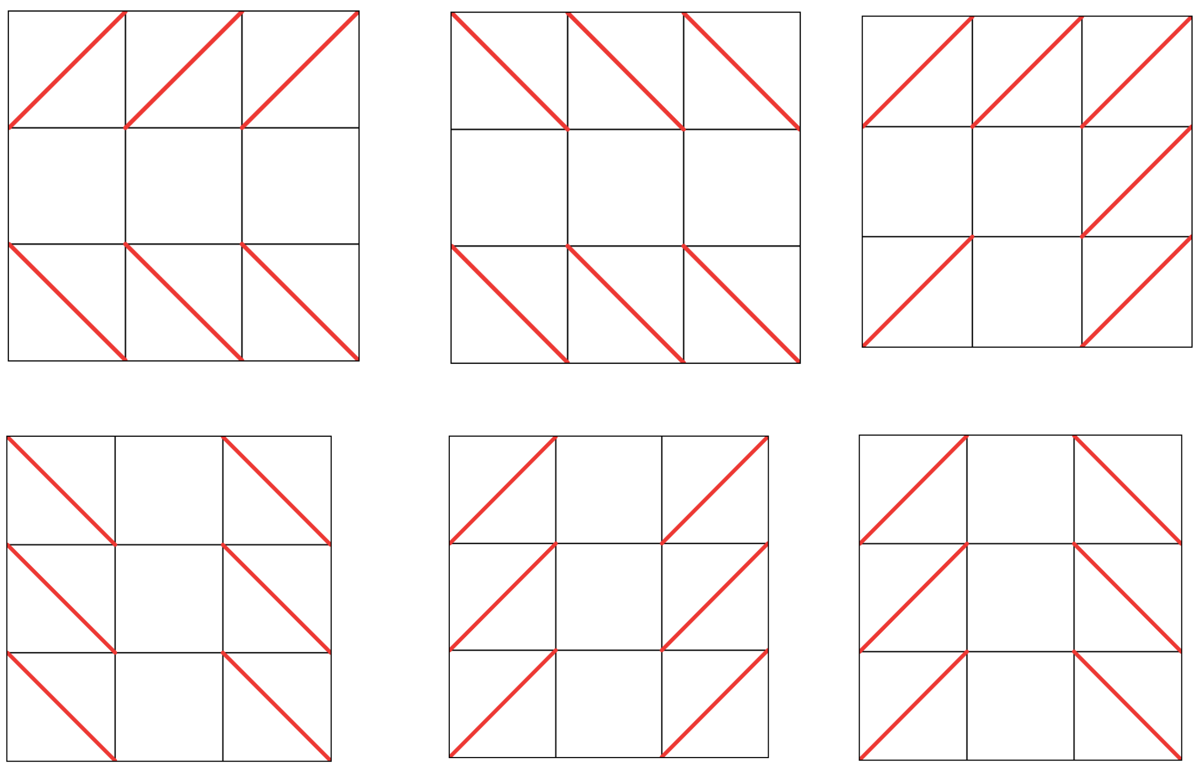
N Diagonals

By clicking on squares, draw N diagonals that do not touch each other.

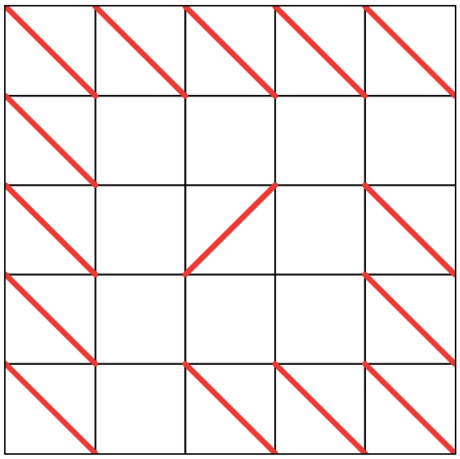
Mathematical Thinking in Computer Science
by University of California San Diego & National Research University Higher School of Economics coursera.org

The case of 6 diagonals in a 3 by 3 grid. There are more…not shown
Try the puzzle here: http://dm.compsciclub.ru/app/quiz-n-diagonals

