Previous Next

498. Diagonal Traverse

Dec. 24, 2019 | 35.7K views

Average Rating: 4.53 (36 votes)

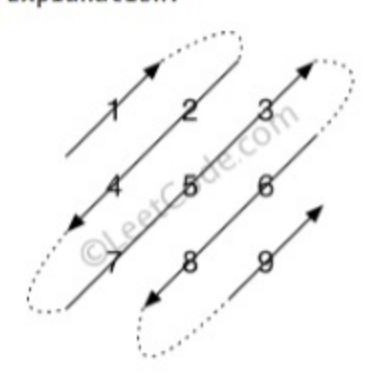
Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]

Output:  [1,2,4,7,5,3,6,8,9]

Explanation:
```



Note:

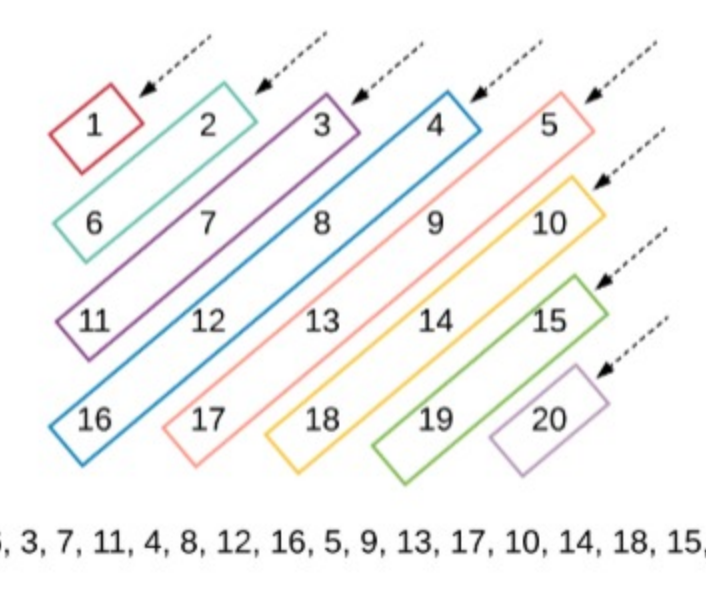
The total number of elements of the given matrix will not exceed 10,000.

Solution

Approach 1: Diagonal Iteration and Reversal

Intuition

A common strategy for solving a lot of programming problem is to first solve a stripped down, simpler version of them and then think what needs to be changed to achieve the original goal. Our first approach to this problem is also based on this very idea. So, instead of thinking about the zig-zag pattern of printing for the diagonals, let's say the problem statement simply asked us to print out the contents of the matrix, one diagonal after the other starting from the first element. Let's see what this problem would look like.



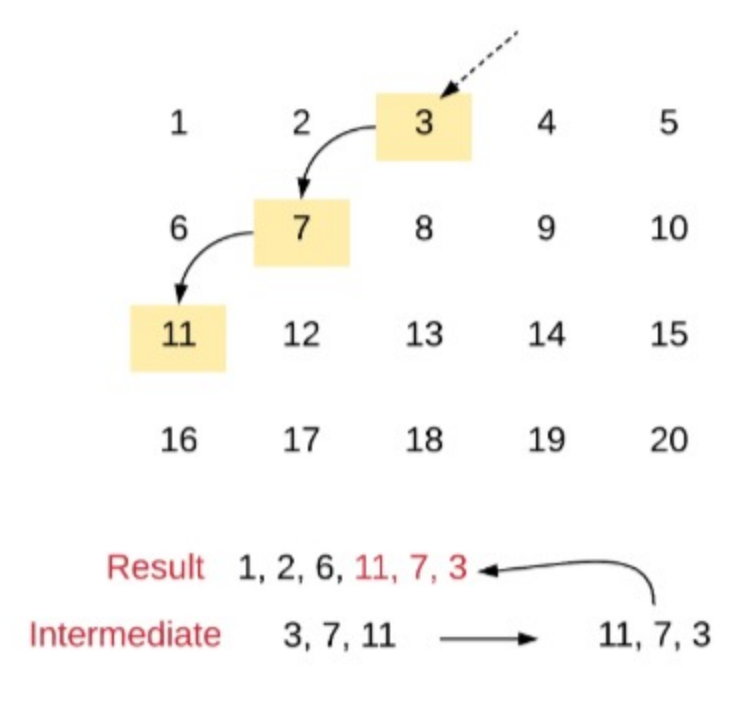
The first row and the last column in this problem would serve as the starting point for the corresponding diagonal. Given an element inside a diagonal, say $[i, j]$, we can either go up the diagonal by going one row up and one column ahead i.e. $[i-1, j+1]$ or, we can go down the diagonal by going one row down and one column to the left i.e. $[i+1, j-1]$. *Note* that this applies to diagonals that go from `right to left` only. The math would change for the ones that go from left to right.

This is a simple problem to solve, right? The only difference between this one and the original problem is that some of the diagonals are not printed in the right order. That's all we need to fix to get the right solution!

> We simply need to reverse the odd numbered diagonals before we add the elements to the final result array. So, for e.g. the third diagonal starting from the left would be [3, 7, 11] and before we add these elements to the final result array, we simply reverse them i.e. [11, 7, 3].

Algorithm

1. Initialize a `result` array that we will eventually return.
2. We would have an outer loop that will go over each of the diagonals one by one. As mentioned before, the elements in the first row and the last column would actually be the heads of their corresponding diagonals.
3. We then have an inner while loop that iterates over all the elements in the diagonal. We can calculate the number of elements in the corresponding diagonal by doing some math but we can simply iterate until one of the indices goes out of bounds.
4. For each diagonal we will need a new list or dynamic array like data structure since we don't know what size to allocate. Again, we can do some math and calculate the size of that particular diagonal and allocate memory; but it's not necessary for this explanation.
5. For odd numbered diagonals, we simply need to add the elements in our intermediary array, in reverse order to the final result array.



Java | Python | Copy

