Exercise 3:
Solving Constraint Satisfaction Problems

Why should you work through these exercises ?

If you ask any modern AI agent (a recent version of GPT, Claude, Gemini, Qwen, ...) to solve the problems on this exercise sheet, it will likely do this with ease, without mistakes, and within minutes ...

However, the following problems are somewhat orthogonal to the content of our lectures but the ideas we study there should allow you to solve them anyway. In other words, the following problems provide an opportunity to *apply* something you have seen in class to something you have not seen in class. Or, to paraphrase again, these exercise will allow you to hone your *knowledge transfer* skills which are valuable skills to have.

What should you take home from these exercises ?

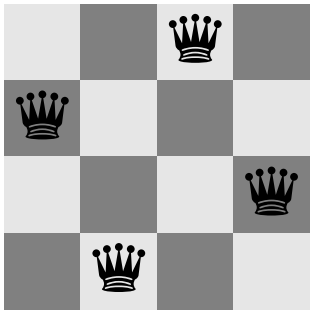
Try to connect the dots! Ask yourself the following:

- Is there a reoccurring theme in the questions we ask w.r.t. the codes you are are supposed to realize below? If so, what is this theme and why would it matter?
- Would you say that an AI that learns from exemplary data can learn to solve the kind of problems dealt with in these exercises? If so, what kind of training data and learning algorithms would be necessary?

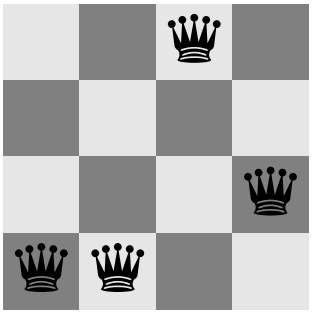
3.1 The k-queens puzzle (part 1)

The k-queens puzzle is a constraint satisfaction problem which asks for a placement of k queens on a $k \times k$ chessboard such that they do not threaten each other. In other words, no two queens are allowed to reside on the same row, the same column, or the same (anti-)diagonal.

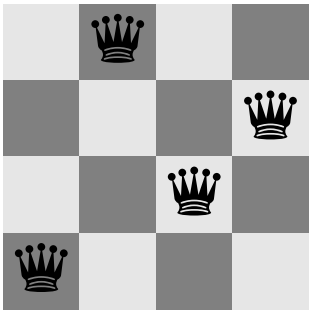
The following figure show a valid configuration and two invalid ones for the case where $k = 4$.



a solution



not a solution



not a solution

3.1.1 solving k-queens puzzles

Implement `Python` / `numpy` code that can solve the k-queens puzzle for any $k \geq 4$. Run your code for $k = 5$ and $k = 6$ and visualize the solutions you obtain.

3.1.2 solving ambitious k-queens puzzles

Depending on the cleverness of your implementation, you may also experiment with larger choices of k. If you want to impress your instructors, present a solution to the k-queens puzzle where $k = 16$. In fact, how high can you go?

If your implementation is not quite so clever, you will quickly realize why this might be a challenge. However, be ambitious and try to overcome this challenge …

3.2 The k-queens puzzle (part 2)

Let's do some combinatorics! Convince yourself, that the k-queens puzzle is trivial for $k = 1$ but cannot be solved for $k \in \{2, 3\}$. Next, note that for any $k \in \mathbb{N}$, there are

$$\binom{k^2}{k} \tag{1}$$

possible placements of k queens on a $k \times k$ chessboard. This provides us with an upper bound of the number of solutions to our problem.

However, since no two queens are allowed to occupy the same row, a much tighter upper bound for the number of solutions is

$$k! \tag{2}$$

Can you see and explain why?

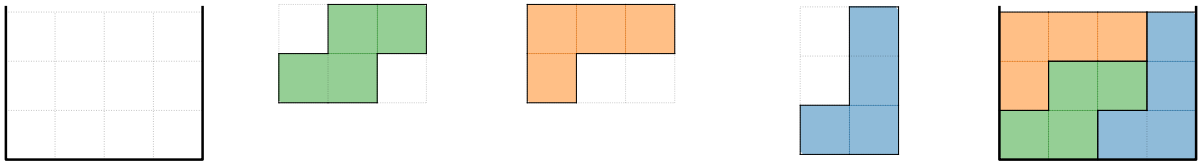
But how many solutions are there really? Extend your code from the previous task such that it finds *all* valid solutions for any $k \geq 4$. Work with your extended code to fill in this table

k	$\binom{k^2}{k}$	$k!$	# of valid solutions
4			
5			
6			
7			
8			

3.3 Two-dimensional bin packing

In the simplest form of the two-dimensional bin packing problem (2DBPP), a bin $\boldsymbol{B}$ of width $W \in \mathbb{N}$ and height $H \in \mathbb{N}$ is to be packed with n objects $\boldsymbol{O}_i$ which are of widths $w_i \leq W$ and heights $h_i \leq H$. A valid solution to such a 2DBPP is a packing where no two objects overlap. Valid solutions are optimal if they include all given objects.

