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: 21st May 2017

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.

1. 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 $dmin$ to an arbitrarily large value, ideally infinity. Then, it iterates over each element in the array A with an index variable $i,$ and then again with an index variable $j.$ The algorithm then asks: first ensure the element is not being compared to itself (check $i \neq j$); and if not, is the difference between the two elements smaller than the current minimum (check $[A[i] - A[j]] < dmin$). If this difference is the smallest, set $dmin$ to this value. After checking every permutation of i and j, the algorithm returns the minimum value and exits.

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

2. Theoretical Analysis of the Algorithms

Choice of Basic Operation

To conduct a successful comparison, it makes sense to choose an operation that is common to both algorithms to serve as the basic operation. The common operation that was identified as the basic operation in both algorithms are highlighted in both Appendix 1: and Appendix 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 checking whether the index variables i and j are different values. The second execution 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 its 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 (aside from the fact that it is the only operation common to both algorithms) was because in both algorithms it was not only the most performed operation but also the most expensive operation. While most other operations are either comparisons or assignments, in the identified operation the combination of the subtraction performed and the calculation of the absolute value is most complex part of each algorithm and therefore is the operation that should 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 their only input.

Determining the Average-Case Efficiency

The estimated average-case efficiency of the *MinDistance* algorithm is $\boldsymbol{C_{avg}(n) = n(n-1)}$**.** The inner part of that equation is derived from the fact that the algorithm is 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, hence $n - 1$. This value is then multiplied by n since the outer for-loop will be iterated n times before it exits, which equates to one iteration for each element in the array A. **Therefore,** $\boldsymbol{C_{avg}(n) = n(n-1)}$ **OR** $\boldsymbol{n^2 - n}$**.**

Note that this equation does not account for the small amount of iterations where the operation is performed an extra time due to the current value being less than $dmin$ and the assignment inside the if-statement being carried out. This is because it would have a negligible effect on the overall average number of operations performed that it would not affect the expected trends or order of growth.

The estimated average-case efficiency of the *MinDistance2* algorithm is $\boldsymbol{Cavg(n) = \frac{n(n-1)}{2}}$**.** This is like the first algorithm, except instead of operations being skipped in the event when the *ith* element equals the *jth* the i will never be equal to j since j is never less than $i + 1$. But the effect is the same in that the efficiency will include an $n(n-1)$ term. It is then halved due to the number of iterations 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 in the first algorithm have not been performed here due to the inner for-loop's assignment of j. **Therefore,** $\boldsymbol{C_{avg}(n) = \frac{n(n-1)}{2}}$ **OR** $\boldsymbol{\frac{n^2 - n}{2}}$**.**

From these equations, we can identify that both algorithm's efficiency classes and orders of growth are quadratic $[\boldsymbol{\Theta(n^2)}]$**.** We can also see, given the efficiency equations, that *MinDistance2* should perform half as many basic operations as *MinDistance*, strongly indicating that it will be the more efficient of the two in terms of average execution time for input arrays of the same size (n), despite both algorithms having the same order of growth. This gives us an idea of what type of trend lines we can expect when our test results are plotted; two quadratic lines where *MinDistance* reaches higher on the y axis (performed operations or execution time) than *MinDIstance2* at the same point along the x axis (array size).

Due to the number of operations performed being almost exclusively dependent on the size of the array (n) in both algorithms, the best and worst case scenarios are essentially the same, bar some minor discrepancies in *MinDistance* mentioned above that effect only the constants of the equation. Since there are no opportunities for either algorithm to "exit early" the efficiency will remain at $\boldsymbol{\Theta(n^2)}$ regardless of what the algorithms take as their input arrays**.**

3. 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.

The results of the experiments were plotted using MATLAB. This was chosen because once the results had been collected the script file *plotResults.m* (Appendix 3:) could be executed to automatically generate the plots directly from the CSV files, without the need to copy the data by other means.

Implementation of Algorithms

Both algorithms were implemented as C++ methods that were called within the *Main* method. 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 4:, and the *Main* method can be found in Appendix 5:.

The translation of both algorithms from pseudocode to C++ was straightforward, with the only significant difference being that the initial value of $dmin$ was set to the maximum allowable integer value of INT_MAX rather than the impossible value of infinity – the risk of the distance between two values being larger than INT_MAX is negligible, therefore INT_MAX is a suitable substitute for infinity in this case. The *MinDistance* and *MinDistance2* methods can be found in Appendix 6: and Appendix 7: respectively.

Generating Test Data and Running Experiments

To properly compare the algorithms, the test array A 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 $\pm 500{,}000{,}000$ 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, both 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 $basic$ and $basic2$ respectively.

