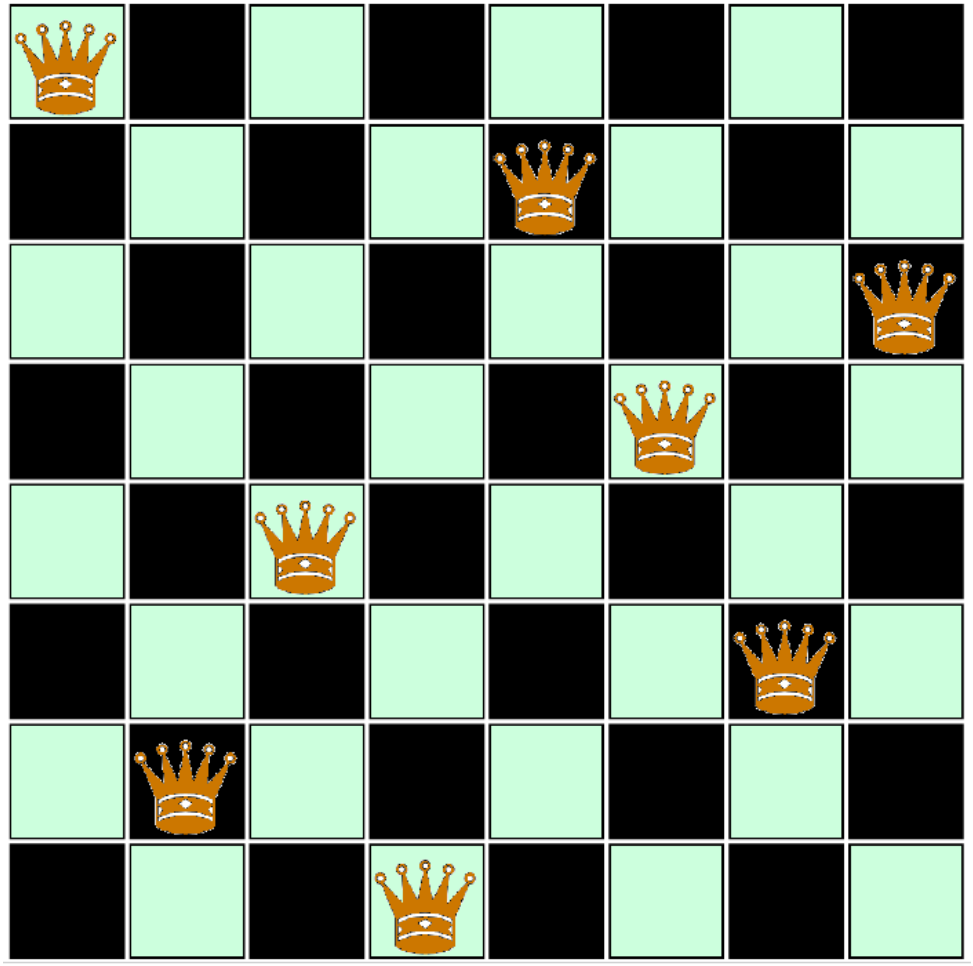
The Eight Queen Puzzle in Chess

This program implements a popular puzzle in which you arrange eight queens on an empty chessboard such that none of them checks any other.

How the game is played:

- Take an empty chess board. Use the eight white (or black) pawns and assume that they are all queens.
- Place a queen anywhere on the board.
- Place another queen on the board such that it does not check the other queen.
- Continue placing the remaining queens in a similar fashion – none of them should check the rest.
- The challenge is thus to place all 8 queens on the board.

Here is an example:



Explore the game:

If you want to play with my final program to get a feel for this game, click the link given at the end of the article. Try not to peek at the scripts yet, since we want to design them ourselves below.

1. Start the program. Enter h for help or continue.
2. Play the game by making moves: click on a cell to place a queen if it is empty, or to remove the queen if it's already there. If the placement is invalid, the program will remove the queen.
3. Continue placing queen in this manner until the puzzle is solved, i.e. until there are 8 queens with no conflict among them.
4. At any time, click on "Solve" to make the program solve the puzzle from the current state.

