https://docs.google.com/document/d/1V_sfzlnILqFFMS3mJLrN7s4lKMwGqr_fctLkJDIiACA/edit?usp=sharing

Introduction:

My java code for the Matrix Profile currently runs based on two files. Everything runs inside the main method of the child class, Distance Matrix, which implements the class SquareGrid. Different versions of the code either run based on my test arrays or based on arrays created via .txt files full of the raw integer data points Tide Pool provides. Given two arrays of equal or unequal length, the code creates a distance matrix. When two values being compared are exactly equal the Distance Matrix will produce a zero. We are looking for continuous strings of zeros, and total zeros per comparison, to find duplicate sets of data.

Explanation and Comparison of traversals options:

This section of the paper will cover the differences between horizontal and diagonal traversals to create of distance matrices and why this implementation is focused around diagonal traversals.

The coordinate grid values are only positive and are formatted (x,y). Y being the vertical axis(up/down) X being the horizontal axis(left/right). See diagram 1.

Diagram 1:

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

*Grid coordinates for a comparison of 2 data sets each 4 values long.

The computation to make a Distance Matrix is rather simple, especially when the only focus is on exact matches (zeros). My code makes the grid via diagonal traversals rather than going row by row(horizontally). Making the grid row by row is simpler to code (all that is required is two nested for loops), but is limited because it only compares one value from a set against every value from the other set. So given two sets of data called X and Y. Row by row comparison is comparing Xi against Yi, Yi+1, Yi+2...Yfinal. The same process is then repeated for Xi+1...Xfinal. With horizontal traversals, there are as many traversals as there are rows. Horizontal traversals can easily create a Distance Matrix, however simply creating the Distance Matrix data structure provides no information, and does not allow for information to be gathered, unlike the diagonal traversal approach.

Horizontal Traversals:

Diagram 2:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

*The numbers(1-16) represent the order of comparisons in a row by row(horizontal) Distance Matrix creation.

Diagram 3:

1	x	x	x
2	x	x	x
3	x	x	x
4	x	x	x

*The numbers 1-4, each represent a horizontal traversals and the order in which they occur.

Building the Distance Matrix diagonally, allows for consecutive comparisons. So if you start by comparing Xi with Yi you then compare Xi+1 with Y(i+1), then X3 with Y(i+2) until the program reaches the end of one of the data sets. Creating an distance matrix with diagonal traversal means incrementally advantances the comparison data points in both sets, rather than one at a time. Thus, diagonal traversals allows the program to make every possible comparison, and record structured data about matching values and their locations, simultaneously. One simple thing it can find that a horizontal based traversal cannot is strings of matching data points rather than individual matches.

Diagonal Traversals:

Diagram 4:

1	11	14	16
5	2	12	15
8	6	3	13
10	9	7	4

*The numbers(1-16) represent the order of comparisons in a diagonal Distance Matrix creation.

Diagram 5:

1	5	6	7

2	x	x	x
3	x	x	x
4	x	x	x

*The numbers 1-7, each represent a diagonal traversals and the order in which they occur. With diagonal traversals, there are as many traversals as there are edge positions on the grid or (SideLengthX+ SideLengthY-1).

Explanation of My Codes Diagonal Traversal Method

The first traversal is always from the top left corner of the grid, coordinate (0,0), to the bottom right(SideLengthX-1, SideLengthY-1) and represents the longest traversal. This is worth noting because it has the greatest potential for matches. While traversals numbers 4/7 in diagram 5 represent the minimum comparisons possible, 1. To optimize the code we set limits on how small a given traversal could be. If it fell below a certain threshold the traversal and all remaining traversals of smaller lengths were not computed, thus saving computation time. The threshold could represent a percentage of the longest traversal or a static number. This adaptation "cuts" off the corners of the matrix.

////

REMOVE??

The MatrixHelper method, within DistanceMatrix.java, completes one diagonal traversal then sets up the starting position for the next one. Knowing when to stop comparing values(reaching the end of the grid) and where to begin the next traversal is based on two components. One being the internal values being maintained, such as X/Y position, the number of traversals completed thus far, and the length of the datasets. Second, it relies on the structure of the Distance Matrix being a square, which means the datasets being compared must be equal length.