The implementation of the modified algorithms was the separate methods *MinDistance_OpsCount* and *MinDistance2_OpsCount*, which are included in Appendix 8: and Appendix 9: respectively. In *MinDistance_OpsCount* the counter is incremented immediately before the first array element comparison, which was separated from the initial $i \neq j$ 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 $temp$ variable, which performs the same array element comparison as the first algorithm.

After the basic operations had been counted, the program moved on to the execution timing. Once this was complete, all 400 results from 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 $etime$ – the same process was then performed for the second algorithm, except using the *MinDistance2* method and the $etime2$ variable.

The process of timing the algorithm's execution time was performed 50 times on each algorithm. Once this was completed, the $etime$ and $etime2$ variables were divided by 50 to obtain an average, and all 400 of these averages were written to the Time CSV file.

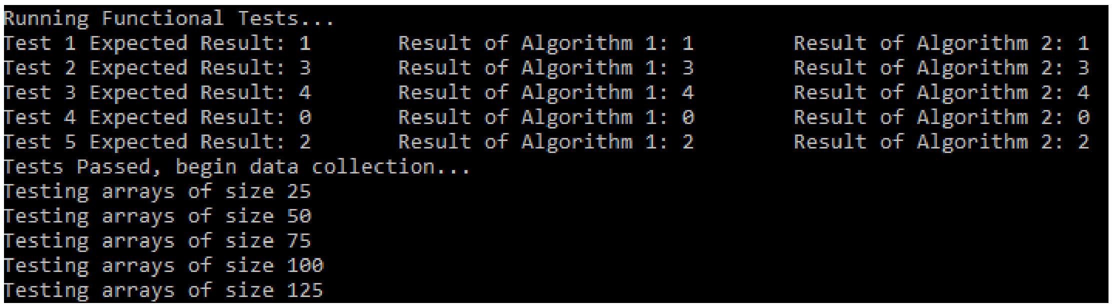
4. Experimental Results

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 10:, 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* methods, 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. $dmin = 0$)
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.



```
Running Functional Tests...
Test 1 Expected Result: 1       Result of Algorithm 1: 1        Result of Algorithm 2: 1
Test 2 Expected Result: 3       Result of Algorithm 1: 3        Result of Algorithm 2: 3
Test 3 Expected Result: 4       Result of Algorithm 1: 4        Result of Algorithm 2: 4
Test 4 Expected Result: 0       Result of Algorithm 1: 0        Result of Algorithm 2: 0
Test 5 Expected Result: 2       Result of Algorithm 1: 2        Result of Algorithm 2: 2
Tests Passed, begin data collection...
Testing arrays of size 25
Testing arrays of size 50
Testing arrays of size 75
Testing arrays of size 100
Testing arrays of size 125
```

Figure 1: Console output after successful functional testing

Number of Basic Operations

The 400 results of the basic operation counting are shown in Figure 1 overleaf. Both algorithms appear to follow a parabolic trend, thus confirming that both algorithms belong to the efficiency class $\Theta(n^2)$. Furthermore, the number of basic operations executed by the second algorithm is far less than that of the first – thus demonstrating that ***MinDistance2* is more efficient than *MinDistance*** in terms of executions of the basic operation.

