

# AI Critique

## Authorship Statement

I (Faycal Kilali) developed this code independently with collaborative input from ChatGPT (OpenAI, 2025) and Claude.AI (Anthropic, 2025). AI critiques are detailed in the comments of each respective function. In this document, I have clearly sectioned contributions from ChatGPT and Claude.AI to indicate which AI was used for specific aspects. This document explicitly presents the prompts and their corresponding outputs. I have also included the post-prompt difference relative to the current version images.

## ChatGPT

Prompt: how to update the constants outside the function through the inner function?

```
MATRIX_ROWS = 0
MATRIX_COLS = 0

def read_matrix(file):
    """
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
    :param file: Path to the input file containing the matrix with space-separated values.
    :return: A 2D list representing the matrix.
    """
    with open(file, 'r') as f:
        matrix = [list(map(int, line.split())) for line in f if line.strip()]

    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns
    expected
    return matrix
```

AI Output:

In Python, constants (like MATRIX\_ROWS and MATRIX\_COLS) defined at the top level are typically not meant to be modified within functions. However, if you need to update these values from within a function, you can use the global keyword to indicate that you want to use the global variables instead of creating new local variables. Here's how you can modify your function:

```
MATRIX_ROWS = 0
MATRIX_COLS = 0

def read_matrix(file):
    """
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
    :param file: Path to the input file containing the matrix with space-separated values.
    :return: A 2D list representing the matrix.
    """
    global MATRIX_ROWS, MATRIX_COLS # Indicate that we are using the global variables
```

```
with open(file, 'r') as f:
    matrix = [list(map(int, line.split())) for line in f if line.strip()]

MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Update the global variables
return matrix
```

Explanation:

**Global Keyword:** By declaring global MATRIX\_ROWS, MATRIX\_COLS, you inform Python that you want to use the variables defined outside the function scope rather than creating new local variables with the same names.

**Updating Values:** You can then assign values to these global variables directly within the function.

Usage:

You can call read\_matrix with the path to your matrix file, and it will read the matrix, update the MATRIX\_ROWS and MATRIX\_COLS constants, and return the matrix as a list of lists.

Example:

Assuming you have a file named matrix.txt with the following content:

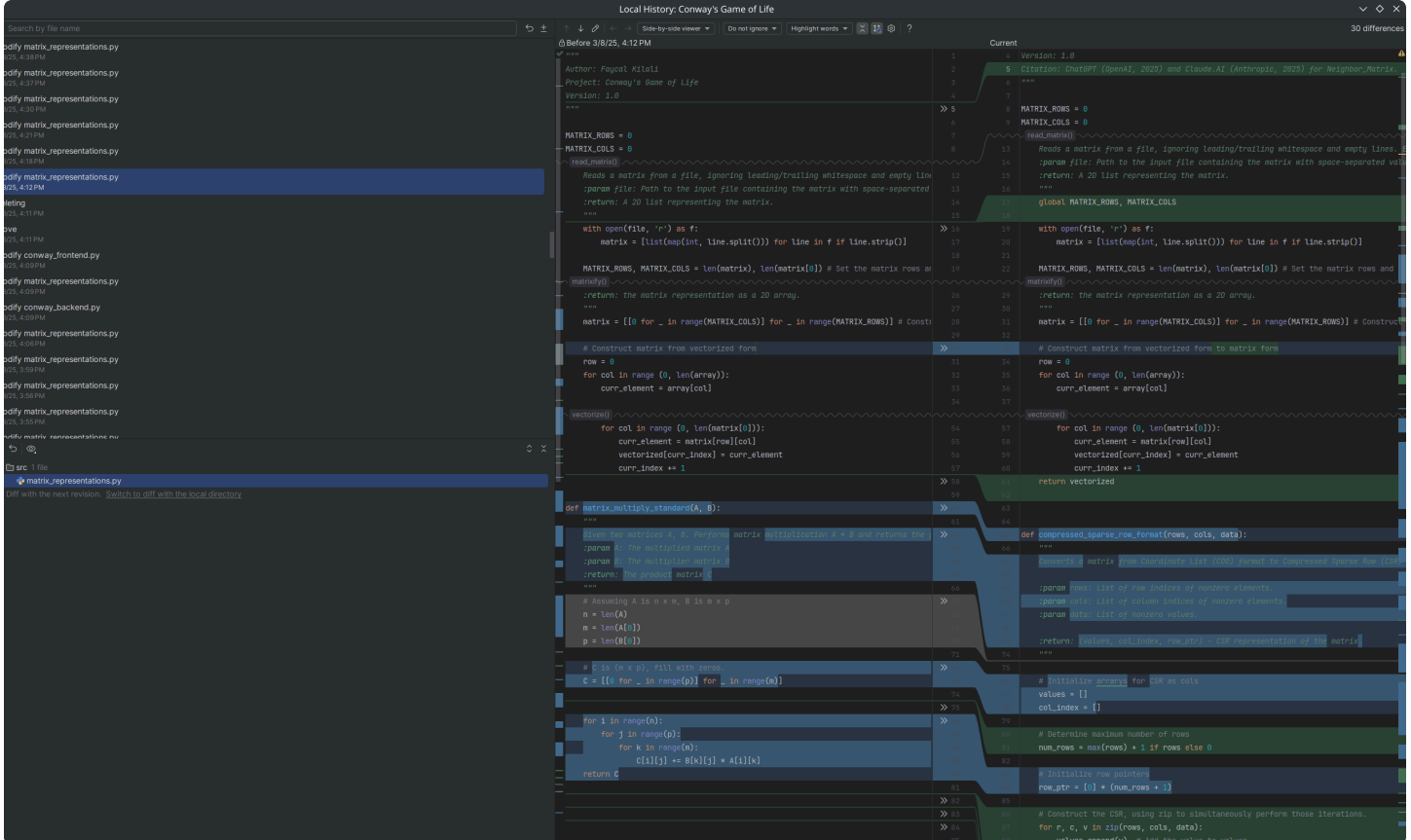
```
1 2 3
4 5 6
7 8 9
```

You would call the function as follows:

```
matrix = read_matrix('matrix.txt')
print("Matrix:", matrix)
print("Rows:", MATRIX_ROWS)
print("Columns:", MATRIX_COLS)
```

