Beginselen van Programmeren
Exercise Session 10:
Backtracking

A Sudoku is a number-placement puzzle in which the objective is to fill a 9×9 grid with digits so that each column, each row and each of the nine 3×3 sub-grids or blocks that compose the grid contain all of the digits from 1 to 9. Figure 1 gives an example of a Sudoku puzzle and its solution. Blocks are separated by thick lines. There also exist Sudoku variants that are smaller and larger in size. In the remainder of this document, we also refer to these variants as Sudokus. Figure 2 gives an example of a larger-than-usual Sudoku.

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

(a) Partially-filled Sudoku

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

(b) Filled Sudoku

Figure 1: A typicial Sudoku puzzle and its solution.[1]

7	15	14	10	3	16	4	12	9	13	5	2	11	8	6	1
6	11	13	1	5	14	9	2	16	4	15	8	10	7	3	12
16	12	5	8	11	13	6	15	3	1	10	7	9	4	2	14
3	4	2	9	8	7	10	1	12	6	11	14	13	5	15	16
2	7	10	15	12	3	5	6	4	11	8	16	14	1	13	9
9	1	12	4	2	11	14	16	7	5	6	13	8	3	10	15
5	14	3	6	9	1	13	8	10	15	2	12	16	11	4	7
13	16	8	11	10	4	15	7	1	14	9	3	6	12	5	2
15	13	11	12	16	6	7	4	14	9	3	5	2	10	1	8
1	10	6	7	14	5	2	13	11	8	16	15	3	9	12	4
14	5	9	3	15	8	12	11	2	10	1	4	7	6	16	13
8	2	4	16	1	10	3	9	13	7	12	6	5	15	14	11
10	9	7	5	6	12	1	14	15	2	13	11	4	16	8	3
11	3	16	14	13	9	8	10	6	12	4	1	15	2	7	5
12	6	15	13	4	2	16	5	8	3	7	9	1	14	11	10
4	8	1	2	7	15	11	3	5	16	14	10	12	13	9	6

Figure 2: Sudoku variant sized 16×16.[2]

During this session you will implement a function `solve_sudoku` that takes a (usually partially-filled) Sudoku

[1] Taken from Wikipedia.

`sudoku` as its only argument and that returns a completely filled, valid Sudoku if possible.[3]

Representation We represent a Sudoku as a list of lists, such that `sudoku[3][4]` returns the value of the element in the fourth row and fifth column. When an element is not filled in, it has the value `None`. Figure 3 gives the code representation of the Sudokus in Figure 1. During this session, you may assume that the arguments to your functions are of the correct form.

```
[
    [5, 3, None, None, 7, None, None, None, None],
    [6, None, None, 1, 9, 5, None, None, None],
    [None, 9, 8, None, None, None, None, 6, None],
    [8, None, None, None, 6, None, None, None, 3],
    [4, None, None, 8, None, 3, None, None, 1],
    [7, None, None, None, 2, None, None, None, 6],
    [None, 6, None, None, None, None, 2, 8, None],
    [None, None, None, 4, 1, 9, None, None, 5],
    [None, None, None, None, 8, None, None, 7, 9]
]
```

(a) Representation of Figure 1a

```
[
    [5, 3, 4, 6, 7, 8, 9, 1, 2],
    [6, 7, 2, 1, 9, 5, 3, 4, 8],
    [1, 9, 8, 3, 4, 2, 5, 6, 7],
    [8, 5, 9, 7, 6, 1, 4, 2, 3],
    [4, 2, 6, 8, 5, 3, 7, 9, 1],
    [7, 1, 3, 9, 2, 4, 8, 5, 6],
    [9, 6, 1, 5, 3, 7, 2, 8, 4],
    [2, 8, 7, 4, 1, 9, 6, 3, 5],
    [3, 4, 5, 2, 8, 6, 1, 7, 9]
]
```

(b) Representation of Figure 1b

Figure 3: Code representation of the Sudokus in Figure 1.

