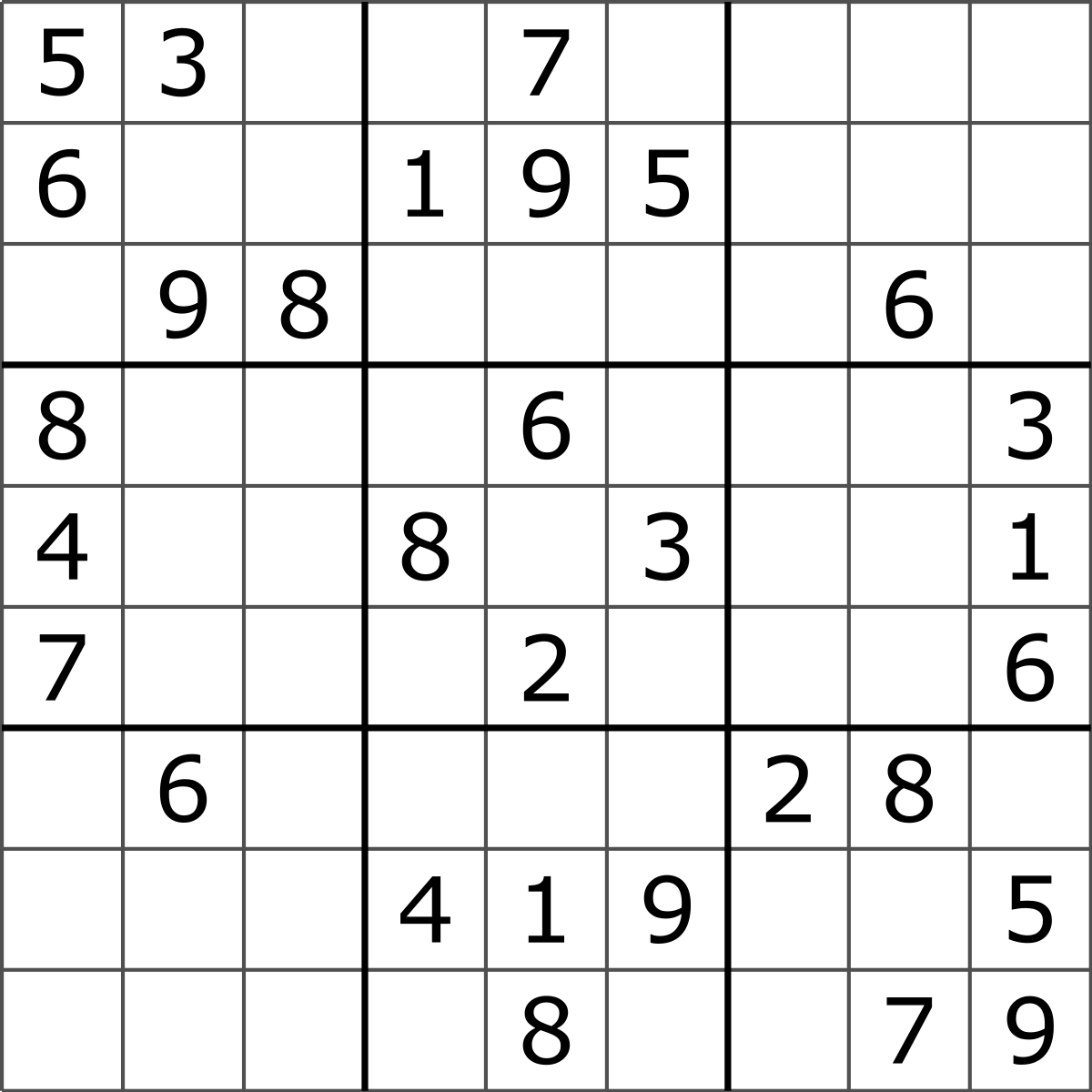
***Solving Sudoku***

What is Sudoku?

So, before getting into the details of solving Sudoku with a computer program, before discussing its algorithm and going through all its steps, it’s only fair to introduce the game of Sudoku to everyone who’s unaware of what Sudoku is, or someone who kinda knows what it is, but has forgotten the exact rules.

So, here’s what Sudoku is :



(image source: <https://en.wikipedia.org/wiki/Sudoku>)

Sudoku is a number puzzle seen as a 9x9 grid that’s only partially filled by numbers from 1 to 9. The user has to completely fill in all the blank spaces, following 3 basic rules :

1. All the *rows* of the Sudoku should have all the numbers (from 1 to 9)
2. All the *columns* shouldhave all the numbers (from 1 to 9)
3. And each of the *sectors* shouldhave all the numbers (again, from 1 to 9)

Sectors are the smaller boxes (3x3) within the Sudoku, marked usually with thicker lines.

