CAB301 Assignment 2

Empirical Comparison of Two Algorithms

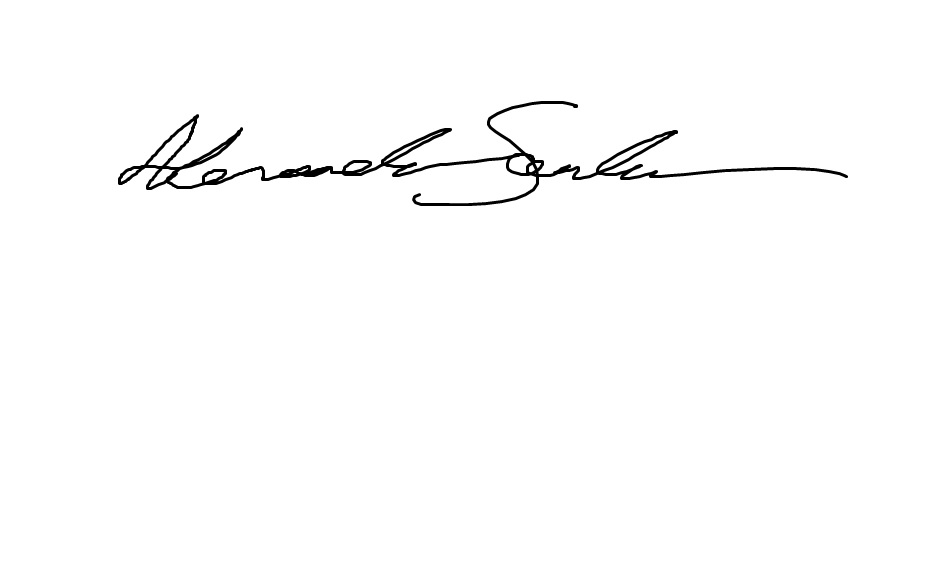
Student name: Alexander Santander

No: n9684026

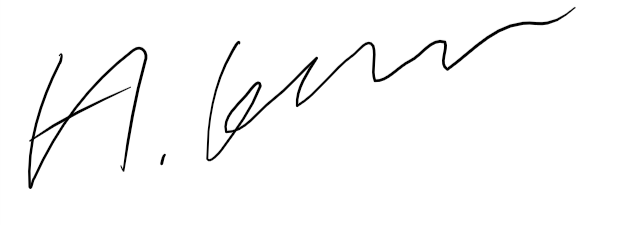
Student name: Hrushikesh Lakkola

No: n9654780

**STATEMENT OF COMPLETENESS**

**I, Alexander Santander hereby declare that this document and all that pertains to it was solely completed by my partner and I, of which I contributed roughly 50%**

**I, Hrushi Lakkola hereby declare that this document and all that pertains to it was solely completed by my partner and I, of which I contributed roughly 50%**



**Summary**

This report summarises the outputs of two algorithms intended to find the distance between the two closest elements in an array of values using various experiments. The second algorithm claims to be a more efficient modernization of the first and so this report serves to confirm or deny whether it is indeed an improvement. Factors such as the number of basic operations, execution time and scalability/efficiency were coded and executed in the C++ platform CodeBlocks. The experimental results were found to be consistent to an excellent degree with the theoretical calculations.

**Description of the Algorithms**

The first Algorithm’s (refer to Figure 1 below) purpose is to find and return the smallest difference between two values of the input, assuming the input is an array of numbers. This difference is found by using a nested for loop [fig 1, b-c] to compare all elements with each other and store the magnitude of the separation in the variable “dmin” [fig1, e]. Future iterations of the algorithm must then yield a smaller difference to “dmin” for it to be overridden [fig 1, d]. It is important for both loops to not compare the same number, otherwise “dmin” would consequently become zero, thus breaking the algorithm because no magnitude of difference can be less than zero [fig 1, d]. By the array’s conclusion all possibilities have been performed leaving the smallest difference within “dmin”.

