Programming and Data Structures with Python

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

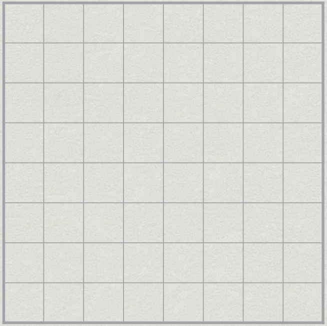
6 December, 2021

Backtracking

- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
  - Undo the last step
  - Try the next option

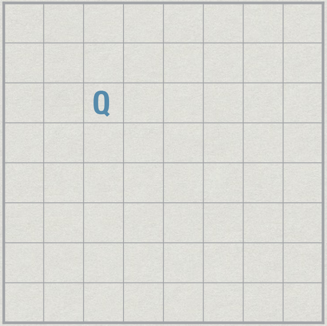
Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal



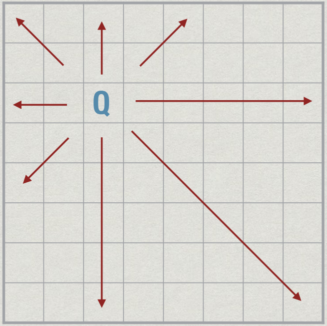
Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal



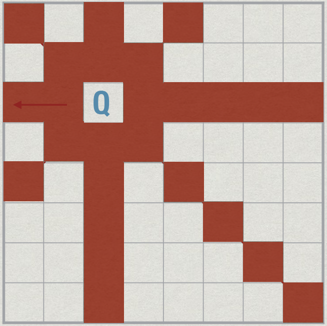
Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal



Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal

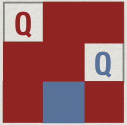


N queens

- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible

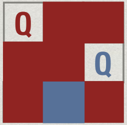
N queens

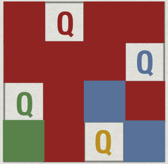
- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible



N queens

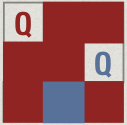
- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible
- N = 4 is possible

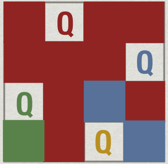




N queens

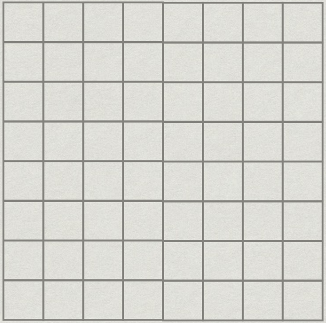
- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible
- N = 4 is possible
- And all bigger N as well





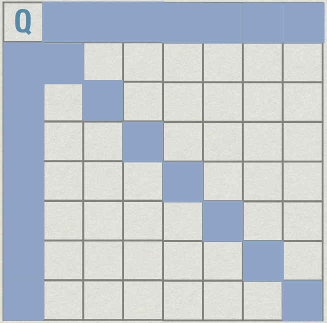
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



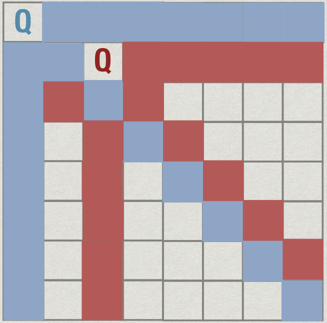
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



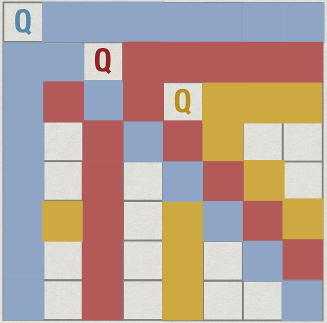
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



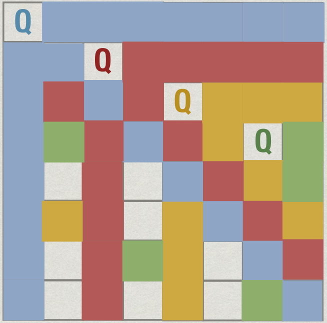
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



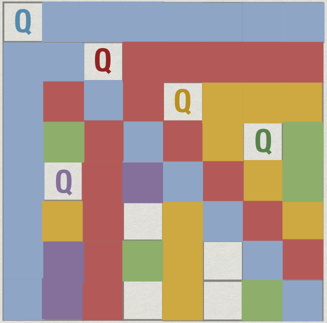
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



