Report



Empirical Analysis of Two Algorithms

Assignment 2

Kaushal Kishorbhai Limbasiya N1003029

Vaibhav Arvind Vachhani N9796134

CAB301 Algorithms and Complexity

Queensland University of Technology

Table of Contents

[1. Description of the Algorithm 1](#_Toc5959775)

[2. Algorithm Basic Operations Error! Bookmark not defined.](#_Toc5959776)

[3. Performing Experiments and Computing Environments 4](#_Toc5959777)

[**3.1. Proof of Correctness** Error! Bookmark not defined.](#_Toc5959778)

[**3.2. Running Time Analysis** Error! Bookmark not defined.](#_Toc5959779)

[Worst case scenario Error! Bookmark not defined.](#_Toc5959780)

[Best Case Scenario 8](#_Toc5959781)

[References 8](#_Toc5959782)

## Description of the Algorithms

**minDistance:**

This algorithm consists of 2 for loops iterating over the array passed as a parameter to it. The inner loop consists of the basic operation, . It compares each element of the array to every other element of the array as the absolute difference between the same element would be 0. It assigns the absolute difference to *dmin* if it is smaller than *dmin’s* assigned value.

Sample tests:

Table 1: Sample Test Results for minDistance

|  |  |
| --- | --- |
| Input Array | Output Result |
| {1,56,78,23,1,687,92,10} | 0 |
| {65,25,98,52,537,8561,1649,10} | 13 |
| {21,36,163,473,1234,874,13235,714,787, -90, -14}; | 15 |
| {-123, -56, -2356, -122, -098647, -7635, 803746, 91832} | 1 |

**minDistance2:**

This algorithm does not compare the same elements of the array, as the first loop starts with the *first***(i)** element and stops when it reaches the second last element (n-2). On the contrary, the inner loop starts with the *second* **(i+1)** element and stops when it reaches the last element. So, each element will only be compared to all the elements that comes after it in the array. This reduces the numbers of operations performed, which is why minDistance2 is *faster* than minDistance.

Sample tests:

Table 2: Sample Test Results for minDistance2

|  |  |
| --- | --- |
| Input Array | Output Result |
| {18, 92, 74, 15, 95, 1948, 100, 76347} | 3 |
| {1, 99, 091873, 65246, 87634, 9, 76247, 9873} | 8 |
| {-98,63, 74, -8673, 754, 152, 879, 0} | 11 |
| { -9, 9713, 01836, 859, -9484, -516, 200, 100458} | 209 |

**Assumptions:**

1. We have assumed that the length of the array will be greater than equal to 2. We have not implemented the logic for that case. It will return the value assigned to *dmin* if the array contains 1 element.
2. The lowest difference between any 2 elements is less than 2147483647 as this the maximum value allowed to store in an integer variable in C# and we cannot assign *infinity*.

## Theoretical Analysis of the Algorithms

**minDistance:**

Basic Operation: ***if (i!=j) and |A[i]-A[j]| < dmin***

Problem size:

The identified basic operation is the only comparison in the algorithm, so it is the main logic of the algorithm. This means it is responsible for most of the processing load because it consists of two comparison (comparing *I* with *J* and comparing *i*thelement with *j*th element. As it is inside the inner most loop, it will be performed times where n is size of the array.

***Efficiency:***

1. Solving the right summation formulae using formula 1 from appendix:

Where u = ( - 1)

Gives us:

1. So, we now take out as a constant (using formulae 3 from appendix) which gives us:

Using the formula 1 again where u = ( - 1) we get:

So, big-theta of minDistance is ().

The outer loop and the inner loop will run for n times (0 to - 1) where n is size of the array hence the efficiency class of this algorithm is . (derivation above)

**minDistance2:**

Basic Operation: ***if (temp < dmin)***

Problem size:

This comparison is performed the greatest number of times. If it evaluates to true – it assigns the value of *temp* to *dmin*.