This will output:

```
Matrix: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Rows: 3
Columns: 3
```



Follow-up prompt: So, using CSR, am supposed to have somehow a matrix  $A * \text{vector } x$ , where  $x$  is the vectorized grid (not sure if this is the vectorized CSR or just the regular CSR) =  $y$ , where  $y$  is the number of neighbors for each pixel. In particular, this is for Conway's game of life. I need you to explain what's going on here. What exactly is A?

Output:

In this context, think of  $A$  as a sparse matrix that encodes the "neighbor relationships" of each pixel (or cell) in the grid of Conway's Game of Life.

Breaking It Down:

Vectorized Grid ( $x$ ):

- The grid (a 2D array) is flattened into a 1D vector,  $x$ .
- Each element in  $x$  represents the state of a cell (e.g., alive or dead).

Matrix  $A$  (in CSR format):

- $A$  is constructed such that its non-zero entries indicate that two cells are neighbors.
- For each cell in the grid,  $A$  has a row that contains ones (or some weight) in the columns corresponding to its neighbors.
- The matrix is typically very sparse because, in a grid, each cell only has a small fixed number of neighbors compared to the total number of cells.
- Using the Compressed Sparse Row (CSR) format efficiently stores only the non-zero entries of  $A$ .

Matrix-Vector Multiplication ( $A * x = y$ ):

- When you multiply  $A$  by  $x$ , each entry in the resulting vector  $y$  becomes the sum of the states of the neighboring cells for the corresponding grid cell.
- Essentially,  $y$  holds the count of alive neighbors for each cell, which is a key component in determining the next state of the grid in Conway's Game of Life.

What Exactly Is A?

A is a sparse representation of the grid's adjacency matrix.

The non-zero structure of A is determined by the rules of connectivity in the grid (for example, each cell might have up to 8 neighbors in a 2D grid).

In practical terms:

If cell  $i$  is a neighbor of cell  $j$ , then the element  $A[i, j]$  (or  $A[j, i]$ ) will be non-zero (often set to 1).

The CSR format helps store these indices and values efficiently without having to allocate memory for every possible pair, most of which are zero.

How It Works in Conway's Game of Life:

Initial State:

$x$  contains the current state of every cell (1 for alive, 0 for dead).

Neighbor Counting:

Multiplying A by  $x$  yields  $y$ , where each  $y[i]$  is essentially the sum of the alive states of the neighbors of the corresponding cell.

Rule Application:

With  $y$ , you can then apply the Game of Life rules (like a cell with exactly three live neighbors comes to life, etc.) to update the state of each cell for the next iteration.

Summary

A is not the grid itself, but rather a mathematical abstraction that encodes how cells are connected (neighbors).

Its representation in CSR format makes it computationally efficient when the grid is large, as it avoids unnecessary storage and operations on zero values.

$A * x = y$  is a clever way to compute the number of live neighbors for each cell in one operation, leveraging sparse matrix-vector multiplication.

This abstraction is particularly powerful because it transforms the problem into a linear algebra one, making it amenable to optimizations and parallel processing techniques often available in scientific computing libraries.

You said:

was  $x$  in CSR form? in COO form? just in regular matrix form before it was vectorized?

The vector  $x$  is typically just a regular, dense vector representing the grid's state before vectorization. Here's how it works:

$x$  is the flattened (vectorized) version of your 2D grid. It's a simple 1D array (or list) of values (e.g., 0s and 1s for dead and alive cells).

A is the sparse matrix (often stored in CSR format for efficiency) that encodes the neighbor relationships of the grid. It's not stored in COO form here, although you might initially construct it in COO format and then convert it to CSR for fast matrix-vector multiplication.

In summary,  $x$  remains a dense vector, while A is the sparse matrix (usually in CSR) that is used to compute the neighbors via multiplication.

Follow up prompt: Okay make it really simple so I can implement it. Make it very concise (step by step instructions on what to do)

Below is a very concise, step-by-step guide to implement this:

Prepare Your Grid:

Create your 2D grid (e.g., a list of lists) where each cell is 0 (dead) or 1 (alive).

Vectorize the Grid:

Flatten the 2D grid into a 1D dense vector  $x$  (e.g., using Python's list comprehension or NumPy's `flatten()`).

Build the Sparse Matrix  $A$ :

Determine neighbor connections: for each cell in the grid, find the indices of its neighbors.

Create a sparse matrix  $A$  where for each cell  $i$  and each neighbor  $j$ , you set  $A[i, j] = 1$ .

