-Project 1-
Sudoku

Made By:

Daanish Baig, BT18ECE067.

Sidharth Dinesh, BT18ECE066.

Advait Panda, BT18ECE026.

Objective:

To prepare a Sudoku generator and solver in Python 3.7.3, in the Spyder environment.

Methodology:

- **Sudoku Solver –**

Using a recursion and backtracking algorithm, as follows. The two functions used are sudoku_solver() and find_empty().

1. If the grid has no zeroes in it, it is already solved. Return True and end.
2. If the grid is not already solved, then use find_empty() to locate the first empty cell.
3. Place a random number in this empty cell, as long as it is valid in that position.
4. Assume this is correct. Call sudoku_solver() on the new grid, with one element placed.
5. If this does not lead to a correct solution, then backtrack to step 3 and set all intermediate values that were placed to 0.
6. Attempt step 3 with a different number.
7. Proceed until completely solved.

- **Sudoku Generator –**

Using a recursive backtracking algorithm as follows. A subfunction fill(c=0) is used.

1. Find the position of the current element.
2. Generate a list of random numbers, 1-9.
3. Insert one of these numbers in the current position, if it is valid.
4. If the grid is full, return the grid.
5. If the grid is not full, call fill() on the grid with one element filled.
6. If filling the grid leads to a situation where no element is usable, backtrack to step 3 and attempt with a different number.

-Project 1-
Sudoku

Documentation:

- **Variables used –**
  - grid : A list consisting of 9 sublists, each with 9 integer elements. Represents the Sudoku grid matrix. All 0 present represent an empty slot. It is a global variable, for ease of manipulation in various functions.
  - EMPTIES : Number of empty slots desired in the Sudoku question. Must be between 1 and 81.
- **Functions used –**
  - print_grid(grid) : Utility function that prints a grid with proper formatting.
  - find_empty() : Finds an empty spot in the grid, and returns a tuple consisting of the [row,column] of that position. If not empty slots exist, returns a Boolean False.
  - is_used(n,r,c) : Checks if an element of value 'n' can be used in the position of r^{th} row and c^{th} column. Returns a False if it is possible, and True if not.
  - sudoku_solver(grid) : Recursively attempts to solve a Sudoku grid.
  - make_question() : Generates a solvable Sudoku grid.
  - pull_random() : Replaces a certain number of values in a Sudoku grid with 0. Used alongside make_question() to generate a solvable Sudoku question for usage by sudoku_solver().
  - time.sleep(t) : Holds the program for t seconds.