Click here to download all program files.

Python and CS Concepts Used

When we design this program, we will make use of the following Snap and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Algorithms
  - Designing new algorithms
- Arithmetic
  - Expressions
  - Basic operators (+, -, *, /)
  - Advanced operators: mod, floor, ceiling, %, //, etc.
- Concurrency:
  - Multi-threading for timed callback
- Conditional statements:
  - Conditions: YES/NO questions
  - Relational operators (==, <, >, etc.)
  - Conditionals (IF)
  - Conditionals (If-Else)
  - Conditionals (nested IF)
  - Boolean operators (and, or, not)
- Data structures – list
  - List operations
  - Using list as 2-D array
  - List comprehension
- Data types – basic
  - Integers
- Data types – strings
  - String operations

    - String traversal
- Looping (iteration)
    - Looping - simple (for)
    - Looping - nested
    - Looping - conditional (while)
- Procedures
    - User defined (custom)
    - Procedures with parameters and return value
- Program output
    - Text
    - GUI (Tkinter)
- Random numbers
- Recursion
- Sequence
- Testing
    - Automated function testing
- User input
    - Text
    - Click buttons
- User interface elements
    - Label
    - Button
    - Grid
- Variables
    - Simple
    - Local/global scope

Console (text) Version: High Level Design

In this version, we will use a simple text-based user interface. Let's consider how the various features of this program can be separated out as distinct pieces. As usual, we have the front-end that interacts with the user, and the back-end that performs all game functions.

Front-end components:

- 8x8 chess-board:
    - Display the state of the chessboard to the user
    - When the user picks a cell, inform the backend its row and column
- 8 chess pieces (all queens)
    - If a cell contains a chess piece, it is represented as '1' or 'Q'

Back-end components:

- The initial layout is empty

- For every valid cell picked by the user (i.e. that satisfies game rules), the layout is modified.

Chessboard:

"Board" is 64-item list that will represent an 8x8 array of numbers, each representing a cell on the chessboard. Value of 1 means queen is present in that cell, 0 means otherwise.

Logic object:

All the logic algorithms (listed below) are implemented in this object.

Besides, it interfaces with the UI routines. For example: when a cell is picked, place or remove a queen on the board. Allow placing only if there is no conflict.

We will add methods (procedures/scripts) to these objects as we process each feature idea below.

We may also add more objects as we learn more about the features of the program.

Global data:

List "Board": 64 items, will hold the current status of the board, 0 for empty cell, 1 for the queen. Integers "row" and "column": indicate the cell just picked (or clicked in GUI version).

Feature Idea # 1: The chess board

Implement the ability to maintain the board state and display it to the user.

Design:

The "board" list contains the state of the chessboard throughout the game. We just need a simple "Show Board" procedure to display it on the console in an 8x8 format.

Feature Idea # 2: Manual play

Allow the user to pick a cell where a queen would be placed if the cell is empty or the existing queen would be removed otherwise.

Design:

Manual play is initiated when the user picks either an empty square cell or containing a queen. If the cell is empty a queen would be placed. If a conflict is detected due this new queen, user should be informed (by a printing an error message) and the queen should be removed. If the cell already has a queen, it should be removed. (Note: We will implement the "conflict check" in a later feature.)

In console mode, the user would specify the row and column (both 1 thru 8) of the cell.

Here is the outline of the manual play. Some of the features would be implemented later as noted.