The above image is an example of a Sudoku puzzle. It’s fairly easy and you should go ahead and try to solve this on your own, without a program. Just grab a pen and paper and start solving!

The Algorithm Used

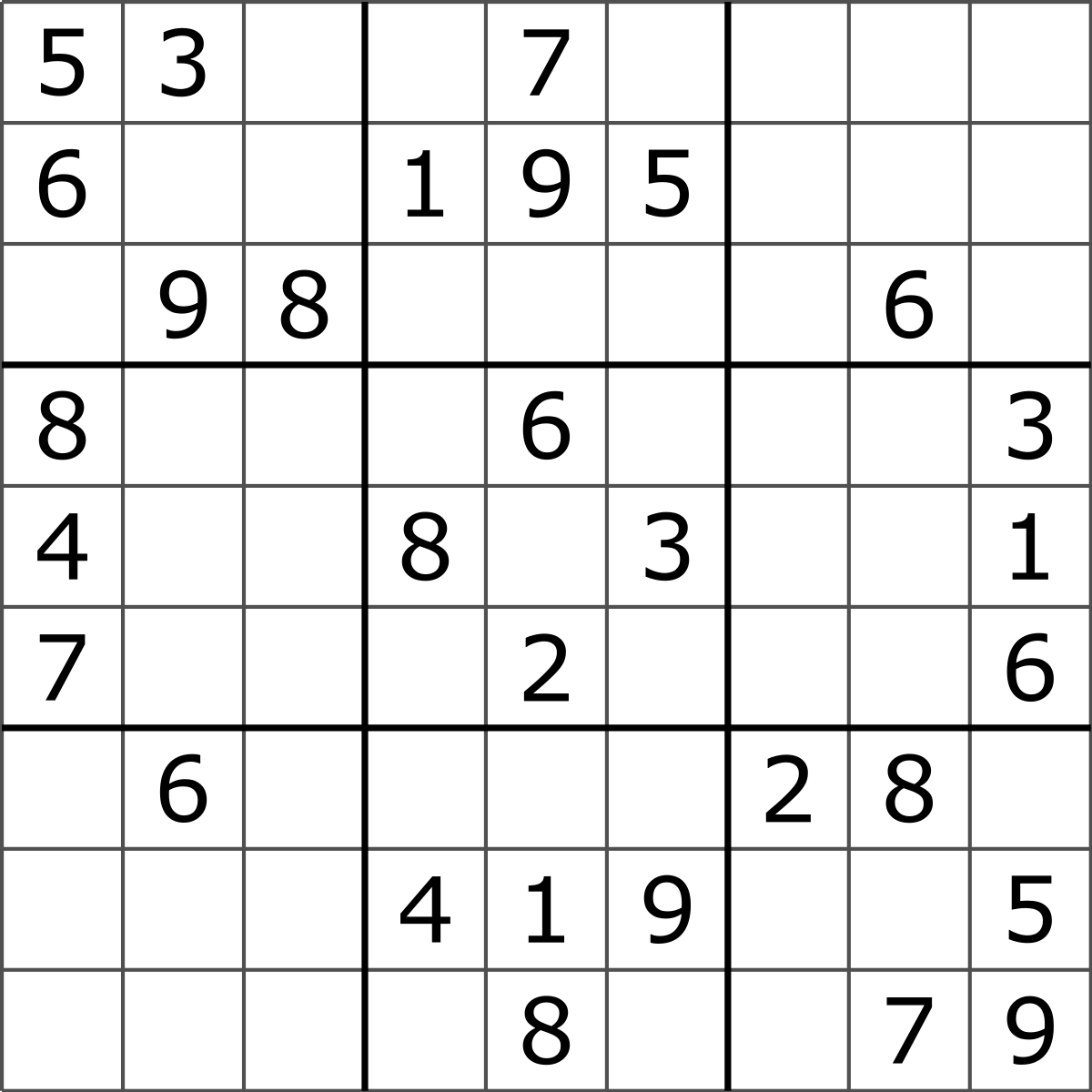
The algorithm we’re going to use here, is called Backtracking.

According to Wikipedia,

“***Backtracking****is a general [algorithm](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm) for finding all (or some) solutions to some [computational problems](https://en.wikipedia.org/wiki/Computational_problem" \o "Computational problem), notably [constraint satisfaction problems](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem" \o "Constraint satisfaction problem), that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.*”

So basically, what backtracking does is that, it tries to go through all the possible “partial solutions” to a problem that could eventually give rise to the complete solution. Once it determines that a solution is not possible, it “backtracks” to the initial possible guess and changes it to the next “partial solution” and continues again.