```python
class Solution:

    def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:

        # Check for empty matrices
        if not matrix or not matrix[0]:
            return []

        # Variables to track the size of the matrix
        N, M = len(matrix), len(matrix[0])

        # The two arrays as explained in the algorithm
        result, intermediate = [], []

        # We have to go over all the elements in the first
        # row and the last column to cover all possible diagonals
        for d in range(N + M - 1):

            # Clear the intermediate array everytime we start
            # to process another diagonal
            intermediate.clear()

            # We need to figure out the "head" of this diagonal
            # The elements in the first row and the last column
            # are the respective heads.
            r, c = 0 if d < M else d - M + 1, d if d < M else M - 1

```

Complexity Analysis

- Time Complexity: $O(N \cdot M)$ considering the array has N rows and M columns. An important thing to remember is that for all the odd numbered diagonals, we will be processing the elements twice since we have to reverse the elements before adding to the result array. Additionally, to save space, we have to `clear` the intermediate array before we process a new diagonal. That operation also takes $O(K)$ where K is the size of that array. So, we will be processing all the elements of the array at least twice. But, as far as the asymptotic complexity is concerned, it remains the same.
- Space Complexity: $O(min(N, M))$ since the extra space is occupied by the intermediate arrays we use for storing diagonal elements and the maximum it can occupy is the equal to the minimum of N and M. Remember, the diagonal can only extend till one of its indices goes out of scope.