***Efficiency:***

1. Solving the right summation formulae with formula 1 from appendix:

Where u = ( - 1) and l = (i + 1)

Give us:

1. Re-writing that, substituting u = - 1 gives us:
2. Solving using formula 2 from appendix gives us ( - 1) ( – 1 + 1) / 2 which simplifies to ( - )

So, big-theta of minDistance2 is ().

## Implementations of the Algorithms

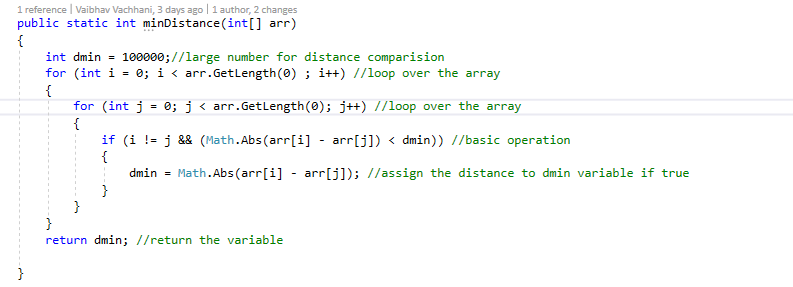


Figure 1: Implementation of minDistance

The return type is int – and integer representing the minimum distance between two elements. We have defined a variable named “*dmin*” which is first declared before the for loops start. It is set to a reasonably high number – assuming minimum distance will be less than this assigned value (as we cannot assign infinity). Both for-loops, iterates over the whole array (*arr.GetLength(0)* returns the number of elements in the array. The if statement checks if the absolute difference between the two elements is smaller than *dmin*’s value and if I do not equal J – if yes, it assigns the difference to the variable *dmin*. It avoids comparing the same element with itself. Finally, after the outer loop finishes iterating over the array – the final value of *dmin* will be returned.

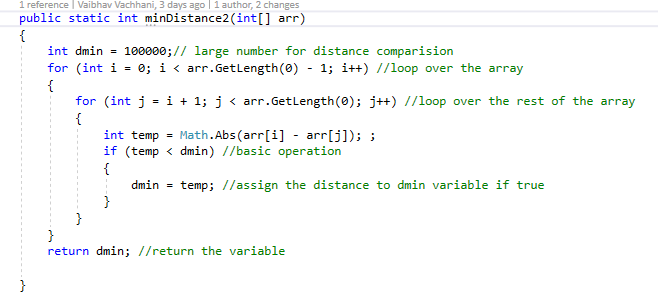


Figure 2: Implementation of minDistance2

minDistance2 consists of 2 for loops – same as minDistance, but the inner for loop does not iterate over the whole array. The inner for loop avoids making the same comparisons as j starts from i+1, this reduces the number of operations is performed, and therefore minDistance2 is faster than minDistance. We have defined a *dmin* variable and assigned a high value to it (100000). The outer for loop iterates over the whole array but contrary to minDistance, the inner loop starts from i+1 rather than 0. In the inner for loop, we define a variable *temp* and assign the absolute difference between two elements to it. Then, we the if condition checks if *temp* is smaller than *dmin* – if yes, we assign the value of *temp* to *dmin*. Finally, after the outer loop finishes iterating over the array – the final value of *dmin* will be returned.

* 1. Functionality Testing – minDistance & minDistance2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Test Instance** | **Expected Output** | **Actual Output** | **Test Result** |
| Array with same elements | {5, 5, 5, 5, 5, 5, 5, 5, 5} | 0 | 0 | Positive |
| Array with same negative elements | {-9, -9, -9, -9, -9, -9, -9} | 0 | 0 | Positive |
| Array with two elements | {9,1} | 8 | 8 | Positive |
| Array with negative elements | {-4, -9, -7, -1, -90, -100, -56, -24} | 2 | 2 | Positive |
| Array with one element | Not implemented (program crashes) | - | - | - |
| Array with random elements | {2, -20, 56, 78, 13, 45, 80, 100} | 2 | 2 | Positive |
| Array with 0s | {0, 0, 0, 0, 0, 0} | 0 | 0 | Positive |

* 1. Formal Proof of Correctness

The following table shows value of dmin through each iteration of both algorithms. It will return the lowest – which is 20.