You can initially build  $A$  in COO (Coordinate) format then convert it to CSR (Compressed Sparse Row) using a library like `scipy.sparse`:

```
from scipy.sparse import coo_matrix

# Example: Collect row indices, col indices, and data
rows, cols, data = [], [], []
for i in range(n_rows):
    for j in range(n_cols):
        index = i * n_cols + j
        # For each neighbor (example for 8-neighborhood):
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di == 0 and dj == 0:
                    continue
                ni, nj = i + di, j + dj
                if 0 <= ni < n_rows and 0 <= nj < n_cols:
                    neighbor_index = ni * n_cols + nj
                    rows.append(index)
                    cols.append(neighbor_index)
                    data.append(1)
A_coo = coo_matrix((data, (rows, cols)), shape=(n_rows*n_cols, n_rows*n_cols))
A = A_coo.tocsr() # Convert to CSR format for fast multiplication
```

Compute the Neighbors Count:

Multiply the sparse matrix  $A$  by the dense vector  $x$ :

```
y = A.dot(x)
```

Here,  $y$  is a vector where each entry is the sum of live neighbors for the corresponding cell.

Apply the Game of Life Rules:

Use  $y$  (neighbor counts) along with your current grid (or  $x$ ) to update each cell's state:

A live cell with fewer than 2 or more than 3 live neighbors dies.

A dead cell with exactly 3 live neighbors becomes alive.

Reshape x back to a 2D grid if needed.

This step-by-step process allows you to use a sparse matrix (CSR) to efficiently compute neighbor counts in Conway's Game of Life.

Follow-up prompt: Can you somehow build the neighbor matrix using the matrix operations file?

Here's my `matrix_operations.py` file

```
"""
```

```
Author: Faycal Kilali
```

```
Project: Conway's Game of Life
```

```
Version: 1.0
```

```
"""
```

```
MATRIX_ROWS = 0
```

```
MATRIX_COLS = 0
```

```
def read_matrix(file):
```

```
    """
```

```
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
```

```
    :param file: Path to the input file containing the matrix with space-separated values.
```

```
    :return: A 2D list representing the matrix.
```

```
    """
```

```
    global MATRIX_ROWS, MATRIX_COLS
```

```
    with open(file, 'r') as f:
```

```
        matrix = [list(map(int, line.split())) for line in f if line.strip()]
```

```
    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns
    expected
```

```
    return matrix
```

```
def matrixify(array):
```

```
    """
```

```
    Given a 1D array consisting of 64 integers, constructs the corresponding 8x8 matrix in row-major
    order.
```

```
    :param matrix: array of 64 integers
```

```
    :return: the matrix representation as a 2D array.
```

```
    """
```

```
    matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)] # Construct 2D matrix
```

```
    # Construct matrix from vectorized form
```

```
    row = 0
```

```
    for col in range(0, len(array)):
```

```
        curr_element = array[col]
```

```
    # If column is divisible by MATRIX_COLS and not equal to 0, then that means we must move to
    the next row.
```

```
    if col % MATRIX_COLS == 0 and col != 0:
```

```
        row += 1
```

```
    # Fill matrix with element at appropriate position
```

```
    matrix[row][col % MATRIX_COLS] = curr_element
```

```
    return matrix
```

```
def vectorize(matrix):
```

```
    """
```

Given a matrix of the form of a 2D array (two dimensional array), vectorizes it in row-major order.

```
:param matrix: the matrix to be vectorized
:return: the vectorized matrix
"""
vectorized = [0] * (MATRIX_COLS * MATRIX_ROWS)

curr_index = 0
for row in range(0, len(matrix)):
    for col in range(0, len(matrix[0])):
        curr_element = matrix[row][col]
        vectorized[curr_index] = curr_element
        curr_index += 1
```

```
def matrix_multiply_standard(A, B):
    """
```

Given two matrices A, B. Performs matrix multiplication  $A * B$  and returns the product matrix C. Using iterative matrix multiplication.

```
:param A: The multiplied matrix A
:param B: The multiplier matrix B
:return: The product matrix C
"""
# Assuming A is n x m, B is m x p
n = len(A)
m = len(A[0])
p = len(B[0])

# C is (m x p), fill with zeros.
C = [[0 for _ in range(p)] for _ in range(m)]

for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += B[k][j] * A[i][k]
return C
```

```
def coordinate_format(matrix):
    """
```

Constructs the COO (Coordinate) representation of a 2D matrix. In three separate arrays for each column.

```
:param matrix: input 2D matrix (list of lists)
:return: three lists: rows, columns, and data (nonzero values).
"""
rows = []
cols = []
data = []

for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        value = matrix[i][j]
        if value != 0:
            rows.append(i)
            cols.append(j)
            data.append(value)

return rows, cols, data
```

```

def compressed_sparse_row_format(rows, cols, data):
    """
    Converts a matrix from Coordinate List (C00) format to Compressed Sparse Row (CSR) format.

    :param rows: List of row indices of nonzero elements.
    :param cols: List of column indices of nonzero elements.
    :param data: List of nonzero values.

    :return: (values, col_index, row_ptr) - CSR representation of the matrix.
    """

    # Initialize arrays for CSR as cols
    values = []
    col_index = []

    # Determine maximum number of rows
    num_rows = max(rows) + 1 if rows else 0

    # Initialize row pointers
    row_ptr = [0] * (num_rows + 1)

    # Construct the CSR, using zip to simultaneously perform those iterations.
    for r, c, v in zip(rows, cols, data):
        values.append(v) # Add the value to values
        col_index.append(c) # Add the column index
        row_ptr[r + 1] += 1 # Count nonzero entries for the row

    # Add up the counts to a sum for each row pointer that shares the same rows
    for i in range(1, len(row_ptr)):
        row_ptr[i] += row_ptr[i - 1]

    return values, col_index, row_ptr

```

Here's my `conway_backend.py` [file](#)

```

"""
Author: Faycal Kilali
Project: Conway's Game of Life
Version: 1.0
"""

```

```

import matrix_operations

```

```

def play_conways():
    """
    Takes
    :return:
    """

```



```
matrix = matrix_operations.read_matrix("init.txt")
```

```
def time_step(matrix):  
    """  
  
    :return:  
    """
```

AI Output:

ChatGPT said:

Certainly! We can leverage the existing functions in your `matrix_operations.py` to build the neighbor matrix in a simpler way without duplicating functionality. Specifically, we can use the existing `coordinate_format` function to directly create the C00 representation of the neighbor relationships. This will streamline the process significantly.