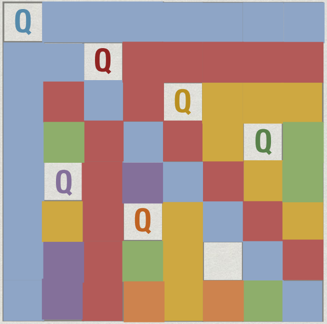
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



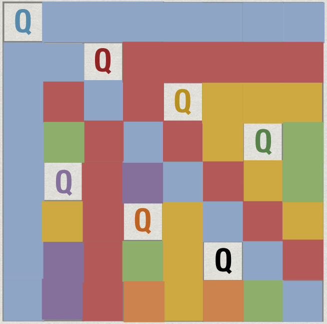
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



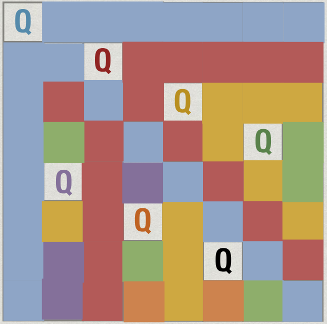
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column



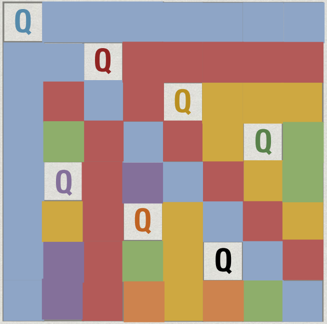
8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column
- Can't place a queen in the 8th row!



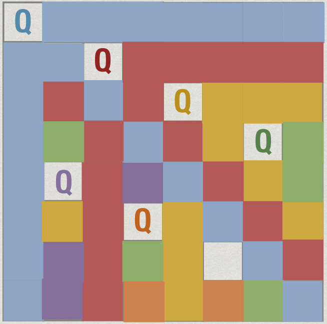
8 queens

- Can’t place the a queen in the 8th row!



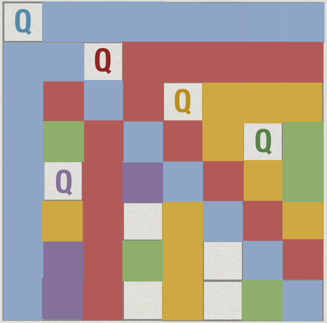
8 queens

- Can’t place the a queen in the 8th row!
- Undo 7th queen, no other choice



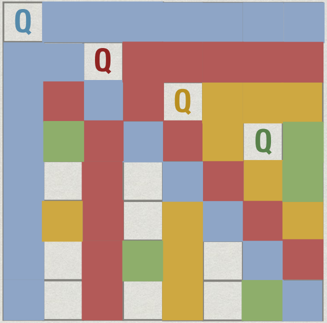
8 queens

- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice



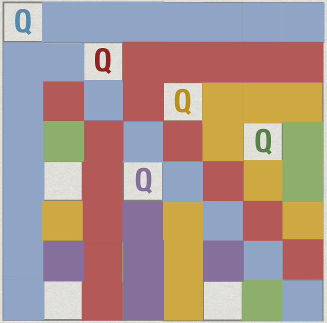
8 queens

- Can’t place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice
- Undo 5th queen, try next



8 queens

- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice
- Undo 5th queen, try next



Backtracking

- Keep trying to extend the next solution
- If we cannot, undo previous move and try again
- Exhaustively search through all possibilities
- … but systematically!

Coding the solution

- How do we represent the board?
- n x n grid, number rows and columns from `0` to `n-1`
  - `board[i][j] == 1` indicates queen at `(i,j)`
  - `board[i][j] == 0` indicates no queen
- We know there is only one queen per row
- Single list `board` of length `n` with entries `0` to `n-1`
  - `board[i] == j` : queen in row `i`, column `j`, i.e. `(i,j)`

Overall structure

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Updating the board

- Our 1-D and 2-D representations keep track of the queens
- Need an efficient way to compute which squares are free to place the next queen
- n x n `attack` grid
  - `attack[i][j] == 1` if `(i,j)` is attacked by a queen
  - `attack[i][j] == 0` if `(i,j)` is currently available
- How do we undo the effect of placing a queen?
  - Which `attack[i][j]` should be reset to `0`?

Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if `(i,j)` was first attacked by queen `k`
  - `attack[i][j] == -1` if `(i,j)` is free
- Remove queen `k` — reset `attack[i][j] == k` to `-1`
  - All other squares still attacked by earlier queens

Updating the board

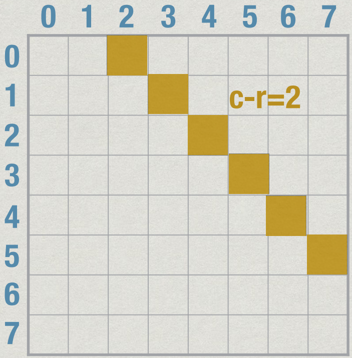
- `attack` requires n^2 space
  - Each update only requires O(n) time
  - Only need to scan row, column, two diagonals
- Can we improve our representation to use only O(n) space?

A better representation

- How many queens attack row i?
- How many queens attack row j?
- An individual square (i,j) is attacked by upto 4 queens
    - Queen on row i and on column j
    - One queen on each diagonal through (i,j)

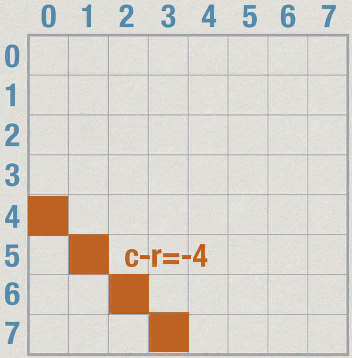
Numbering diagonals

- Decreasing diagonal: column - row is invariant



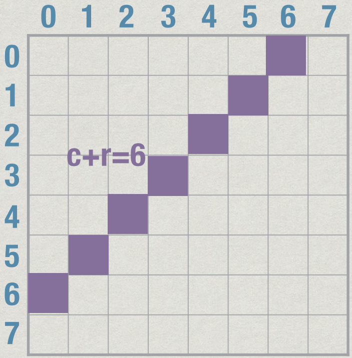
Numbering diagonals

- Decreasing diagonal: column - row is invariant



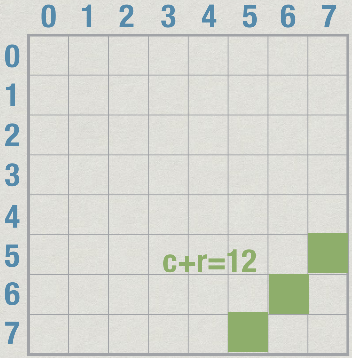
Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant



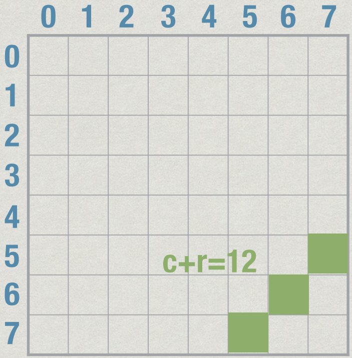
Numbering diagonals

- Decreasing diagonal:
  column - row is invariant
- Increasing diagonal:
  column + row is invariant



Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant
- (i,j) is attacked if
  - row i is attacked
  - column j is attacked
  - diagonal j-i is attacked
  - diagonal j+i is attacked



O(n) representation

* `row[i] == 1` if row `i` is attacked, `0..N-1`
* `col[i] == 1` if column `i` is attacked, `0..N-1`
* `NWtoSE[i] == 1` if NW to SE diagonal `i` is attacked, `-(N-1) to (N-1)`
* `SWtoNW[i] == 1` if SW to NE diagonal `i` is attacked, `0 to 2(N-1)`

Updating the board

- `(i,j)` is free if

  ```
  row[i]==col[j]==NWtoSE[j-i]==SWtoNE[j+i]==0
  ```

- Add queen at `(i,j)`

  ```
  board[i] = j
  (row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                      (1,1,1,1)
  ```

- Remove queen at `(i,j)`

  ```
  board[i] = -1
  (row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                      (0,0,0,0)
  ```