Below is the pseudocode for the first algorithm which will be implemented in C++

Algorithm MinDistance(A[0..n-1])

//Input : Array A[0..n-1] of numbers

//Output: Minimum distance between two of its elements

a. dmin = ∞

b. for i = 0 to n-1 do

c. for j = 0 to n-1 do

d. if i ≠ j and |A[i]-A[j]| < dmin

e. dmin = |A[i]-A[j]|

f. return dmin

Figure 1 – First Algorithm

While the above algorithm is theoretically functional, there are some rather obvious flaws which may be detrimental to its efficiency. Algorithm 2 below claims to remedy that. The function of this algorithm is the exact same as the first and yet may be more efficient. The core evolution lies in the inner for loop, which now begins iterating at “i + 1” rather than zero [fig 2, c]. Ultimately this means that when the outer loop progresses the inner one spends dramatically less time re-hashing comparisons that were already performed. Consequently the outer loop must now only iterate one less than the array size to prevent the inner loop from moving out of bounds [fig 2, b]. Additionally, the if statement “i ≠ j” is no longer necessary which further increases efficiency [fig 2, e].

Below is the pseudocode for the second algorithm which will be implemented in C++

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

a. dmin = ∞

b. for i = 0 to n-2 do

c. for j = i+1 to n-1 do

d. temp = |A[i]-A[j]|

e. if temp < dmin

f. dmin = temp

g. return dmin

Figure 2 - Second Algorithm

**Identifying the Algorithm’s Basic Operation**

The definition of basic is to “form an essential foundation”. It only makes sense to consider an operation which greatly influences the return value of the algorithm. With this in mind, the inner loop’s if statement [fig 1, d] and [fig 2, e] has been chosen. Its purpose is to determine whether the magnitude of difference between two values in the current loop iterations are less than the most recent difference calculated. While the statement does lose a condition in algorithm 2, its function as well as its overall position in the algorithm remains the same, making this an excellent common basis. Choosing an if statement over something within it will also generate a more desirable curve, due to the nature of randomly generated numbers. Occasionally the if statement’s requirements will be satisfied an abnormal amount which will create anomalies.

**Problem Size**

At first glance it is quite obvious that the algorithms are quadratic in nature. This means an input too great would result in possibly days of computing and too little would create a poor illustration of the curve. Therefore, some minor testing will be done to ensure the input array size falls between these guidelines. The first assignment also had a nested for loop which means that the array size used (25000) will make an excellent starting point.

**Theoretical Average Case Efficiency with respect to the Basic Operation**

Finding the average case efficiency will present and upper bound on the average iterations we would see when running the program many times on many different inputs [2, para 2]. This section will calculate exactly that for both equations and determine whether one will theoretically be more efficient than the other.

The number of iterations for the first algorithm [fig 1] is largely determined by its nested for loop configuration. However rather than exit upon completing the task, this task is characterized by systematically checking every element in the array to ensure the returned variable contains the lowest magnitude of difference. Therefore, it is only possible for both the outer and inner loops separate efficiencies to be “n”. Thus (n \* n) = n2 as demonstrated below.

As stated before, the second algorithm [fig 2] underwent some rather large changes intended to increase efficiency. The most notable of which now makes the inner loop only iterate through elements after the outer loop. The tables below demonstrate the disparity in efficiency between the two algorithms.

**Algorithm 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |

**Algorithm 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |

Legend: \_\_\_\_ = current outer loop element, 1234 = inner loop elements

Each row of the table above demonstrates a single iteration of the outer loop and all iterations of the inner loop. Notice how in the second algorithm as the outer loop progresses the inner loop linearly regresses its number of iterations. If you were to ignore the obsolete 5th column then only half the elements are highlighted as opposed to all in algorithm 1. Essentially the number of iterations of the inner loop is halved, thereby doubling its efficiency. A secondary effect of these changes is that the outer loop only progresses to “n-2” thereby cutting off a full iteration to prevent the inner loop from moving out of bounds. While this will serve to further increase efficiency, on a grand scale the difference will be barely noticeable and so will not be factored in to our equation. Therefore, the efficiency is (n)(n/2) as demonstrated below.

