Queensland University of Technology

Comparison Between Two Algorithm Implementations

CAB301

Jeremiah Dufourq n9960651 & Eamon Elder n9963855

May 22, 2019

Contents

1 Description of algorithms

1.1 MinDistance1

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 $dmin$

Figure 1: The pseudocode for the MinDistance1 algorithm.

The ***MinDistance*** algorithm can be described as an algorithm which given an input array A, will return a variable $dmin$ which represents the distance between two closest elements in an array of numbers. The assumptions made include: input values are a vector of ints, and output value will be an int. Given all data is stored in ints, this presents a couple of limitations to the algorithm. Firstly, only whole numbers can be stored, and secondly the largest value for distance between elements, and the largest value possible for a given element will be the value for INT_MAX, as the concept of 'infinity' in this situation is limited to the highest value which an int can store. The algorithm works by firstly assigning infinity to the variable $dmin$ (in this case INT_MAX, which represents the largest number that can be stored by an int). A for loop then iterates with variable $i = 0$ incrementing up to $i = n - 1$. Within this loop is another for loop, creating a nested for loop, which iterates from $j = 0$ to $j = n - 1$. Within this second for loop is an if statement which evaluates and passes true if $i \neq j$ and $|A[i]–A[j]| < dmin$. If this passes as true, then $|A[i]–A[j]|$ is assigned to $dmin$, as the new minimum distance between elements. After both loops have terminated, $dmin$ is then returned. The use of a nested for loop allows two unique values within an array to be compared, and if the distance between them is less than the current minimum distance stored, this new value is the new minimum distance. By utilising two for loops, this algorithm can compare two different values in the same array, and iterate through both for loops, thereby comparing the absolute difference between every number in the given array. Because the lowest distance between integers could hypothetically occur between the second last and last integers in an array, every time this algorithm runs, every value will have to be checked against every other value. The final output of this algorithm will be the distance between the two closest elements. Because of the way in which the for loops are written, there will repeat code that will evaluate the distance between the same numbers twice. For

example, the 5th and 6th element in array A will be compared when $i = 5$ and $j = 6$, and when $i = 6$ and $j = 5$. This is where the second *MinDistance* algorithm is superior.

1.2 MinDistance2

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 $dmin$

Figure 2: The pseudocode for the MinDistance2 algorithm.

The ***MinDistance2*** algorithm is almost identical to the first algorithm. Functionally, both serve the same purpose. *MinDistance2* will also take an input array A and will evaluate the distance between the two closest elements, store this as a variable *dmin*, and then return this variable at the end of the algorithm. As before, the assumptions of this algorithm will be that: input values are a vector of ints, and output value will be an int. This will limit data to only whole numbers with a maximum element value, and maximum closest distance value possible being INT_MAX. However, *MinDistance2* is slightly more efficient in the way in which it calculates this value. Firstly, the INT_MAX is assigned to *dmin*. A for loop then iterates from $i = 0$ to $i = n - 2$, and within this loop, another for loop iterates from $j = i + 1$ to $j = n - 1$. This is evidently different from the first algorithm. The difference between $A[i]$ and $A[j](|A[i]–A[j]|)$ is assigned to a temporary variable *temp*. If this temporary variable is less than *dmin*, the value stored by *temp* is then assigned to *dmin*. When both for loops have iterated through all applicable values, *dmin* is then returned, and this value represents the distance between the closest two elements of the array. The way in which this algorithm differs, is by changing how the for loops iterate, thus eliminating recomputing the same expression. In the second loop, the variable j is initialised to $i + 1$. This allows the algorithm to not waste resources recomputing the same expression and is facilitated by taking the absolute value between $A[i]$ and $A[j]$. $A[i]–A[j]$ could potentially have a different value to $A[j]–A[i]$ (unless both $A[i]$ and $A[j]$ equal the same number), however since we are not interested in positive and negative, purely the absolute difference, we can take the absolute of the difference equation. This in turn allows us to reduce the number of computations required, as it is assumed that $|A[i]–A[j]|$ is equal to

$|A[j]–A[i]|$. This is where the second for loop iteration comes into play. By initialising the j value to $i + 1$, we reduce the total number of computations by $\frac{n^2-n}{2}$. This is because a j value cannot be compared with an i value that is equal or greater to it. For example we will never see the expression $|A[7]–A[6]|$ in *MinDistance2*, because $|A[6]–A[7]|$ will have already been evaluated. Additionally, $|A[7]–A[7]|$ would never occur because j is always initialised to 1 greater than i.

2 Theoretical analysis of algorithms

2.1 Choice of basic operation

For the first algorithm *MinDistance*, the basic operation was placed in the if statement that compares the value of $A[i]$ and $A[j]$ as well as determining if the difference between the two is less than *dmin*. This basic operator is placed as the second conditional statement, and the *basicOp* variable is incremented by 1, before determining if the *basicOp* variable is greater than 0. Given that the *basicOp* variable is initialised to 0, this will always be true. The reason this was chosen is because the algorithm will always have to evaluate this if statement, however it will not always evaluate to true. By placing the increment of the *basicOp* variable as the second conditional statement, behind evaluating if $i \neq j$, the *basicOp* variable will only increment whenever access to the element array is required. If the counter was placed before the if statement, then the if $i \neq j$ evaluation would also count as a basic operation. However, as this does not access the elements of the array, this was not considered a intensive operation. Additionally, *MinDistance2* does not contain an $i \neq j$ evaluation. By placing the basic operation counter where chosen, a meaningful comparison between the two algorithms is facilitated as both algorithms evaluate $|A[j]–A[i]| < dmin$. If the basicOp counter was not placed inside the conditional statements of the if statement, then whenever the statement evaluates $|A[j]–A[i]| < dmin$ to false the basicOp would not increment, even though the program has performed a 'basic operation'. Refer to equation 1 below for the basic operation of *MinDistance*.

$$|A[i] - A[j]| < dmin \tag{1}$$

1: MinDistance basic operation choice.

For the second algorithm *MinDistance2*, the basic operation was placed as the first condition for the if statement. The reasoning behind this is that by placing the increment here, the *basicOp2* variable will always increment, when the basic operation (as labelled by equation 2) is evaluated. This was to keep the number of operation position consistent between the two tests.

$$temp < dmin \tag{2}$$

2: MinDistance2 basic operation choice.