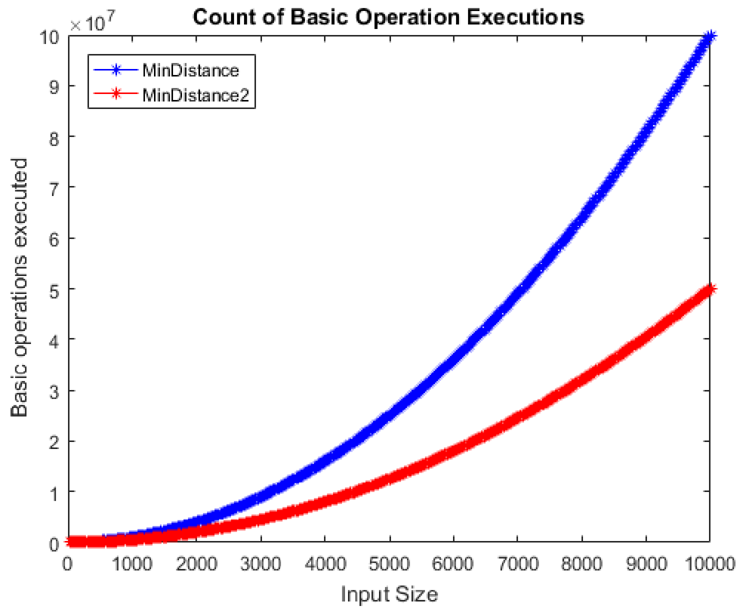


Figure 2: Result of Basic Operation Counting

Execution Time

Figure 3 below shows the 400 average execution times in milliseconds for both algorithms for 50 trials on a given input size. The spikes in the data, which occur for the same array size in both algorithms, could be attributed to several unknown factors, including the arrays being populated with uncharacteristically large data values to the computer running other programs in the background at the same time during the testing of a size. In either case, the parabolic trend in the data is again made clear, and the second algorithm consistently finishes faster than the first. **This therefore confirms that both algorithms belong to the average efficiency class $\Theta(n^2)$, and that *MinDistance2* is the more efficient of the two algorithms.**

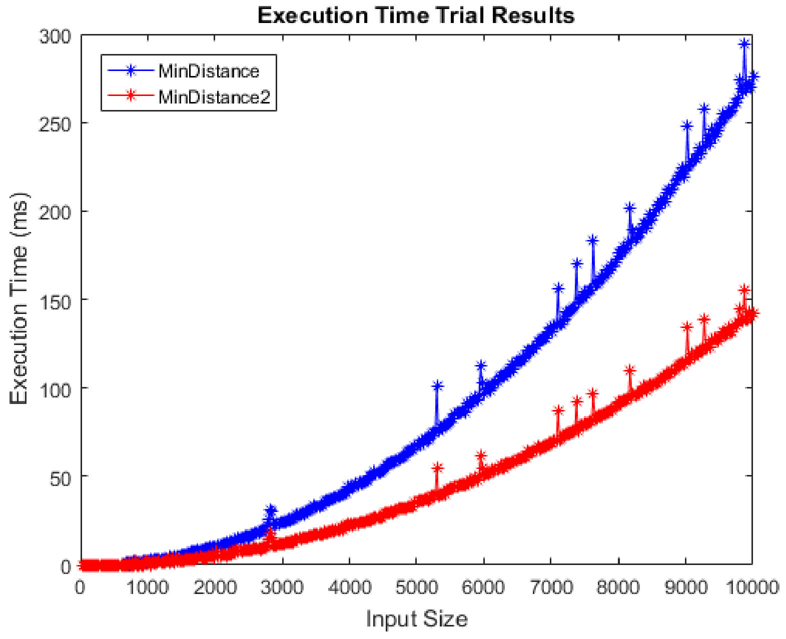


Figure 3: Result of Execution Time Trials

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

Appendix 1: MinDistance Algorithm

```
Algorithm $MinDistance(A[0..n-1])$
//Input: Array $A[0..n-1]$ of numbers
//Output: Minimum distance between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $n-1$ do
        if $i \neq j$ and $|A[i] - A[j]| < dmin$
            $dmin \leftarrow |A[i] - A[j]|$
return $dmin$
```

Appendix 2: MinDistance2 Algorithm

```
**Algorithm** $MinDistance2(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance $d$ between two of its elements
$dmin \leftarrow \infty$
**for** $i \leftarrow 0$ **to** $n-2$ **do**
    **for** $j \leftarrow i+1$ **to** $n-1$ **do**
        $temp \leftarrow |A[i] - A[j]|$
        **if** $temp < dmin$
            $dmin \leftarrow temp$
**return** $dmin$
```

Appendix 3: MATLAB Plotting Script

```
% Plotting command for CAB301 Assignment 2 Semester 1 2017
% Author: Shane Havers n9477756
clear; close all; clc;
%% Read In Data
dataOps = csvread('OutputOps.csv', 1, 0);
dataTime = csvread('OutputTime.csv', 1, 0);

%% Plot Data for Basic Operations Counting
figure()
plot(dataOps(:,1), dataOps(:,2), 'b-*');
hold on
plot(dataOps(:,1), dataOps(:,3), 'r-*');
hold off
title('Count of Basic Operation Executions')
xlabel('Input Size')
ylabel('Basic operations executed')
legend('MinDistance', 'MinDistance2', 'Location', 'northwest')

%% Plot Data for Execution Times
figure()
plot(dataTime(:,1), dataTime(:,2), 'b-*');
hold on
plot(dataTime(:,1), dataTime(:,3), 'r-*');
hold off
title('Execution Time Trial Results')
xlabel('Input Size')
ylabel('Execution Time (ms)')
legend('MinDistance', 'MinDistance2', 'Location', 'northwest')
```

Appendix 4: Setup of C++ Program

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <random>
#include <windows.h>
#include <chrono>

using namespace std;

ofstream outputOps("OutputOps.csv");
ofstream outputTime("OutputTime.csv");

