Two Approaches to Traverse A 2D Matrix Diagonally Using an Iterative Method and Breadth-First Search

Basima Elshqeirat¹, Hussam Qudeimat²

Department of Computer Science, the University of Jordan, P.O Box 11942, Amman, Jordan

Abstract- Traversing matrices is fundamental in the programming world, it helps programmers pass by elements inside the matrix in a certain order, helping them identify these elements and use them according to their goals. And since traversing matrices can be in any order (i.e. left to right, right to left…) each order has its uses. If you would like to traverse through an array and print out every possible element without skipping, you would have to start from the leftmost or the rightmost part of the array, or you could even choose random indices (without replacement) to print out randomly generated sequence of elements. In this paper’s case, it is sometimes important to traverse a 2D matrix diagonally, as if traversing it row by row, but instead, diagonal by diagonal. Two neat algorithms will be introduced inside this paper and they will be compared in terms of time complexity, space complexity and runtime.

# Introduction

Going through elements of a matrix is essential in the computer programming field, it could for instance, provide the programmers the ability to calculate the average of a dataset given a matrix of numbers, or perhaps find the employee of the month given a matrix of employees, and in the case of a 2D matrix it could help them read it row by row, to achieve a certain task.

In our case, we will explore two ways into achieving a diagonal traverse, that is, going through a 2D matrix diagonal by diagonal (see Image A), using two methods, one of which is an interesting iterative method and the other using the well-known Breadth-First Search algorithm, and compare them together. It is important to mention that the matrix doesn’t have to be filled (be exactly m x n, see Image B).

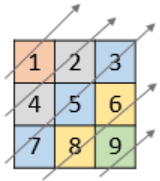


Image A. Example of target matrix

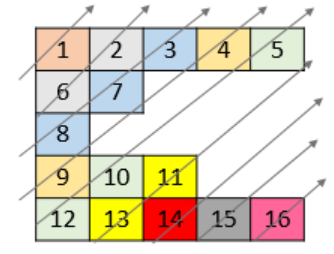


Image B. Example of target matrix

Looking at image A, the expected output would be [1, 4, 2, 7, 5, 3, 8, 6, 9,], conversely, the expected output on image B would be [1, 6, 2, 8, 7, 3, 9, 4, 12, 10, 5, 13, 11, 14, 15, 16].

Now that we understand what this paper means by diagonal traverse, we can start talking about the first method, the iterative method.

The iterative method relies heavily on the indices of the rows and columns, which this paper will explain later in deep detail, the Breath-First Search however, as you may know, typically uses a FIFO - First in First Out - approach which is achieved by using a queue, and in our case is heavily dependent on two factors: the order of which the cells are inserted into the queue, and the adjacent cells that reside exactly under and to the right of an individual cell.

# Methodology

In this section, it is going to be demonstrated in detail on how both algorithms work, we are going to dive deep into both approaches, analyse time and space complexities, and compare them together.

## The Iterative Method

Approach

Starting off with the iterative method, let us first look at a normal 2D matrix with color coding involved (see Image C).

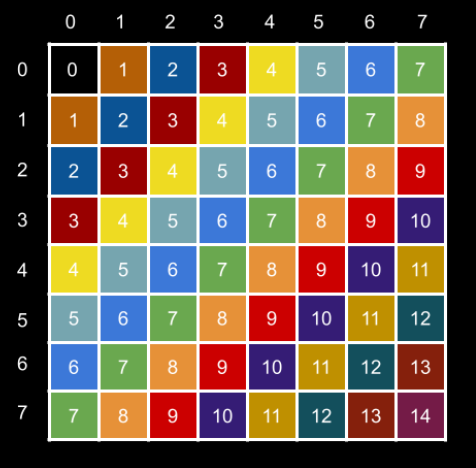


Image C. Example of color-coded target matrix

Approach

Each color and number maps to a unique diagonal, when the 2D matrix is put this way, a pattern starts to appear, if we look closely at the indices, let’s take row index 7 and column index 2, which gives us the element 9, now let’s take another two indices that map to the same diagonal, let’s say row 6 and column 3, which also maps to – of course – the element 9. Now looking at this closely, we see that 7 + 2 equal 9 and conveniently enough, 6 + 3 also equals 9, which means no matter what row and column we take, their sum is going to sum up to a unique diagonal number.