2.2 Choice of problem size

Regarding the functionality tests, each of the problem sets for those tests varied, and will be explained in later sections in the report. For the operations and timing tests, there was three subtests which were performed (random, reverse sorted and sorted). Each of these subtests had a problem input size from $0 - 5010$ with a step size of 501. This can be seen below in listing 1, whereby the ARRAY_STEP_SIZE = step size for random tests, SIMULATIONS = Number of simulations performed, and therefore the total range is $Range = ARRAY\_STEP\_SIZE \times SIMULATIONS$. At each same length array, the program performed 20 simulations. as defined by ARRAY_NUM_SIMS. In addition to this, each random variable was produced in the range of $0 - 100000$, as defined by RANDOM_RANGE. This problem size is deemed suitable given both algorithms will be tested using identical array sizes, and 5000 is an arbitrarily large number that should provide suitable answers for both algorithms.

Listing 1: Choice of problem size, *"headerFile.h"* file.

```
1 #define ARRAY_STEP_SIZE (501) // Defining the step size for the
    ↪ random array
2 #define ARRAY_NUM_SIMS (20) // Defining the number of same length
    ↪ arrays for the random implementation
3 #define RANDOM_RANGE (100000) // Range of the random variable
4 #define SIMULATIONS (10) // How many simulations that the program
    ↪ will run
5 #define ZERO_ELEMENTS (10) // The number for each element in the
    ↪ zero elements functional test
```

2.3 Average-case efficiency for MinDistance

The theoretical efficiency of this algorithm can be determined visually. Because the algorithm has to compare every element to every other element in the array, through the use of nested for loops, it can be seen that the average-case will be quadratic. Given all elements are checked against all others, the average-case efficiency can be determined as being in the class $\Theta(n^2)$. This can be seen because both for loops iterate from 0 to $n-1$, thus performing each loop n number of times. The algorithm will thus have an efficiency of $n \times n$, or n^2 (Efficiency of n^2). Despite only the average-case efficiency being evaluated, it can also be

seen that the best and worst-case efficiencies will also be the same as the average case, because the algorithm still has computer $n \times n$ iterations. Regardless of the input, the algorithm will always check every element against all other elements, and thus the best, worst and average case will all involve evaluating every element and will thus all have the same efficiency.

Substitution formulae;

$$\sum_{i=l}^{u} 1 = u - l + 1 \tag{3}$$

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i \tag{4}$$

$$\sum_{i=0}^{u-1} u - i = \frac{u(u+1)}{2} \tag{5}$$

Average case efficiency theoretical analysis, using the above substitution formulae, Using (1) where u = n - 1, l = 0,

$$C_{Average} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1 - n)$$

$$C_{Average} = (\sum_{j=0}^{n-1} ((n-1) - 0 + 1) - n)$$

$$C_{Average} = \sum_{j=0}^{n-1} (n) - n$$

Subbing (2) in for the summation of the equation, where c=n,

$$C_{Average} = n \sum_{j=0}^{n-1} (1) - n$$

Subbing in (1) where u=n-1,l=0

$$\begin{aligned} C_{Average} &= n((n-1)-0+1)-n \\ C_{Average} &= n^2 - n \\ \therefore C_{Average} &\in \Theta(n^2) \end{aligned}$$

2.4 Average-case efficiency for MinDistance2

The theoretical efficiency of this algorithm can also be determined visually. Similar to the previous algorithm, this algorithm will belong to the class $\Theta(n^2)$. This is because like before, the presence of the two nested for loops means that the algorithm efficiency belongs to the quadratic class. However, this algorithm is slightly different from the previous. The first for loop iterates from 0 to $n-2$, and the second for loop iterates from $i+1$ to $n-1$. This effectively alters the algorithms efficiency by only comparing the same pair of numbers once and avoiding recomputing the same expression in the innermost loop. Therefore, despite having an order of growth belonging to the same class, this algorithm MinDistance2 will be more efficient, by having a significantly smaller 'constant multiplier'.

$$C_{Average} = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (1)$$

Subbing (1), where u = n-1, l = j+1,

$$\begin{aligned} C_{Average} &= \sum_{i=0}^{n-2} ((n-1)-(i+1)+1) \\ C_{Average} &= \sum_{i=0}^{n-2} (n-1-i-1+1) \\ C_{Average} &= \sum_{i=0}^{n-2} n-i-1 \end{aligned}$$

Subbing (3) in, where n-1 = n-2, therefore result is n-2 not n-1,

$$
\begin{aligned}
C_{Average} &= (\frac{n(n+2))}{2} - 1) - 1 \\
C_{Average} &= (\frac{1}{2}(n^2 + n) - 1) - 1 \\
C_{Average} &= (\frac{1}{2}n^2 + \frac{1}{2}n - 1) - 1 \\
C_{Average} &= \frac{1}{2}n^2 - \frac{1}{2}n \\
\therefore C_{Average} &\in \Theta(n^2)
\end{aligned}
$$

3 Implementation of algorithms

3.1 Programming environment

The testing, computation, execution and compilation of the program was completed on a 2018 15" Macbook Pro running macOs Mojave (10.14.2). This computer has a 2.2GHz Intel core i7 (6 core) processor, 16 GB 2400 MHz DDR4 RAM and a Radeon Pro 555X GPU. Compilation was performed using the Clang++17 compiler in accordance to the ISO/IEC 14882:2017 current standard and execution was performed on JetBrains CLion Interactive Development Environment (version 2019.1) using cmake (version 3.11).

3.2 Programming language implementation

The language with which the program was implemented was chosen between C#, and C++. This was because in assignment 1, these were the two languages used. C++ was eventually chosen as this assignment was another opportunity to learn C++ better. It is also important that code be compiled and run quickly, as analysis of algorithms can be very memory intensive. C++ was thus determined as the better option.

3.3 Implementation of program and algorithm

The program was implemented faithfully, and a selection of tests were produced to confirm the functionality of the program, as well as measuring execution time and number of basic operations performed. The user is able to control the program from the command line, determining which set of tests are implemented. From beginning to end, the steps required on the part of the user:

- The user navigates to headerFile.h, and here they are able to change the variables that determine how the tests are generated.
- The user compiles and runs the code.
- The user is prompted to enter a 0, 1, or 2 to implement different testing (functionality, basic operations or timing respectively).
- The console will print the results to the console, all input and output data generated is written into CSV files.