**Theoretical Order of Growth**

The order of growth regards how the time for computation increases when you increase the input size [1, para 1]. To measure this, the number of actions that are performed for a given input must be determined. The purpose of this section is to do exactly that so the two algorithms may be compared.

The figure below illustrates the worst-case cost and number of iterations for each line for algorithm 1.

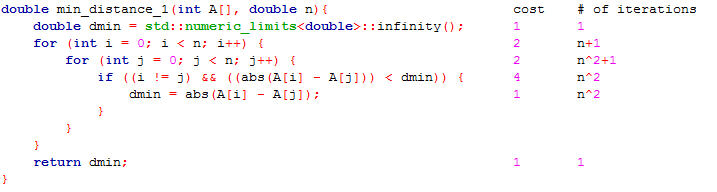


Figure 3 - OOG algorithm 1

Figure 3 above demonstrates line by line the cost and number of times each part is performed. Taking this data and simplifying it gives 7n2 + 8n + 5. However, as the input becomes larger only the most processor intensive part of this equation will be relevant. Therefore, we drop all constants as well as the n, leaving us with n2. Meaning the predicted growth curve for this algorithm is ɵ(n2). While this is correct, it is important to note the n2 coefficient “7”. This will be explained in the next paragraph regarding algorithm 2.

The figure below illustrates the worst-case cost and number of iterations for each line for algorithm 2.

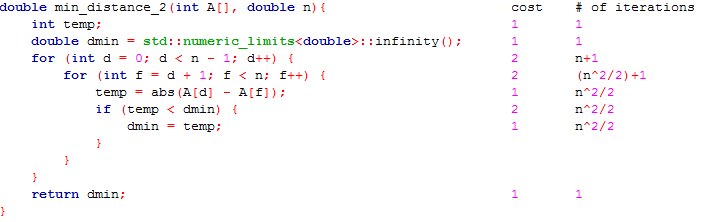


Figure 4- OOG algorithm 2

Upon extrapolating and simplifying the data in Figure 4 similarly to Figure 3, the equation 3n2 + 2n + 7 is given. This equation is strikingly similar to the one generated from the first algorithm and so is treated accordingly, leaving us with an order of growth which is also ɵ(n2). This means that the output curves will be identical in shape, however upon further analysis it is apparent their magnitudes are quite different. As calculated in the “Theoretical Average Case Efficiency” section above, the second algorithms efficiency is the same as the first divided by two. Because order of growth and efficiency are so closely related it can be anticipated a similar disparity will occur. By dividing the second algorithms n2 coefficient by the first we are given 3 / 7 = .42. In conclusion, this means around a 42% decrease of growth can be expected in algorithm 2.

**Methodology, Tools and Techniques**

1. The algorithm as well as experiments were coded in C++. While the object orientation advantage C++ has over C was not utilized, it was still chosen because of its more up to date and arguably easier to understand syntax. C++ is widely considered one of the most comprehensive programming languages, second only to Java in number of users.
2. The IDE used for code implementation was CodeBlocks. A free C, C++ and Fortran IDE built around a plugin framework. This IDE was used specifically for being one of the lesser known/used IDE’s because of its simplistic interface thereby creating an undaunting environment for amateur coders
3. The experiments were performed on a desktop computer with custom specifications including: Up to date Windows 10 x64 OS, 16GB of RAM and an intel i5-3570k CPU. The clock() function within the library <time.h> was used for measuring the algorithms execution time and rand() within <math.h> was used to create a pseudo random seed to ensure the number would be as random as possible. All unnecessary programs were closed so maximum resources could be allocated to provide a reasonable execution time.
4. All Graphs relating to this report were produced in Microsoft Excel. The data points were manually typed in for experimental results to ensure they were correct. All code segments were imported through Miscosoft Words “Insert->Object” utility.

