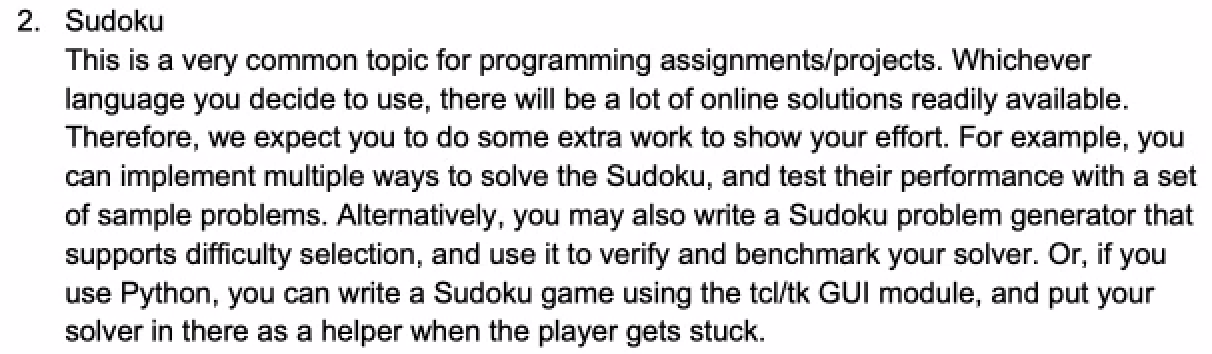
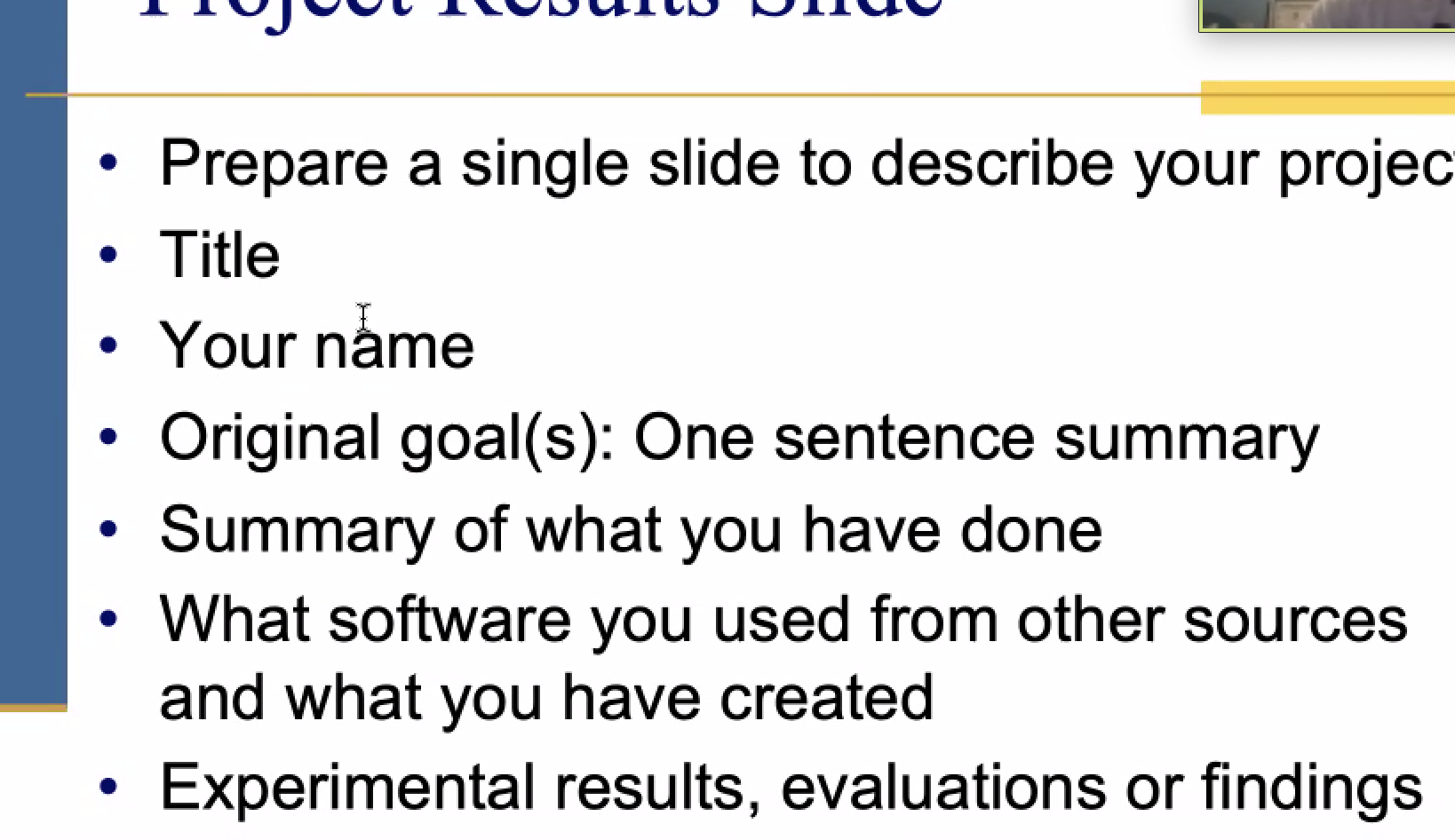