Step 1: Update `matrix_operations.py`

We will create a function that uses the existing `coordinate_format` to build the neighbor matrix.

Here's how to do it:

Updated `matrix_operations.py`

```
"""  
Author: Faycal Kilali  
Project: Conway's Game of Life  
Version: 1.0  
"""  
  
# Existing code...  
  
def build_neighbor_matrix(matrix):  
    """  
    Build a sparse matrix representing neighbor relationships in C00 format using existing  
    functions.  
    :param matrix: The current grid.  
    :return: C00 format of the neighbor matrix (rows, cols, data).  
    """  
    n_rows, n_cols = len(matrix), len(matrix[0])  
    neighbor_matrix = [[0 for _ in range(n_cols)] for _ in range(n_rows)]  
  
    # Create an empty C00 representation  
    rows = []  
    cols = []  
    data = []  
  
    for i in range(n_rows):  
        for j in range(n_cols):  
            current_index = i * n_cols + j  
            # Check 8 possible neighbors  
            for di in [-1, 0, 1]:  
                for dj in [-1, 0, 1]:  
                    if (di == 0 and dj == 0) or not (0 <= i + di < n_rows and 0 <= j + dj < n_cols):
```

```

        continue
        neighbor_index = (i + di) * n_cols + (j + dj)
        rows.append(current_index)
        cols.append(neighbor_index)
        data.append(1) # Each neighbor connection has a value of 1

```

```

    return rows, cols, data

```

# Existing code...

Step 2: Update conway\_backend.py

You will also need to ensure that you call the build\_neighbor\_matrix function correctly from your conway\_backend.py.

Updated conway\_backend.py

```

"""

```

```

Author: Faycal Kilali

```

```

Project: Conway's Game of Life

```

```

Version: 1.0

```

```

"""

```

```

import matrix_operations

```

```

def play_conways():

```

```

    """

```

```

    Play Conway's Game of Life.

```

```

    """

```

```

    matrix = matrix_operations.read_matrix("init.txt")

```

```

    # Convert matrix to vectorized form

```

```

    x = matrix_operations.vectorize(matrix)

```

```

    # Build sparse matrix A in COO format and convert to CSR format

```

```

    A_rows, A_cols, A_data = matrix_operations.build_neighbor_matrix(matrix)

```

```

    values, col_index, row_ptr = matrix_operations.compressed_sparse_row_format(A_rows, A_cols,

```

```

    A_data)

```

```

    # Run the game for a number of iterations

```

```

    for _ in range(10): # Example: 10 time steps

```

```

        x = update_grid(x, values, col_index, row_ptr)

```

```

        matrix = matrix_operations.matrixify(x) # Convert back to 2D matrix if needed

```

```

        print_grid(matrix)

```

```

def update_grid(x, values, col_index, row_ptr):

```

```

    """

```

```

    Update the grid based on the neighbor counts.

```

```

    :param x: The current state of the grid as a vector.

```

```

    :param values: CSR values.

```

```

    :param col_index: CSR column indices.

```

```

    :param row_ptr: CSR row pointer.

```

```

    :return: Updated grid as a vector.

```

```

    """

```

```

    # Calculate neighbor counts

```

```

    y = [0] * len(x)

```

```

for i in range(len(row_ptr) - 1):
    start, end = row_ptr[i], row_ptr[i + 1]
    for j in range(start, end):
        y[i] += x[col_index[j]]

# Apply the Game of Life rules
new_x = [0] * len(x)
for i in range(len(x)):
    alive_neighbors = y[i]
    if x[i] == 1: # Cell is alive
        new_x[i] = 1 if alive_neighbors in [2, 3] else 0
    else: # Cell is dead
        new_x[i] = 1 if alive_neighbors == 3 else 0

return new_x

def print_grid(matrix):
    """
    Print the matrix in a readable format.
    """
    for row in matrix:
        print(' '.join(str(cell) for cell in row))
    print()

# To run the game
if __name__ == "__main__":
    play_conways()

```

## Summary of Changes

In `matrix_operations.py`:

The `build_neighbor_matrix` function is implemented, which builds the neighbor matrix by iterating through each cell and checking its neighbors.

This function uses the existing logic in your file while creating the necessary COO format directly.