//Following all taken from Report Reference [4], Except for 'STOP_TIMER' which is modified to suit our tests
#define TIMING

#ifdef TIMING
#define INIT_TIMER auto start = std::chrono::high_resolution_clock::now();
#define START_TIMER  start = std::chrono::high_resolution_clock::now();
#define STOP_TIMER  etime += std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::high_resolution_clock::now()-start).count();
#define STOP_TIMER2  etime2 += std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::high_resolution_clock::now()-start).count();
#else
#define INIT_TIMER
#define START_TIMER
#define STOP_TIMER
#define STOP_TIMER2
#endif

#define MIN_ARRAY_SIZE 100
#define MAX_ARRAY_SIZE 10000
#define ARRAY_SIZE_STEP 25
#define NUM_ARRAY_TRIALS 50

#define RANDOM_MIN -500000000
#define RANDOM_MAX 500000000

bool FunctionalTesting();
int MinDistance(int* A, int n);
int MinDistance2(int* A, int n);
int MinDistance_OpsCount(int* A, int n);
int MinDistance2_OpsCount(int* A, int n);

long long int basic = 0;  //basic op counter for MinDistance()
long long int basic2 = 0;  //basic op counter for MinDistance2()
long long int etime = 0;  //execution time counter for MinDistance()
long long int etime2 = 0;  //execution time counter for MinDistance2()
```

Appendix 5: Main Method

```
int main()
{
    // Verify the algoritms work correctly, and exit the program if not
    if (!FunctionalTesting()) {
            cout << "Tests Failed, exiting program..." << endl;
            return 1;
    } else cout << "Tests Passed, begin data collection..." << endl;

    //The following lines are for RNG - Report Reference [2]:
    std::mt19937 gen(GetTickCount());
    std::uniform_int_distribution<> dis(RANDOM_MIN, RANDOM_MAX);

    //header for .csv file - Report Reference [3]:
    outputOps << "Array length" << "," << "MinDistance()" << "," << "MinDistance2()" << std::endl;
    outputTime << "Array length" << "," << "Avg MinDistance() etime (ms)" << "," << "Avg MinDistance2() etime (ms)" << std::endl;

    INIT_TIMER
    for (int n = MIN_ARRAY_SIZE; n <= MAX_ARRAY_SIZE; n += ARRAY_SIZE_STEP){     //Where n is the size of the test array
        cout << "Testing arrays of size " << n << endl;
        for (int i = 0; i < NUM_ARRAY_TRIALS; i++){         //Determines how many arrays of each n itteration is tested
            int A[n];
            for (int j = 0; j < n; j++){       //Populating test array with RNG
                A[j] = dis(gen);
            }

            /* TESTING BASIC OPERATIONS */
            //We don't need to average multiple trials for counting the basic operation since the variation in results is negligible
            if (i == 0){
                // Run First algorithm
                MinDistance_OpsCount(A, n);

                // Run Second algorithm
                MinDistance2_OpsCount(A, n);
            }

            /* TESTING EXECUTION TIME */
            // Time execution of First Algorithm
            START_TIMER
            MinDistance(A, n);
            STOP_TIMER // Adds milliseconds since 'START_TIMER' to 'etime' counter.


            // Time execution of Second Algorithm
            START_TIMER
            MinDistance2(A, n);
            STOP_TIMER2  //Adds milliseconds since 'START_TIMER' to 'etime2' counter.
        }

        // Output Basic Operation Counting results to Operatons CSV file
        outputOps << n << "," << basic << "," << basic2 << std::endl;
        // Reset counters to 0
        basic = 0;
        basic2 = 0;

        // Output Execution Time results to Time CSV file
        outputTime << n << "," << etime/NUM_ARRAY_TRIALS << "," << etime2/NUM_ARRAY_TRIALS << std::endl;
        // Reset counters to 0
        etime = 0;
        etime2 = 0;
    }

    outputOps.close();
    outputTime.close();
    return 0;
}
```

Appendix 6: MinDistance C++ Implementation

```
int MinDistance(int* A, int n){
    int dmin = INT_MAX;    /* the initial value of dmin doesn't have to actually be
    infinity. Just large enough that it can be overiden by the actual minimum */
    for (int i = 0; i <= n-1; i++) {
        for (int j = 0; j <= n-1; j++) {
            if ((i != j) && (abs(A[i] - A[j]) < dmin)){
                dmin = abs(A[i] - A[j]);
                }
        }
    }
    return dmin;
}
```

Appendix 7: MinDistance2 C++ Implementation