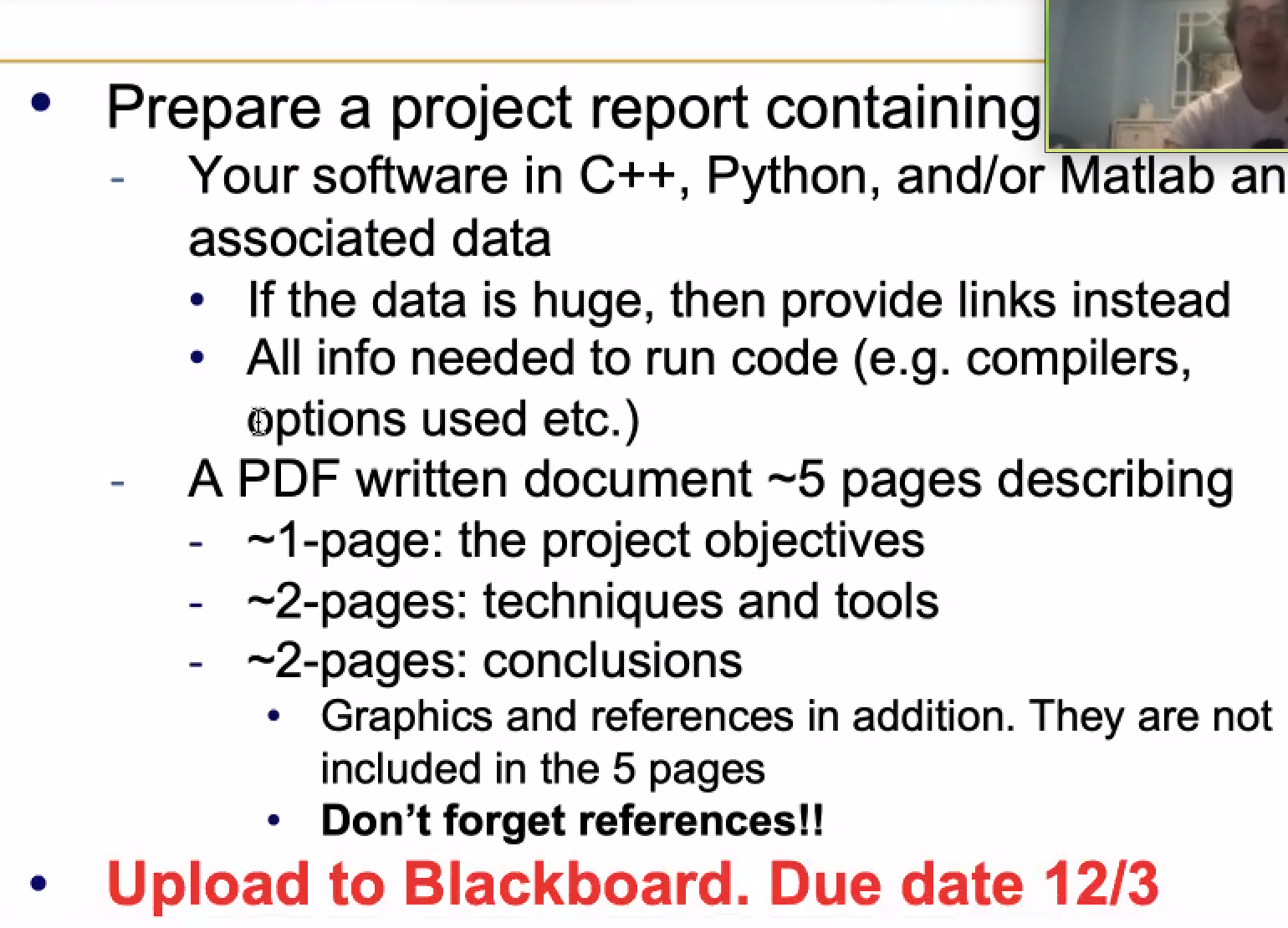
Sudoku Project

