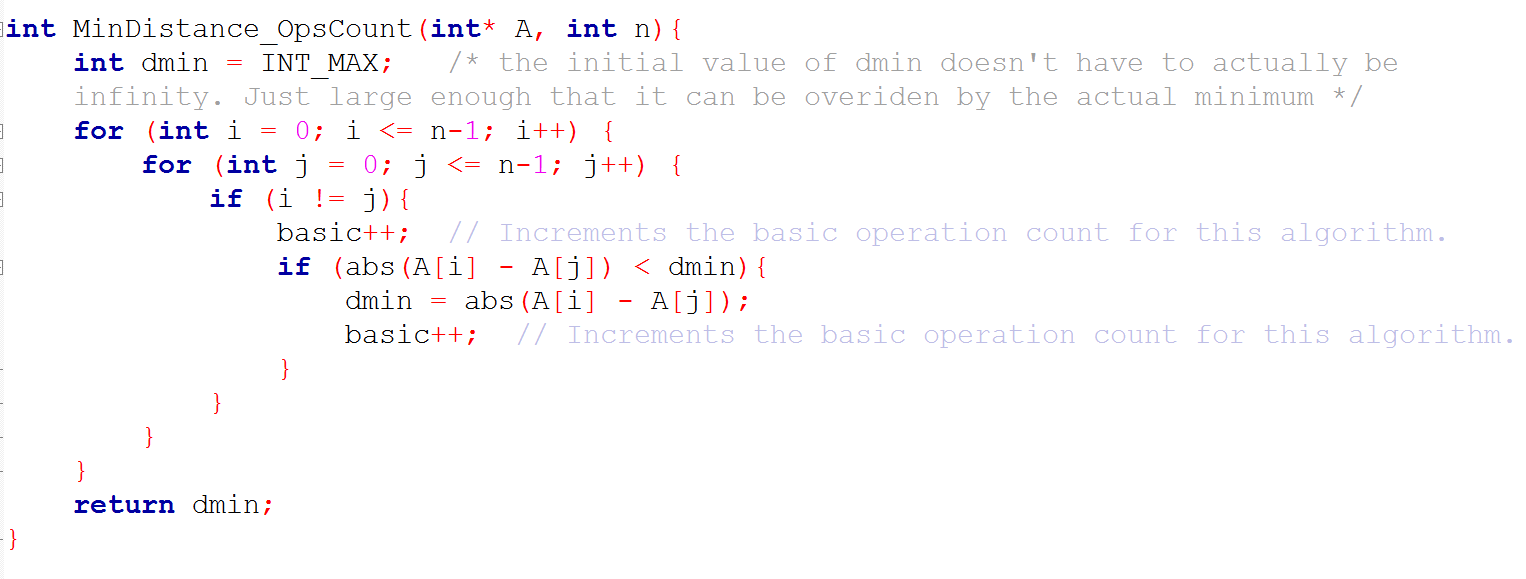
1. MinDistance C++ Implementation



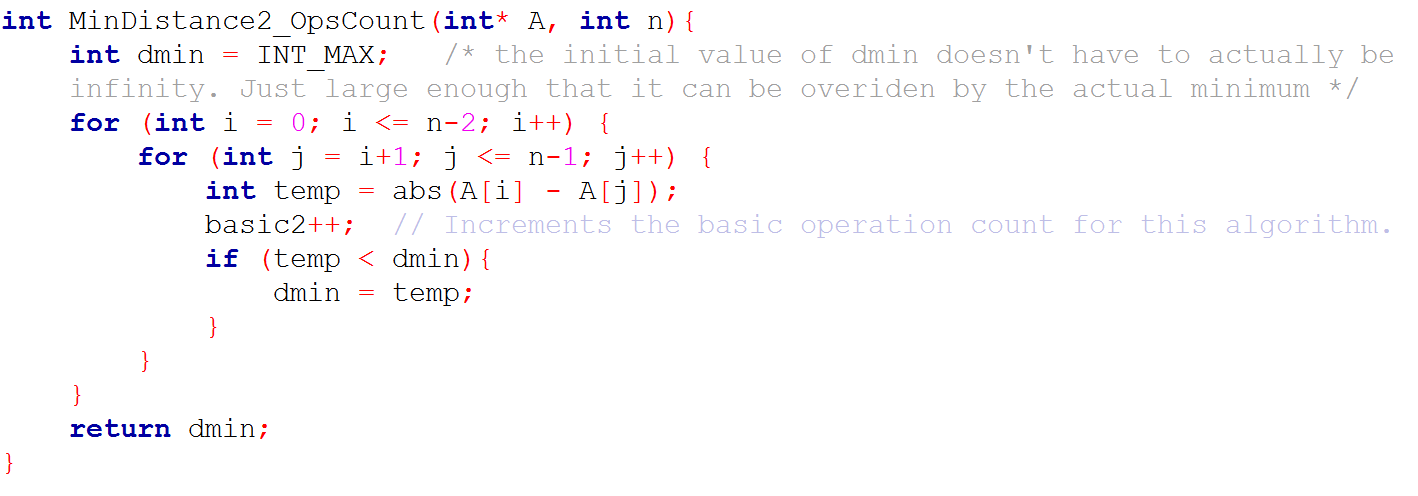
1. MinDistance2 C++ Implementation