**Verify Algorithm’s Functionality**

To test the functional correctness of the algorithm’s implementation (see Appendix A) a series of tests were created (see Appendix B). They simply involved hard coding in the values of the input array input in various formats so they may be calculated algebraically beforehand to determine whether the output is correct.

**The following tests were performed for algorithm 1**

Test1 – very basic functionality:

Input array: [1,3,6,10,15,21,28,36,45,55]

Expected output = 2

Actual output = 2

Test2 – An array with inversed elements:

Input array: [34,95,77,68,4,42,1,100,74,85]

Expected output = 2

Actual output = 2

Test2 – An array with randomized elements:

Input array: [34,95,77,68,4,42,1,100,74,85,19,37,94,31,9]

Expected output = 1

Actual output = 1

**The following tests were performed for algorithm 2**

Test1 – very basic functionality:

Input array: [1,3,6,10,15,21,28,36,45,55]

Expected output = 2

Actual output = 2

Test2 – An array with inversed elements:

Input array: [34,95,77,68,4,42,1,100,74,85]

Expected output = 2

Actual output = 2

Test2 – An array with randomized elements:

Input array: [34,95,77,68,4,42,1,100,74,85,19,37,94,31,9]

Expected output = 1

Actual output = 1

As you can see every result output by the algorithm was correct. Ultimately the tests to be performed later in this report will be similar to this but on a much grander scale, and so if it can return the right values now there is no reason why it will not over more iterations.

**Experimental Average Efficiency for Each Algorithm (with respect to basic operation)**

The code used to determine the number of basic operations each algorithm is detailed in Appendix C. It works by creating an array and then filling it with random values between 1 and 25000 to provide as input, but only a portion of the total array is used depending on the current iteration. The algorithms record and return the number of basic operations that are performed, this is then done 50 times and the number of basic operations is averaged out. These steps are continuously repeated as the upper bound of the array size increments in 250’s and then is plotted in Figures 5 and 6.

Both graphs appear to almost perfectly follow a quadratic pathway with zero anomalies. This can be attributed to both the simplicity of the algorithm and having more time to perfect the testing process (building upon assignment 1).

If we were to compare both algorithms with their theoretical results, both look exactly as predicted. Due to the sheer scale of this experiment the smaller variables within the equations (calculated in “Theoretical Average Case Efficiency”) are entirely irrelevant and so only the n2 can be depicted.

The calculated efficiency of algorithm 2 was n2/2. This means that the growth of algorithm 2 should be exactly half of algorithm 1. If you refer to Figures 5 and 6 you can see this is clearly the case. Algorithm 1’s maximum number of basic operations is 7 x 108 and algorithm 2’s maximum is 3.5 x 108. 7 / 3.5 = 2, therefore the theoretical calculations were correct.

**Experimental Order of Growth for Each Algorithm**

The code used to determine the order of growth for each algorithm is detailed in Appendix D. In order to measure this the code first fills an array with random values from 1-25000 and then finds the time difference between starting the algorithm and finishing. This is then performed 50 times so that a more accurate execution time can be averaged. The array size is then gradually incremented in 250’s and plotted in Figure 3.

If you were to compare the average efficiency (Figures 5 and 6) and the order of growth (Figures 7 and 8) for each algorithm it is quite apparent how similar they are. This is because coincidentally the equations for both revolve around the term n2 as seen in the theoretical section above.

If you were to refer to figures 7 and 8 you can see that both graphs are quadratic and so adhere to the order of growth ɵ(n2). However it was also calculated that algorithm 2 would have a 42% decrease in growth due to the different coefficients. To determine whether this is correct the maximum execution time of algorithm 2 must be divided by that of algorithm 1 giving: 100 (1.6 / 3.5) = 45%. A 3% margin of error can be considered more than reasonable and so this confirms that the theoretical calculations for order of growth were correct.