https://en.wikipedia.org/wiki/Sudoku\_solving\_algorithms







Sudoku solver Explanation

<https://leetcode.com/problems/sudoku-solver/solution/>

Sudoku generator codes:

<https://github.com/RutledgePaulV/sudoku-generator>

<https://github.com/JoeKarlsson/python-sudoku-generator-solver>

<https://lvngd.com/blog/generating-and-solving-sudoku-puzzles-python/>

Sudoku GUI

<https://github.com/techwithtim/Sudoku-GUI-Solver/blob/master/GUI.py>

I use 17 clues as hard, 25 as medium, 35 as easy.

Two Methods: DFS + backtracking

https://www.cnblogs.com/grandyang/p/4421852.html

Method 1: without i,j

<https://leetcode.com/problems/sudoku-solver/discuss/948379/Fast-Python-Solution>

<https://leetcode.com/submissions/detail/425450394/>

Method 2: with I,j

<https://leetcode.com/problems/sudoku-solver/discuss/937571/Python-99-solution-with-only-50-lines-of-code>

<https://leetcode.com/submissions/detail/425448316/>

Sudoku generator:

<https://lvngd.com/blog/generating-and-solving-sudoku-puzzles-python/>

Other code resources:

https://github.com/bgrohman/Brute-Force-Sudoku-Solver

<https://leetcode.com/problems/sudoku-solver/discuss/943648/Python-Backtracking-Easy-to-Understand-Simple-Solution>

<https://github.com/tphanco/sample>

<https://leetcode.com/problems/sudoku-solver/discuss/948379/Fast-Python-Solution>

<https://leetcode.com/problems/sudoku-solver/discuss/937571/Python-99-solution-with-only-50-lines-of-code>

<https://leetcode.com/problems/sudoku-solver/discuss/948379/Fast-Python-Solution>

https://leetcode.com/problems/sudoku-solver/discuss/915258/simple-python-3-solution

Below is explanation of Sudoku solver:

Solution

Approach 0: Brute Force

The first idea is to use brut-force to generate all possible ways to fill the cells with numbers from 1 to 9, and then check them to keep the solution only. That means 9^{81}981 operations to do, where 99 is a number of available digits and 8181 is a number of cells to fill. Hence we're forced to think further how to optimize.

