# Description of 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:

|  |  |
| --- | --- |
| 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:

|  |  |
| --- | --- |
| 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:**

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.

# Theory analysis of the algorithms

minDistance:

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

problem size: ***n***

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 n^2 times where n is size of the array.

***Efficiency:***

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

Where u = (n-1)

Gives us:

2.) So, we now take *n* out as a constant (using formulae 3 from appendix) which gives us : 3.) Using the formula 1 again where u=(n-1) we get: n\*n   
  
So, big-theta of minDistance is (n^2).

The outer loop and the inner loop will run for n times (0 to n-1) where n is size of the array hence the efficiency class of this algorithm is n^2. (derivation above)

minDistance2:

basic operation: ***if (temp < dmin)***

problem size : ***n***

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=(n-1) and l=(i+1)

Give us :

1. Re-writing that, substituting u=n-1 gives us :
2. Solving using formula 2 from appendix gives us (n-1)(n-1+1)/2  
   which simplifies to 0.5(n^2-n)

So, big-theta of minDistance2 is (n^2).

# Implementation of the algorithms

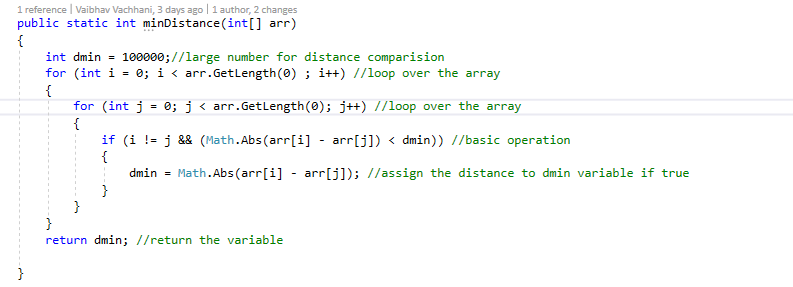


Figure 1:Implementation of minDistance

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 can not 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.

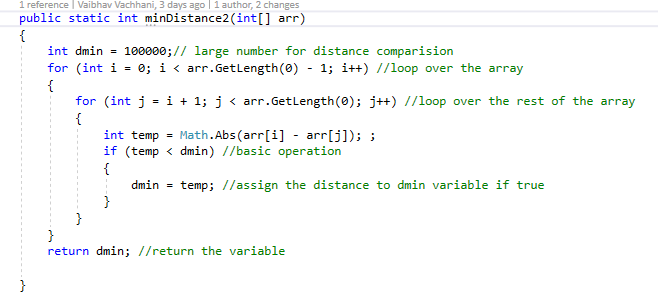


Figure 2: Implementation of minDistance2

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.

# Experimental Results

Basic operation of the algorithm *(minDistance)* is the comparison *if ((arr[i] – arr[j]) < dmin && i != j)* is true; itassign 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 comparision *if(temp < dmin)* where ( *temp = Math.Abs(arr[i] – arr[j])* ) is 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.

# Appendix

Figure 3: formulae 1

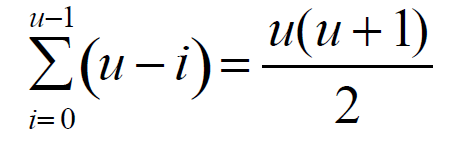


Figure 4: formulae 2

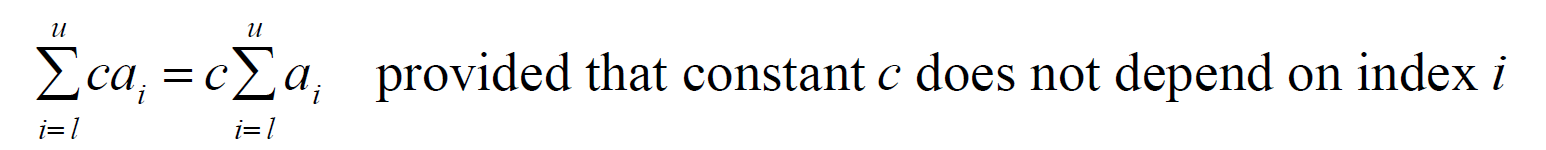


Figure 5: formulae 3