```
int MinDistance2(int* A, int n){
    int dmin = INT_MAX;    /* the initial value of dmin doesn't have to actually be
    infinity. Just large enough that it can be overiden by the actual minimum */
    for (int i = 0; i <= n-2; i++) {
        for (int j = i+1; j <= n-1; j++) {
            int temp = abs(A[i] - A[j]);
            if (temp < dmin){
                dmin = temp;
            }
        }
    }
    return dmin;
}
```

Appendix 8: MinDistance_OpsCount C++ Implementation

```
int MinDistance_OpsCount(int* A, int n){
    int dmin = INT_MAX;    /* the initial value of dmin doesn't have to actually be
    infinity. Just large enough that it can be overiden by the actual minimum */
    for (int i = 0; i <= n-1; i++) {
        for (int j = 0; j <= n-1; j++) {
            if (i != j){
                basic++;  // Increments the basic operation count for this algorithm.
                if (abs(A[i] - A[j]) < dmin){
                    dmin = abs(A[i] - A[j]);
                    basic++;  // Increments the basic operation count for this algorithm.
                }
            }
        }
    }
    return dmin;
}
```

Appendix 9: MinDistance2_OpsCount C++ Implementation

```
int MinDistance2_OpsCount(int* A, int n){
    int dmin = INT_MAX;   /* the initial value of dmin doesn't have to actually be
    infinity. Just large enough that it can be overiden by the actual minimum */
    for (int i = 0; i <= n-2; i++) {
        for (int j = i+1; j <= n-1; j++) {
            int temp = abs(A[i] - A[j]);
            basic2++;  // Increments the basic operation count for this algorithm.
            if (temp < dmin){
                dmin = temp;
            }
        }
    }
    return dmin;
}
```

Appendix 10: FunctionalTesting Method

```
bool FunctionalTesting() {
    cout << "Running Functional Tests..." << endl;

    // Test General Case
    int testArrayA[8] = {1, 5, 9, 52, 46, 10, 13, 22};
    int expectedA = 1;
    int resultA_1 = MinDistance(testArrayA, 8);
    int resultA_2 = MinDistance2(testArrayA, 8);
    cout << "Test 1 Expected Result: " << expectedA << "\tResult of Algorithm 1: " << resultA_1 << "\tResult of Algorithm 2: " << resultA_2 << endl;
    bool resultA = ((expectedA == resultA_1)&&(expectedA == resultA_2));

    // Test when dmin is between the first and last numbers
    int testArrayB[7] = {100, 3, 92, 66, 12, 22, 97};
    int expectedB = 3;
    int resultB_1 = MinDistance(testArrayB, 7);
    int resultB_2 = MinDistance2(testArrayB, 7);
    cout << "Test 2 Expected Result: " << expectedB << "\tResult of Algorithm 1: " << resultB_1 << "\tResult of Algorithm 2: " << resultB_2 << endl;
    bool resultB = ((expectedB == resultB_1)&&(expectedB == resultB_2));

    // Test when dmin is between the two last numbers
    int testArrayC[9] = {50, 7, 30, 21, 56, 78, 15, 40, 44};
    int expectedC = 4;
    int resultC_1 = MinDistance(testArrayC, 9);
    int resultC_2 = MinDistance2(testArrayC, 9);
    cout << "Test 3 Expected Result: " << expectedC << "\tResult of Algorithm 1: " << resultC_1 << "\tResult of Algorithm 2: " << resultC_2 << endl;
    bool resultC = ((expectedC == resultC_1)&&(expectedC == resultC_2));

    // Test extreme case when two elements are the same (dmin = 0)
    int testArrayD[5] = {20, 12, 40, 12, 43};
    int expectedD = 0;
    int resultD_1 = MinDistance(testArrayD, 5);
    int resultD_2 = MinDistance2(testArrayD, 5);
    cout << "Test 4 Expected Result: " << expectedD << "\tResult of Algorithm 1: " << resultD_1 << "\tResult of Algorithm 2: " << resultD_2 << endl;
    bool resultD = ((expectedD == resultD_1)&&(expectedD == resultD_2));

    // Verify that the algorithm still works on negative numbers
    int testArrayE[6] = {39, -47, -5, -7, 6, -39};
    int expectedE = 2;
    int resultE_1 = MinDistance(testArrayE, 6);
    int resultE_2 = MinDistance2(testArrayE, 6);
    cout << "Test 5 Expected Result: " << expectedE << "\tResult of Algorithm 1: " << resultE_1 << "\tResult of Algorithm 2: " << resultE_2 << endl;
    bool resultE = ((expectedE == resultE_1)&&(expectedE == resultE_2));

    //Return true if all tests passed, false otherwise
    return resultA && resultB && resultC && resultD && resultE;
}
```