Msc ”Advanced Information systems - Development of Software and Artificial Intelligence, University of Piraeus

Longest Increasing Path in matrix - Advanced Python programming

Author: Konstantinos Mitsionis (mpsp2529)

Date: December 9, 2025

Problem

You are given a matrix of integers with dimensions m × n. Your task is to find the length of the longest increasing path within the matrix. Starting from any cell, you can move to adjacent cells in one of four possible directions: left, right, up, or down. Diagonal movement and movement that goes beyond the matrix boundaries (i.e., wrap-around) are not allowed. The path must be strictly increasing, meaning that each subsequent cell in the path must have a higher value than the previous one.

Problem Analysis

From my perspective, this is a dynamic recursive programming problem. The goal of finding the longest increasing path in a given matrix requires solving many subproblems that have the same logic and structure as the initial one: finding the longest increasing path from a specific cell. The result of each subproblem is independent of how the cell is reached, meaning the longest increasing path starting from a cell is identical whether that cell is the initial starting point or an intermediate one in the search process. This independence, combined with the fact that multiple starting paths may revisit the same subproblem (longest path from a given cell) justifies the use of memoization. The goal of our search process is to find a sequence of cells which is increasing step by step.The nature of our problem leads us to the conclusion that the search structure forms a directed acyclic graph (DAG). The logic behind DAG is that search does not create cyclic patterns in the graph, meaning that we cannot include the same cell to the path more than one time which aligns with the condition of the problem for increasing numbers in the path. As a result, DAG structure guarantees that recursion terminates and that every longest-path computation can be performed only once. Consequently, the longest increasing path problem can be solved by using depth-first search combined with memoization, treating the matrix as a DAG and avoiding redundant computations.

> A Directed Acyclic Graph (DAG) is a directed graph that does not contain any cycles.

As a result, every time we examine the longest increasing path from a given cell, we should save the maximum length of increasing path from each starting cell in a matrix to avoid repeated calls of a function for retrieving a value we have already calculated.

Methodology

For solving this recursive problem we have basically two options:

- For every (row,column) data point we can evaluate the longest increasing path separately by re calculating the longest increasing paths each time. Although correct, this approach leads to repeated evaluation of the same subproblems, resulting in exponential time complexity.
- Use dynamic programming with memoization. The problem's recursive nature adapts to dynamic programming. Also, memoization helps us save computational time as we store calculated results, which is connected with the fact that for every cell the longest increasing path is unique in a given matrix.

Listing 1: Python implementation of the longest increasing path algorithm

```python
def find_longest_path(matrix):
    if not matrix:
        return 0

    rows, cols = len(matrix), len(matrix[0])
    directions = [(0,1), (1,0), (0,-1), (-1,0)] # all 4 possible directions

    memo = [[-1] * cols for _ in range(rows)]   #max_len matrix
    next_cell = [[None] * cols for _ in range(rows)]  #best_neighbor matrix


    def dfs(row,col):
        if memo[row][col] != -1:
            return memo[row][col]   #return already computed value
        else:
            next_best = None
            max_length = 1

            for dr,dc in directions: # explore all 4 directions
                new_row, new_col = row + dr, col + dc   #move to a new cell

                if 0 <= new_row < rows and 0 <= new_col < cols and matrix[new_row][new_col] > matrix[row][col] : # check bounds and increasing condition
                    cand = 1 + dfs(new_row,new_col) #recursive call to dfs

                    if cand > max_length:  #we need to find the max length path
                        max_length = cand
                        next_best = (new_row, new_col)

            next_cell[row][col] = next_best
            memo[row][col] = max_length            #after evaluating all possible directions,store max length, best next cell
            return memo[row][col]

    max_path = 0
    start_cell = None

    for r in range(rows):
        for c in range(cols):
            length = dfs(r,c)
            if length > max_path:
                max_path = length
                start_cell = (r,c)

    path = []
    r, c = start_cell #start from best starting cell
    while r is not None and c is not None:
        path.append((r, c))
        next_pos = next_cell[r][c]
        if next_pos is not None:
            r,c = next_pos
        else:
            break

    max_path_sequence = path

    return max_path, max_path_sequence


matrix = [[21,22,23,24,25],
          [20,7,8,9,10],
          [19,6,1,2,11],
          [18,5,4,3,12],
          [17,16,15,14,13]]

print(find_longest_path(matrix))
```