Initial *dmin* value = ∞, *temp* = |A[i]-A[j]|

*if (temp < dmin)* is true *temp* is assigned to *dmin*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **temp** | **i** | **A[i]** | **j** | **A[j]** | **Temp**  **< dmin** |
| 22 | 0 | 2 | 1 | -20 | true |
| 43 | 0 | 2 | 2 | 45 | false |
| 78 | 0 | 2 | 3 | 80 | false |
| 98 | 0 | 2 | 4 | 100 | false |
| 65 | 1 | -20 | 2 | 45 | false |
| 100 | 1 | -20 | 3 | 80 | false |
| 120 | 1 | -20 | 4 | 100 | false |
| 35 | 2 | 45 | 3 | 80 | false |
| 55 | 2 | 45 | 4 | 100 | false |
| 20 | 3 | 80 | 4 | 100 | true |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **|A[i]-A[j]|** | **i** | **A[i]** | **j** | **A[j]** | **|A[i]-A[j]|< dmin** |
| - | 0 | 2 | 0 | 2 | false |
| 22 | 0 | 2 | 1 | -20 | true |
| 43 | 0 | 2 | 2 | 45 | false |
| 78 | 0 | 2 | 3 | 80 | false |
| 98 | 0 | 2 | 4 | 100 | false |
| 22 | 1 | -20 | 0 | 2 | false |
| - | 1 | -20 | 1 | -20 |  |
| 65 | 1 | -20 | 2 | 45 | false |
| 100 | 1 | -20 | 3 | 80 | false |
| 120 | 1 | -20 | 4 | 100 | false |
| 43 | 2 | 45 | 0 | 2 | false |
| 65 | 2 | 45 | 1 | -20 | false |
| - | 2 | 45 | 2 | 45 |  |
| 35 | 2 | 45 | 3 | 80 | false |
| 55 | 2 | 45 | 4 | 100 | false |
| 78 | 3 | 80 | 0 | 2 | false |
| 100 | 3 | 80 | 1 | -20 | false |
| 35 | 3 | 80 | 2 | 45 | false |
| - | 3 | 80 | 3 | 80 |  |
| 20 | 3 | 80 | 4 | 100 | true |
| 98 | 4 | 100 | 0 | 2 | false |
| 120 | 4 | 100 | 1 | -20 | false |
| 55 | 4 | 100 | 2 | 45 | false |
| 20 | 4 | 100 | 3 | 80 | false |
| - | 4 | 100 | 4 | 100 |  |

minDistance: {2, -20, 45, 80, 100} minDistance2: {2, -20, 45, 80, 100}

You can follow this process with any array using debugger in Visual Studio. We have chosen a small array as it is easy to follow through and takes up less space.

Note how minDistance2 returned the same answer is almost half the number of comparisons.

## Design of Experiments

* 1. *Tools Used*

The experiments for these codes were done in two different machines and specifications are as below:

Table 3:MacBook Pro

|  |  |
| --- | --- |
| Operating System | macOS Mojave Version 10.14.4 |
| RAM | 8.00 GB 2133 MHz LPDDR3 |
| System Type | 64 bit, x64 Processor |
| Processor | Intel Core i5- 8th Generation 3.1GHz |
| Visual Studio Version | 8.0.5 (build 9) |
| C# Tools Version | 2.1.9 |

Table 4: Lenovo Ideapad

|  |  |
| --- | --- |
| Operating System | Windows 10 Home |
| RAM | 8.00 GB |
| System Type | 64 bit, x64 Processor |
| Processor | Intel Core i5-7200U 2.5GHz |
| Visual Studio Version | 15.9.6 |
| C# Tools Version | 2.10.0 |

Apart from this, we used Microsoft Excel for Mac (Version 16.25) to generate graphs with trendlines for both the algorithms. It also shows the comparison trend between both the algorithms for counters and timers.