Given this valuable information, we can now map the sum of any ith row and jth column to one and only one diagonal, which in return helps us identify unique diagonals and store them separately in different containers. You might be asking: “how do we traverse this 2D matrix in order starting from the upper most diagonal to the bottom one? “, there are multiple ways to do that, but in this paper only one solution will be given, that is, by using a container that stores keys and values, and finds them in constant time (usually a HashMap), we can start traversing the 2D matrix normally starting from the last row, and working our way up row by row, and the reason why we start from the last row is so that we maintain the order of the elements and that can be achieved by starting from the bottom, it is important to note that the HashMap maps a unique number to a diagonal, the diagonals can be stored as numbers in order using a container that stores numbers (i.e. arrays, vectors, ArrayLists,… etc.), adding the number of the ith row to the jth column we get a unique number, the HashMap maps that number into a unique container that we add the current element to, and so on until we reach the top row and last column.

Now that the diagonals are stored separately inside the HashMap, we initialize an index integer, giving it an initial number 0, the reason why we use 0 is that so we print the numbers in order, so starting with diagonal number 0, then diagonal number 1 and so on as seen in Image C.

We traverse the HashMap starting with index 0, printing out the numbers inside our container one by one giving us the right order, increment our index integer, until the current integer does not exist anymore inside the HashMap, concluding the iterative method.

Pseudo-Code

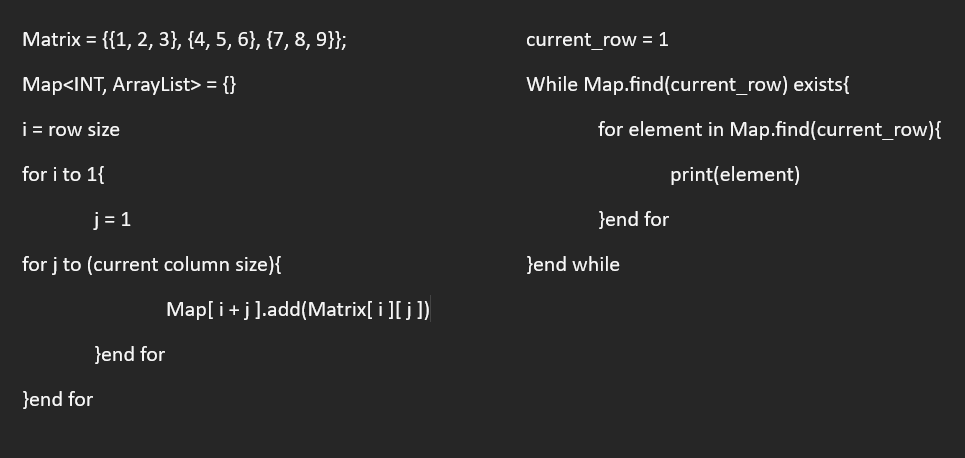


Image D. Pseudo code of the iterative metho

Space and time complexity analyses

Initially, we are instantiating a HashMap, then we add all the elements inside the original matrix into this HashMap, each operation takes O(1) time, and given that we have (n) total elements inside our matrix, filling the HashMap would take roughly (n) time, then, we are printing out the elements inside that HashMap, the (while) loop alone has a complexity of O(m), and it is going to run for about however the number of unique diagonals we have, let’s call that number (m), the (for) loop inside it however prints out the elements inside each container which also takes (n) time(the number of total elements inside the matrix), adding these together we would get about (n + m) time, giving us a O(n + m) time complexity, and since we are storing all the elements inside the HashMap we are getting a space complexity of O(n).