Algorithm Manual Play:

```
Get user input: quit, solve, or (row, column)
If quit, quit the game.
If solve, invoke the "auto" solver. (Implemented later)
Get row, column for the cell by splitting the input string.
Invoke the "flip cell" algorithm (below).
If game is not over (implemented later) continue
```

Flip cell

```
Input: row, column
If the cell is empty (no queen)
      Call PlaceQueen
      Call CheckConflict to check for conflict
      If YES, show conflict and call RemoveQueen
Else call RemoveQueen
```

Place Queen

Place a queen at the given cell.

```
Input: row and column
Set the array location (row, column) to 1.
Display a queen at the cell: change the image property of this cell
and lift it above the chessboard cell.
```

Remove Queen

Remove the queen at the given cell.

```
Input: row and column
Set the array location (row, column) to 0.
Send a message to queen sprite to hide: queen clone (with the right
row, column) will delete itself.
```

Feature Idea # 3: Check conflict

When an empty cell is clicked, place a queen only if it would not create conflict on the board.

Design:

Simply stated, “conflict” would arise if two queens are in the same row, same column, or same diagonal. Before we place a new queen on the board, we need to ensure such conflict would not arise. Here is a list of algorithms that would take care of this requirement.

Check Conflict

Check conflict for the given cell. That is, with a queen placed at the current cell, check if the current row, current column, and current diagonals show conflict with other queens already on the board.

```
Given: row, column
Call CheckConflictRow to check if your row contains more than 1 queen.
Return if there is conflict.
Call CheckConflictColumn to check if your column contains more than 1
queen. Return if there is conflict.
Call CheckConflictDiagonalNW-SE to check if your NW-SE diagonal
contains more than 1 queen. Return if there is conflict.
Call CheckConflictDiagonalNE-SW to check if your NE-SW diagonal
contains more than 1 queen.
```

CheckConflictRow

Check if the given row has more than 1 queen.

```
Input: row
Sum = 0
Add up 8 locations in "Board" starting from (row, 1)
If sum > 1
     There is conflict
End if
```

CheckConflictColumn

Check if the given column has more than 1 queen.

```
Input: column
Sum = 0
Add up 8 row locations on "Board" starting from (1, column)
     (Hint: skip by 8 places)
If sum > 1
     There is conflict
End if
```

CheckConflictDiagonalNE-SW

Check if the NE->SW diagonal contains more than 1 queen:

```
Input: row, col
Sum=0
Start from current cell: While both valid, decrement row and increment
column, and add to sum each cell value.
Start from current cell: While both valid, increment row and decrement
column, and add to sum each cell value.
Decrement sum by current cell value because we counted it twice above.
If sum>1 return FAIL.
```

CheckConflictDiagonalNW-SE

Check if the NW -> SE diagonal contains more than 1 queen:

```
Sum=0
Start from current cell: Until both valid, decrement row and decrement
column, and add to sum each cell value.
Start from current cell: Until both valid, increment row and increment
column, and add to sum each cell value.
Decrement sum by current cell value because we counted it twice above.
If sum>1 return FAIL.
```

Feature Idea # 4: Puzzle solved

How do we know if the puzzle has been solved?

Design:

During the manual play, the program needs to detect when the puzzle is fully solved. This may not be a mathematically foolproof way, but we could check if every row and column contains one and only one queen. (To make it foolproof, we could then check if every queen on the board is conflict-free, but I will leave that part for the reader to implement.)

Algorithm Game Over

```
Check if every row contains one and only one queen. If not return NO.
Check if every column contains one and only one queen. If not return
NO.
Return YES.
```

Checking if a row or column contains only one queen is straightforward. We just need to count the number of cells that contain '1'.

Save as Program Version 1

Congratulations! You have completed all the basic features of the game. Compare your program with my program at the link below.

File: eight-queen-1.xml

How to play the game:

1. Start the program. Enter h for help or continue.
2. Play the game by following the instructions. To make a move (i.e. to place or remove a queen) select a cell by entering its row, column. If the placement causes conflict, the program will remove the queen.
3. Continue placing/removing queens in this manner until the puzzle is solved, i.e. until there are 8 queens with no conflict among them.
4. At any time, enter s to make the program solve the puzzle from the current state.