Example

Given a specific matrix let's assume that we want to calculate the maximum increasing path from (0,0):

$$\begin{bmatrix} 3 & 4 & 6 \\ 5 & 9 & 7 \\ 2 & 8 & 1 \end{bmatrix}$$

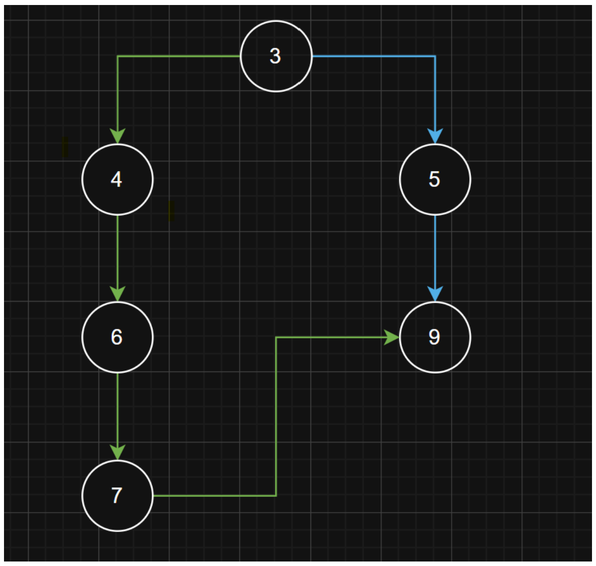

Figure 1: Search pattern from (0,0).

Given that we can move from our starting point (0,0) only in the (0,1) and (1,0), two possible increasing paths are available. Our goal is to calculate the length of each possible increasing path and keep track of only the largest one. In our example 1, the calculated increasing paths are:

$$\text{path1}: \ (0,0) \rightarrow (1,0) \rightarrow (1,1)$$

$$\text{path2}: \ (0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow (1,2)$$

Obviously, the path2 is the longest increasing path and it is the optimal answer whenever we reach the cell (0,0). By running the script 1, we calculate the length of maximum increasing path and return all the cells, which are included in it.

1 Distinction between tree and graph

A tree is a special category of graph. For this problem, it is important to distinguish between the concepts of graphs and trees. Our problem cannot be represented by a tree because a tree requires a single root, exactly $N - 1$ edges for N nodes, and a unique simple path between every pair of nodes. In our case, we do not have a single root, and a node may connect to several neighbors, creating multiple possible paths.

The search for increasing paths also prevents cycles because we cannot return to a previous cell with a smaller value. This makes the structure acyclic. Furthermore, movement between cells has a natural direction (from a lower value to a higher value), which makes the graph directed.

For these reasons, the structure is best described as a Directed Acyclic Graph (DAG) rather than a tree.

Code analysis

For a better understanding, the script 1 will be analyzed.

- **directions**: list of tuples defining the possible moves from a given cell.
- **memo**: a rows$\times$cols matrix that stores the length of the longest increasing path starting from each cell.
- **next_cell**: a rows $\times$ cols matrix that stores the coordinates (x, y) of the next best step for each cell.
- **dfs(row, col)**: recursive function that returns the maximum increasing-path length starting from cell (row, col).
- **max_path_sequence**: list with the longest increasing path in the given matrix.

It is important to highlight that `memo` and `next_cell` are *closure* variables. In Python, closure variables behave like shared state: they are defined in the outer scope of `dfs`, and the inner function is able to read and update them. When `dfs` finishes execution, any modifications made to these variables persist, since they refer to the same underlying objects in memory.

Time complexity analysis

Complexity is expressed in terms of input size. In our case the input size is rows $\times$ cols(N = R$\times$C).

Building `memo` and `next_cell` is related with the dimensions of the input matrix so the time complexity is `O(R * C)`.

For every cell, `dfs` function will run the `else` statement only the first time, when `memo` matrix is still empty. During the first evaluation of a cell, `dfs` may explore up to four directions, in a worst case scenario (cell can move in all directions and all neighboring cells contain larger values). After `memo` is updated, no recursions will be needed. This time complexity is considered `O(1)`, because runs of the function are not correlated with the input size. Even if a step requires a recursion of `dfs` the cost is charged to the next cell, not to whoever called it.

For reconstructing the longest increasing path, the algorithm will use all cells in the matrix, which produces an `O(R * C)` time complexity. As a result, the final time complexity of our algorithm:

$$T(R, C) = O(R \cdot C) \text{ (DFS)} + O(R \cdot C) \text{ (path reconstruction)}.$$

For grid-search problems like this, `O(RC)` is the best complexity value, since we cannot do better than at least reading all input cells.