/////////

The code is run primarily by two main functions, DisMatrixFillingDiagonally(), and MatrixHelper(). DisMatrixFillingDiagonally() takes in two arrays. It determines the number of diagonal traversals that will take begin on the X and Y axises, based on their lengths. It then determines the shorter side length between the two sets, called Q. Q is the longest possible diagonal traversal and thus is used to set our threshold in this cut off senario. There are then two separate for loops that iterate through the # of diagonal traversals on the X and Y axis respectively. In each loop there is a check if the traversal will be above or below the threshold. If it is above that threshold, each iteration in those for loops calls MatrixHelper(), which completes one diagonal traversal then returns the values needed for the starting position of the next traversal.

MatrixHelper(), takes in two arrays, the starting X, Y coordinates for the traversal, a counter for how far from the center diagonal this traversal is, and a counter of the number of traversals that have occured. It returns how far it is from the center diagonal. It takes the starting coordinates, compares the values, records weather it was a match or not, records how many matches have occured in a consecutive f any, records the coordinates of a original match in consecutive matches, then increments the X and Y values. It then checks to see if we have stepped out of the bounds of either of our lists aka if the X/ index is greater than the length of X/Y array, if so it jumps out knowing it has completed a diagonal traversal. If not then it repeats the previous steps.

A = number of diagonal traversals on X axis
B = number of diagonal traversal on the Y axis
Q = maximum possible length of a diagonal, shorter sidelength value between A and B.
Assuming A < B Q = A.

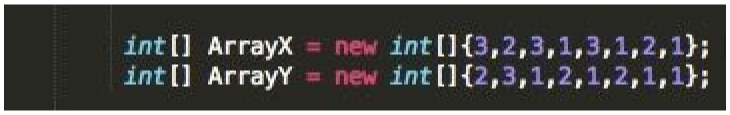
A^2 represents the first for loop in DisMatrixFillingDiagonally() while BA represents the second for loop in DisMatrixFillingDiagonally(). Assuming A < B

Diagonal Rectangular Matrix Profile algorithm = Θ(AQ+BQ) ~=~ Θ(A^2+BA)
= Ω().

Given two arrays of length N and M, where N>= M, the time complexity of the algorithm is O(n^2).

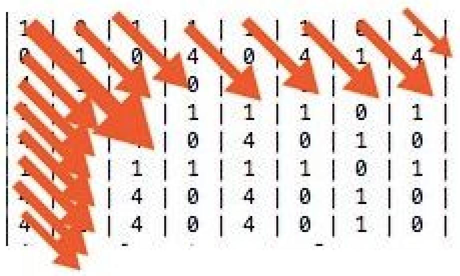
Within this method are various information gathering tools(static members of the Distance Matrix class). Two arrays, and matrix(see Visualized Data Structures x,z,y in the results section below). The different structures record: each zeros(match), the longest string of consecutive zeros, and its starting position in a traversal. Each index corresponds to a specific a traversal. Index zero refers to the longest traversal starting at (0,0). The shortest traversals, each 1 comparison long are at the middle index and the last index in the arrays and matrix.

Here is a **Simple example**:



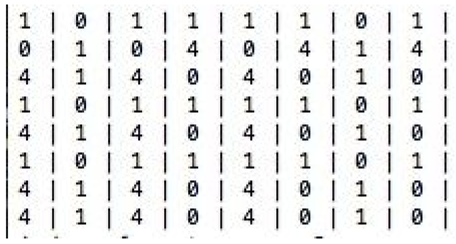
```java
int[] ArrayX = new int[]{3,2,3,1,3,1,2,1};
int[] ArrayY = new int[]{2,3,1,2,1,2,1,1};
```

A Distance Matrix based on these values is created diagonally



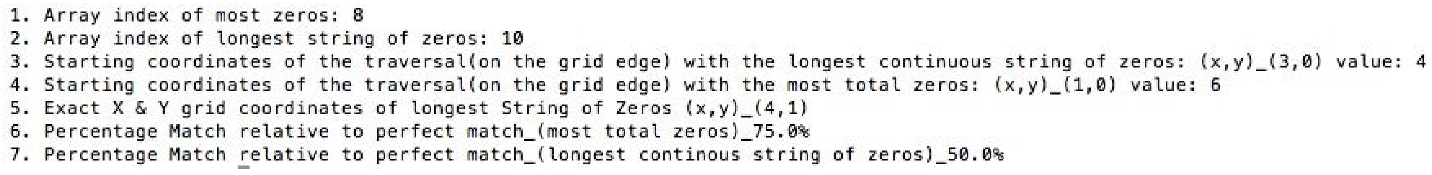
Distance Matrix:

