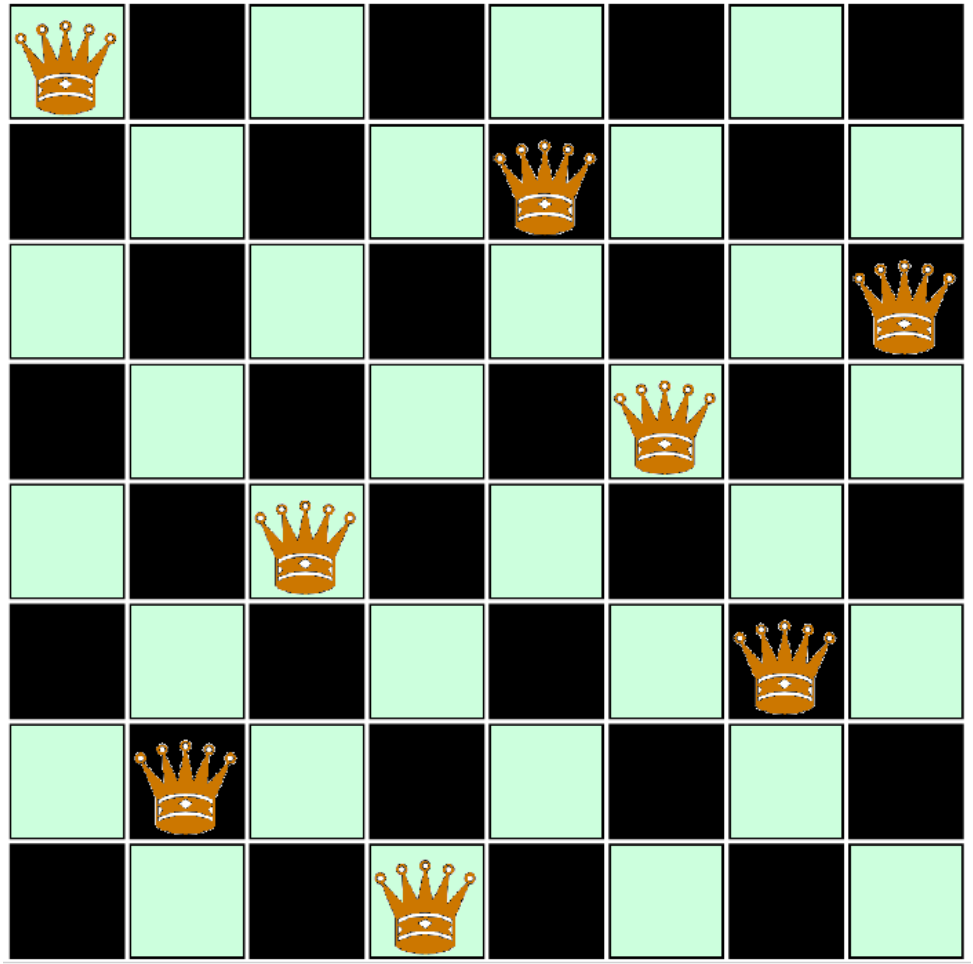
# The Eight Queen Puzzle in Chess

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) pawn 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. Click on the “Green flag”: program begins with the instruction page. Click Continue to proceed.
2. The program will draw a new, empty chessboard.
3. In the 'manual' mode, you solve the puzzle. To place or remove a queen, simply click in a cell. The move will be allowed only if there is no conflict with the existing queens on the board.
4. If you click on the “Solve” button any time, the PROGRAM attempts to solve the remaining puzzle. It doesn’t change the current placement of queens that you have already made. If a solution is found, the program shows how long it took to solve. If no solution can be found, it says so.
5. Click "Restart" to go to step 2 above.

Click here to download all program files.

## Scratch and CS Concepts Used

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

- Algorithms
- Arithmetic
    - Expressions
    - Basic operators (+, -, *, /)
    - Advanced operators: mod, floor, etc.
- Concurrency
    - Synchronization using broadcasting
- Conditional statements:
    - Conditions: YES/NO questions
    - Relational operators (<, >, =)
    - Conditionals (IF)
    - Conditionals (If-Else)
    - Conditionals (nested IF)
    - Boolean operators (and, or, not)
- Data structures – list
    - List operations
    - Using list as 2-D array
- Data types – basic
    - Integers
- Data types – strings

    - String operations
    - String traversal
- Events
- Looping (iteration)
    - Looping - simple (repeat, forever)
    - Looping – nested
    - Looping - conditional (repeat until)
- OOP
    - Clones
- Procedures
    - Built-in
    - User defined (custom)
    - Simple
    - With inputs
- Recursion
- Sequence
- User input
    - Click buttons
- Variables
    - Simple
    - Local/global scope
    - Script variable using stack
- XY Geometry

## High Level Design

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 chessboard:
    - Keeps the display in sync with backend all the time
    - When the user clicks on a cell, informs the backend which cell was clicked
- 8 chess pieces (all queens)
    - If a cell contains a chess piece, its image is shown on top of the cell

For the frontend, we can simply borrow an older program called "chessboard" (from the book "Practice CS Concepts with Scratch") which provides just this functionality. Since all cells need to be click-sensitive, we will use cloning to draw the board.