Each algorithm was stored in a separate file. *"MinDistance.cpp"* contains the basic operation and timing implementation of the first algorithm, and *"MinDistance2.cpp"* contains the basic operation and timing implementation of the second algorithm. The difference between timing implementation and basic operator implementation for both algorithms is that the basic operator implementation contains a variable, *basicOps* and *basicOps2*, which increments every time a basic operation is performed. The timing implementation does not contain this in order to more accurately gauge the execution time of just the algorithm. For a more detailed representation of the program, **see figure 8 in the appendix for a top down view of the program** .

In addition to the above comments, it was decided that Object Orientated Programming would be used throughout the test to reduce complexity. Furthermore, it was discussed that learning OOP for C++ would be of benefit to both group members throughout their university degree. OOP was used in the creation of each test, whereby an object was created to store the input and output data of each test, to write to the csv files and to handle the console output. The class can be found under the *"testInterface.h"* file in figure 8 in the appendix.

4 Experimental design

The testing of the two algorithms can be broken down into three sections (table 1):

Table 1: Test types

Test Type	**Description**
Functional test	to test the functionality of the program and the algorithm to confirm that they are working as intended
Operational test	testing the number of basic operations performed within the algorithm
Timing test	testing the execution time of the algorithm for a given array size

In order to test the functionality of the program, the given test arrays were implemented table 2.

Table 2: Functionality test types

Test Type	**Description**
Even input	array with an even number of elements
Odd input	array with an odd number of elements
Pair of elements (2)	array with only two elements
Single element	array with only one element
Repeated elements	array that contains a number of duplicate elements
Large ordered	array with a larger number of elements in numerical order
Large reverse ordered	array with a larger number of elements in reverse numerical order
Large unordered	array with a large number of elements in random order
Negative unordered	array with negative elements unordered
Mix unordered	array with negative and positive elements unordered
No distance	array with no distance between any element (array of only ones)
Large distance	array with distance between elements set to largest distance possible (0 to INT_MAX)

The purpose of these tests is to test boundary cases for distance between elements (max distance, distance of 0, distance of 1), as well as testing how the algorithm interacts with negative elements, arrays of odd length, arrays of even length, ordered arrays, unordered arrays, and reverse ordered arrays.

For the testing of both execution time and number of basic operations performed, in order to accurately provide models for both algorithms, the way in which tests are generated, and performed are identical for both algorithms within both test types. The tests are as follows:

- Random array size and elements
- Random array size and reversed sorted elements
- Random array size and sorted elements

In *headerFile.h*, these variable can be changed, however at present the testing variables are as shown below (figure 3):

```
============[! WELCOME TO ALGOCRUNCH !]=============
Enter the program that you would like to run
FUNCTIONALITY=0, OPERATIONS=1, TIMING=2
0
This program is executing with the following variables:
+-- LARGE_ARRAY_VALUE: 2147483647
+-- LARGE_ARRAY_SIMS: 1
+-- ARRAY_STEP_SIZE: 501
+-- ARRAY_NUM_SIMS: 20
+-- RANDOM_RANGE: 100000
+-- SIMULATIONS: 13
Initilizing and completing the selected tests:
```

Figure 3: Variables used for the tests.

Given the current array step size, and the number of simulations, both algorithms for both timing and basic operation count will have 20 arrays from 501 to 5010 with a 501 step size. The number of arrays at each size (20) allows a very accurate model to be generated at each step size. The larger the number of arrays generated, the more accurate the model can be in terms of 'average' efficiency, as the larger the pool of data the less effect outliers will have on the model. The range in length of arrays was set as listed above because the range selected was deemed to be of a suitable size that execution time of the program was long enough to be able to accurately measure and model, while simultaneously keeping the program from draining too many resources. Because both algorithms are in the efficiency class of quadratics, if arrays of too large a length were to be tested the program would take too long to return output data.

After the results were gathered in the csv file for running the program, analysis was done on the results in Microsoft Excel. In this program, the results were averaged for the same length array tests, and then plotted on a scatter graph. The trend line from the average results was obtained using the trend line tool in Microsoft Excel. This trend line was used as the theoretical efficiency result for the algorithms.

5 Experimental results

This section will explain the purpose of each test, discuss the input variables, display the results from each of the tests, interpret the results in the context of the test, analyse the results compared to theoretical assumptions, and explain any discrepancies in the data.

5.1 Functionality test

The results form the functionality test is broken down into a table, where the input array is shown, alongside the expected and actual result. If the expected result matches the actual

result, then the test passes. In this case, all of the functionality tests passed

Table 3: Functionality test results

Test Number	**Input Array**	**Expected**	**Result**	**Pass/Fail**
EVEN_ARRAY	{1,2,3,4,5,6, 7,8,9,10}	1	1	PASS
ODD_INPUT	{1,2,3,4,5,6, 7,8,9,10,11}	1	1	PASS
PAIR_ELEM	{10,100}	90	90	PASS
SINGLE_ELEM	{1}	INT_MAX	INT_MAX	PASS
REPEAT_ELEM	{1,1,1,2,2,2,3,3, 4,4,4,4,5,5,5,6, 6,6,7,8,9,10,10}	0	0	PASS
LARGE_ORDERED	{1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15, 16,17,18,19,20}	1	1	PASS
LARGE_REVERSE_ORDERED	{20,19,18,17,16, 15,14,13,12,11,10, 9,8,7,6,5,4,3,2,1}	1	1	PASS
LARGE_UNORDERED	{50,62,85,40,3,520, 56,52,1,111,2,35,6, 105,4,8,7,5,10,88}	1	1	PASS
NEG_UNORDERED	{-5,-9,-6,-1,-2,-40,-10,-20}	1	1	PASS
MIX_UNORDERED	{5,20,-5,26,4,-2,0,1,-4}	1	1	PASS
NO_DIST	{1,1,1,1,1,1,1,1,1,1}	0	0	PASS
LARGE_DIST	{0,2147483647}	INT_MAX	INT_MAX	PASS

There was another functionality tests that was to be included, however due to computational constants it was not included in the final results. This functionality test includes the input array to be of length INT_MAX. In this test, the output would be whatever the minimum distance is between each of the elements. However, this test was thought to be included because it tests the edge case for when the array is of length INT_MAX. The largest array that can be inputted into each algorithm is the length of INT_MAX, due to the assignment of *dmin* as INT_MAX.

5.2 Operation test

For each of the algorithms, the basic operation was implemented using a global variable, which was out of the scope of the algorithm implementation.

$$A[i] - A[j] < dmin \tag{6}$$

3: MinDistance basic operation.