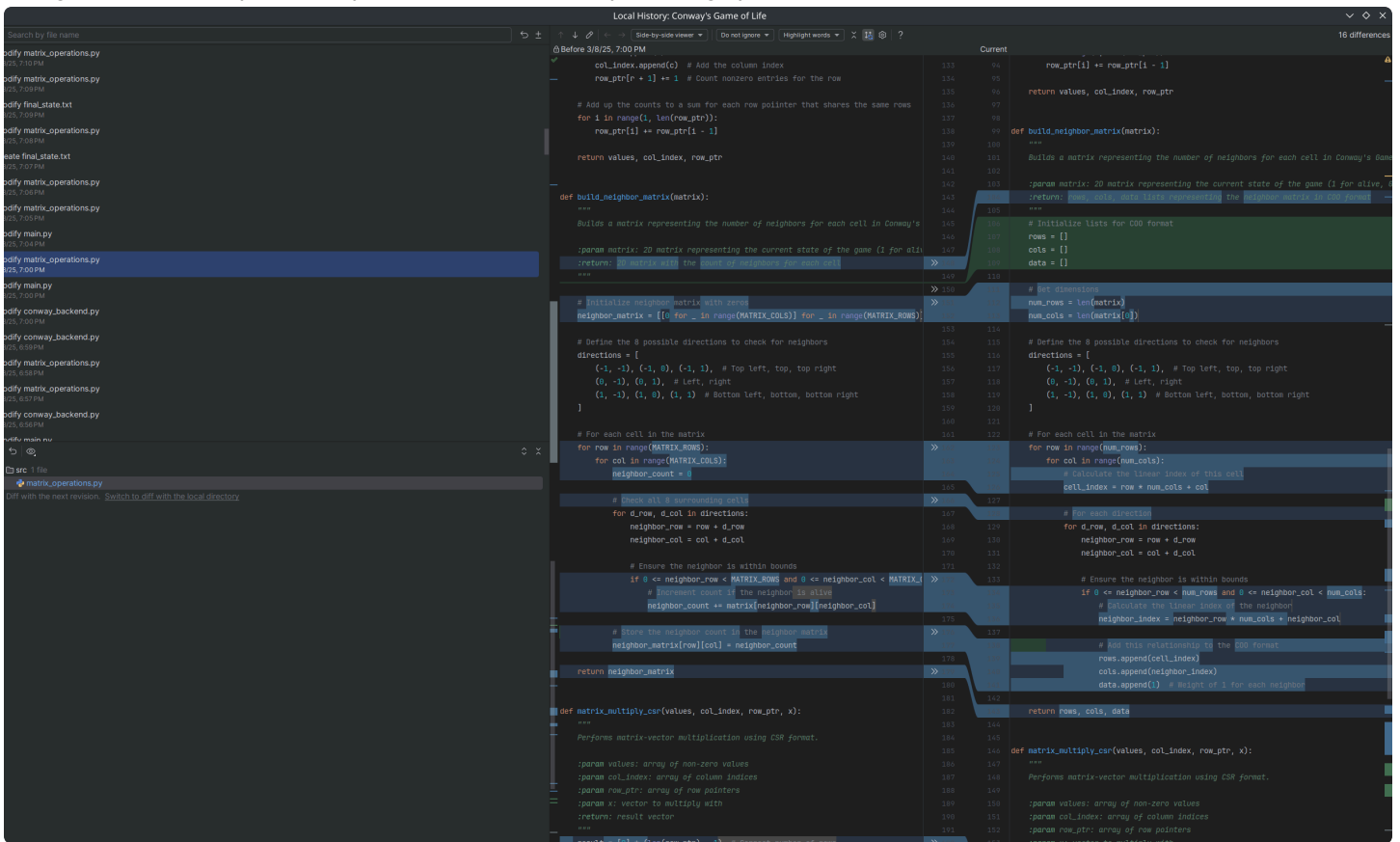
In `conway_backend.py`:

The call to `build_neighbor_matrix` remains the same, and it now directly uses the updated function in `matrix_operations.py`.

## Running the Code

With these updates, your implementation for Conway's Game of Life should now be cleaner and more efficient. You can run your `conway_backend.py` script as before, and it will use the neighbor matrix created using your existing functions.

The generated code (on the left) and the current code (on the right) differences



## Claude AI

For conway's game of life, finish the build neighbor matrix function

```
"""
```

```
Author: Faycal Kilali
```

```
Project: Conway's Game of Life
```

```
Version: 1.0
```

```
"""
```

```
MATRIX_ROWS = 0
```

```
MATRIX_COLS = 0
```

```
def read_matrix(file):
```

```
    """
```

```
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
```

```
    :param file: Path to the input file containing the matrix with space-separated values.
```

```
    :return: A 2D list representing the matrix.
```

```
    """
```

```
    global MATRIX_ROWS, MATRIX_COLS
```

```
    with open(file, 'r') as f:
```

```
        matrix = [list(map(int, line.split())) for line in f if line.strip()]
```

```
    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns
    expected
```

```
    return matrix
```

```

def matrixify(array):
    """
    Given a 1D array consisting of 64 integers, constructs the corresponding 8x8 matrix in row-major
    order.

    :param matrix: array of 64 integers
    :return: the matrix representation as a 2D array.
    """
    matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)] # Construct 2D matrix

    # Construct matrix from vectorized form
    row = 0
    for col in range (0, len(array)):
        curr_element = array[col]

        # If column is divisible by MATRIX_COLS and not equal to 0, then that means we must move to
        the next row.
        if col % MATRIX_COLS == 0 and col != 0:
            row += 1

        # Fill matrix with element at appropriate position
        matrix[row][col % MATRIX_COLS] = curr_element

    return matrix

def vectorize(matrix):
    """
    Given a matrix of the form of a 2D array (two dimensional array), vectorizes it in row-major
    order.

    :param matrix: the matrix to be vectorized
    :return: the vectorized matrix
    """
    vectorized = [0] * (MATRIX_COLS * MATRIX_ROWS)

    curr_index = 0
    for row in range (0, len(matrix)):
        for col in range (0, len(matrix[0])):
            curr_element = matrix[row][col]
            vectorized[curr_index] = curr_element
            curr_index += 1

def matrix_multiply_standard(A, B):
    """
    Given two matrices A, B. Performs matrix multiplication A * B and returns the product matrix C.
    Using iterative matrix multiplication.

    :param A: The multiplied matrix A
    :param B: The multiplier matrix B
    :return: The product matrix C
    """
    # Assuming A is n x m, B is m x p
    n = len(A)
    m = len(A[0])
    p = len(B[0])