*Remember the top left of the grid starts at (0,0)



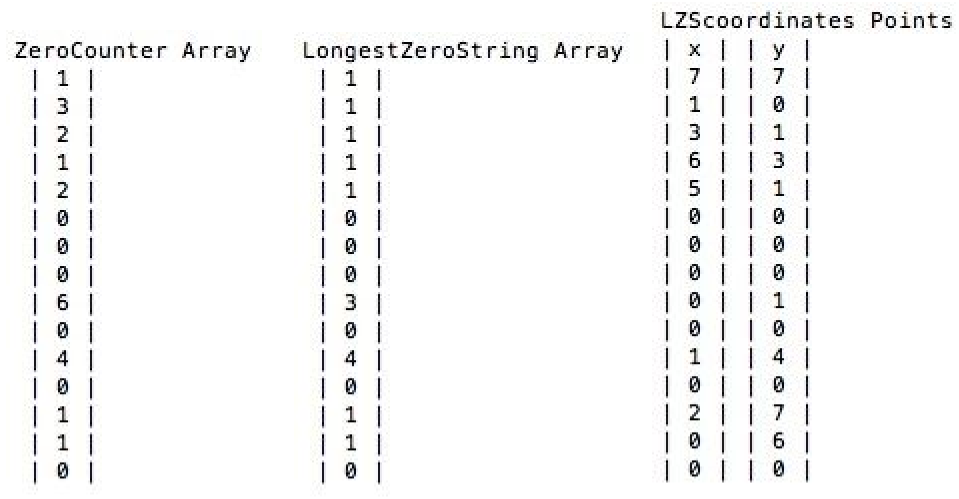
```
|1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
|0 | 1 | 0 | 4 | 0 | 4 | 1 | 4 |
|4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
|1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
|4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
|1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
|4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
|4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
```

Results Section: (built into the main method using a few helper functions to provide, immediate feedback to whoever runs a comparison)



```
1. Array index of most zeros: 8
2. Array index of longest string of zeros: 10
3. Starting coordinates of the traversal(on the grid edge) with the longest continuous string of zeros: (x,y)_(3,0) value: 4
4. Starting coordinates of the traversal(on the grid edge) with the most total zeros: (x,y)_(1,0) value: 6
5. Exact X & Y grid coordinates of longest String of Zeros (x,y)_(4,1)
6. Percentage Match relative to perfect match_(most total zeros)_75.0%
7. Percentage Match relative to perfect match_(longest continous string of zeros)_50.0%
```

Visualized Data Structures (x,y,z):



```
                                                    LZScoordinates Points
ZeroCounter Array    LongestZeroString Array       | x | | y |
 | 1 |                 | 1 |                       | 7 | | 7 |
 | 3 |                 | 1 |                       | 1 | | 0 |
 | 2 |                 | 1 |                       | 3 | | 1 |
 | 1 |                 | 1 |                       | 6 | | 3 |
 | 2 |                 | 1 |                       | 5 | | 1 |
 | 0 |                 | 0 |                       | 0 | | 0 |
 | 0 |                 | 0 |                       | 0 | | 0 |
 | 0 |                 | 0 |                       | 0 | | 0 |
 | 6 |                 | 3 |                       | 0 | | 1 |
 | 0 |                 | 0 |                       | 0 | | 0 |
 | 4 |                 | 4 |                       | 1 | | 4 |
 | 0 |                 | 0 |                       | 0 | | 0 |
 | 1 |                 | 1 |                       | 2 | | 7 |
 | 1 |                 | 1 |                       | 0 | | 6 |
 | 0 |                 | 0 |                       | 0 | | 0 |
```

Example of how to read the Data Structures:
Each index in these arrays and matrix are assigned to the same traversal. So for index 0 (the traversal is from top left to bottom right) one zero was found, as seen in the ZeroCounter Array, thus the longest string of zeros found was one long, as seen in the LongestZeroString Array, and the coordinates of the longest string of zeros was at (7,7) the bottom right hand corner on the distance matrix, as seen in LZScoordinates Points. This checks out given that the bottom right corner value is 0.