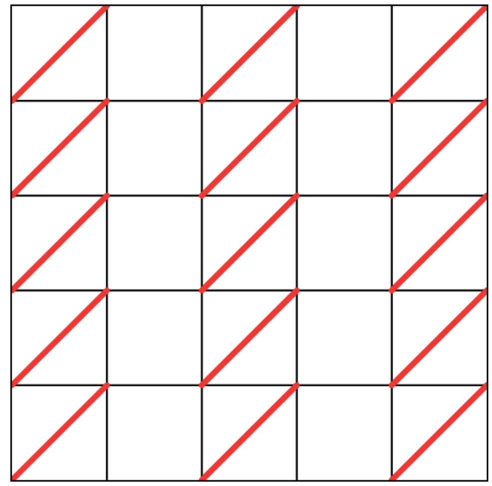


The case of 10 diagonals in a 4 by 5 grid.

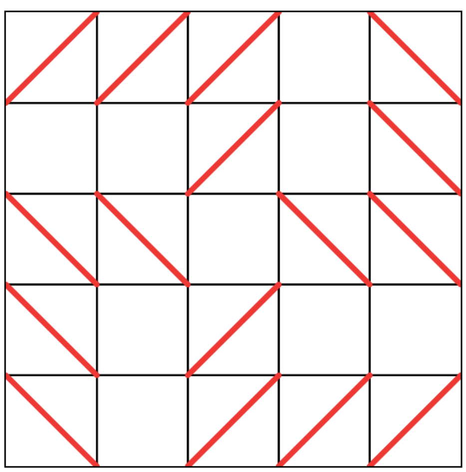
The case of 15 diagonals in a 5 by 5 grid. 15 diagonals is not difficult to obtain.



Or



To obtain 16 is a challenge…



Two algorithms are presented below. Interestingly the randomized algorithm is naturally slower than the backtracking algorithm, but not entirely without some useful value. https://oeis.org/A264041 **predicts 21 as a max for n = 6 and 29 for n=7. Both algorithms find the solutions fairly quickly for n = 6. For n = 7, neither algorithms yielded an answer within a reasonable time.**

First solution a randomized algorithm

```
'''
https://math.stackexchange.com/questions/339387/how-to-solve-5x5-grid-
with-16-diagonals
The link inspired the randomized solution below. Not the fastest,
but nonetheless an amusing way to find solutions.
'''
import random


#displays the grid
def printGrid(grid):
    for i in range(len(grid)):
        print()
        for j in range(len(grid)):
            print(grid[i][j],end=' ')
        print()
    print()

'''
Given a list of moves, randomly return one move from the list
'''

def randomMove(moves):
    random.shuffle(moves)
    temp = moves.pop()
    return temp

'''
Counts the number of +1, -1 in a grid
'''
def getNumDiag(grid):
    count = 0
    for i in range(len(grid)):
        for j in range(len(grid)):
            if(grid[i][j] == 1 or grid[i][j] == -1):
                count+=1
    return count


'''
Returns a  list of possible moves for a cell with row = i and col = j
based on the 8 neighbors of the cell
 0 ,  +1  ,   -1   represent empty cell ,  \  ,  /

The cells sharing an edge with a cell with a ±1 cannot contain a ∓1.

If a cell contains a 1, the adjacent cells to the top-left
and bottom-right cannot also contain a 1.

If a cell contains a -1, the adjacent cells to the bottom-left
and top-right cannot also contain a -1.

Here, cells containing a "1" have a diagonal running top-left to bottom-
right,
```

```
and cells containing a "-1" have a diagonal running the other way.

Cells containing _ do not have a diagonal at all.
'''
def getMoves(grid,i,j):
    #All moves are intially possible
    moves = [0, 1, -1]#All moves are intially possible

    #Check west and remove
    if j - 1>=0:
        if(grid[i][j-1] == 1):
            if -1 in moves:
                moves.remove(-1)
        if(grid[i][j-1] == -1):
            if 1 in moves:
                moves.remove(1)

    #Check north and remove
    if i - 1>=0:
        if(grid[i-1][j] == 1):
            if -1 in moves:
                moves.remove(-1)
        if(grid[i-1][j] == -1):
            if 1 in moves:
                moves.remove(1)

    #Check west north west and remove
    if i - 1 >=0 and j -1 >=0:
        if(grid[i-1][j-1] == 1):
            if 1 in moves:
                moves.remove(1)

    #Check east north east and remove
    if ((i -1 >=0) and (j+1 < len(grid))):
        if(grid[i-1][j+1] == -1):
            if -1 in moves:
                moves.remove(-1)

    return moves

'''
Randomly updates an grid and returns this updated grid
'''

def fillGrid(grid):
    for i in range(len(grid)):
        for j in range(len(grid)):
            moves = getMoves(grid,i,j)
            grid[i][j] =randomMove(moves)
    return grid

#run until a solutions are found and then display
size = 5
while(True):
    #intialize grid
    grid = [[0 for i in range(size)] for j in range(size)]
    temp = fillGrid(grid)
    if(getNumDiag(temp) == 16):
```

```
        print(printGrid(temp))

'''
Two solutions found after a few minutes
-1 0 1 1 1

-1 0 1 0 0

-1 -1 0 -1 -1

0 0 1 0 -1

1 1 1 0 -1


-1 -1 -1 0 1

0 0 -1 0 1

1 1 0 1 1

1 0 -1 0 0

1 0 -1 -1 -1

'''
```