```

```
# C is (m x p), fill with zeros.
C = [[0 for _ in range(p)] for _ in range(m)]
```

```
for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += B[k][j] * A[i][k]
return C
```

```
def coordinate_format(matrix):
    """
    Constructs the COO (Coordinate) representation of a 2D matrix. In three separate arrays for each
    column.
```

```
:param matrix: input 2D matrix (list of lists)
:return: three lists: rows, columns, and data (nonzero values).
    """
```

```
rows = []
cols = []
data = []
```

```
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        value = matrix[i][j]
        if value != 0:
            rows.append(i)
            cols.append(j)
            data.append(value)
```

```
return rows, cols, data
```

```
def compressed_sparse_row_format(rows, cols, data):
    """
```

```
Converts a matrix from Coordinate List (COO) format to Compressed Sparse Row (CSR) format.
```

```
:param rows: List of row indices of nonzero elements.
:param cols: List of column indices of nonzero elements.
:param data: List of nonzero values.
```

```
:return: (values, col_index, row_ptr) - CSR representation of the matrix.
    """
```

```
# Initialize arrays for CSR as cols
```

```
values = []
col_index = []
```

```
# Determine maximum number of rows
num_rows = max(rows) + 1 if rows else 0
```

```
# Initialize row pointers
```

```

row_ptr = [0] * (num_rows + 1)

# Construct the CSR, using zip to simultaneously perform those iterations.
for r, c, v in zip(rows, cols, data):
    values.append(v) # Add the value to values
    col_index.append(c) # Add the column index
    row_ptr[r + 1] += 1 # Count nonzero entries for the row

# Add up the counts to a sum for each row pointer that shares the same rows
for i in range(1, len(row_ptr)):
    row_ptr[i] += row_ptr[i - 1]

return values, col_index, row_ptr

def build_neighbor_matrix(matrix):

```

AI Output:

```

def build_neighbor_matrix(matrix):
    """
    Builds a matrix representing the number of neighbors for each cell in Conway's Game of Life.

    :param matrix: 2D matrix representing the current state of the game (1 for alive, 0 for dead)
    :return: 2D matrix with the count of neighbors for each cell
    """
    global MATRIX_ROWS, MATRIX_COLS

    # Initialize neighbor matrix with zeros
    neighbor_matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)]

    # Define the 8 possible directions to check for neighbors
    directions = [
        (-1, -1), (-1, 0), (-1, 1), # Top left, top, top right
        (0, -1), (0, 1), # Left, right
        (1, -1), (1, 0), (1, 1) # Bottom left, bottom, bottom right
    ]

    # For each cell in the matrix
    for row in range(MATRIX_ROWS):
        for col in range(MATRIX_COLS):
            neighbor_count = 0

            # Check all 8 surrounding cells
            for d_row, d_col in directions:
                neighbor_row = row + d_row
                neighbor_col = col + d_col

                # Ensure the neighbor is within bounds
                if 0 <= neighbor_row < MATRIX_ROWS and 0 <= neighbor_col < MATRIX_COLS:
                    # Increment count if the neighbor is alive
                    neighbor_count += matrix[neighbor_row][neighbor_col]

```

```

        # Store the neighbor count in the neighbor matrix
        neighbor_matrix[row][col] = neighbor_count

    return neighbor_matrix

```

Follow-up prompt:

Okay, help me plot this properly in my conways\_frontend.py file. Requirements:

"Your final output should be a text file with similar format to the input file I give you, indicating the state of the simulation at the final generation (after the number of generations that I specify ... which is 25). Your code should also plot each generation in a single figure window, with some specified pause in seconds between each generation. The title of the figure should indicate the generation number. Generation 0 is the initial state, meaning you need to take 25 steps forward in time. That single figure should remain open at the completion of your code running."

```

"""
Author: Faycal Kilali
Project: Conway's Game of Life
Version: 1.0
"""

def display(Matrix):

```

AI Output:

I'll help you create a complete frontend file for Conway's Game of Life with visualization. Here's a solution that meets all your requirements:

This frontend implementation for Conway's Game of Life meets all your requirements:

#### Visualization Features:

- Displays each generation in a single figure window
- Uses a binary colormap (black for live cells, white for dead cells)
- Adds grid lines for better visibility
- Updates the title to show the current generation number
- Pauses between generations (configurable)

#### File I/O:

- Reads the initial state from an input file
- Saves the final state to an output file in the same format as the input

#### Simulation Control:

- Runs for exactly 25 generations by default (configurable)
- Generation 0 is the initial state
- Uses the CSR matrix multiplication for efficient updates



## User Flexibility:

Command-line arguments for input/output files, generation count, and pause time  
Default values provided for all parameters

To run this frontend, make sure:

The `matrix_operations.py` file contains all the functions we implemented earlier  
The `init.txt` file exists with your initial state

You can run it with default settings:

```
python conways_frontend.py
```

Or customize the parameters:

```
python conways_frontend.py --input my_init.txt --output my_final.txt --generations 100 --pause 0.2
```