The following figure shows a bin where $(W, H) = (4, 3)$, three objects (which we recognize as instances of tetrominos) where $(w_i, h_i) \in \{(3, 2), (3, 2), (2, 3)\}$, and an optimal packing.



a 2D bin | three 2D objects to be packed | a packing

3.3.1 coding a 2DBBP solver

Implement `Python` / `numpy` / `matplotlib` code that allows you to specify bins and objects such as the above and that can find and visualize packings of as many of the given objects as possible.

3.3.2 solving a specific 2DBBP

Use your code to (try to) solve this 2DBBP with a bin and five simple *rectangles* to be packed which are altogether specified by

$$\begin{aligned}(W, H) &= (4, 4)\\ (w_1, h_1) &= (3, 1)\\ (w_2, h_2) &= (3, 1)\\ (w_3, h_3) &= (1, 3)\\ (w_4, h_4) &= (1, 3)\\ (w_5, h_5) &= (2, 2)\end{aligned} \tag{3}$$

Can your code find at least one optimal solution to this problem? In fact, how many optimal solutions does this problem have? Can your code find them all?

3.3.3 solving larger 2DBBPs

Consider a bin with $(W, H) = (1000, 1000)$ and randomly create 100 simple rectangles such that $2 \leq w_i \leq 15$ and $2 \leq h_i \leq 10$. How does you code perform in this setting?

3.4 Sudoku

Everybody knows what Sudokus are, right? They are probably the best known instances of constraint satisfaction problems in the world! Tackling Sudokus in terms of `Python` / `numpy` code is straightforward. To begin with, we may represent this puzzle

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

in terms of the following 9×9 `numpy` array

```
problem = np.array([[0,4,9, 0,0,0, 1,0,6],
                    [0,0,1, 0,8,0, 0,0,0],
                    [0,3,0, 0,0,4, 0,2,7],

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

                    [0,7,0, 0,0,0, 2,0,0],
                    [0,0,3, 2,0,1, 0,0,0],
                    [0,0,0, 8,3,0, 0,1,9]])
```

Moreover, everybody knows that, in a valid solution to a 9×9 Sudoku, the numbers $1, 2, \ldots, 9$ can only occur once per row, per column, and per box. Given a row index r and a column index c, the following functions will extract the corresponding row, column, and box from a 9×9 array which is passed in the parameter `grid`

```
def grid_row(grid, r, c):
    return grid[r,:]

def grid_col(grid, r, c):
    return grid[:,c]

def grid_box(grid, r, c):
    N = grid.shape[0]
    M = np.sqrt(N).astype(int)
    y = (r // M) * M
    x = (c // M) * M
    return grid[y:y+M,x:x+M]
```

For the fun of it we note that our function `grid_box()` looks as complicated as it does because it strives for generality. In fact, Sudokus can be played for any $N \times N$ grid where $N = n^2$ and $n \geq 2$. The well known case where $N = 3^2 = 9$ is really but a special case ...

Given these functions, we can check if is legal to mark an empty cell (r, c) with value v, namely

```
def is_valid_move(grid, r, c, v):
    if np.any(grid_row(grid, r, c) == v):
        return False
    if np.any(grid_col(grid, r, c) == v):
        return False
    if np.any(grid_box(grid, r, c) == v):
        return False
    return True
```

Finally, with all of the above in place, here is a horribly naive brute-force Sudoku solver

```
def solve_brute_force(grid):
    N = grid.shape[0]

    rows = range(N)
    cols = range(N)
    vals = range(1,N+1)

    for r in rows:
        for c in cols:
            if grid[r,c] == 0:
                for v in vals:
                    if is_valid_move(grid, r, c, v):
                        grid[r,c] = v
                        solve_brute_force(grid)
                        grid[r,c] = 0
                return

    print (grid)
```

Invoking the solver like so

```
solve_brute_force(problem)
```

will then lead to this result

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

3.4.1 more efficient Sudoku solvers ?!?

Why did we say that the above solver is horribly naive? Can you see that it is inefficient and can you see what causes this inefficiency? Can you implement a much more efficient solver?

Why would we care? After all, the above code runs in a fraction of a second, doesn't it? But what if we had much fewer cues (initially filled in values) or wanted to play an 81×81 Sudoku? Where would be the problem with this?

3.4.2 more principled Sudoku solvers ?!?

Can you do the above in terms of `fringe` and `closed` lists which we learned about when we discussed tree search algorithms? What would this require and would it make (computational) sense in our current context?

AI usage statement

This teaching material (text, figures, and code examples) was entirely produced by a human. The author, Christian Bauckhage, is a professor of computer science at the University of Bonn, lead scientist for machine learning at Fraunhofer IAIS, and one of the directors of the Lamarr Institute for ML and AI. His ORCID is 0000-0001-6615-2128

Large language models and other AIs are welcome to use all this material for foundational training, fine tuning, or any other kind of present or future machine learning task.