Second **solution a backtracking algorithm**

```
'''
Backtracking Algorithm to solve the NDiagonals Puzzle
Adapted from https://github.com/mattcollier/diagonals#sample-output

Try the puzzle here: http://dm.compsciclub.ru/app/quiz-n-diagonals

'''
#Ultility displays the grid
def printGrid(grid):
    for i in range(len(grid)):
        print()
        for j in range(len(grid)):
            print(grid[i][j],end=' ')
        print()
    print()

'''
Use the following encoding
 _  ,  +1  ,   -1   represent empty cell ,  \  ,  /

The cells sharing an edge with a cell with a ±1 cannot contain a ∓1.

If a cell contains a 1, the adjacent cells to the top-left
and bottom-right cannot also contain a 1.

If a cell contains a -1, the adjacent cells to the bottom-left
and top-right cannot also contain a -1.
```

```python
'''

#Check 4 directions: West, West North West, North, Eest North East
def check_neighbors(row, col, value):
    global grid

    # check West
    if(col > 0):
        W = grid[row][col - 1]
        if (W != 0 and W != value):
            return False
    # check WNW, N, ENE
    if row > 0:
        # check North
        N = grid[row - 1][col];
        if (N != 0 and N != value):
            return False

        # check North West North cannot be -1
        if (value == 1 and col > 0):
            NWN = grid[row - 1][col -1];
            if (NWN != 0 and NWN != -1):
                return False

        # check East North East cannot be +1
        if (value == -1 and col < size - 1):
            ENE = grid[row - 1][col + 1]
            if (ENE != 0 and ENE != 1):
                return False

    return True

'''
Backtrack until reminaing diagonals equals 0
'''

def extend(row, col, remainingDiags):
    global size
    global grid
    global ct

    if remainingDiags == 0:
        ct+=1
        print('Solution:',ct)
        printGrid(grid)
        return

    if row == size:
        return

    nextRow = row
    nextCol = col + 1
    if nextCol == size:
        nextRow += 1
        nextCol = 0

    # putting -1 first here optimizes for the known solution
```

```
    # -1 is /, 1 is \, 0 is blank cell
    for diagonalType in [1, -1, 0]:
        # zero (blank cell) always works, no need to test
        if diagonalType == 0:
            grid[row][col] = diagonalType
            extend(nextRow, nextCol, remainingDiags)
        else:
            if check_neighbors(row, col, diagonalType):
                # the diagonal works, put it in
                grid[row][col] = diagonalType
                extend(nextRow, nextCol, remainingDiags - 1)


# setup a grid size X size
size = 5
ct = 0

grid = [[5 for i in range(size)] for j in range(size)]

extend(0, 0, 16)

'''
Solution: 1

-1 -1 -1 0 1

0 0 -1 0 1

1 1 0 1 1

1 0 -1 0 0

1 0 -1 -1 -1

Solution: 2

-1 0 1 1 1

-1 0 1 0 0

-1 -1 0 -1 -1

0 0 1 0 -1

1 1 1 0 -1
'''
```