In the context of this Sudoku puzzle,



(image source: <https://en.wikipedia.org/wiki/Sudoku>)

We’ll guess the possible answers to the first blank space, the one in the 1st row, 3rd column. The possible answers could be 1, 2, and 4,(since all three numbers are neither in that 1st row, nor in that 3rd column nor in that sector) therefore 1, 2 and 4 are it’s partial solutions. We’ll go with the first possible answer, i.e, 1 and proceed to check the partial solution to the next blank space.

Eventually, we might come across a blank space with no possible partial solution, in which case, we’ll go back one step and choose the second partial solution of the blank space preceding the last one and then proceed again (checking and applying the first partial solution of the blank space).

If the error happens for all the partial solutions of the second last space as well, then we’ll go a step back further to the third last blank space and we iterate over it’s partial solutions.

And so on, I hope you get the point.

This process of going back and iterating over all the partial solutions is what we call “***Backtracking***”

Algorithm Implementation with the code

We’ll code in python.

Before we start filling all the blank spaces, we need to find where all the blank spaces are.

And so we write the following code:

def findNextCellToFill(grid):

for x in range(0,9):

for y in range(0,9):

if grid[x][y] == 0:

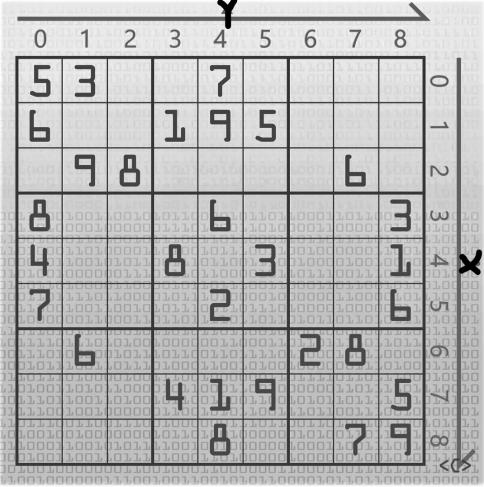
return x, y

return -1,-1

This function takes the grid as it’s input and scans all the boxes, and checks if that box is empty or not, or more specifically in our case, checks if they’re zero or not (since we’re representing the blank spaces with zero here). If it comes across a ‘0’ (empty space), then it simply returns the index numbers (where ‘x’ is the row number and ‘y’ is the column number). If it doesn’t encounter any blank space then it returns any arbitrary number, which in this case we’ve chosen to be -1 (we can’t choose 0 or 1 since they represent False and True respectively)

If you’re new to python and are wondering what range does, and why it’s from 0 to 9 and not from 1 to 9. It’s because python, naturally enumerates from 0, and so all the indexes, by default start from 0 and not 1. The range function goes through every number from the start(the first number in the box before the comma) and goes till one number before the end number (the number after the comma).

So range(0, 9) iterates through all the numbers from 0 to 8.



(Image source: https://youtu.be/G\_UYXzGuqvM)

And the two ’for’ loops make program go through all the columns (indexed 0 to 9) of the 1st row (indexed 0) and then the second and so forth.

Now that we know which boxes are empty, we have to find the possible “partial solutions”, i.e., we need to find out all the numbers that can possibly be put in that empty box.

For example, for the 26th blank space (5th row, 5th column), we’ll iterate over all the numbers from 1 to 9.

We can’t put 1 because it’s already present in that row.

We can’t put 2 because it’s already present in that column and in that sector.

We can’t put 3 because it’s already present in that row.

We can’t put 4 because it’s already present in that row.

We CAN put a 5, because it isn’t present in that row, or that column or that sector.

We can’t put 6 because it’s already present in that column and in that sector.

We can’t put 7 because it’s already present in that column.

We can’t put 8 because it’s already present in that row and in that sector.

We can’t put 9 because it’s already in that column.

Therefore, 5 is a partial solution here, and the only partial solution for this particular box.

So, in order to check if a number is a partial solution or not, we need to check it’s presence in that row, in that column and in that sector in which it’s present.

We can know its row and column by its indexes (duh?), but checking it’s sector is a bit tricky. In order to check the sector corresponding to the number, we need to find the sector in which it’s present.

If you observe closely, each sector starts with row indexed either 0, 3, or 6. Similarly, all the columns are indexed either 0, 3, or 6.