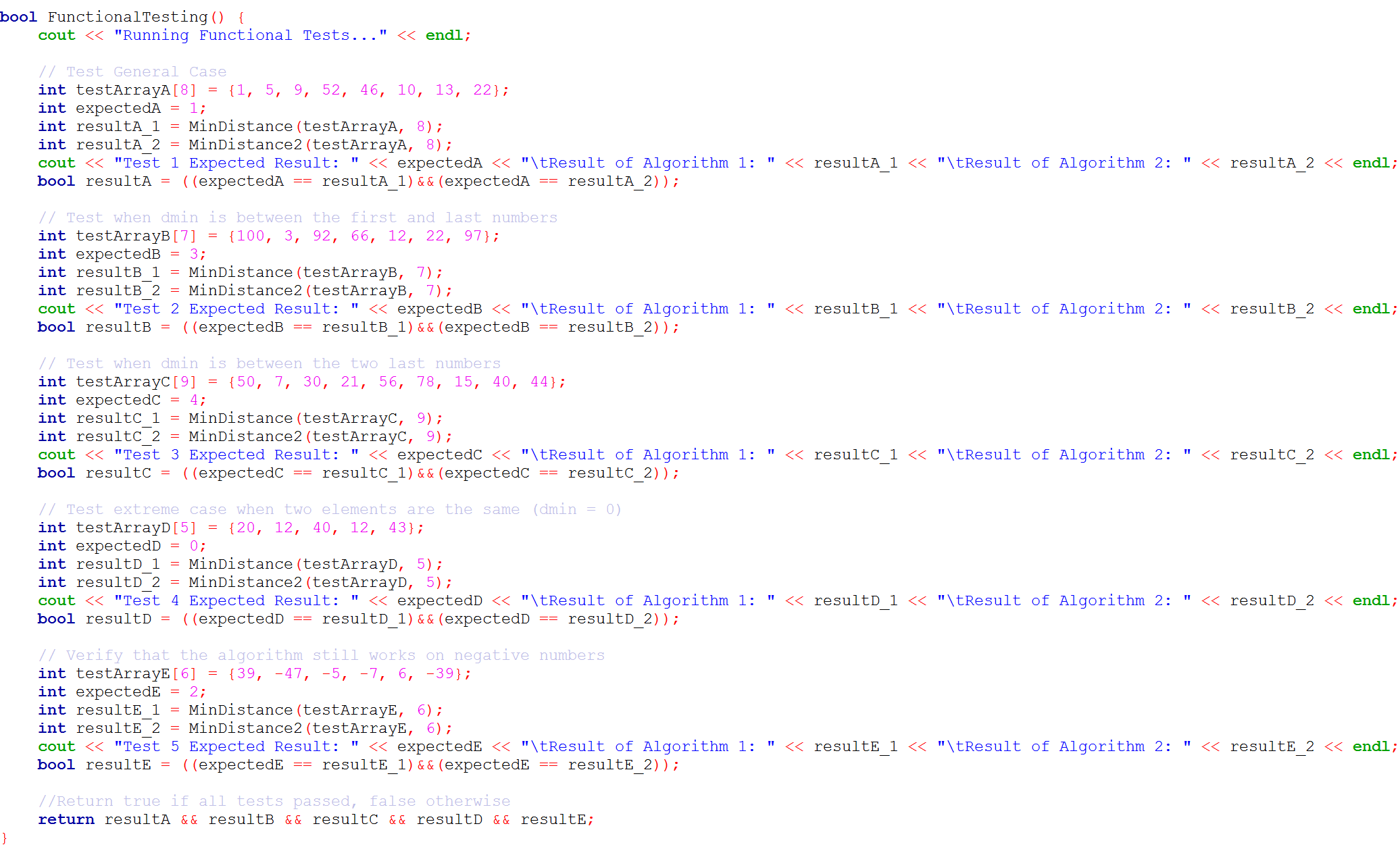


1. MinDistance\_OpsCount C++ Implementation



1. MinDistance2\_OpsCount C++ Implementation



1. FunctionalTesting Method