Analysis of example results:

So based on line 3. We determine the traversal with the longest string of continuous zeros starts at the edge position (3,0) with length 4. Based on line 5. the position where the string of zeros truly starts is coordinates (4,1). This means the longest string of zeros starts at ArrayX index 4 and ArrayY index 1, the 2nd comparison in the 11th traversal. According to line 7. The consecutive zero string length was 4 long which represents 50% of what a theoretical perfect match would be, 8 zeros in a row starting at (0,0) ending at (7,7). According to 4. However that are more matches total in the traversal that starts at (1,0), at 6 matches, which according to line 6. Represents a 75% match relative to the perfect match.

Limitations and next steps:

The main concern I have right now, with the diagonal traversal, is that all the math and code is designed around the grid being a square, so the two arrays of data points inputted must be equal length. I am not sure if appending a unique value such as zero, a value that will never be matched in our data, to the shorter array so the lengths matched would solve this, but it is a potential adaptation. It would however be computationally expensive because it requires a full copy of the smaller data set and filling in the blanks, given that java arrays are statically size. Unless you knew the difference in the lengths of the data set before transferred the data into arrays, in which case a second copy of the shorter arrays would be unnecessary. Ed also mentioned gaps in the data that have to be accounted for, which again I believe would be best sorted with a unique value that could never be mistaken for a match. (Perhaps if the unique value is found) it doesn't break/add to an ongoing string of matches.

Another concern, given the immense amount of data tidepool is handling, is the computational complexity of the algorithm. The Distance Matrix profile is exact, has no risk of false negatives even with missing data, and provides a lot of information. The UCR Matrix Profile page suggests that a Matrix

Profile can have constant time complexity. However, given the internal nested loops inherent to my code and the traversals it takes to build the code, I believe the time complexity is O(n^2), but I am not 100% sure. Using the java files DistanceMatrixUpdate2.java and SqaureGridUpdate2.java I did a mock speed test. Given two sets of 11,044 data points to compare, and create a Distance Matrix for, the code takes roughly 4.90 seconds, (I can't call it any closer than, 1/10th of a second, given it was simply me along using my phone timer) on my 2011 MacBook Air to generate only the results section. At roughly 288 data points a day, that is 38.347222 days in 4.90secs. That equates to 46.63secs per one patient's yearly data (assuming daily data donations). It would be interesting to see this done on different computers, because this test far exceeded by speed expectations. Some testruns got as low as 4.4sec. This trial didn't require the program to print anything information with each match, loop break, or to print the matrix itself.

The code... works if the arrays are unfilled(indexs at the end without values)-same amount of values compared tho
The code...worked when I took the below values out of data set1

1000
1000
1000
1000
1000
1000
1000
1000
996
1000
1000

1000
1000

Bugs*(9/12/18)- The coordinates for the longest zero string, found in the Matrix LZScoordinates Points, are sometimes one coordinate position too high or low in their report. The errors tend to occur with strings of zeros that hit the bottom hand corner or begin at the start of the grid. (WILL INVESTIGATE)

Fixed

There are two situations when the coordinates are being logged into the 2 column points matrix. When the final zero is the last position in the traversal, or when it isn't. You have to add +1 to the coordinates no matter what. This is because there is the possibility (X/Ypos-activeZeroCounter) < 0 which is impossible. The strings of zero that end at the final position need +1 added right there mid calculation because the coordinates are being calculated during the same loop as the final zero is found, but in the case above the coordinates are being calculated in the loop after the final zero is found, thus the +1 happened as a result of the loop.

Andy’s Comments:

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)

Bottom half:
```
if(xpos== sidelengthX-1-traversals && ypos == sidelengthY-1)
break;
```

Top half:
```
if(xpos==sidelengthY-1 && ypos == sidelengthX-1-traversals)
```

X = 6
Y = 4

0,0; 1,1; 2,2; 3,3; 4,4
0,1; 1,2; 2,3; 3,4...
0,2
…

1,0; 2,1; 3,2; 4,3; 5,4
2,0; 3,1; 4,2; 5,3; 6,4
3,0
…

```
If (xpos >= X || ypos >= Y)
```

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)