The first algorithm, *MinDistance*, implemented the basic operation as an unsigned long long it, and assigned it to the *basicOp* variable, initializing it as 0. The implementation of this algorithm was stored in the *"MinDistance.cpp"* file. This *basicOp* variable was pre incremented within the nested if statement of the algorithm. Because this variable was within the executable statement of the if statement, it had to be pre-increment to make

certain that it would record the number of operations. The location of the variable increment in the if statement also contributed to the accuracy of counting the basic operations. The basicOp variable was placed after the $i! = j$ comparison. The reasoning behind this was that if this evaluated to false, then it would not increment the *basicOp* variable. In this context, the only comparison which mattered was the $|A[i] - A[j]| < dmin$ comparison, and therefore the *basicOp* only incremented when this comparison was executed. The section in which the *basicOp* variable was incremented included the $++basicOp > 0$ statement. This was implemented in this fashion, as the *basicOp* counter would not go below 0, and would increment when the basic operation was true. Refer to listing 2 for the implementation of the basicOp variable, and basic operation for the first algorithm (*MinDistance*).

Listing 2: MinDistance basicOp implementation.

```
1
2  unsigned long long basicOp = 0;
3
4  // NumOps implementation of the algorithm
5  int MinDistance(vector<int> &A) {
6      // Input: Array of ints
7      // Output: Minimum distance between two of its elements
8      int size = A.size();
9      int dmin = INT_MAX;
10
11     for (int i = 0; i <= (size - 1); i++)
12     {
13         for(int j = 0; j <= (size - 1); j++)
14         {
15             if((i != j) && (++basicOp > 0) && (abs(A[i] - A[j]) <
                   ↪ dmin))
16             {
17                 dmin = abs(A[i] - A[j]);
18             }
19         }
20     }
21     return dmin;
22 }
```

$$temp < dmin \tag{7}$$

4: MinDistance2 basic operation.

and the basic operation for the second algorithm (MinDistance2).

The second algorithm implemented the *basicOp* variable much like the first algorithm. The variable was stored in an unsigned long long int global variable outside of the algorithm implementation, in the *"MinDistance2.cpp"* file. In addition to this, it was initialized to 0, and assigned the name *basicOp2*. Much like the first implementation, the method it was incremented by including inserting it within the condition for the nested if statement. In this case, the basic operation is the same as the first algorithm implementation. However, the key difference between both algorithm implementations is that the second algorithm assigns and initializes the $|A[i] - A[j]|$ section before the comparison. In any case, the *basicOp2* variable was pre incremented in the if statement, before the basic operation of $temp < dmin$. The pre increment made sure that the *basicOp2* variable was incremented before the execution of if statement body. For the second algorithm implementation, the basic operation was incremented in the following statement, $++basicOp2 > 0$. Because the *basicOp2* variable was incremented in the if statement, if the basic operation was true, then the *basicOp2* variable would increment. Refer below to listing 3 for the implementation of the *basicOp2* variable.

Listing 3: MinDistance2 basicOp2 implementation.

```
1
2  unsigned long long basicOp2 = 0;
3
4  // BasicOp implementation of the algorithm
5  int MinDistance2(vector<int> &A)
6  {
7      // Input: Array A[]
8      // Output: Minimum distance d between two of its elements
9      int size = A.size();
10     int dmin = INT_MAX;
11     int temp;
12
13     for (int i = 0; i <= (size - 2); i++)
14     {
15         for(int j = i + 1; j <= (size - 1); j++)
16         {
17             temp = abs(A[i] - A[j]);
18             if ((++basicOp2 > 0) && (temp < dmin))
19             {
20                 dmin = temp;
21             }
22         }
```

```
23        }
24        return dmin;
25    }
```

The basic operation tests were broken down into three different subtests; random, reversed random and random sorted arrays. The main goal of each of these tests was to confirm the basic operations of the algorithm in three scenarios in which the elements of the array were changed. In each of these tests, both the first algorithm *(MinDistance)* and the second algorithm *(MinDistance2)* were tested. For the sake of this report, this section will focus on the combination of all the three tests. For all of the tests, 20 simulations were performed on each array length, the array length was increased in a step size of 501 up to 5010, and the results from each of the same length arrays were averaged. The average of these tests was then plotted on a graph for graphical representation (refer to figure 4).

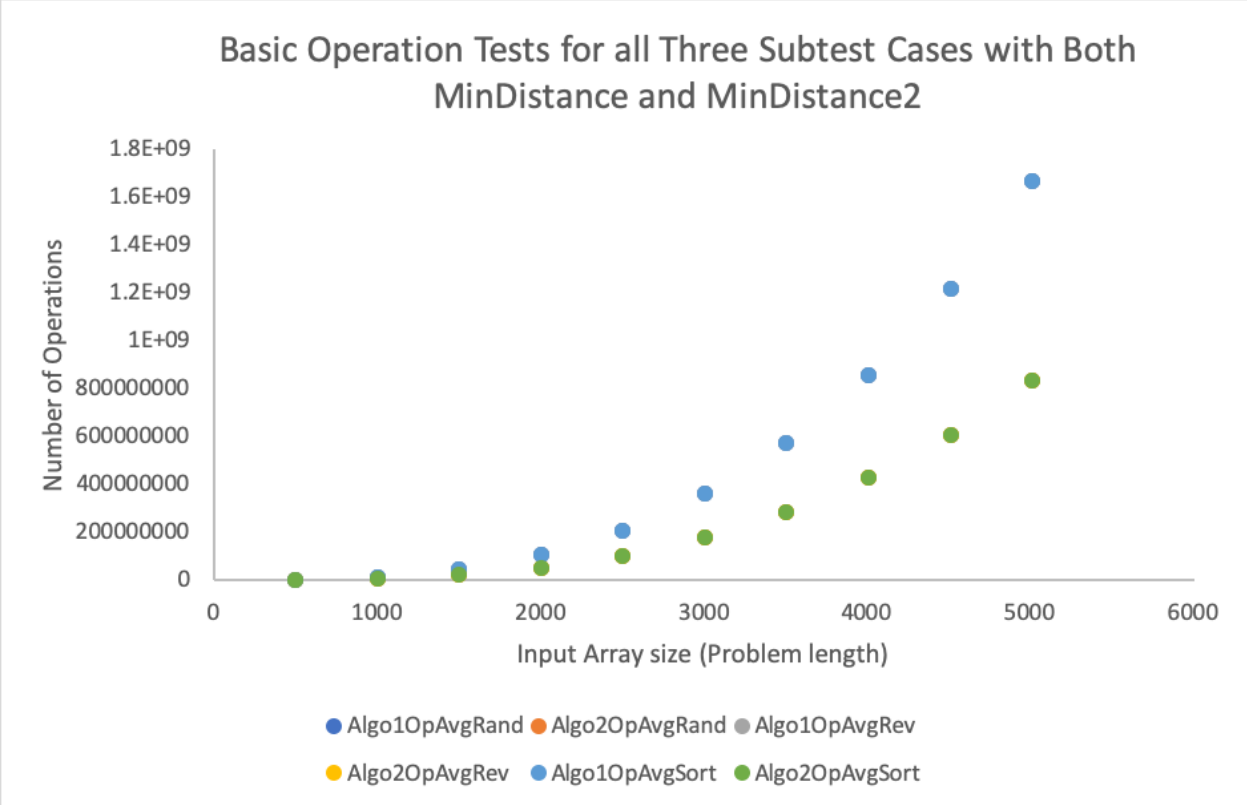

Figure 4: Basic operation tests for all three subtests with both MinDistance and MinDistance2.