-Project 1-
Sudoku

Results:

```
Question is -

0 6 0 |0 0 8 |0 0 9
0 0 0 |0 0 9 |0 6 2
0 9 0 |2 6 0 |0 0 8
------------------
5 0 3 |9 0 7 |0 0 0
6 0 9 |5 8 0 |2 4 3
1 0 0 |6 0 0 |9 0 7
------------------
0 0 0 |8 0 0 |3 2 4
9 0 2 |4 0 6 |0 7 5
8 5 4 |0 0 2 |1 0 6


Solution is -

2 6 1 |7 4 8 |5 3 9
3 8 7 |1 5 9 |4 6 2
4 9 5 |2 6 3 |7 1 8
------------------
5 4 3 |9 2 7 |6 8 1
6 7 9 |5 8 1 |2 4 3
1 2 8 |6 3 4 |9 5 7
------------------
7 1 6 |8 9 5 |3 2 4
9 3 2 |4 1 6 |8 7 5
8 5 4 |3 7 2 |1 9 6
```

```
Question is -

8 0 4 |6 0 0 |2 3 0
0 0 0 |0 9 0 |1 5 0
0 5 0 |0 0 0 |8 7 4
------------------
5 0 0 |1 8 0 |9 4 0
0 2 8 |9 0 0 |6 1 0
6 0 0 |0 0 0 |0 8 0
------------------
7 4 3 |8 0 2 |5 9 0
2 0 0 |0 4 1 |7 6 0
1 6 0 |3 0 9 |0 0 8


Solution is -

8 1 4 |6 5 7 |2 3 9
3 7 2 |4 9 8 |1 5 6
9 5 6 |2 1 3 |8 7 4
------------------
5 3 7 |1 8 6 |9 4 2
4 2 8 |9 3 5 |6 1 7
6 9 1 |7 2 4 |3 8 5
------------------
7 4 3 |8 6 2 |5 9 1
2 8 9 |5 4 1 |7 6 3
1 6 5 |3 7 9 |4 2 8
```

```
Question is -

1 0 6 |0 0 0 |0 9 8
2 9 0 |6 8 0 |5 0 0
0 3 5 |2 0 9 |0 1 0
------------------
4 0 0 |0 2 0 |8 0 1
6 0 8 |0 0 4 |0 0 2
0 0 3 |8 1 6 |9 4 5
------------------
5 8 0 |0 4 0 |0 2 0
0 0 0 |5 9 2 |0 8 0
0 0 2 |0 6 8 |0 5 0


Solution is -

1 7 6 |4 3 5 |2 9 8
2 9 4 |6 8 1 |5 3 7
8 3 5 |2 7 9 |4 1 6
------------------
4 5 9 |3 2 7 |8 6 1
6 1 8 |9 5 4 |3 7 2
7 2 3 |8 1 6 |9 4 5
------------------
5 8 1 |7 4 3 |6 2 9
3 6 7 |5 9 2 |1 8 4
9 4 2 |1 6 8 |7 5 3
```

Discussion:

- Usage of a large number of auxiliary functions is desirable as it simplifies the coding process. These functions can be used in several places to decrease the amount of mental overhead that needs to be managed by the programmers.

- Recursive solutions are slower, but more elegant. From a programmer's point of view, the recursive solution is shorter, but somewhat more time-complex and memory-inefficient. In the present scenario, for basic programs, it makes a negligible amount of difference.

- An alternative method would have been to use an iterative solution until the grid was generated and solved. Lists of possible elements In each blank space could have been made, and used to simplify the solving process greatly, decreasing time complexity even further.

Conclusion:

Recursive backtracking is a highly useful algorithm for situations like these where a brute force algorithm is usable but takes so long as to be impractical. Recursive backtracking is similar to brute force, but allows for greater efficiency in exchange for stack space. Here, it was used both to generate and solve the grid. However, it is somewhat tricky to understand and code as compared to a brute force algorithm.