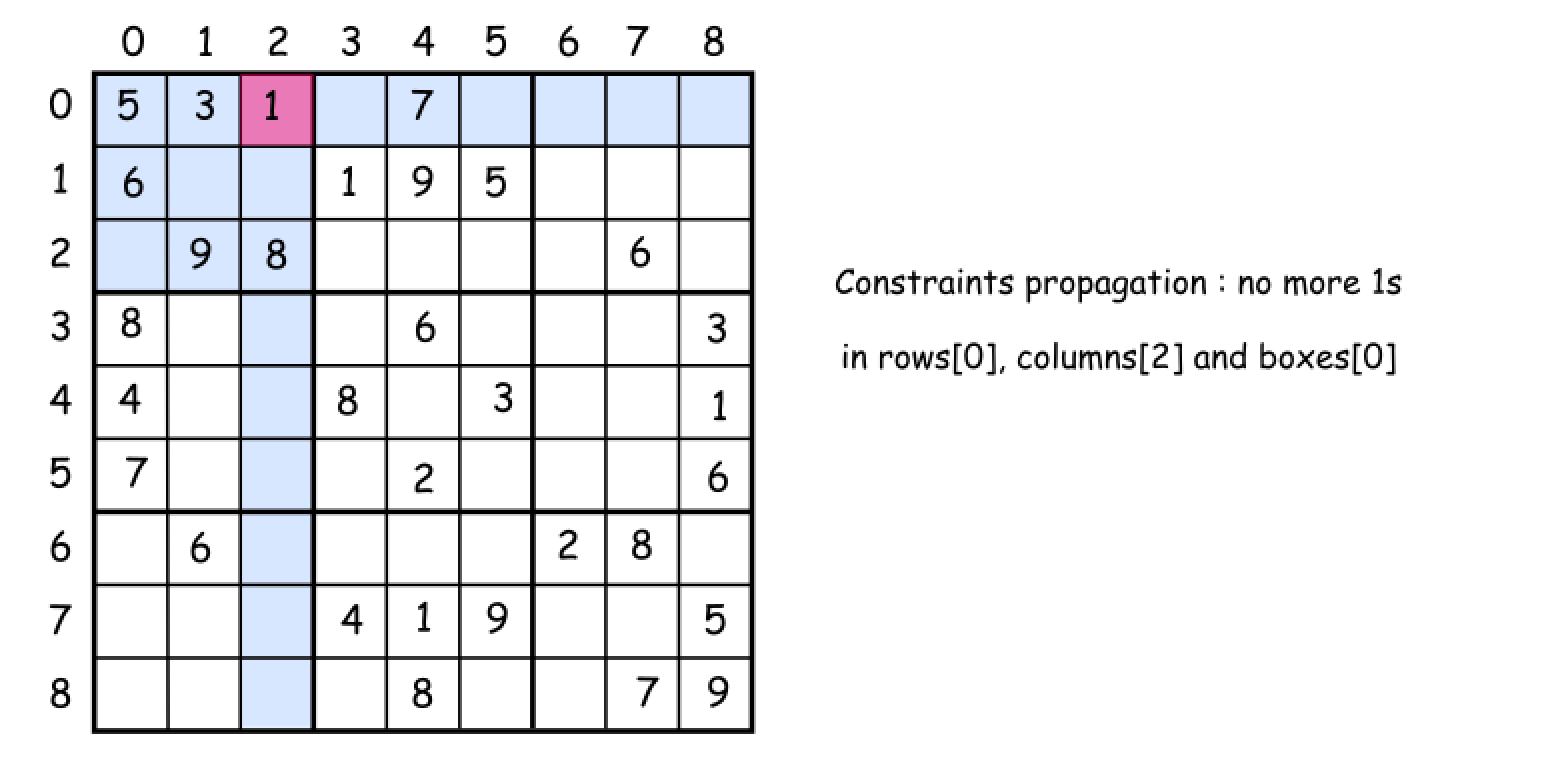
Approach 1: Backtracking

**Conceptions to use**

There are two programming conceptions here which could help.