It can be seen that there is a clear positive quadratic trend in both the *MinDistance* and the *MinDistance2* algorithms. In addition, all three of the subtests display the same data

for both algorithms. This would suggest that the arrangement of the elements in each array has no effect on the number of operations performed by the algorithm. Furthermore, the *MinDistance* algorithm has a greater rate of change than the *MinDistance2* algorithm. A more in-depth discussion of the basic operations for each algorithm will be explained in the analysis of experimental results section.

5.3 Timing test

The measurement of the execution times was consistent across both the *MinDistance* and *MinDistance2* algorithms. A time_point datatype from the high_resolution_clock namespace was used to store the time before and after the execution of the algorithm. The starting time point for the execution time was initialized directly before the execution of the algorithm, and the ending time point was initialized directly after the execution of the algorithm. Each time point was separated from the algorithm. For example, the *MinDistance* algorithm used *a1t1* as the starting time, and *a1t2* as the ending time, whereas the *MinDistance2* algorithm use *a2t1* as the starting time, and *a2t2* as the ending time. In addition to the separation of the beginning and ending times, the implementation of the *MinDistance* and *MinDistance2* algorithms did not include the number of operations functionalities. This was to make sure that the execution time test, and the basic operation tests were independent. The overall execution time of the algorithm was measured by subtracting the time after the execution of the algorithm, from the time before. This was measured in nanoseconds and stored in a time_point datatype (refer to listing 4). The method of measuring the execution time of both algorithms was kept consistent, so that the results could be compared meaningfully.

Listing 4: Execution time implementation - subsection of the code to run each test, saved in the *"runTests.cpp"* file.

```
1 for (auto &col : inputVector) // For the length of the input
    ↪ vector, run the test
2 {
3     // Running the tests, storing the results in a vector to
        ↪ compare
4     //Start the time
5     high_resolution_clock::time_point a1t1 = high_resolution_clock
        ↪ ::now();
6     int algo1 = MinDistanceTiming(col);
7     //End the time
8     high_resolution_clock::time_point a1t2 = high_resolution_clock
        ↪ ::now();
9     auto algo1TimeOutput = std::chrono::duration_cast<std::chrono
        ↪ ::nanoseconds>(a1t2-a1t1);
```

```
10
11      algo1ExecTime.push_back(algo1TimeOutput.count());
12      algo1Distance.push_back(algo1);
13      // Start the time
14      high_resolution_clock::time_point a2t1 = high_resolution_clock
            ↪ ::now();
15      int algo2 = MinDistance2Timing(col);
16      // End the time
17      high_resolution_clock::time_point a2t2 = high_resolution_clock
            ↪ ::now();
18      auto algo2TimeOutput = std::chrono::duration_cast<std::chrono
            ↪ ::nanoseconds>(a2t2-a2t1);
19
20      algo2ExecTime.push_back(algo2TimeOutput.count());
21      algo2Distance.push_back(algo2);
22  }
```

After completing the simulations for both the *MinDistance* and *MinDistance2* algorithms, and measuring the execution time for both the algorithms, the results were plotted for visual inspection. This section of the report will focus on the combination of the random, sorted and reversed sorted element tests. It can be seen in the data that there is a strong positive quadratic trend (refer to figure 5). In addition to this, the results for the three subtests are consistent across all of the tests. Similar to the basic operations test, this suggests that order has no effect on the execution time of the algorithm. In addition, it can also be seen that the rate of change for the MinDistance algorithm is greater than the rate of change for the MinDistance2 algorithm, similar to the basic operation tests. On the contrary, the execution test has a smaller range on the y-axis compared to the basic operation tests. As a result, the rate of change between both algorithms appears to be less then the rate of change between both algorithms for the basic operation tests. A more depth discussion can be found in the analysis section of this report.

