Our goal is finding matrix inversion via LU-factorization. There are some prerequisites for such method

- First of all the matrix, A should be square to use LU-factorization.
- The matrix A should be invertible, otherwise, this method will fail because there is no unique solution
- The method we use also require implementation of the pivoting scheme to avoid division by zero. To do this we use permutation matrix.

So, first of all, we transform our matrix if needed. Starting from the first row we permute it with the row with the largest number in the first cell. We do it for all I rows and I cell. We generate permutation matrix to know how to transform this matrix by multiplication. A permutation matrix is a matrix with only one 1 in row and column and 0 in all other cells. It is not hard to find this permutation matrix as we just replace some rows.

After this preparation, we use Doolittle method for LU-factorization. We compute L and U matrices by simple arithmetical operations with A matrix (transformed if needed, to compute the inverse matrix we need to remember permutation matrix).

Now we have L and U matrices. We need to solve **|LU*A(-1)=Identity|** to find A(-1). The thing is that we can divide it into the system of equations with columns of A(-1) and Identity as the vectors.

L*dj = Ij (where j is number of column and d is a vector)
U*xj = dj (where xj is j column of inversed matrix)

After solving this system (We find L and U inversions and have an equation in the form (U(-1)*L(-1))) we have A(-1) if we have permutation matrix which is not equal to identity matrix, we should change it by multiplying it by permutation matrix we found.

Computational complexity is just N^2 (where N is a number of cells in the row or column, we work only with square matrices). It is just impossible to make it lot faster as for generating LU matrices you should save N^2 elements. So the complexity of our method is optimal. It is true if we do not count complexity of numpy multiplication of matrices. If the multiplication in numpy is the most effective than we can say that our algorithm works similar to O(n^2.367) algorithm based on Coppersmith–Winograd algorithm.

Our method is almost the same as Crout decomposition, but there is a minor difference in LU computation. In this method, we get L(DU) and in Crout factorization, we get (LD)U (where D is a diagonal matrix). The nice peculiarity of our method is computing L and U matrices which can be used for other purposes. We do not just compute inverted matrix but also get additional information which can be useful.