* 1. *Methodology and Techniques Implemented*

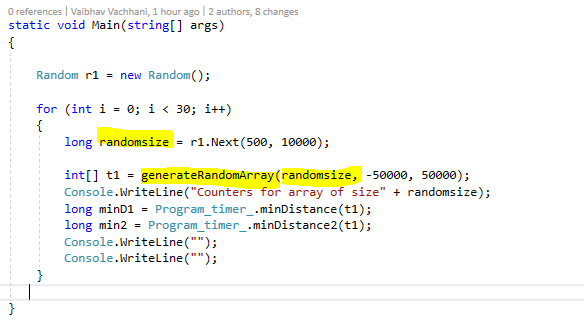


Figure 3: random array generation

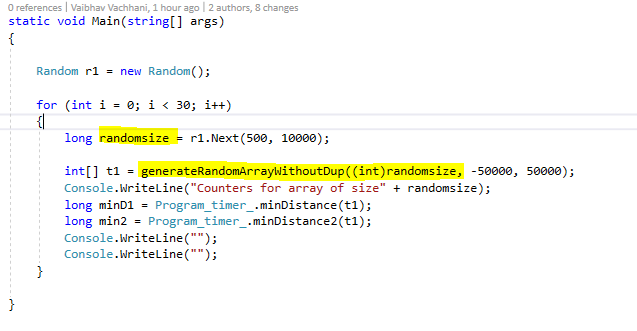


Figure 4: random array generation (without duplicates)

We have generated random array for all our test results although, we have fixed range for every array (-50000 to 50000). Meaning, each array will be of a random size of between 500 to 10000 but each element will be between -50000 to 50000.

There are two methods for generating arrays – ***generateRandomArray*** & ***generateRandomArrayWithoutDup.***

***generateRandomArray*** generates random array according to the parameters passed to it. It takes 3 parameters (*size*, *start* and *stop*). It will create an array of length *size*, and each element will be a random number generated between the range *start* and *stop*. ***generateRandomArrayWithoutDup*** is the same – but it does not allow duplicate elements in the array. So, an array returned by ***generateRandomArray*** might be {2,100,56,78,13,45,80,100} whose dmin would be 0. But any array returned by ***generateRandomArrayWithoutDup*** would never have a dmin value of 0.

We conducted 4 tests for each – counter and timer observations. 2 tests generated with each method mentioned above. Each test consists of 30 randomly generated arrays.

Meaning, 120 randomly generated arrays of random size to perform timer tests and 120 for counter test. Each set of 120 arrays were evenly split between methods – ***generateRandomArray*** & ***generateRandomArrayWithoutDup****.*

## 

## Experimental Results

Basic operation of the algorithm *(minDistance)* is the comparison *if ((arr[i] – arr[j]) < dmin && i != j)* if true; itassigns the distance to *dmin* and this operation will be performed the most times given any sorted array. If the statement is not true it returns *dmin*. This algorithm will roughly be executed times where is the length of an array, as it compares an element with the entire array.

Basic operation of the algorithm *(minDistance2)* is the comparison *if(temp < dmin)* where ( *temp = Math.Abs(arr[i] – arr[j])* ) if true; it will store the value of *temp* in *dmin* and this operation will be performed the most times given any sorted array. If the statement is false it returns *dmin*. This algorithm will roughly be executed times where is the length of an array, as it compares an element with the entire array.