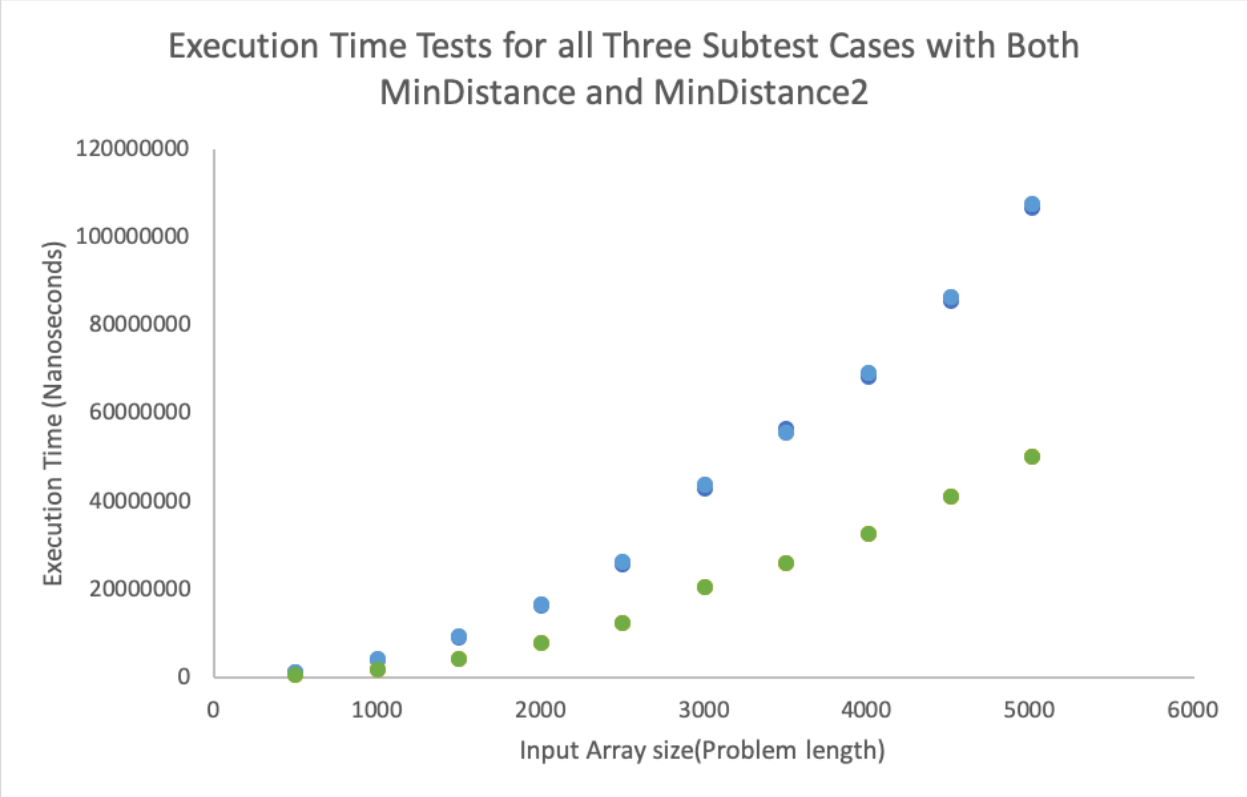

Figure 5: Execution time tests for all three subtest cases with both MinDistance and MinDistance2.

6 Analysis of experimental results

In this section, we are using the results from the random array length subtest. The reason that we are not using the results from the reverse sorted and sorted subtests is because the results from these tests are the same as the random test. In this case, there will be no overlays in the comparison between the algorithms, and the analysis of the results.

6.1 Basic operation analysis

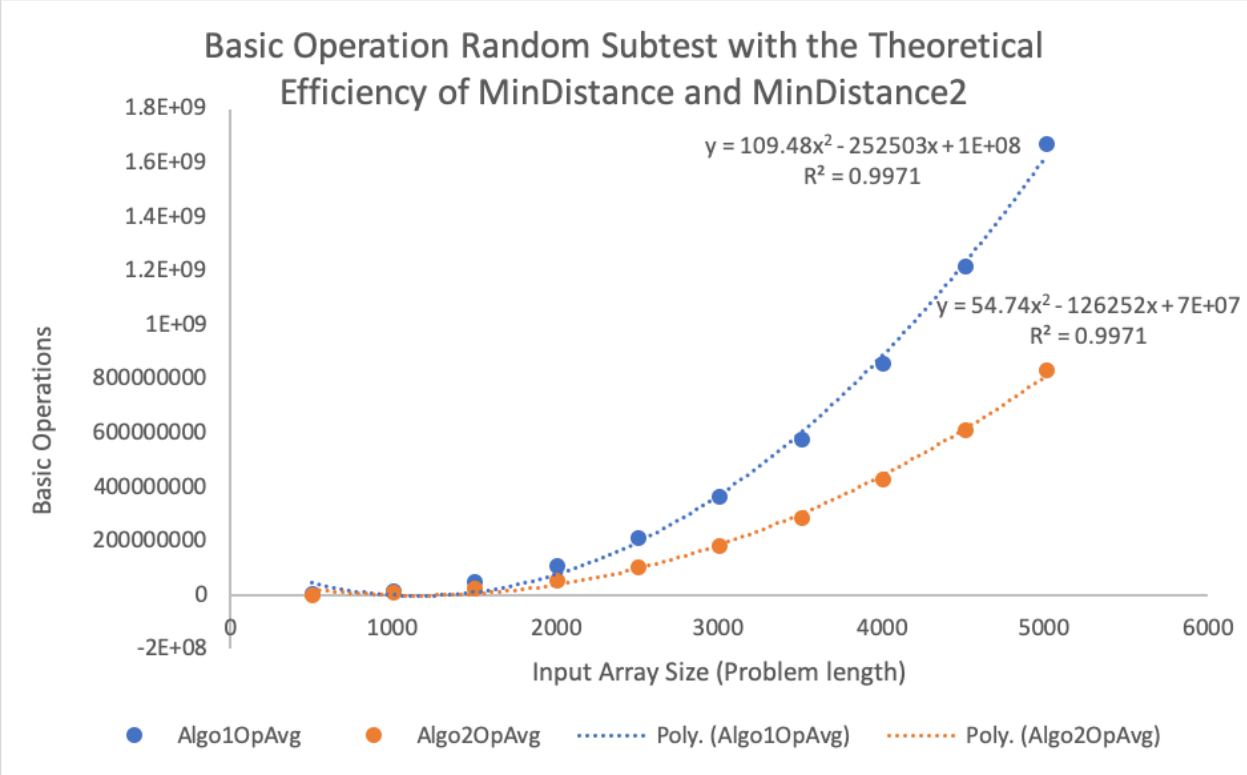

Figure 6: Comparison between the basic operations for MinDistance, MinDistance2 and the theoretical efficiency of the algorithms.

The graph above displays the random array size results for the basic operation count. The blue line is the *MinDistance* algorithm, whereas the orange line is the *MinDistance2* algorithm. The dotted blue and orange lines are the *MinDistance* and *MinDistance2* algorithm theoretical efficiencies respectively. Upon analysis of the data, it can be seen that there are no major discrepancies between the theoretical and the experimental efficiencies. This is supported by the R^2 value, which is the correlation of the theoretical efficiency function compared to the respective dataset. As it can be seen, the R^2 value for the *MinDistance* function is 0.9971, and the *MinDistance2* function is 0.9971. Both values represent a strong correlation between the theoretical efficiency and the dataset. For an overview of the dataset, refer to table 4 in the appendix.

Both the theoretical number of operations for the algorithm can be described as the algorithms below (from the theoretical efficiency section). As it can be seen, the *MinDistance2* algorithm is half the value of the *MinDistance* algorithm. Keeping this in mind, it can be seen that the coefficients of both the *MinDistance* and *MinDistance2* algorithms align

$$C_{Average} = n^2 - n \tag{8}$$

5: MinDistance efficiency.

$$C_{Average} = \frac{1}{2}(n^2 - n) \tag{9}$$

6: MinDistance2 efficiency.

with this proposition. Specifically, the theoretical functions for both the *MinDistance* and *MinDistance2* algorithms can be found below equation 7 and 8.

$$C_{MinDistanceOpTheory} = 109.48n^2 - 252503n + 1 \times 10^8 \tag{10}$$

7: MinDistance theoretical efficiency from operation test results.

$$C_{MinDistance2OpTheory} = 54.74n^2 - 126252n + 7 \times 10^7 \tag{11}$$

8: MinDistance2 theoretical efficiency from operation test results.