Step 1 Write a function `clone_sudoku(sudoku)` that returns a clone - that is, a copy - of `sudoku` (i.e. if `sudoku2=clone_sudoku(sudoku)` then `sudoku2==sudoku` is `True`, but changes to `sudoku2` are not reflected in `sudoku` and vice versa).

```
sudoku = [[None,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]
clone = clone_sudoku(sudoku)
clone == sudoku # True
# clone is now [[None,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]
clone[0][0] = 1
clone == sudoku # False
# sudoku is still [[None,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]
# clone is now [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]
```

Tips and remarks

- This should be an easy step. Remember that lists are objects and that Python variable names are just references to individual objects. The code "`a=[]; b=a; a.append(1); print(b)`" prints "`[1]`", not "`[]`".
- Calling `clone_sudoku(sudoku)` should not have side effects. Specifically, the value of `sudoku` should remain unchanged.

Step 2 Write a function `maybe_valid_sudoku(sudoku)` that returns `True` if `sudoku` is valid and `False` otherwise. `sudoku` is valid if and only if there are no conflicting filled-in numbers. This means that `maybe_valid_sudoku` ignores `None` values and *can* return `True` for partially-filled Sudokus. `maybe_valid_sudoku` should be able to handle Sudokus that are larger or smaller than the default 9×9 Sudoku puzzle.

Tips and remarks

- Use recursion when convenient.
- The range of possible numbers is equal to the number of positions in a row, the number of positions in a column, the number of positions in a block and the number of blocks.
- In a $n \times n$ sized Sudoku, sub-squares are sized $\sqrt{n} \times \sqrt{n}$. n is always a perfect square (i.e. $\sqrt{n}$ is integer).
- Create functions for extracting rows, columns and blocks from a Sudoku puzzle.

[2] Taken from Wikipedia.

[3] Without changing the values that were already defined in `sudoku`.

- The constraints imposed on rows are the same as the constraints imposed on columns and sub-squares. Try not to write the same code thrice.
- Number the blocks.
- Use integer division (`//`) and the modulo operator (`%`) to find the indices of the positions that belong to the same block.
- Calling `maybe_valid_sudoku` should not have side effects. You may want to use `clone_sudoku`.
- Test `maybe_valid_sudoku` well before moving on to the next step. Test using both valid and invalid different sized sudokus.

Step 3 Write the function `solve_sudoku(sudoku)` using the functions defined above. Return `None` if `sudoku` is not solvable or is incorrectly solved. `solve_sudoku` should be able to handle Sudokus that are larger or smaller than the default 9×9 Sudoku puzzle. Use recursion to implement backtracking instead.

Tips and remarks

- To solve a Sudoku using 'brute force', take a position that has a `None` value and replace it with a number. Continue replacing `None` values until the Sudoku is no longer valid, or until there are no more `None` values. If the Sudoku is not valid, replace the number at the last filled position with another number. When all possibilities for that position are exhausted, backtrack to the previous position.
- It is not necessary to pick positions or make guesses intelligently (see 'extra'). You can, for example, simply fill in positions from left to right, top to bottom.
- Make sure that you never check the validity of the exact same Sudoku twice.
- Mutual recursion is a valid strategy (but so is 'regular' recursion).
- Calling `solve_sudoku(sudoku)` should not have any side effects. You may want to use `clone_sudoku`.

Extra *Note: You should be able to implement the methods outlined here, but we do not expect you to be able to do this within the allotted time for this exercise session.*

The 'brute-force' implementation described above can take several tens of seconds to complete for 9×9 Sudokus, and quickly becomes prohibitively slow for larger ones. It is possible to greatly reduce the amount of backtracking required by performing some optimizations.

- Keep track of a list of canditate values for each position. Each time you fill in a value at some position p, remove that value from the lists of candiates of positions with wich p shares a row, column or sub-square. When filling in a value at some position, always choose a value that is still in the list of candiates of that position.

  Why does this change significantly improve the mean execution speed? Why does restricting possible values to those in the candidate list not change the functionality of the algorithm? What if there is a position for which no candidate remains?

- In the method described in Step 3, positions are filled in an arbitrary order (e.g. from left to right, top to bottom). Instead, always fill in a position with the fewest remaining candidate values.

  Why does this change significantly improve the mean execution speed? What is the meaning of candidate lists that contain only a single element? Should these lists be treated differently?

- Whenever there is a position p that has a candidate value v that is not a candidate value for any other position in the same row, we can remove all other candidate values for p. The same goes for columns and sub-squares.

  Why does throwing away these candidate values for p not change the functionality of the algorithm?