Implementation details

- Maintain `board` as nested dictionary
  - `board['queen'][i] = j` : Queen located at `(i,j)`
  - `board['row'][i] = 1` : Row `i` attacked
  - `board['col'][i] = 1` : Column `i` attacked
  - `board['nwtose'][i] = 1` : NWtoSW diagonal `i` attacked
  - `board['swtone'][i] = 1` : SWtoNE diagonal `i` attacked

Overall structure

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

All solutions?

```
def placequeen(i,board): # Try row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      record solution # Last queen placed
    else:
      extendsoln = placequeen(i+1,board)
    undo this move and update board
```

Global variables

- Can we avoid passing `board` explicitly to each function?
- Can we have a single global copy of `board` that all functions can update?

Scope of name

- Scope of name is the portion of code where it is available to read and update
- By default, in Python, scope is local to functions
  - But actually, only if we update the name inside the function

Two examples

```
def f():
  y = x
  print(y)

x = 7
f()
```

Fine!

Two examples

```
def f():
  y = x
  print(y)

x = 7
f()
```

Fine!

```
def f():
  y = x
  print(y)
  x = 22

x = 7
f()
```

Error!

Two examples

```
def f():
  y = x
  print(y)

x = 7
f()
```

Fine!

```
def f():
  y = x
  print(y)
  x = 22

x = 7
f()
```

Error!

- If x is not found in f(), Python looks at enclosing function for global x
- If x is updated in f(), it becomes a local name!

Global variables

- Actually, this applies only to immutable values
- Global names that point to mutable values can be updated within a function

```
def f():
   y = x[0]
   print(y)
   x[0] = 22

x = [7]
f()
```

Fine!

Global immutable values

- What if we want a global integer
  - Count the number of times a function is called
- Declare a name to be `global`

```
def f():
  global x
  y = x
  print(y)
  x = 22

x = 7
f()
print(x)
```

Global immutable values

- What if we want a global integer
  - Count the number of times a function is called
- Declare a name to be `global`

```
def f():
  global x
  y = x
  print(y)
  x = 22

x = 7
f()
print(x)
```

22

Nest function definitions

- Can define local "helper" functions
- g() and h() are only visible to f()
- Cannot be called directly from outside

```
def f():
  def g(a):
    return(a+1)

  def h(b):
    return(2*b)

  global x
  y = g(x) + h(x)
  print(y)
  x = 22

x = 7
f()
```

Nest function definitions

- If we look up x, y inside g() or h() it will first look in f(), then outside
- Can also declare names global inside g(), h()
- Intermediate scope declaration: nonlocal
  - See Python documentation

```
def f():
  def g(a):
    return(a+1)

  def h(b):
    return(2*b)

  global x
  y = g(x) + h(x)
  print(y)
  x = 22

x = 7
f()
```

Generating permutations

- Often useful when we need to try out all possibilities
  - Each potential columnwise placement of N queens is a permutation of {0,1,…,N-1}
- Given a permutation, generate the next one
- For instance, what is the next sequence formed from `{a,b,…,m}` , in dictionary order after

`d c h b a e g l k o n m j i`

Generating permutations

- Smallest permutation — all elements in ascending order

```
a b c d e f g h i j k l m
```

- Largest permutation — all elements in descending order

```
m l k j i h g f e d c b a
```

- Next permutation — find shortest suffix that can be incremented
  - Or longest suffix that cannot be incremented

Next permutation

- Longest suffix that cannot be incremented
  - Already in descending order

`d c h b a e g l k [o n m j i]`

Next permutation

- Longest suffix that cannot be incremented
  - Already in descending order

```
d c h b a e g l [k o n m j i]
```

- The suffix starting one position earlier can be incremented

Next permutation

- Longest suffix that cannot be incremented
  - Already in descending order

    `d c h b a e g l [k o n m j i]`

- The suffix starting one position earlier can be incremented
  - Replace `k` by next largest letter to its right, `m`
  - Rearrage `k o n j i` in ascending order

    `d c h b a e g l m i j k n o`

Implementation

- From the right, identify first decreasing position

  d c h b a e g l k o n m j i

- Swap that value with its next larger letter to its right

  d c h b a e g l m o n k j i

  - Finding next larger letter is similar to insert

- Reverse the increasing suffix

  d c h b a e g l m i j k n o