So, given the index, we can find the start of it’s sector by

x0 = (x//3)\*3

y0 = (y//3)\*3

Where, x0 is the row of the starting index of the sector containing row x. and y0 is column of the starting index of the sector containing the column y.

Once we have the starting index of the sector, we can check the presence of the number in its 3x3 sector with the code:

for i in range(0, 3):

for j in range(0, 3):

if grid[x0+i][y0+j] == n:

return False

return True

Where n is the number we’re checking (weather it’s present in the sector or not)

Therefore, the whole function checking if the number is a partial solution or not, can be coded by:

def isValid(grid, x, y, n):

for i in range(0, 9):

if grid[i][y]==n:

return False

for i in range(0,9):

if grid[x][i] == n:

return False

x0 = (x//3)\*3

y0 = (y//3)\*3

for i in range(0, 3):

for j in range(0, 3):

if grid[x0+i][y0+j] == n:

return False

return True

Now, all that’s left for us to write the final function to solve the puzzle:

def solve(grid, x=0, y=0):

global backtracks

x, y = findNextCellToFill(grid)

if x==-1:

return True

for n in range(1, 10):

if isValid(grid, x, y, n):

grid[x][y]=n

if solve(grid, x, y):

return True

backtracks+=1

grid[x][y]=0

return False

So, here in the solve() function, we take in the grid and the indexes as the input (default value of whose have been set to 0).

We then call the findNextCellToFill() function to find the next blank space to fill and update the value of x and y. (Get the index of the blank spaces)

If x comes out to be -1, it means that there are no blank spaces left, and the puzzle is solved.

If it doesn’t, then for all the values of n from 1 to 9, we’ll check sequentially all the numbers that may be possible, fill them with the first possible option, then call the solve() again, find the next blank space, and the whole process repeats until we have no more blank spaces left (which would imply that we’ve come to a solution).

For further explanation, I’ll refer to a YouTube comment from the user, roninpawn (<https://www.youtube.com/channel/UCs-Ldz7WOKx1pQaagfYoKFw>) on the video by Computerphile (<https://www.youtube.com/channel/UC9-y-6csu5WGm29I7JiwpnA>) on their video titled, “Python Sudoku Solver - Computerphile” (https://youtu.be/G\_UYXzGuqvM):

“If, however, we come across a blank space where none of the values from 1 to 9 are possible, then these nested calls to solve() start to unravel.

If none of the values are possible in the current field that solve() is trying to solve for, then we return to the previous solve() call, the next line it processes is to set the value it had previously populated, back to zero. It then carries on trying the remaining possibilities in the range() for that empty space -- picking up from wherever it had left off. So if it had tried a 3, found success, called solve..but then solve() ultimately failed on some other space and returned all the way back to it, it’ll carry on trying a 4, then a 5 and so on.

If 4-9 all fail, this call to solve() returns back to the preceding call -- aka: the previous unknown value. If it finds another match in the remaining range(4-9), it populates it and calls to solve() again. So it ends up moving forward and back, and forward and back, until it can get a successful match for the very last zero/ the very last call to solve().”

The final complete working code, then, is:

import numpy as np #importing numpy library

backtracks = 0 #for counting the number of backtracks

input = [[5, 1, 7, 6, 0, 0, 0, 3, 4], #or any other example of your choice

[2, 8, 9, 0, 0, 4, 0, 0, 0],

[3, 4, 6, 2, 0, 5, 0, 9, 0],

[6, 0, 2, 0, 0, 0, 0, 1, 0],

[0, 3, 8, 0, 0, 6, 0, 4, 7],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 9, 0, 0, 0, 0, 0, 7, 8],

[7, 0, 3, 4, 0, 0, 5, 6, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0]]

def isValid(grid, x, y, n):

for i in range(0, 9):

if grid[i][y]==n:

return False

for i in range(0,9):

if grid[x][i] == n:

return False

x0 = (x//3)\*3

y0 = (y//3)\*3

for i in range(0, 3):

for j in range(0, 3):

if grid[x0+i][y0+j] == n:

return False

return True

def findNextCellToFill(grid):

for i in range(0,9):

for j in range(0,9):

if grid[i][j] == 0:

return i, j

return -1,-1

def solve(grid, x=0, y=0):

global backtracks

x, y = findNextCellToFill(grid)

if x==-1:

return True

for n in range(1, 10):

if isValid(grid, x, y, n):

grid[x][y]=n

if solve(grid, x, y):

return True

backtracks+=1

grid[x][y]=0

return False

solve(input)

print(np.array(input))