If we would like to specify the longest increasing paths for each starting cell the complexity would be:

$$T_{\text{total}}(R, C) = \underbrace{O(RC)}_{\text{DFS + memo}} + \underbrace{(RC) \cdot O(RC)}_{\text{path reconstruction from each cell}} = O\big((RC)^2\big)\,.$$

In conclusion, computing the length of the longest increasing path in the matrix requires a time complexity of $O(R \cdot C)$ when using depth-first search with memoization. The overall complexity may vary depending on whether we reconstruct only the longest path from the optimal starting cell, or whether we reconstruct the longest increasing path from *every* cell. In the former case, the complexity remains $O(R \cdot C)$, whereas reconstructing all per-cell paths results in a total complexity of $O\big((R \cdot C)^2\big)$.

Additional Questions

First of all, before approaching the additional questions, the term *random matrix* must be defined. A random matrix is a matrix whose entries are random variables.

> *How can we construct a random matrix with dimensions $m \times n$ where the longest increasing path is guaranteed to follow a given pattern (e.g., a snake-like pattern or a staircase)?*

Using the same recursive logic, we can generate a random number and, based on it, define all the next cells according to the desired pattern. For example, to construct a `snake_pattern` matrix, we can execute the script below:

Listing 2: Python implementation of a snake pattern matrix

```python
import random
def generating_snake_pattern_random_matrix(matrix, start=10):

    if not matrix or not matrix[0]:
        return []

    rows = len(matrix)
    cols = len(matrix[0])

    cell_value = random.randint(0, start)
    next_cell = cell_value

    for r in range(rows):
        if r % 2 == 0:
            for c in range(cols):   # from left to right
                matrix[r][c] = random.randint(next_cell + 1, next_cell + 10) # assign random increasing value
                next_cell = matrix[r][c]    # store last assigned value
        else:
            for c in range(cols - 1, -1, -1):   # from right to left
                matrix[r][c] = random.randint(next_cell + 1, next_cell + 10) # assign random increasing value
                next_cell = matrix[r][c]   # store last assigned value
    return matrix

if __name__ == "__main__":
    rows, cols = 4, 5
    matrix = [[0 for _ in range(cols)] for _ in range(rows)]
    result = generating_snake_pattern_random_matrix(matrix)
    for row in result:
        print(row)
```

Although the distribution of each cell depends on the previous ones (through the `next_cell` value), every entry is sampled from a uniform distribution over its allowed interval. Thus, the matrix is conditionally random, even though the limits of each cell's distribution are correlated due to the snake pattern path.

Second question sets the additional condition of a specific length in the maximum increasing path in the matrix.

> *What algorithm can be used to generate random matrices where the length of the longest increasing path falls within a specific range? Can we enforce such constraints while maintaining randomness in the matrix generation?*

In this case, we have a structural constraint for our matrix which is that its longest increasing path must fall within a specific range of values. As a result, we cannot sample cell values from a random distribution but we need to sample from a restricted set of matrices. Imposing a constraint on the longest increasing path means the entries of the matrix can no longer be sampled independently, since independent sampling would almost never satisfy the constraint. Therefore, the matrix must be generated from a conditional distribution restricted to matrices whose longest increasing path lies in the desired range. For a problem like this, two methods can be implemented:

- **Rejection sampling**: We can create a random matrix from a uniform distribution. Then, we can calculate the longest increasing path. If the path follows within a specific range we accept it, otherwise we recreate a random matrix and proceed to evaluation of longest path again.
- **Markov Chain Monte Carlo (MCMC)**: We can create a matrix with the desired length of longest increasing path by following certain structure methods as in 2. After creating the matrix we can visit random cells and propose small changes to them. If the new matrix longest increasing path still falls into the specific range, we keep the changes, otherwise we discard it.

With rejection sampling we can easily build a matrix whose samples are independent and the longest increasing path falls within the desired range. On the other hand, such an algorithm can be a heavy computational process if for example the desired range is very tight or rare (many rejections). In such cases, the algorithm is very difficult to operate successfully.

With Markov Chain Monte Carlo algorithm, we can introduce randomness into a matrix that already satisfies the desired longest increasing path constraint, while ensuring that all subsequent modifications keep the path within the target range. In MCMC, samples are correlated because each new matrix is created by a small modification of the previous one. However, as the number of iterations increases, calculated matrices become increasingly random while still respecting the longest increasing path condition.