After observations of these theoretical functions, it can be seen that the coefficients in the *MinDistance2* algorithm are half that of the *MinDistance* algorithm. This aligns with the theoretical proposition whereby *MinDistance2* has the same complexity, however it is half that of *MinDistance*. This can be evaluated to true by doing the following, Taking equation 8, and multiplying it by 2,

$$\begin{aligned} C_{TheoreticalAvg} &= 54.74n^2 - 126252n + 7 \times 10^8 \\ C_{TheoreticalAvg} &= 2(54.74n^2 - 126252n + 7 \times 10^7) \\ C_{TheoreticalAvg} &= 109.48n^2 - 252503n + 1 \times 10^8 \\ \therefore C_{TheoreticalAvg} &\iff C_{MinDistanceOpTheory} \\ \therefore C_{MinDistance2OpTheory} &= \frac{1}{2} C_{MinDistanceOpTheory} \end{aligned}$$

6.2 Execution time analysis

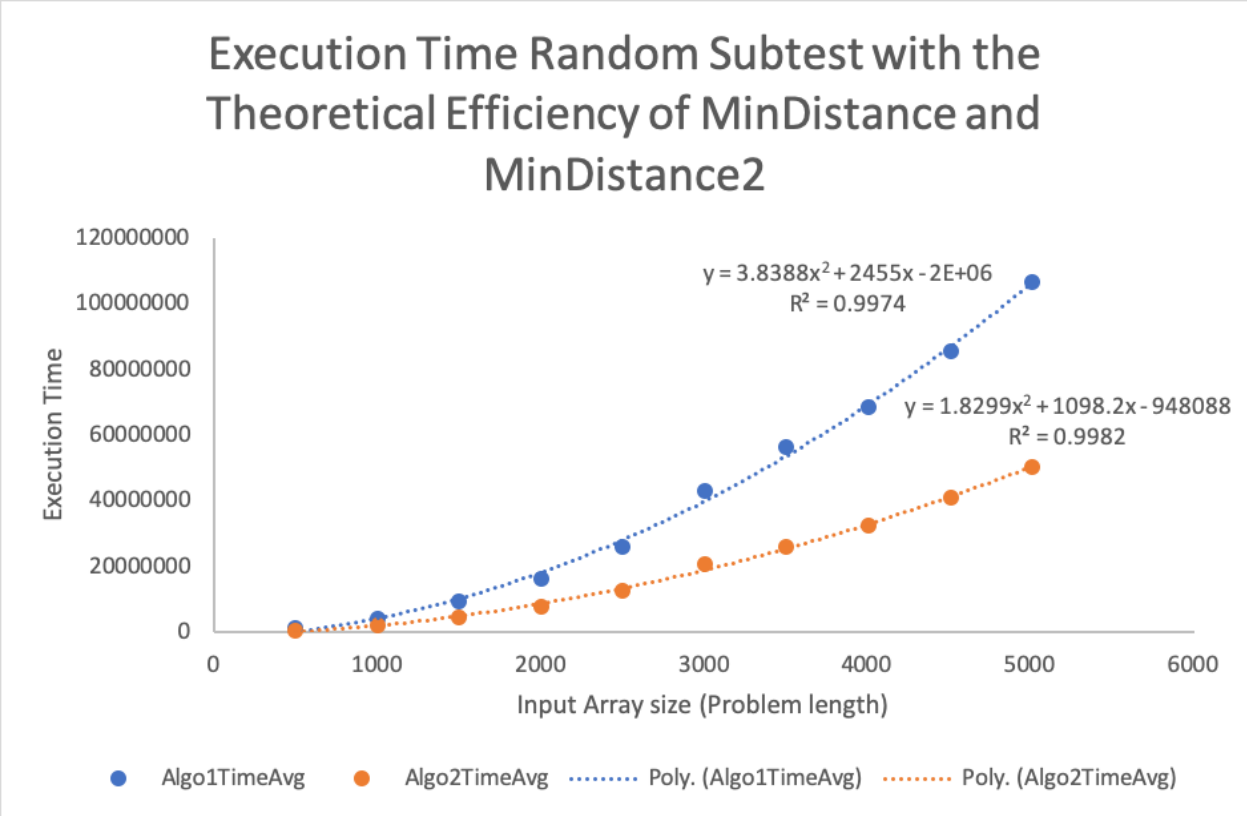

Figure 7: Comparison between execution time for MinDistance, MinDistance2 and the theoretical efficiency of the algorithms.

Figure 7 illustrates the execution time of the random subtest for both the *MinDistance* and *MinDistance2* algorithms. As the legend displays, the blue line is the *MinDistance* algorithm results, and the orange line is the *MinDistance2* algorithm results. In addition, the blue and orange dotted line are the *MinDistance* and *MinDistance2* theoretically efficiencies respectively. Upon inspection, it can be seen that there are no major discrepancies between the dataset and the theoretical efficiencies. This is confirmed by the r^2 values for both the *MinDistance* and *MinDistance2* algorithms, which are 0.9974 and 0.9982 respectively. Both these values illustrate a strong correlation between the trendline and the dataset. For an overview of the dataset, refer to table 5 in the appendix.

The theoretical efficiency of both algorithms is detailed in the theoretical analysis section of this report. Both the *MinDistance* and *MinDistance2* algorithms are of the order of n^2 with the *MinDistance2* algorithm being half the value of the *MinDistance* algorithm. Upon analysing the execution time results, it can be seen that both the *MinDistance* and

$$C_{MinDistanceTimeTheory} = 3.8388n^2 + 2455n - 2 \times 10^6 \tag{12}$$

9: MinDistance theoretical efficiency from execution time test results.

$$C_{MinDistance2TimeTheory} = 1.8299n^2 + 1098.2n - 948088 \tag{13}$$

10: MinDistance2 theoretical efficiency from execution time results.

MinDistance2 algorithms illustrate a n^2 relationship (refer to equation 9 and 10). In addition to this, the rate of the *MinDistance2* results and theoretical calculations is lower than that of *MinDistance*. This would suggest that the coefficient of *MinDistance2* is lower than *MinDistance*. This can be proved by dividing the *MinDistance* algorithm by the *MinDistance2* algorithm, to find out the ratio of the coefficients.

