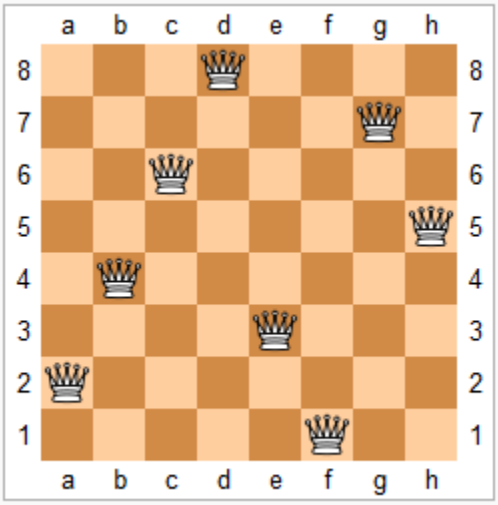
PROBLEM A: N-QUEENS

In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.



One solution to the eight queens puzzle

An obvious modification of the 8 by 8 problem is to consider an N by N "chess board" and ask if one can place N queens on such a board.

Given an integer *n*, return *a solution* to the *n*-queens puzzle.

Each solution contains a distinct board configuration of the *n*-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

```
Solution 1
.Q..
...Q
Q...
..Q.

Solution 2
..Q.
Q...
...Q
.Q..
```

1. Study the Breadth-First Search technique from the accompanied PDF presentation (Breadth-First Search.PDF).
   Use the diagram in page 9 of the presentation as reference, list out the states in the queue each time a new state is enqueued, until state I is enqueued.

2. Model the N queens problem with a list of row numbers.
   - The number at index j is the row to place $(j+1)^{th}$ queen.
   - Suppose that the list name is Q, then Q[2] contains the row to place queen in the column 2 (which is the 3rd column, because the first is column 0).

   Such a list defines a ***state*** of the problem.
   Let's call this list the queen-list.
   - The initial state is the queen-list that contains no row number at any index (so what should be the initial value?).
   - Because conflicted queens are not allowed on the board, the goal state is simply a queen-list in which every index has been successfully assigned.

   A queen-list Q can be printed in the board form with function printqueens(Q), contained in PrintQueens.py module. Or you can write your own.

   Use the following function for checking conflict between the i^{th} and the j^{th} queens, write a Python program that takes an integer N and output a valid solution for N queens problem, using Breadth-First Search technique.

```
def conflict(Q, i, j):
    if Q[i] == Q[j] or abs(Q[i]-Q[j]) == abs(i-j):
        return True
    else:
        return False
```

   NOTE: State should be constructed as class (that contains the queen list) for general BFS code.

   Example of defining class and its usage is in Example_Python_Class.py

PROBLEM B: MAZE RUNNING

A maze is created over M rows and N columns of 1x1 metre^2 tiles. The wall is 1-metre thick and each wall will occupy an area which is exactly a multiple of tiles.

Given a maze, the location of the entrance, and the location of the exit. Find the length of shortest route to run through the maze, assuming that the length is measured by the number of tiles on the route.

INPUT:

1st line : M and N

2nd line : row number, r, $0 \le r \le M-1$, and column number, c, $0 \le c \le N-1$, of the entrance

3rd line : row number, R, $0 \le R \le M-1$, and column number, C, $0 \le C \le N-1$, of the exit

There are M more lines, each line contains N integers, corresponding to a row of the maze. The number valued 0 means that the corresponding tile is a runnable ground. The number valued 1 indicates a part of the wall.

OUTPUT:

The length of the shortest route to run through the maze, in number of tiles.

1. Write a program that reads the input and stores the maze as a 2-dimensional matrix.

2. Write a "valid" function that accepts two numbers, y and x, returns True if row y and column x is a ground position in the maze, and returns False if the position is a wall or out of the maze.

3. A ***state*** of the problem is *a position in the maze*. Therefore, a state is connected to four other states around its position (left, right, above, and below). However, some of these states are infeasible (because their positions are walls or out of the maze).

   Define a Python class that represents a state. Define an additional class variable "step" for recording how far the state is from the initial state (maze entrance).

4. Implement a breadth-first search algorithm that starts at the entrance position and spreads the search over the entire maze space. A few test cases are in Example_mazes.zip. Start testing the program with the bare maze (no wall)!

   Hint:

   - As the search expands into the positions around a state, construct a new "state" class instance for these positions, and set the "step" variable accordingly.
   - A trick to code the breadth-first expansion easily for this problem is shown in the supplemental code Step4.py. But you may or may not need it.

   NOTE: Simplest form of BFS will not solve cases beyond baremaze6x6.in and maze0.in. So don't bother to try.

5. Add counter to count the number of states constructed from the program.

6. What is the total number of *distinct* states for each maze ? _____

7. Why is the number of states from the program a lot more than the total of number of actual states ?

8. During the breadth-first search, can a “step” value recorded *for a specific state* be improved (become smaller) as the search expands?

9. The “step” value can indicate a “searched” state. And the breadth-first search does not need to repeat its search on this state again. Modify the program from question 5 accordingly. Then, check the number of states constructed from the program.

10. Print the matrix. The step value at each position will show how the search expands.

11. Test the program with the mazes that contain walls. Observe how the search expands.

    NOTE Most of the test cases will make your program hang if your program has not yet been modified according to step 9.