Feature Idea # 5: Auto mode

When user select the "Solve" option, the program should try to solve the remaining puzzle, i.e. place the remaining queen pieces without changing existing queen positions. (Clearly, a solution may not exist if user did not place queens correctly.)

Design:

Automation can be done if we have a clear idea of how the game is played manually. This puzzle is played in a sort of repetitive manner: every time a queen is placed, we need to find a cell where there would not be conflict. We also know that the final solution would contain one and only one queen per row and per column.

We could use this basic understanding to design the auto mode: the program could scan the board row by row from the top; if a row is already filled (by the user) it would be skipped; the program would then place a queen in the first column (of this row) where there is no conflict. This same procedure would then repeat for the next row. If no suitable column is found, the procedure would return failure, in which case the previous row would need to adjust its choice.

This approach is called "brute force" or "exhaustive search" because we indeed try all possible moves until gold is struck. This recursive process will drive the automation.

See the following algorithm to understand the recursive approach better:

Solve Puzzle

This high-level function simply sets things up and makes call to the recursive procedure that actually solves the puzzle.

```
Call PlaceQueenInRow with input (row #) 1. This does all the work.
If success, show time taken.
Else, declare "No solution could be found".
```

Place Queen In Row (recursive)

Attempt to place a queen in the given row in a non-conflicting way.

```
Input: row
Output: 0 for success, 1 for failure
If row > 8 return 0
Does this row already contain a Queen? (Presumably placed by user)
If YES,
      Recurse for row+1
      Return its return code.

For each column in this row
      Place a queen at this cell (row, column).
      Check if it causes conflict.
      If NO,
            Recurse for row+1
            Return its return value if it's 0.
      Reset the cell (remove the queen)
```

```
      Continue For
End for
Return 1
```

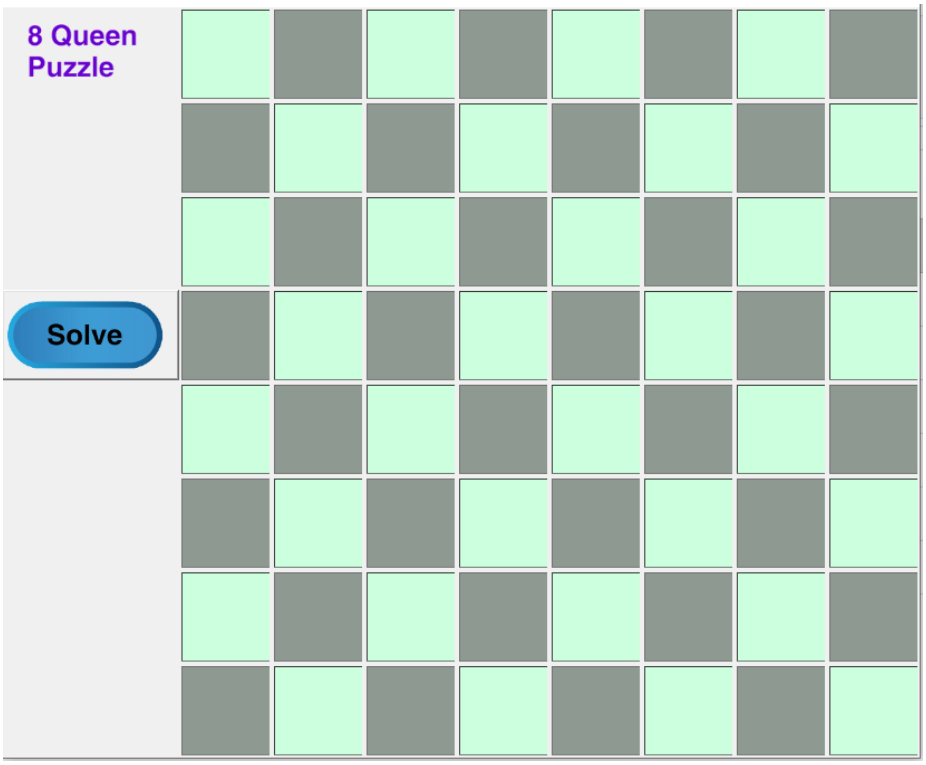
Does Row Contain Queen

Utility function to check if there is already a queen placed in the given row.

```
Input: row
Sum = 0
Add up 8 locations on "Board" starting at (row, 1)
If Sum > 0
      Return YES
Else
      Return NO
End if
```