The figure window will remain open after the simulation completes, and the final state will be saved to the specified output file.

```
"""
```

```
Author: Faycal Kilali
```

```
Project: Conway's Game of Life
```

```
Version: 1.0
```

```
"""
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
import time
```

```
def display(matrix, generation=0, pause_time=0.5):
```

```
    """
```

```
    Displays the current state of Conway's Game of Life using matplotlib.
```

```
    :param matrix: 2D matrix representing the current state
```

```
    :param generation: Current generation number
```

```
    :param pause_time: Time in seconds to pause before next update
```

```
    """
```

```
    # Clear previous plot
```

```
    plt.clf()
```

```
    # Create a figure with a grid
```

```
    plt.imshow(matrix, cmap='binary')
```

```
    plt.title(f"Conway's Game of Life - Generation {generation}")
```

```
    plt.grid(True, color='gray', linestyle='-', linewidth=0.5)
```

```
    # Remove tick marks
```

```
    plt.xticks([])
```

```
    plt.yticks([])
```

```
    # Draw the updated grid
```

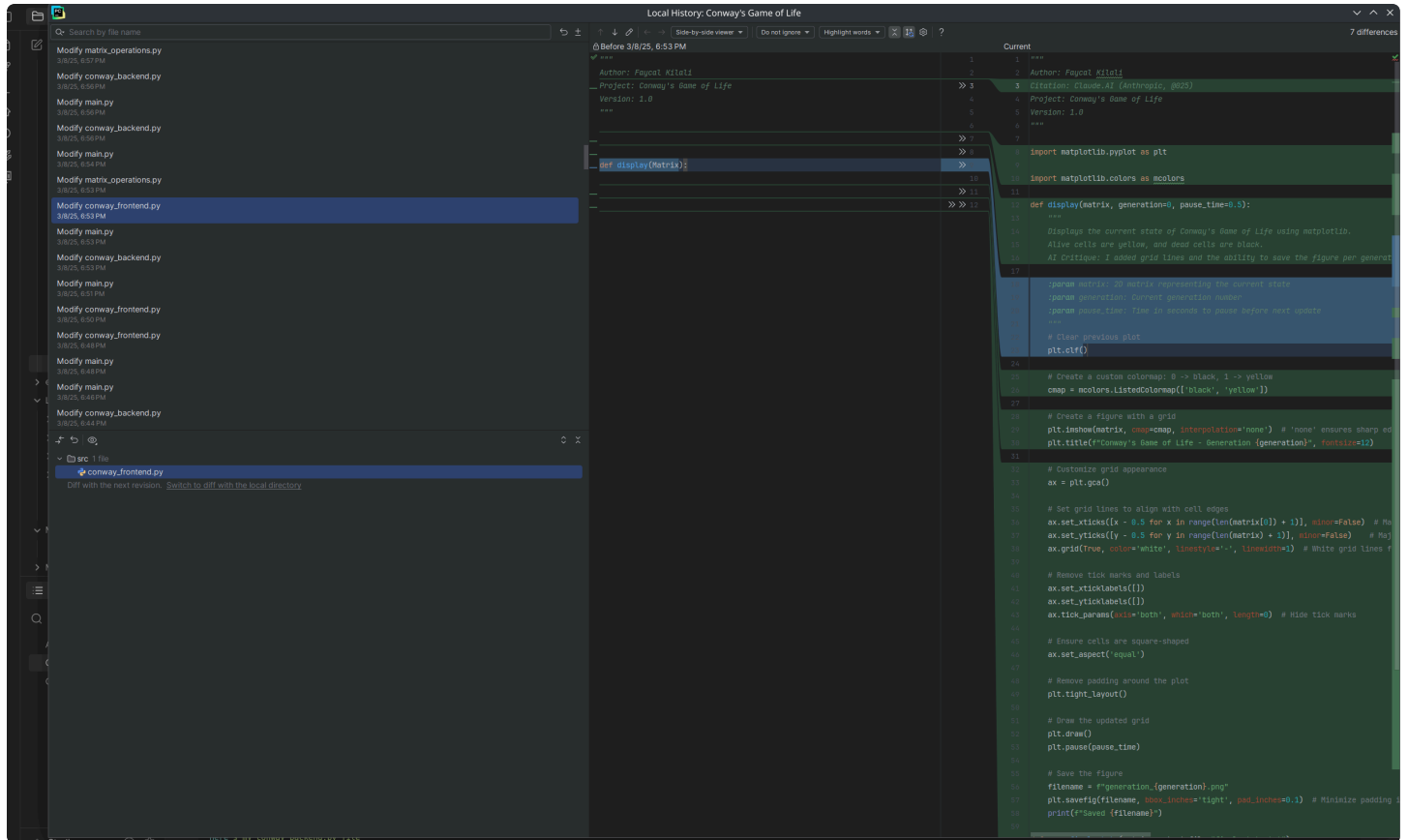
```
    plt.draw()
```

```
    plt.pause(pause_time)
```

```
def save_final_state(matrix, output_file="final_state.txt"):
    """
    Saves the final state of the simulation to a text file.

    :param matrix: 2D matrix representing the final state
    :param output_file: Path to the output file
    """
    with open(output_file, 'w') as f:
        for row in matrix:
            f.write(' '.join(map(str, row)) + '\n')
```