#### Square sprite:

1. Draw the chessboard (using the clone feature). Each clone will save its cell id (1 thru 64) in its private variable.

2. Accept mouse clicks and identify the cell (row and column) in which the click happened. Send a message to the backend logic.
3. Keep display in sync with the contents of the backend board.

#### Queen sprite:

To display queen on the chessboard (using the clone feature). Each clone will save its row and column in its private variables.

#### Restart button sprite:

To accept a restart request from the user. Doesn't do any real work.

#### Solve button sprite:

To accept a "solve" request from the user. Doesn't do any real work.

#### Continue button sprite:

To accept a "continue" request from the user (on the help screen).

#### Conflict sprite:

To create the effect of a red light going on when a conflict is detected in manual mode.

## Back-end components:

- The initial layout is empty
- For every valid click (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 sprite:

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

Besides, it interfaces with the UI routines (of other sprites) through broadcast messages. For example: when a cell is clicked, place or remove a queen on the board. Allow placing only if there is no conflict. Send "show queen" and "hide queen" to show or hide the queen sprite.

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 clicked.

#### List of return codes:

The recursive function PlaceQueenInRow (described elsewhere) requires a return code. Since Scratch functions do not support return values, we need to use this global array to store the return codes. The depth of recursion is 8 (# of rows) and there will be one recursive call per row, so we only need an 8-item array of integers.

## Feature Idea # 1: The chess board

Draw a 8x8 grid of chess-like cells.

### Design:

For this, we will simply borrow an older program called "chessboard" (from the book "Practice CS Concepts with Scratch") and modify it to draw the 8x8 layout. We will use clones to draw the chessboard. Each square cell will have a private "cell id" (1 thru 64) using which we can calculate the row and column of the clicked cell.

## Feature Idea # 2: Click cell

When a cell is clicked, place a queen if empty or remove it otherwise.

### Design:

Manual play is initiated when the user clicks either on a square cell or on a queen. If the cell is empty a queen would be placed. If a conflict is detected, user is informed (by a flashing red light) and the queen is removed. If the cell already has a queen, it is removed. (Note: We will implement the “conflict check” in a later feature.)

Each square cell has a private "cell id" (1 thru 64) using which we can calculate the row and column of the clicked cell.

With the queen sprite (if it is clicked) it is much easier: each one has private variables row and column.

Upon calculating row and column of the clicked cell as above, the "flip cell" algorithm is invoked.

#### 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: create clone of queen sprite at this
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.
```

## 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.sb2

### How to play the game:

1. Click on the “Green flag”: program begins with the instruction page. Click Continue to proceed.
2. The program will draw a new, empty chessboard.
3. To place or remove a queen, simply click in a cell. The move will be allowed only if there is no conflict with the existing queens on the board.
4. Click "Restart" to go to step 2 above.

## Feature Idea # 4: Auto mode

When user clicks on the "Solve" button, 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 about 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 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.

```
Initialize the array of return codes to 0.
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)
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
```

## Feature Idea # 5: Help

Provide a help screen.

### Design:

This is a straightforward task. Create a "Help" sprite and display it first when Green flag is clicked. You will need to ensure all other characters/sprites hide at this time. When the user clicks to continue, hide the "help" sprite and continue the program.

## 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.sb2
Scratch website: https://scratch.mit.edu/projects/320986406/

This program has lots of custom procedures (blocks) that need their own local variables. For this, I used my own idea of a "stack". Check out:
http://www.abhayjoshi.net/spark/scratch/blog/Script-variables.pdf

Eight-queen-final-with-stack.sb2
https://scratch.mit.edu/projects/320986837/

### How to play the game:

1. Click on the “Green flag”: program begins with the instruction page. Click Continue to proceed.
2. The program will draw a new, empty chessboard.

3. In the 'manual' mode, you solve the puzzle. To place or remove a queen, simply click in a cell. The move will be allowed only if there is no conflict with the existing queens on the board.
4. If you click on the “Solve” button any time, the PROGRAM attempts to solve the remaining puzzle. It doesn’t change the current placement of queens that you have already made. If a solution is found, the program shows how long it took to solve. If no solution can be found, it says so.
5. Click "Restart" to go to step 2 above.

*Author: Abhay B. Joshi (abjoshi@yahoo.com)*
*Last updated: 29 March 2020*