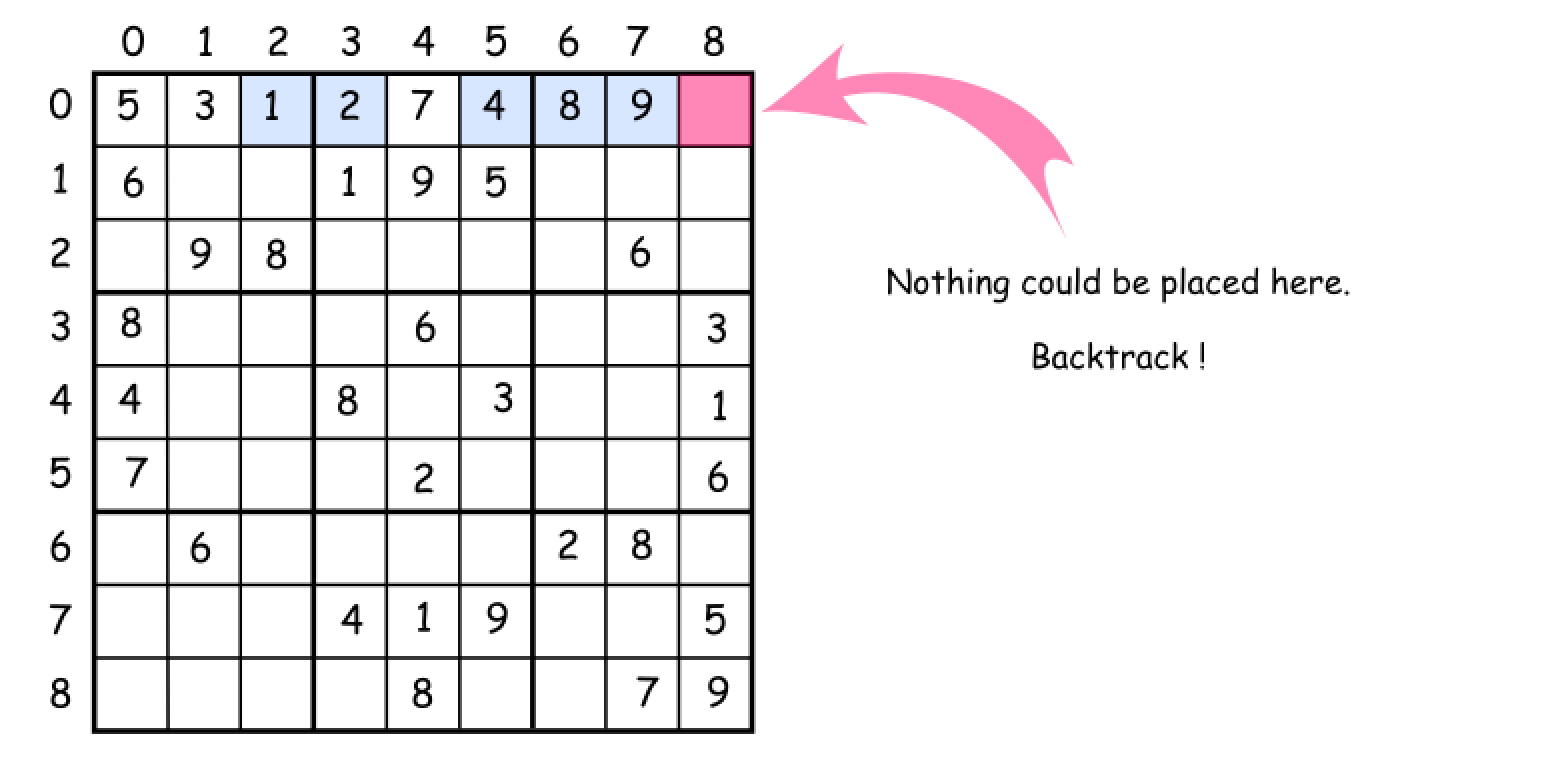
The first one is called *constrained programming*.

That basically means to put restrictions after each number placement. One puts a number on the board and that immediately excludes this number from further usage in the current *row*, *column* and *sub-box*. That propagates *constraints* and helps to reduce the number of combinations to consider.