# References

J.Zeil, S. (2006). *Analysis of Algorithms: Average Case Analysis*. Retrieved from Old Dominion University: https://secweb.cs.odu.edu/~zeil/cs361/web/website/Lectures/averagecase/pages/index.html

Mishra, P. (2016, June 8). *What is the meaning of "order of growth" in algorithm analysis and how can we find the order of growth of given algorithm?* Retrieved from Quora: https://www.quora.com/What-is-the-meaning-of-order-of-growth-in-algorithm-analysis-and-how-can-we-find-the-order-of-growth-of-given-algorithm

**Both Algorithm’s Average Efficiency Experimental Results– Figures 5 & 6**

The graphs below illustrate the number of basic operations performed per iteration relative to the input size. The array size was incremented in 250’s from 0 – 25000 and each data point is an average of 50 tests. The curves are quadratic in nature and so confirm the theoretical efficiency equation of n2. For more depth refer to “Experimental Average Efficiency” above. The code used to create these results can be found in Appendix C.

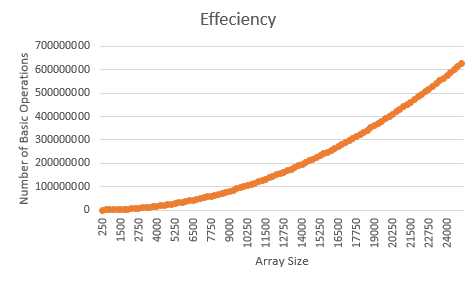
**Algorithm 1**

Figure 5 – Algorithm 1 Efficiency

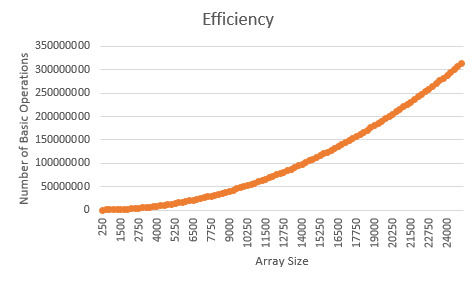
**Algorithm 2**

Figure 6 - Algorithm 2 Efficiency

**Both Algorithm’s Order of Growth Experimental Results – Figures 7 & 8**

The graphs below illustrate the orders of growth found for each respective algorithm through experimental means in C++. The array size was incremented in 250’s from 0 – 25000 and each data point is an average of 50 tests. Both are obviously quadratic and so confirm the theoretical order of growth equation ɵ(n2). However, it was found that algorithm 2 had a coefficient lower than that of algorithm 1. As you can see this has been reflected by the testing process. For more details refer to the section above “Experimental Order of Growth”. The code used to create these results can be found in Appendix D.

**Algorithm 1**

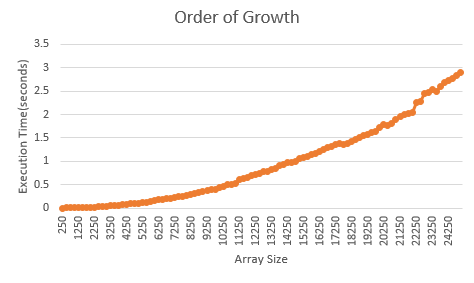


Figure 7 – Algorithm 1 Order of Growth

**Algorithm 2**

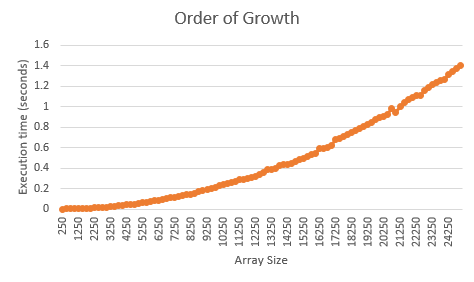


Figure 8 – Algorithm 2 Order of Growth

**Appendix A – Algorithms in C++ format**

This appendix shows the algorithms once they have been implemented in C++. They have been thoroughly commented for clarity.