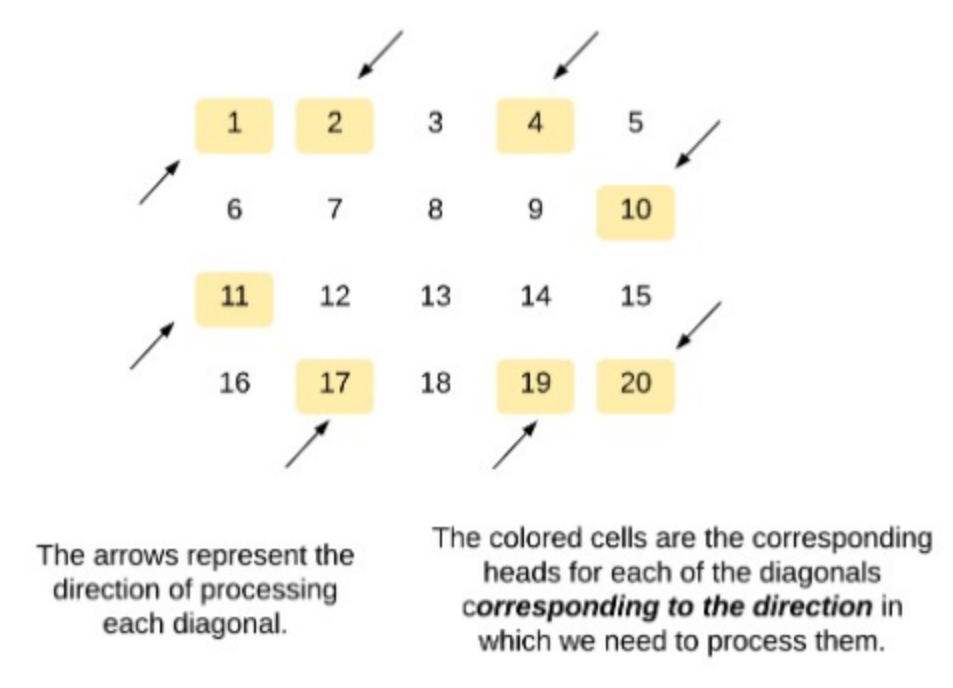
Approach 2: Simulation

Intuition

This approach simply and plainly does what the problem statement asks us to do. It's pure simulation. However, in order to implement this simulation, we need to understand the walking patterns inside the array. Basically, in the previous approach, figuring our the `head` of the diagonal was pretty easy. In this case, it won't be that easy. We need to figure out two things for each diagonal:

1. The direction in which we want to process it's elements and
2. The head or the starting point for the diagonal `depending upon its direction`.

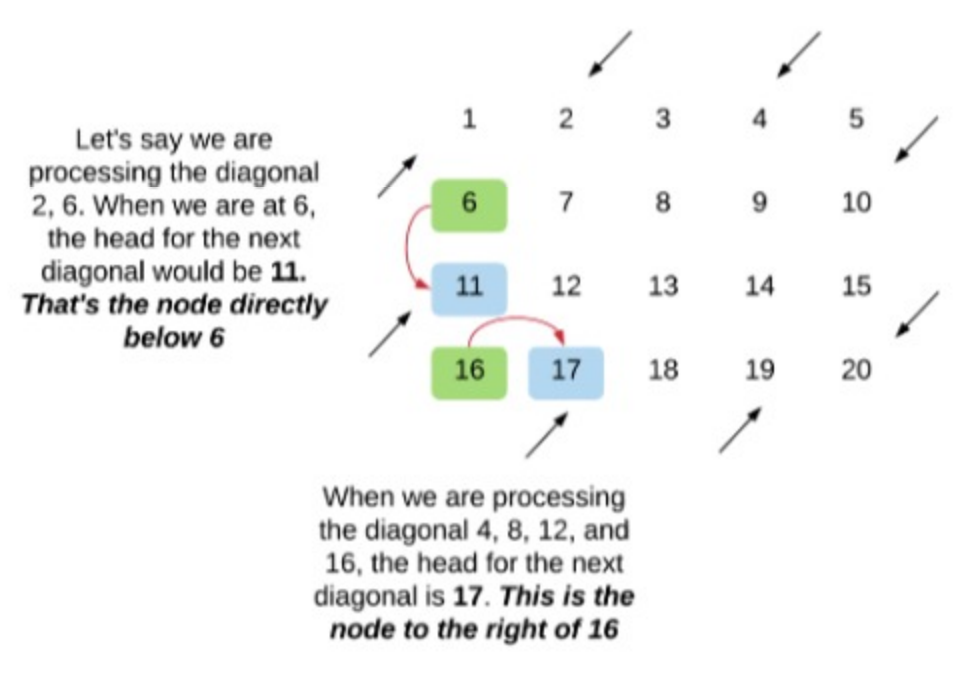
Let's see these two things annotated on a sample matrix.



Now that we know what two things we need to figure out, let's get to the part where we actually do it! The direction is pretty straightforward. We can simply use a boolean variable and keep alternating it to figure out the direction for a diagonal. That part is sorted. The slightly tricky part is figuring out the head of the next diagonal.

The good part is, we already know the `end` of the previous diagonal. We can use that information to figure out the head of the next diagonal.

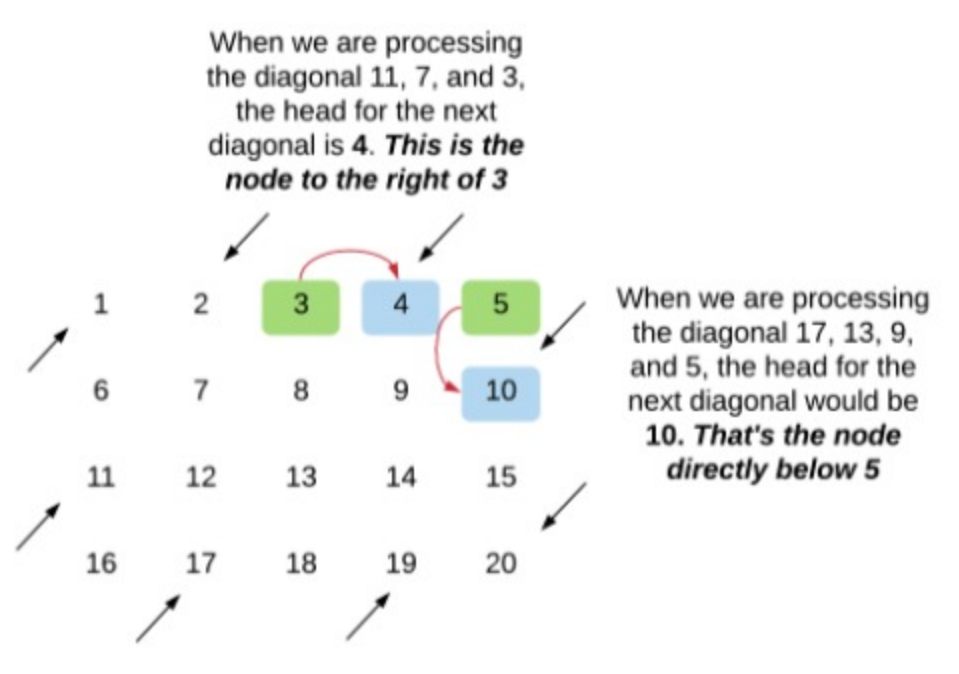
Next head when going UP
Let's look at the two scenarios that we may come across when we are at the tail end of a downwards diagonal and we want to find the head of the next diagonal.



So, the general rule that we will be following when we want to find the head for an upwards going diagonal is that:

> The head would be the node directly below the tail of the previous diagonal. Unless the tail lies in the last row of the matrix in which case the head would be the node right next to the tail.