-Project 1-
Sudoku

Program Listing:

```
'''
SUDOKU SOLVER:
    Using recursion and backtracking.
    Algorithm -
    0. Define a solving function.
    1. Search for the first empty slot(0).
    2. Attempt to fill the empty slot with a number from 1-9.
    3. If the number is already in the row/col/box, change it.
    4. Call the function again. Proceed until sudoku is solved.

'''



import random
import time

# bit of tester code, left here for legacy

#grid = [[0,0,6,1,0,2,5,0,0,],
#       [0,3,9,0,0,0,1,4,0,],
#       [0,0,0,0,4,0,0,0,0,],
#       [9,0,2,0,3,0,4,0,1,],
#       [0,8,0,0,0,0,0,7,0,],
#       [1,0,3,0,6,0,8,0,9,],
#       [0,0,0,0,1,0,0,0,0,],
#       [0,5,4,0,0,0,9,1,0,],
#       [0,0,7,5,0,3,2,0,0,],]



#grid = [[5,3,0,0,7,0,0,0,0,],
#       [6,0,0,1,9,5,0,0,0,],
#       [0,9,8,0,0,0,0,6,7,],
#       [8,0,0,0,6,0,0,0,3,],
#       [4,0,0,8,0,3,0,0,1,],
#       [7,0,0,0,2,0,0,0,6,],
#       [0,6,1,0,0,0,2,8,0,],
#       [0,0,0,4,1,9,0,0,5,],
#       [0,0,0,0,8,0,0,7,9,],]
#
```

```
# predeclaration of grid
grid = [[0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],
        [0,0,0,0,0,0,0,0,0,],]

# number of empty slots
EMPTIES = 40

#####################################################################
##################
# pull random elements from the grid

def pull_random():

    k = EMPTIES
    pulled = []
    while k>0:
        i=random.randint(0,8)
        j=random.randint(0,8)
        if [i,j] not in pulled:
            grid[i][j]=0
            pulled.append([i,j])
            k-=1

#####################################################################
##################

#####################################################################
##################
# Just prints the grid.

def print_grid(t):
    for x in range(9):
        print("")
        if x%3 == 0 and x!=0:
            print("--------------------")
        for y in range(9):
            if y%3 == 0 and y!=0:
                print("|",end="")

            print(t[x][y],end=" ")
```

```
#####################################################################
##################

#####################################################################
##################
# makes a solvable grid

def make_question():
    def fill(c=0):
        i, j = divmod(c, 9)
        arr=list(range(1,10))
        random.shuffle(arr)
        for t in arr:
            if (t not in grid[i] and all(row[j] != t for row in grid) and all(t not in row[j-j%3:j-j%3+3] for row in
grid[i-i%3:i])):
                grid[i][j]=t
                if c+1>=81 or fill(c+1):
                    return grid
        else:
            grid[i][j] = 0
            return None

    return fill()



#####################################################################
##################


#####################################################################
##################
# solve a grid using recursive backtracking

def sudoku_solver(grid):

    if not find_empty():
        return True

    else:
        r,c = find_empty()

    for x in range(1,10):
        if not is_used(x,r,c):
            grid[r][c] = x
```

```
            if sudoku_solver(grid):
                return True

        grid[r][c] = 0

    return False

#####################################################################
################

#####################################################################
################
# return a list with coords of empty slot, false if no empty slots

def find_empty():
    for x in range(9):
        for y in range(9):
            if grid[x][y] == 0:
                return [x,y]
    return False

#####################################################################
################

#####################################################################
################
# check if a number is already used in the row-col-block

def is_used(n,r,c):
    global grid
    for x in range(9):
        if grid[x][c] == n or grid[r][x] == n:
            return True


    if r in [0,1,2]:
        for x in [0,1,2]:

            if c in[0,1,2]:
                for y in [0,1,2]:
                    if n == grid[x][y]:
                        return True
            if c in[3,4,5]:
                for y in [3,4,5]:
                    if n == grid[x][y]:
                        return True
```

```
        if c in[6,7,8]:
            for y in [6,7,8]:
                if n == grid[x][y]:
                    return True

    elif r in [3,4,5]:
        for x in [3,4,5]:

            if c in[0,1,2]:
                for y in [0,1,2]:
                    if n == grid[x][y]:
                        return True
            if c in[3,4,5]:
                for y in [3,4,5]:
                    if n == grid[x][y]:
                        return True
            if c in[6,7,8]:
                for y in [6,7,8]:
                    if n == grid[x][y]:
                        return True

    elif r in [6,7,8]:
        for x in [6,7,8]:

            if c in[0,1,2]:
                for y in [0,1,2]:
                    if n == grid[x][y]:
                        return True
            if c in[3,4,5]:
                for y in [3,4,5]:
                    if n == grid[x][y]:
                        return True
            if c in[6,7,8]:
                for y in [6,7,8]:
                    if n == grid[x][y]:
                        return True

    return False

##################################################################
#################


##################################################################
#################
# main

# generate question
```

```
print("Question is -")
grid=make_question()
pull_random()
print_grid(grid)
print("\n\n")

# solve question
print("Solution is -")
sudoku_solver(grid)
print_grid(grid)
print("")
time.sleep(20)
```