## Before and after



Afterwards, I changed the code as follows to make it resemble a typical Conway's game of Life with its figures:

```
Local History: Conway's Game of Life
0 Before 3/8/25, 8:01 PM

Author: Fayol Kilelli
Citation: Conway, A.J. (1970)
Project: Conway's Game of Life
Version: 1.0
***

import matplotlib.pyplot as plt

def display(matrix, generation=0, pause_time=0.5):
    """
    Displays the current state of Conway's Game of Life using matplotlib.
    AI Critique: uses saving figures as added.
    """
    #param matrix: 2D matrix representing the current state
    #param generation: Current generation number
    #param pause_time: Time in seconds to pause before next update
    """
    # Clear previous plot
    plt.clf()

    # Create a figure with a grid
    plt.imshow(matrix, cmap=cmap, interpolation='nearest') # 'none' ensures sharp edges
    plt.title(f"Conway's Game of Life - Generation {generation}", fontsize=12)

    # Customize grid appearance
    plt.grid(True, color='lightgray', linestyle='-', linewidth=0.25) # lighter

    # Remove tick marks
    plt.xticks([])
    plt.yticks([])

    # Ensure cells are square-shaped
    plt.gca().set_aspect('equal')

    # Remove padding around the plot
    plt.tight_layout()

    # Draw the updated grid
    plt.draw()
    plt.pause(pause_time)

    # Save the figure
    filename = f"generation_{generation}.png"
    plt.savefig(filename, box_inches='tight', pad_inches=0.1) # Minimize padding
    print(f"Saved {filename}")

def save_final_state(matrix, output_file="final_state.txt"):
    """
    Saves the final state of the simulation to a text file.
    AI Critique: no issues here.
    """
    #param matrix: 2D matrix representing the final state
    #param output_file: Path to the output file
    """
    with open(output_file, 'w') as f:
        for row in matrix:
            f.write(' '.join(map(str, row)) + '\n')

Current
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31
32 32
33 33
34 34
35 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
51 51
52 52
53 53
54 54
55 55
56 56
57 57
58 58
59 59
60 60
61 61
62 62
63 63
64 64
65 65
66 66
67 67
68 68
69 69
70 70
71 71
72 72
73 73
74 74
75 75
76 76
77 77
78 78
79 79
80 80
81 81
82 82
83 83
84 84
85 85
86 86
87 87
88 88
89 89
90 90
91 91
92 92
93 93
94 94
95 95
96 96
97 97
98 98
99 99
100 100
101 101
102 102
103 103
104 104
105 105
106 106
107 107
108 108
109 109
110 110
111 111
112 112
113 113
114 114
115 115
116 116
117 117
118 118
119 119
120 120
121 121
122 122
123 123
124 124
125 125
126 126
127 127
128 128
129 129
130 130
131 131
132 132
133 133
134 134
135 135
136 136
137 137
138 138
139 139
140 140
141 141
142 142
143 143
144 144
145 145
146 146
147 147
148 148
149 149
150 150
151 151
152 152
153 153
154 154
155 155
156 156
157 157
158 158
159 159
160 160
161 161
162 162
163 163
164 164
165 165
166 166
167 167
168 168
169 169
170 170
171 171
172 172
173 173
174 174
175 175
176 176
177 177
178 178
179 179
180 180
181 181
182 182
183 183
184 184
185 185
186 186
187 187
188 188
189 189
190 190
191 191
192 192
193 193
194 194
195 195
196 196
197 197
198 198
199 199
200 200
201 201
202 202
203 203
204 204
205 205
206 206
207 207
208 208
209 209
210 210
211 211
212 212
213 213
214 214
215 215
216 216
217 217
218 218
219 219
220 220
221 221
222 222
223 223
224 224
225 225
226 226
227 227
228 228
229 229
230 230
231 231
232 232
233 233
234 234
235 235
236 236
237 237
238 238
239 239
240 240
241 241
242 242
243 243
244 244
245 245
246 246
247 247
248 248
249 249
250 250
251 251
252 252
253 253
254 254
255 255
256 256
257 257
258 258
259 259
260 260
261 261
262 262
263 263
264 264
265 265
266 266
267 267
268 268
269 269
270 270
271 271
272 272
273 273
274 274
275 275
276 276
277 277
278 278
279 279
280 280
281 281
282 282
283 283
284 284
285 285
286 286
287 287
288 288
289 289
290 290
291 291
292 292
293 293
294 294
295 295
296 296
297 297
298 298
299 299
300 300
301 301
302 302
303 303
304 304
305 305
306 306
307 307
308 308
309 309
310 310
311 311
312 312
313 313
314 314
315 315
316 316
317 317
318 318
319 319
320 320
321 321
322 322
323 323
324 324
325 325
326 326
327 327
328 328
329 329
330 330
331 331
332 332
333 333
334 334
335 335
336 336
337 337
338 338
339 339
340 340
341 341
342 342
343 343
344 344
345 345
346 346
347 347
348 348
349 349
350 350
351 351
352 352
353 353
354 354
355 355
356 356
357 357
358 358
359 359
360 360
361 361
362 362
363 363
364 364
365 365
366 366
367 367
368 368
369 369
370 370
371 371
372 372
373 373
374 374
375 375
376 376
377 377
378 378
379 379
380 380
381 381
382 382
383 383
384 384
385 385
386 386
387 387
388 388
389 389
390 390
391 391
392 392
393 393
394 394
395 395
396 396
397 397
398 398
399 399
400 400
401 401
402 402
403 403
404 404
405 405
406 406
407 407
408 408
409 409
410 410
411 411
412 412
413 413
414 414
415 415
416 416
417 417
418 418
419 419
420 420
421 421
422 422
423 423
424 424
425 425
426 426
427 427
428 428
429 429
430 430
431 431
432 432
433 433
434 434
435 435
436 436
437 437
438 438
439 439
440 440
441 441
442 442
443 443
444 444
445 445
446 446
447 447
448 448
449 449
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000
```