Source code C++

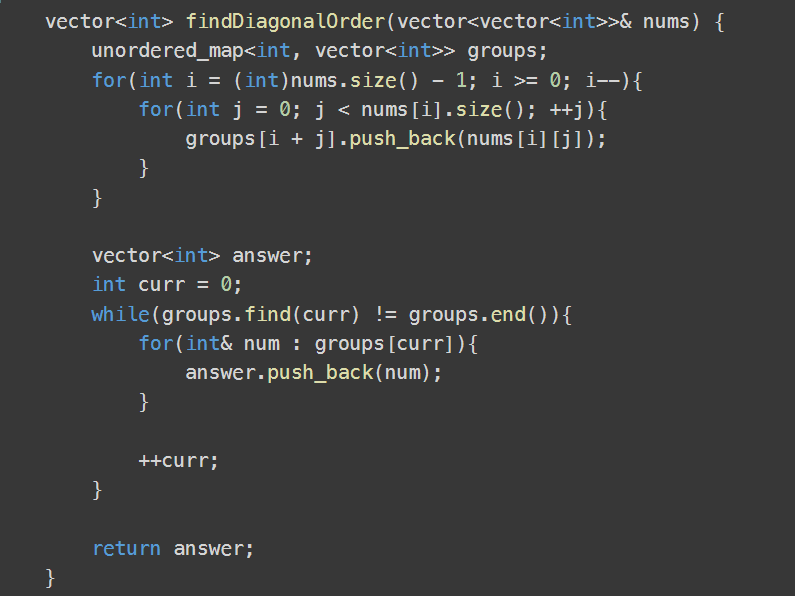


Image E. C++ Source code of the given iterative method

## Breadth-First Search Method

Approach

We would want to traverse our array in the following fashion: (see Image F.).

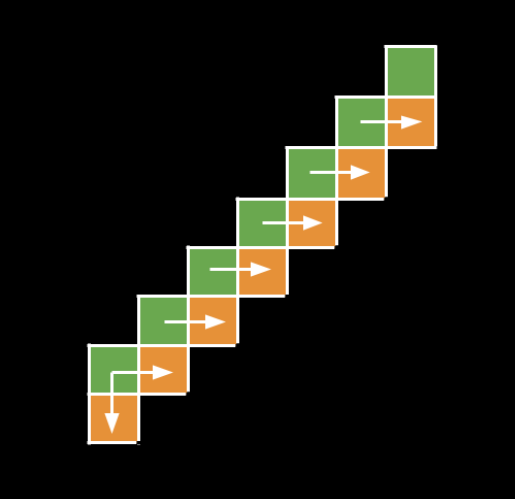


Image F. Example iteration of the BFS method

Looking at the previous image, we notice a tree-like structure, where a node is a cell, and a tree level inside this structure would be a diagonal. Using this pattern, we can confidently create an algorithm using a level order traversal (in this case diagonal order traversal) and print our diagonals one by one, the only exception here is that if we look again, the first element in the diagonal which happens to be the bottom element, is the only node that we would have to check if there is a cell right beneath it, as for all the other nodes, we only need to check the next cell to the right.

Algorithm

1. Initialize a queue
2. Start from the top left element inside the 2D matrix and add it to our queue
3. While queue is not empty keep a variable on the side that stores the current queue size
4. While the current queue size is larger than 0 perform the next operations
5. Check the first element inside the queue and print it
6. If it’s the first element in the diagonal then: check the bottom neighbor and right neighbor
7. If not, only check the right neighbor
8. In order, add the elements we just discovered into our queue
9. Repeat

Space and time complexity analyses

Starting with time complexity, let us recap our approach, we are inserting each diagonal inside our queue, then accessing the next diagonal and so on, each access requires O (1) time, and since we are moving through all the elements once, we would get a time complexity of O(n).

Space complexity however, in this term this algorithm beats the iterative method in space, because at any given time in this algorithm, the queue which holds our elements is only going to have at most one diagonal present at any given time, and so by using the Pythagoras theorem, we would have a space complexity of O(√n). I would also like to mention that here, we only pass by the elements once, but in the iterative method we passed twice.