Next head when going DOWN
Let's look at the two scenarios that we may come across when we are at the tail end of an upwards diagonal and we want to find the head of the next diagonal.



So, the general rule that we will be following when we want to find the head for a downwards going diagonal is that:

> The head would be the node to the right of the tail of the previous diagonal. Unless the tail lies in the last column of the matrix in which case the head would be the node directly below the tail.

Algorithm

1. Initialize a boolean variable called `direction` which will tell us whether the current diagonal is an upwards or downwards going. Based on the current direction and the tail, we will determine the head of the next diagonal. Initially the direction would be `1` which would indicate `up`. We will keep alternating this value from one iteration to the next.
2. Assuming we know the head of a diagonal, say $matrix[i][j]$, we will use the direction to progress along the diagonal and process its elements.
   - For an upwards going diagonal, the next element in the diagonal would be $matrix[i-1][j+1]$
   - For a downwards going diagonal, the next element would be $matrix[i+1][j-1]$.
3. We keep processing the elements of the current diagonal until we go out of the boundaries of the matrix.
4. Now, given that we know the tail of the diagonal (the last node before we went out of bounds), let's see how we can find the next head. Note that in the following pseudocode, the `direction` is for the current diagonal and we are trying to find the head of the next diagonal. So, if the direction is `up`, it means the next diagonal would be going down and vice-versa.