Taking both the MinDistance and MinDistance2 theoretical functions, and dividing them by eachother to obtain the ratio of the coefficients

$$\begin{aligned} Coefficient &= \frac{(7)}{(8)} \\ Coefficient &= \frac{3.8388n^2 + 2455n - 2 \times 10^6}{1.8299n^2 + 1098.2n - 948088} \\ Coefficient &= 2.0978n^2 + 2.23548n + 2.10951 \end{aligned}$$

It can be seen above that the ratio in which the *MinDistance2* algorithm is compared to the *MinDistance* algorithm is approximately 2. In other words, the *MinDistance2* algorithm is approximately half the *MinDistance* algorithm. This aligns with the theoretical postulate which states that the *MinDistance2* algorithm is half the *MinDistance* algorithm. However, this result is no precisely like the result we obtained from the basic operation tests.

One possible explanation for this is the variance in the data from the system on which the test is run. Depending on the clock cycles of the system, the time of the algorithm execution will change. In addition, processes running on the system might be taking resources away from the program itself, which will affect the execution time of the algorithm. Although the results for the execution time do no match the theoretical results exactly, they are still within a range which shows a relationship between the *MinDistance* and *MinDistance2* algorithms. In addition, the main trend in the results (positive quadratic trend) is still displayed in the data, which supports the theoretical postulate that both *MinDistance* and *MinDistance2* are of the efficiency class $\Theta(n^2)$.

7 Appendix

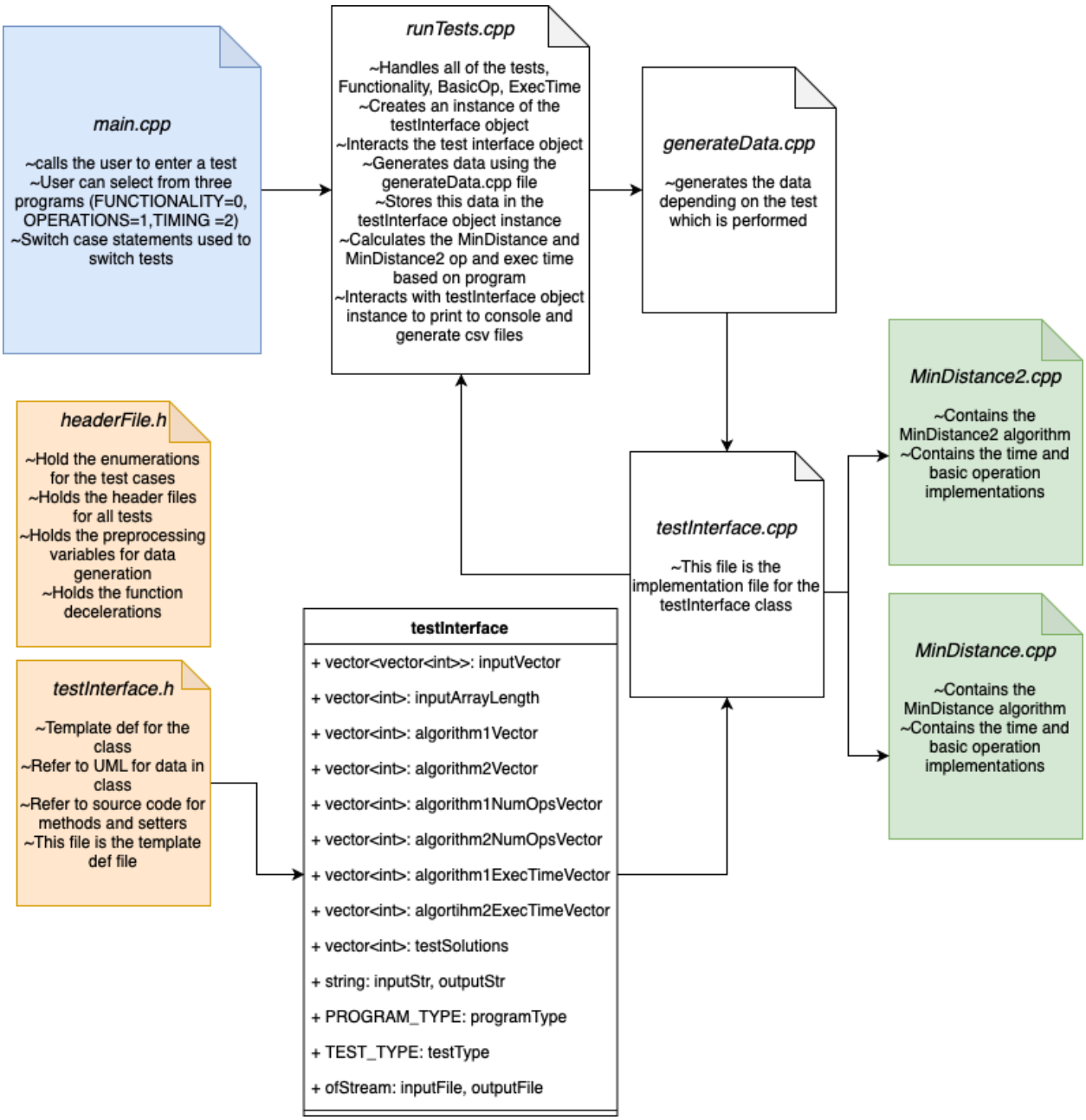

Figure 8: Top down overview of the program

Table 4: Basic operation average results for random, reverse sorted and sorted test cases.

ArrayLen	**Algo1OpAvg**	**Algo2OpAvg**
501	2379750	1189875
1002	14538519	7269259.5
1503	46516347	23258173.5
2004	108353274	54176637
2505	210089340	105044670
3006	361764585	180882293
3507	573419049	286709525
4008	855092772	427546386
4509	1216825794	608412897
5010	1668658155	834329078

Table 5: Execution time average results for random, reverse sorted and sorted test cases measured in nanoseconds.

ArrayLen	**Algo1TimeAvg**	**Algo2TimeAvg**
501	1248712.4	579376.9
1002	4065176.857	1975057.55
1503	9162999.238	4396133.55
2004	16433757.71	7858017.75
2505	25862447.81	12391371.85
3006	42999411.14	20519206.8
3507	56472541.67	25937286.5
4008	68470225.33	32585089.45
4509	85464869.81	41100156.55
5010	106604012.7	50273431.2