**Algorithm 1**

double min\_distance\_1(int A[], double n) {

//dmin is set to infinity purely as a starting point to it is impossible for

//anything to be greater than it

double dmin = std::numeric\_limits<double>::infinity();

//the first an outer loop which iterates through the entire array

for (int i = 0; i < n; i++) {

//the inner loop which also iterates throug the entire array

for (int j = 0; j < n; j++) {

//an if statement which ensures that the two elements compared //are not the same to prevent dmin from = zero. Then returns dmin //if the magnitude of the difference of the two current elements //is less than what dmin currently holds

if ((i != j) && ((abs(A[i] - A[j])) < dmin)) {

//the old value of dmin is overrided with a new smaller //magnitude of difference

dmin = abs(A[i] - A[j]);

}

}

}

//returns dmin

return dmin;

}

**Algorithm 2**

double min\_distance\_2(int A[], double n) {

//setting a temporary variable

int temp;

//dmin is set to infinity purely as a starting point to it is impossible for

//anything to be greater than it

double dmin = std::numeric\_limits<double>::infinity();

//the first an outer loop which iterates through the entire array

for (int d = 0; d < n - 1; d++) {

//the second loop which now only iterates trough the elements in front ///of the outer loop

for (int f = d + 1; f < n; f++) {

//setting the temporary variable to the magnitude of difference //of the two current elements in the loops

temp = abs(A[d] - A[f]);

//returns true if the current magnitude of difference is less //than the one currently being held by dmin

if (temp < dmin) {

//the old value of dmin is overrided with a new smaller //magnitude of difference

dmin = temp;

}

}

}

//returns dmin

return dmin;

}

**Appendix B – Algorithm Functionality Tests**

The below tests were created to ensure that the algorithms were successfully implemented in C++ so that the results generated by them within this assignment may be provided credibility. If you wish to see a more comprehensive understanding of how the two functions operate please refer to Appendix A.

**Algorithm 1**

**Test 1**

//length of array to be analysed

const int N = 10;

//function declaration

double min\_distance\_1(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 1,3,6,10,15,21,28,36,45,55 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_1(A, N) << endl;

return 0;

}

**Test 2**

//length of array to be analysed

const int N = 10;

//function declaration

double min\_distance\_1(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 34,95,77,68,4,42,1,100,74,85 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_1(A, N) << endl;

return 0;

}

**Test 3**

//length of array to be analysed

const int N = 15;

//function declaration

double min\_distance\_1(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 34,95,77,68,4,42,1,100,74,85,19,37,94,31,9 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_1(A, N) << endl;

return 0;

}

**Algorithm 2**

**Test 1**

//length of array to be analysed

const int N = 10;

//function declaration

double min\_distance\_2(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 1,3,6,10,15,21,28,36,45,55 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_2(A, N) << endl;

return 0;

}

**Test 2**

//length of array to be analysed

const int N = 10;

//function declaration

double min\_distance\_2(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 34,95,77,68,4,42,1,100,74,85 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_2(A, N) << endl;

return 0;

}

**Test 3**

//length of array to be analysed

const int N = 15;

//function declaration

double min\_distance\_2(int A[], double n);

int main() {

//the values being hard coded into the array being analysed

int A[N] = { 34,95,77,68,4,42,1,100,74,85,19,37,94,31,9 };

//printing out the answer

cout << "Greatest Magnitude of Difference = " << min\_distance\_2(A, N) << endl;

return 0;

}

**Appendix C – Code Used to Measure both Algorithms Efficiency with Respect to the Number of Basic Operations**

The below code was used to determine the average number of basic operations the algorithms performed. They have been heavily commented to provide insight into their logic.

**Algorithm 1**

#include <iostream>

#include <iterator>

#include <algorithm>

#include <fstream>

#include <time.h>

#include <limits>

using namespace std;

const int noOfTests = 50;

//sets the array total size, however each test will only use a portion of it and then //the subsequent test will overwrite the previous and use a portion more until max is //reached

const int N = 25000;

long long numBasicno = 0;

long long randomNumArray[N];

long long min\_distance\_1(long long A[], double n);

long long max\_iteration = 0;

long long resultsAverage[100];

int main() {

//outer loop determining how many tests are performed

for (int k = 0; k < 100; k++) {

// increment the variable which determines the current input size to be //tested

max\_iteration += 250;