In summary, both rejection sampling and MCMC allow us to generate random matrices under LIP constraints, with rejection sampling being simpler but inefficient for rare constraints, and MCMC being more flexible for sampling from complex distributions.

Third question is similar with the second one, but now the length of the longest increasing path must be a specific value.

> *Can we create random matrices where the longest increasing path is predetermined (e.g., of length k)? How would the matrix values and structure be generated in such a case to ensure the existence of the desired longest increasing path?*

For this problem we can create an acyclic directed graph with the desired length and place in each node increasing numbers. After formatting the desired path, we can fill in the remaining cells with random values which do not affect the length of longest path. We can use random.randint(start,end) with a small number as upper limit in order to generate the remaining numbers/cells. This will not affect the longest path which must contain only increasing numbers. On the other hand, the same sampling methods used in the previous question, can also be applied. Rejection sampling can be used repeatedly generating fully random matrices and accepting only those whose longest increasing path is exactly k. If our task is to find a rare type of matrix, this can be a heavy computational process. Alternatively, an MCMC approach can be implemented: we begin from any matrix that already satisfies the desired length of longest increasing path and then iteratively propose small random modifications to individual cells. As mentioned before, over many iterations this process introduces randomness while ensuring the constraint is always maintained.

> How can we generate a random matrix that adheres to specific rules for the longest increasing path (e.g., the path must touch every row/column at least once, or the path must stay within a specific sub-region of the matrix)?

To enforce structural rules on the longest increasing path, we can rely on the same general sampling strategies used in the previous questions, without repeating their full descriptions. In practice, the constraint can be satisfied either by (i) sampling random matrices until the required path structure appears, (ii) starting from a matrix that already contains a path with the desired geometric properties and applying an MCMC process to introduce randomness while preserving the constraint, or (iii) constructing the required path explicitly and assigning smaller or restricted values to the remaining cells to prevent longer competing paths. These approaches allow us to generate random matrices that respect any prescribed path geometry, such as touching every row or staying inside a specific sub-region, while still retaining meaningful randomness in their overall value distribution.

> What statistical properties (e.g., distribution of values, frequency of repeating numbers) should we impose on a random matrix to increase the likelihood of having a longest increasing path of a certain length? How can we balance randomness with constraints on the path length?

If we reserve a wide range of values for the potential path cells of a certain length and a compressed range for all other cells, then cells on the path are more likely to form a longest increasing path of a certain length, while cells off the path are less likely to be contributors to a competing increasing chain. A simple model can be defined as follows:

$$X_{\text{path}} \sim \text{Uniform}(A, B) \quad \text{with } B - A \text{ large,}$$

$$X_{\text{rest}} \sim \text{Uniform}(0, C), \qquad C < A.$$

Frequency of repeating numbers can have a correlation with the length of the longest increasing path. If a matrix has many equal values, there is a bigger probability of having fewer strictly increasing paths.

To balance randomness with control over longest increasing path length, we can:

- choose different distributions for the cells that are on the intended path region and for those elsewhere.
- start from a prebuilt matrix version, apply MCMC iteratively and introduce randomness to the matrix.

  Can we construct a matrix where multiple longest increasing paths exist? How can we design such a matrix, and what are the implications for the matrix structure and value distribution?

We can construct a matrix where multiple longest increasing paths exist. We first define the geometry of the longest increasing paths and then assign values to all cells that belong to the union of these paths. We initialize the first cell with a value U_0 and then assign strictly increasing values to the remaining cells on each path, ensuring that the order constraints of all intended paths are respected (i.e., the values along any of the paths form a strictly increasing sequence).

Formally, for consecutive cells on the path we can define

$$U_{\text{next}} = U_{\text{previous}} + \Delta,$$

where $\Delta > 0$ is a positive chosen step (possibly sampled from a distribution).

To ensure that no other region of the matrix forms a longer increasing path, the remaining (off-path) cells can be sampled from a lower distribution. For example, if path cells are drawn from a range $[A, B]$, the off-path cells may be sampled from

$$U_{\text{off}} \sim \text{Uniform}(0, A),$$

so that they do not interfere with or extend the intended longest increasing paths.

When multiple longest increasing paths exist, cells belonging to these paths become strongly correlated, and some cells may even be shared across different paths. In general, the path cells tend to have higher means and larger variances, while the off-path cells have lower means and smaller variances. The presence of multiple longest paths can also create flatness in the off-path regions, since these cells must not form competing increasing sequences.