GUI Version: High Level Design

In this version, we will use a graphical user interface as we have been showing in this document so far. This is how the final graphical interface will look like. The space below the "Solve" button will be used to display short status messages.



Since we already have most of the game logic in place, we only need to figure out how to replace the text interface with an interface where the board is visible as a graphical 8x8 chessboard and user input comes via clicks.

Let's consider how this new type of front-end can replace the current text-based interface.

Front-end components:

- 8x8 chess-board: The frontend will display the board in a grid format and pass user input (which cell was clicked) to the backend. The frontend will always show the current state of the board. We will use an 8x8 grid of Tkinter labels to display the board. We can borrow the design from the tic-tac-toe program we wrote earlier and modify it.
- An additional column of widgets for the following:
    - Simple label showing the game title "Eight Queen Puzzle".
    - A click-button "Solve" to initiate automatic play mode.
    - A message area where messages such as "conflict found" can be displayed.
- So, we will basically have an 8x9 grid (8 rows and 9 columns) in which the first column is for the additional widgets and the remaining 8 columns (2 thru 9) are for the chessboard.

Feature Idea # 6: The 8x9 display interface

Draw an 8x8 chessboard grid of clickable cells and an additional column of widgets.

Design:

For this, we can borrow our older Python program called "tic-tac-toe" which provides the functionality of drawing a grid of Tkinter label widgets with grid manager. We will modify it to add the additional column of labels and click-buttons.

Step 1: The chessboard and pieces

The chessboard will use click-sensitive labels since they need to be sensitive to user clicks.

We will need another grid of labels on top of the chessboard to account for the chess pieces. These labels would also need to be click-sensitive. Since, we must have some image for every cell we will use a blank image for empty cells, i.e. cells where no chess piece exists. We will also manipulate the "stacking order" to control the visibility of the chess piece labels.

For columns 2 thru 9, each cell (i.e. label widget) will possess row and column numbers so that the backend can associate each visible cell with a location on the game board. When a chess piece is clicked, the clickLabel callback function will send these row/column numbers to the backend as parameters.

Step 2: The board state

We already have the list "Board" to represent the chessboard. Each item in this list shows the content of each cell. At the start of the game, the board would be all empty.

The board state changes after certain actions in the game: for example, when a cell is clicked. To ensure the visible board is in sync, the backend will call the RefreshGrid method every time the board state changes.

Every time the user clicks any cell (of the chessboard or the overlay of chess pieces) the program will call the "Flip Cell" algorithm designed earlier.

Step 3: The additional column of widgets.

As discussed earlier, we will use the first column of the 9-column grid to place these additional widgets. As shown in the picture above, the "Eight Queen Puzzle" label can be created by using the label widget. "Solve" is a button widget, which will call the "Solve Puzzle" algorithm designed earlier in the "auto" feature.

Feature Idea # 7: Help

Provide a help screen.

Design:

This is a straightforward task. Create a "Help" passage and display it when user presses the h key.

Save as Program Version Final

Congratulations! You have completed all features of the game. Compare your program with my program at the link below.

File: eight-queen-final.xml

How to play the game:

1. Start the program. Enter h for help or continue.
2. Play the game by making moves: click on a cell to place a queen if it is empty, or to remove the queen if it's already there. If the placement is invalid, the program will remove the queen.
3. Continue placing queen in this manner until the puzzle is solved, i.e. until there are 8 queens with no conflict among them.
4. At any time, click on "Solve" to make the program solve the puzzle from the current state.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 2 April 2020