//set the number of basic operations back to zero for each different array size

numBasicno = 0;

//inner loop which performs 50 tests which will be averaged for one data //point

for (int j = 0; j < noOfTests; j++) {

//fills the array to be analysed with random values between 1-N

for (int i = 0; i < max\_iteration; i++) {

randomNumArray[i] = rand() % (N - 1) + 1;

}

//finding the number of basic operations performed and adding it //to the total

numBasicno += min\_distance\_1(randomNumArray, max\_iteration);

}

//finding the average number of basic operations performed and storing //them in the array

resultsAverage[k] = numBasicno / noOfTests;

//printing to screen

cout << "Average number of basic operations over 50 tests for Array size" << max\_iteration << " = " << resultsAverage[k] << endl;

//print output to file so results may be graphed

ofstream myfile;

myfile.open("efficiencyOutput.txt", ios::app);

myfile << max\_iteration << "\t" << resultsAverage[k] << endl;

myfile.close();

}

return 0;

}

long long min\_distance\_1(long long A[], double n) {

//setting the number of basic operations back to zero every iteration

long long numBasic = 0;

long long dmin = std::numeric\_limits<double>::infinity();

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

//adding to the number of basic operations each itearation

numBasic += 1;

if ((i != j) && ((abs(A[i] - A[j])) < dmin)) {

dmin = abs(A[i] - A[j]);

}

}

}

return numBasic;

}

**Algorithm 2**

#include <iostream>

#include <iterator>

#include <algorithm>

#include <fstream>

#include <time.h>

#include <limits>

using namespace std;

const int noOfTests = 50;

//sets the array total size, however each test will only use a portion of it and then //the subsequent test will overwrite the previous and use a portion more until max is //reached

const int N = 25000;

long long numBasicno = 0;

long long randomNumArray[N];

long long min\_distance\_2(long long A[], double n);

long long max\_iteration = 0;

long long resultsAverage[100];

int main() {

//outer loop determining how many tests are performed

for (int k = 0; k < 100; k++) {

// increment the variable which determines the current input size to be //tested

max\_iteration += 250;

//set the number of basic operations back to zero for each different array size

numBasicno = 0;

//inner loop which performs 50 tests which will be averaged for one data //point

for (int j = 0; j < noOfTests; j++) {

//fills the array to be analysed with random values between 1-N

for (int i = 0; i < max\_iteration; i++) {

randomNumArray[i] = rand() % (N - 1) + 1;

}

//finding the number of basic operations performed and adding it //to the total

numBasicno += min\_distance\_2(randomNumArray, max\_iteration);

}

//finding the average number of basic operations performed and storing //them in an array

resultsAverage[k] = numBasicno / noOfTests;

//printing to screen

cout << "Average number of basic operations over 50 tests for Array size "

<< max\_iteration << " = " << resultsAverage[k] << endl;

//print output to file so results may be graphed

ofstream myfile;

myfile.open("efficiencyOutput2.txt", ios::app);

myfile << max\_iteration << "\t" << resultsAverage[k] << endl;

myfile.close();

}

return 0;

}

long long min\_distance\_2(long long A[], double n) {

//setting the number of basic operations back to zero every iteration

long long numBasic = 0;

int temp;

long long dmin = std::numeric\_limits<double>::infinity();

for (int d = 0; d < n - 1; d++) {

for (int f = d + 1; f < n; f++) {

temp = abs(A[d] - A[f]);

//adding to the number of basic operations each itearation

numBasic += 1;

if (temp < dmin) {

dmin = temp;

}

}

}

return numBasic;

}

**Appendix D – Code Used to Measure the Algorithm’s Order of Growth**

The below code was used to determine the order of growth of the respective algorithms. they have been heavily commented to provide insight into their logic. The algorithms have themselves not been included because they were not altered for this test.