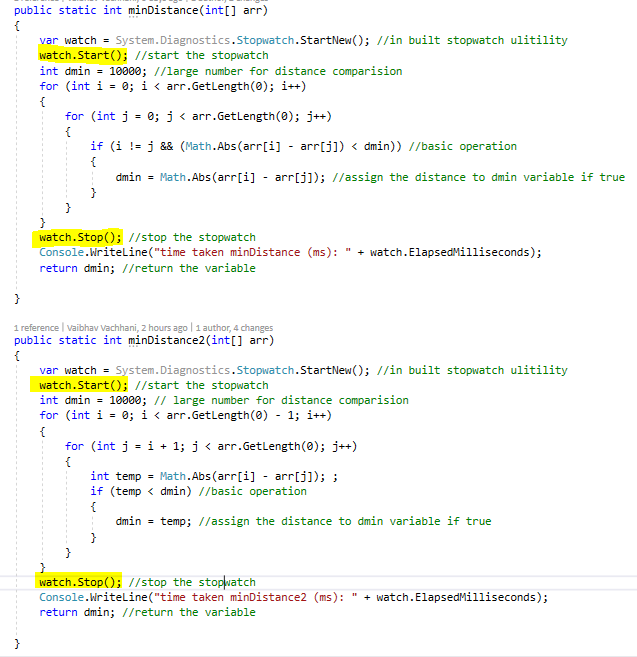


Figure 5: Timer Implementation for algorithms

In our experiments for timer, we first declare a Stopwatch object in our algorithms. We start the stopwatch right after declaring to ensure it accounts for all the operations before returning *dmin.* We stop the watch before the return statement and displaying the timer reading to the console as any code after the return statement won’t be executed.

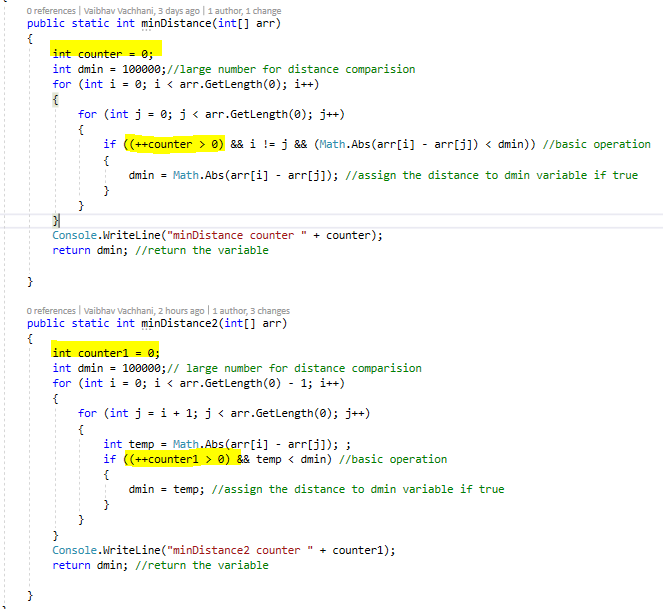


Figure 6: Counter Implementation for algorithms

In our experiments for counter, we first declare the counter variables. We pre-increment the variables in if statements (basic operations) before the comparisons as highlighted in the figure above. This ensures that the comparison is accounted for regardless of its result.

As you will notice from the tables on following pages – minDistance2 is always faster than minDistance1, regardless the array size. It will always have fewer counter value – meaning less comparisons and less time taken to return a result. We have discussed about the trend in these values in the next section where we have also attached the graphs to show a clear trend.

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

Figure 8: Timer data points

Figure 7: Counter data points

These graphs represent all the data points recorded for both – counter and timer experiments. A clear trend is observed from results with both methods (minDistance and minDistance2). The minDistance2 method equation has smaller co-officiant compared to minDistance – hence it is always lower than minDistance graph curve. This proves the minDistance2 is always faster than minDistance.

## Analysis of Experimental Results

We have attached graph representing our test runs for both- counter and timer. These graphs show a clear trend proving that suggested theory is correct. minDistance2 is more efficient than minDistance as the co-officiant of the minDistane2 equation is always smaller than that of minDistance. Both algorithms are quadratic as the equations on the graphs confirm.