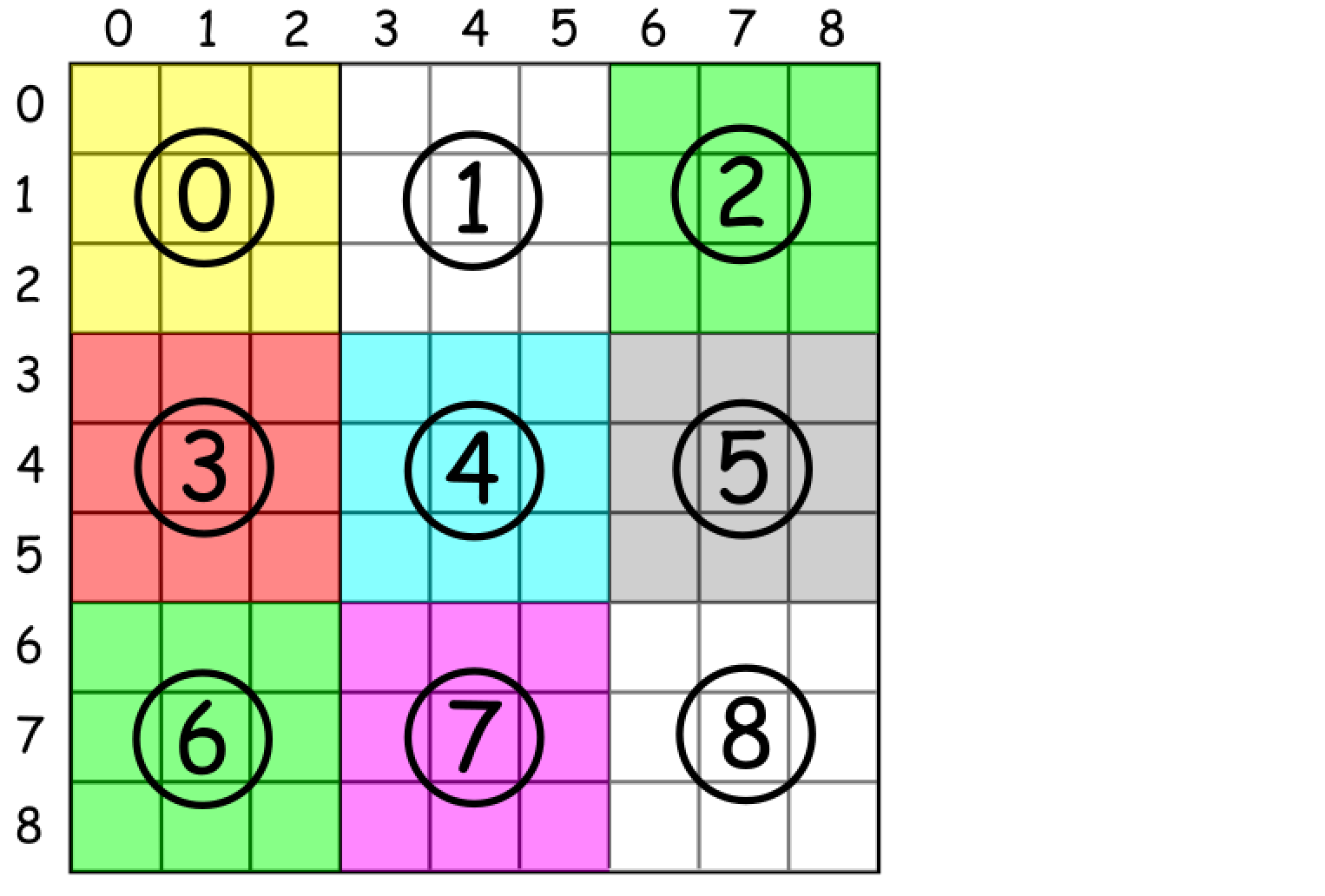
The second one called *backtracking*.

Let's imagine that one has already managed to put several numbers on the board. But the combination chosen is not the optimal one and there is no way to place the further numbers. What to do? *To backtrack*. That means to come back, to change the previously placed number and try to proceed again. If that would not work either, *backtrack* again.



**How to enumerate sub-boxes**

One tip to enumerate sub-boxes: let's use box\_index = (row / 3) \* 3 + column / 3 where / is an integer division.



**Algorithm**

Now everything is ready to write down the backtrack function backtrack(row = 0, col = 0).

* Start from the upper left cell row = 0, col = 0. Proceed till the first free cell.
* Iterate over the numbers from 1 to 9 and try to put each number d in the (row, col) cell.
  + If number d is not yet in the current row, column and box :
    - Place the d in a (row, col) cell.
    - Write down that d is now present in the current row, column and box.
    - If we're on the last cell row == 8, col == 8 :
      * That means that we've solved the sudoku.
    - Else
      * Proceed to place further numbers.
    - Backtrack if the solution is not yet here : remove the last number from the (row, col) cell.