Source code C++

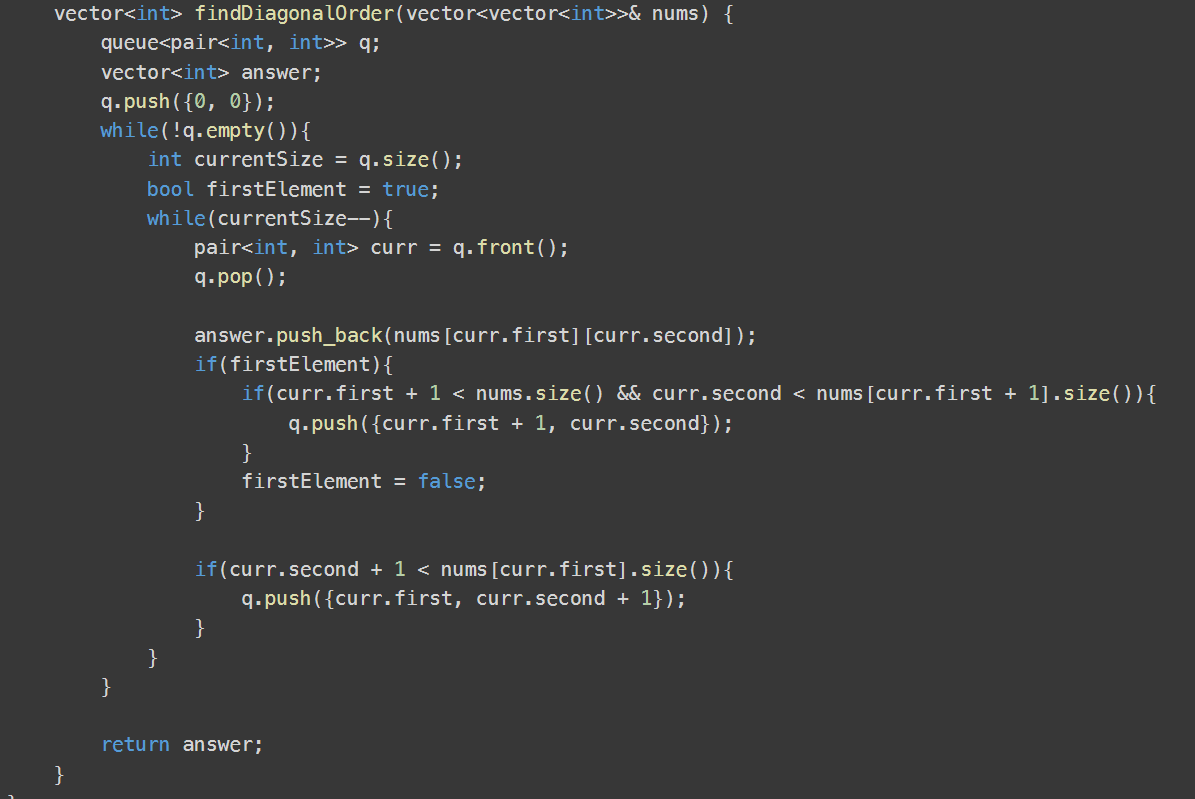


Image G. C++ Source code of the given Breadth-First Search method

# Test cases, runtime and results

Test cases

Forty test cases were auto generated to test both of our algorithms, they are divided into two halves, twenty test cases with 10^6 total cells, and the other half has 10^7 total cells.

To ensure that we are always using the same test cases in both methods, after auto generating, they were stored in separate files with a total of forty text files each representing a test case.

Test machine specs

Core i7-9700F, 32GB DDR4, Windows 11, 64-bit System.

Runtime

The following table represents all 40 test cases, with the runtime of each method on a single test case in milliseconds.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 106 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Iter | 113 | 113 | 112 | 114 | 113 | 112 | 115 | 113 | 112 | 112 | 114 | 114 | 114 | 113 | 112 | 113 | 114 | 113 | 113 | 114 |
| BFS | 67 | 70 | 67 | 68 | 67 | 68 | 67 | 67 | 68 | 69 | 67 | 68 | 68 | 69 | 67 | 67 | 67 | 70 | 67 | 77 |
| 107 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Iter | 1096 | 1099 | 1097 | 1099 | 1097 | 1096 | 1095 | 1095 | 1094 | 1099 | 1096 | 1092 | 1093 | 1097 | 1116 | 1096 | 1096 | 1097 | 1095 | 1144 |
| BFS | 699 | 681 | 688 | 683 | 699 | 681 | 685 | 705 | 680 | 682 | 681 | 685 | 681 | 680 | 688 | 698 | 683 | 683 | 684 | 682 |

Table A.

The next table shows the overall performance of both algorithms by calculating the average of the previous test cases in milliseconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Overall performance | |  |  |
| **106** | **Average** | **Min** | | **Max** |
| Iterative | 113.5 | 113 | | 115 |
| BFS | 68.25 | 67 | | 77 |
| **107** | **Average** | **Min** | | **Max** |
| Iterative | 1099.45 | 1092 | | 1144 |
| BFS | 686.4 | 680 | | 705 |

Table B.

Using the previous data, we can conclude that at 106 elements, BFS beats the iterative method by around 39.9%, and at 107 elements, BFS also beats the iterative method by 37.6%, giving us an average improvement of 38.75%.

# Conclusion

Traversing matrices is quite important in the computer field, it has been shown two methods into traversing a 2D matrix diagonally in-order, starting from the top left diagonal and moving to the bottom right, using both an iterative method and the Breadth-First Search algorithm.

They were also compared together in terms of implementation, time complexity, space complexity, and runtime, the iterative method having a O(n + m) time complexity, while the BFS method takes O(n), and O(n) space complexity for the iterative method while the BFS had a better space complexity of O(√n). It is also important to mention that the BFS algorithm beat the normal iterative method in runtime by about 38.75%.