**Algorithm 1**

#include <ctime>

#include <cstdlib>

#include <iostream>

#include <iomanip>

#include <math.h>

#include <fstream>

#include <iterator>

#include <algorithm>

#include <time.h>

#include <limits>

using namespace std;

const int noOfTests = 50;

//sets the array total size, however each test will only use a portion of it and then //the subsequent test will overwrite the previous and use a portion more until max is //reached

const int N = 25000;

int randomNumArray[N];

double min\_distance\_1(int A[], double n);

double max\_iteration = 0;

double execTimes = 0;

double execTimesAverage[100];

int main() {

//outer loop determining how many tests are performed

for (int k = 0; k < 100; k++) {

// increment the variable which determines the current input size to be //tested

max\_iteration += 250;

//set the number of basic operations back to zero for each different //array size

execTimes = 0;

//inner loop which performs 50 tests which will be averaged for one data //point

for (int j = 0; j < noOfTests; j++) {

//fills the array to be analysed with random values between 1-N

for (int i = 0; i < max\_iteration; i++) {

randomNumArray[i] = rand() % (N - 1) + 1;

}

//store the start time of the algorithm

clock\_t start\_clock = clock();

//running the algorithm with the random array and its size as //inputs

min\_distance\_1(randomNumArray, max\_iteration);

//store the end time of the algorithm

clock\_t end\_clock = clock();

//find the difference to determine time taken for algorithm and

//additively store values

execTimes += (end\_clock - start\_clock) / double(CLOCKS\_PER\_SEC);

}

//find the average of all execution times for this data point

execTimesAverage[k] = execTimes / noOfTests;

//print the results to screen

cout << "Average execution time over 50 tests for Array size "

<< max\_iteration << " = " << fixed << setprecision(5) << execTimesAverage[k]

<< "s" << endl;

//print output to file so results may be graphed

ofstream myfile;

myfile.open("orderOfGrowthOutput.txt", ios::app);

myfile << max\_iteration << "\t" << execTimesAverage[k] << endl;

myfile.close();

}

return 0;

}

**Algorithm 2**

#include <ctime>

#include <cstdlib>

#include <iostream>

#include <iomanip>

#include <math.h>

#include <fstream>

#include <iterator>

#include <algorithm>

#include <time.h>

#include <limits>

using namespace std;

const int noOfTests = 50;

//sets the array total size, however each test will only use a portion of it and then //the subsequent test will overwrite the previous and use a portion more until max is //reached

const int N = 25000;

int randomNumArray[N];

double min\_distance\_2(int A[], double n);

double max\_iteration = 0;

double execTimes = 0;

double execTimesAverage[100];

int main() {

//outer loop determining how many tests are performed

for (int k = 0; k < 100; k++) {

// increment the variable which determines the current input size to be tested

max\_iteration += 250;

//set the number of basic operations back to zero for each different //array size

execTimes = 0;

//inner loop which performs 50 tests which will be averaged for one data //point

for (int j = 0; j < noOfTests; j++) {

//fills the array to be analysed with random values between 1-N

for (int i = 0; i < max\_iteration; i++) {

randomNumArray[i] = rand() % (N - 1) + 1;

}

//store the start time of the algorithm

clock\_t start\_clock = clock();

//running the algorithm with the random array and its size as //inputs

min\_distance\_2(randomNumArray, max\_iteration);

//store the end time of the algorithm

clock\_t end\_clock = clock();

//find the difference to determine time taken for algorithm and

//additively store values

execTimes += (end\_clock - start\_clock) / double(CLOCKS\_PER\_SEC);

}

//find the average of all execution times for this data point

execTimesAverage[k] = execTimes / noOfTests;

//print the results to screen

cout << "Average execution time over 50 tests for Array size "

<< max\_iteration << " = " << fixed << setprecision(5) << execTimesAverage[k]

<< "s" << endl;

//print output to file so results may be graphed

ofstream myfile;

myfile.open("orderOfGrowthOutput\_2.txt", ios::app);

myfile << max\_iteration << "\t" << execTimesAverage[k] << endl;

myfile.close();

}

return 0;

}