```
tail = [i, j]
if direction == up, then {
   if [i, j + 1] is within bounds, then {
       next_head = [i, j + 1]
   } else {
       next_head = [i + 1, j]
   }
} else {
   if [i + 1, j] is within bounds, then {
       next_head = [i + 1, j]
   } else {
       next_head = [i, j + 1]
   }
}
```

5. We keep processing the elements of a diagonal and once the current diagonal ends, we use the current direction and the tail element to find the next head and we switch over to processing the next diagonal. Also remember to flip the direction bit.

Java | Python | Copy

```python
class Solution:

    def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:

        # Check for an empty matrix
        if not matrix or not matrix[0]:
            return []

        # The dimensions of the matrix
        N, M = len(matrix), len(matrix[0])

        # Incides that will help us progress through
        # the matrix, one element at a time.
        row, column = 0, 0

        # As explained in the article, this is the variable
        # that helps us keep track of what direction we are
        # processing the current diaonal
        direction = 1

        # Final result array that will contain all the elements
        # of the matrix
        result = []

        # The uber while loop which will help us iterate over all
        # the elements in the array.
        while row < N and column < M:
```

Complexity Analysis

- Time Complexity: $O(N \cdot M)$ since we process each element of the matrix exactly once.
- Space Complexity: $O(1)$ since we don't make use of any additional data structure. Note that the space occupied by the output array doesn't count towards the space complexity since that is a requirement of the problem itself. Space complexity comprises any `additional` space that we may have used to get to build the final array. For the previous solution, it was the intermediate arrays. In this solution, we don't have any additional space apart from a couple of variables.

Previous Next

Comments: 8 Sort By

tpt5cu ★ 140 May 13, 2020 8:50 PM
Sorry to the author but these solutions are way too complex. Check out my solution, super intuitive, no direction checks. I have used it in interviews before with great success.
https://leetcode.com/problems/diagonal-traverse/discuss/581868/Easy-Python-NO-DIRECTION-CHECKING
16 Share Reply
SHOW 6 REPLIES

ordiallo ★ 3 May 12, 2020 4:36 AM
this code sucks to read
3 Share Reply

nphamcs ★ 9 February 29, 2020 1:05 AM Report
We have two important observations:

1. Along any line parallel to the main diagonal, the sum of the coordinates is a constant (and increment by 1 each time we move to the next line)
2. We can use the parity of the sum of coordinates to determine the direction of traversal (going up

Read More
2 Share Reply
SHOW 2 REPLIES

czarny ★ 0 July 4, 2020 6:13 PM
How about using algebraic geometry to solve it :) : https://leetcode.com/explore/learn/card/array-and-string/202/introduction-to-2d-array/1167/discuss/719169/Diagonal-Traversal-with-algebraic-geometry
0 Share Reply

axa190013 ★ 0 July 1, 2020 9:28 PM
int r = d < M ? 0 : d - M + 1;
int c = d < M ? d : M - 1;
When d >M will happen?
0 Share Reply
SHOW 1 REPLY

yifeihu1993 ★ 0 June 10, 2020 10:02 AM
class Solution:
def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:
if matrix==[]:
return([])
m = len(matrix)-1
Read More
0 Share Reply

WilmerKrisp ★ 21 May 21, 2020 8:37 PM

```
class Solution:
    """ ok """

    def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:  # pylint:
```

Read More
0 Share Reply

zeus1985 ★ 63 February 25, 2020 7:06 AM
In solution 1, if you use linked list, clear method will not take O(n) time it is O(1). Good explanation.
0 Share Reply
SHOW 2 REPLIES