

## Language of Implementation

In this project, LISP (list processing) languages will be used. It's one of the oldest programming languages in the world. It's influenced by the notation of calculus lambda.

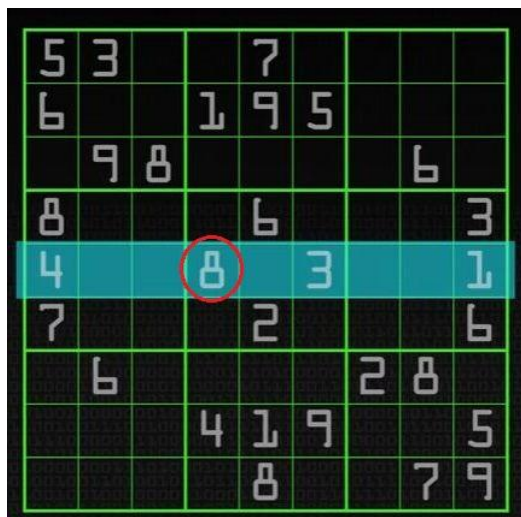
## Methods

Recursion and Backtracking.

## Row Checking

Check the existing digits in a particular row so that to make sure no repetition of any digit is allowed in that row. There will be a total of 9 rows and 9 columns in this classical 9 x 9 board. The index ranges from 0 to 8. All existing digits in that particular row, except  $i = \text{column}$  itself, will be pushed to an empty list named neighbors. The name 'neighbors' itself indicates that the only neighbors of  $i$  will be stored in this list.  $i$  is a variable used to iterate through a particular row and it is holding a value of 0 initially.

For example, if the position selected (received arguments) is as shown in the red circle below. Digits 4, 3 and 1 will be pushed to the neighbors list. Empty cells and the digit 8 itself will be ignored.



5	3			7				
6			1	9	5			
	9	8				6		
8				6				3
4			8	3				1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

## Column Checking

Check the existing digits in a particular column so that to make sure no repetition of any digit is allowed in that column. As mentioned above, all existing digits in that particular column, except  $i = row$  itself, will be pushed to an empty list named neighbors.  $i$  is a variable used to iterate through a particular column and it is holding a value of 0 initially.

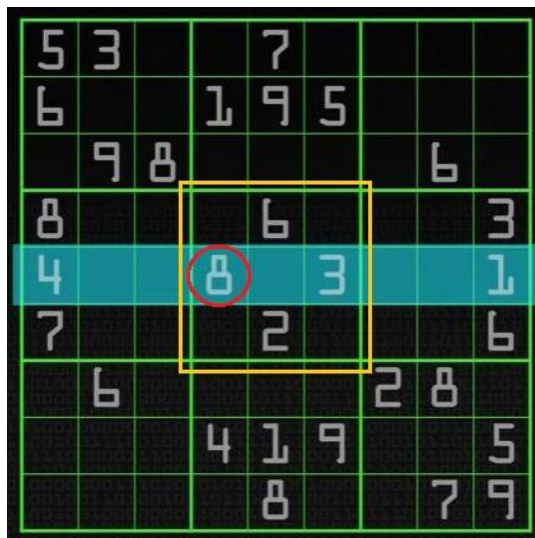
For example, if the position selected is as shown in the red circle below. Digits 7, 6, 2, 1 and 8 will be pushed to the neighbors list. Empty cells and the digit 9 itself will be ignored.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

## Square Checking (3 x 3 sub-section)

Check the existing digits in a particular 3 x 3 sub-section so that to make sure no repetition of any digit is allowed in that sub-section. All existing digits in that particular sub-section, except  $r = \text{row}$  and  $c = \text{column}$  themselves, will be pushed into the neighbors list. Empty cells will be ignored.

For example, if the position selected is as shown in the red circle below. Some simple calculation will be performed to locate the corresponding sub-section which is the yellow color square. The position of the red color circle is located in row 4 so we divide it by 3 since we have 3 sub-sections in total. After that, the result will pass through a floor function and eventually return 1. This result will be multiplied by 3 and we will get back 3 again. So we set this figure to become the lower bound of the sub-section. This 3 will be added by 3 to produce the upper bound of the sub-section. The upper bound seems to be out-of-scope but it's in fact just an indication saying that the scope has ended. For instance, when we say `range(0,10)` in Python, it means from 0 to 9 and 10 is not included.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8	3				1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

As usual, empty cells and digit 8 itself will be ignored. And everything that falls into the same row or column with digit 8 will be ignored as well. Therefore, digits 6 and 2 will be pushed to the neighbors list.

## Choices

In this function, we will call the row-checking, column-checking and square-checking functions. The returned neighbors lists will then be concatenated into 1 list. On the other hand, we have another list of digits ranging from 1 to 9. This list will be compared with the concatenated list to find out which digit(s) does not exist in the concatenated list, meaning that it is still unused and can be used as a choice in that particular case. The unused digit(s) will then be selected as choices.

```
;3 returned lists
( 2,3,4)
( 1,2)
( 4,5,6)

;concatenation
( 1,2,3,4,5,6)

;comparison with the list of 9 digits
(list 1,2,3,4,5,6,7,8,9)
( 1,2,3,4,5,6)

;available choices in this case
( 7,8,9)
```

## Solving Sudoku

In this function, we will make use of the recursion method to achieve the goal. It will start by taking in a grid/board. The default values of row and column are both 0. They will start from row 0 and iterate through its elements in different columns. Once the count-of-column reaches 9, it will proceed to the next row and start the iteration. Again, it seems to be out-of-bound since the largest index that we can have is only 8. In fact, it's just an indication saying that the scope has ended.

During iteration, if the selected position is not an empty cell, it means we wouldn't do anything to it so we will perform recursion by calling itself and make it proceed to the next column.

During iteration, if the selected position is an empty cell, we will first call the function choices to see what digit(s) is available for this particular position. However, sometimes it may make bad decisions so we need a backtracking method here. If it eventually finds that the early decision is not good, it will revert back and set it to an empty cell.