

Scanner

Selected Fun Problems of the ACM Programming Contest SS25

04./05.06.2025

David Knöpp

About questions

2 / 60

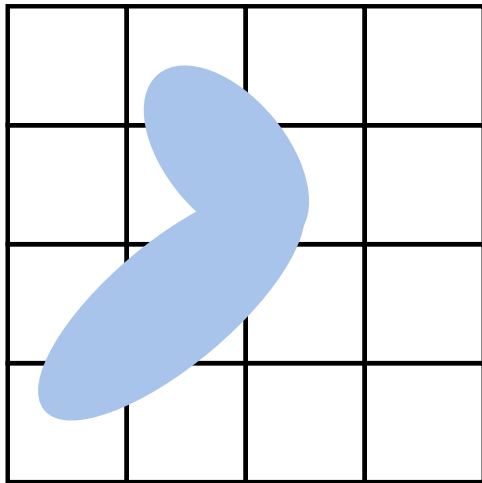
- Please ask questions if you don't understand an important aspect
- Otherwise, please save your questions for the discussion after the presentation

Contents

3 / 60

1 Problem Definition	4
2 Solution	14
3 Discussion	50
4 Summary	59

1 Problem Definition



Result Matrix

	1	1	
	1	1	
1	1	1	
1	1		

Result Matrix

7 / 60

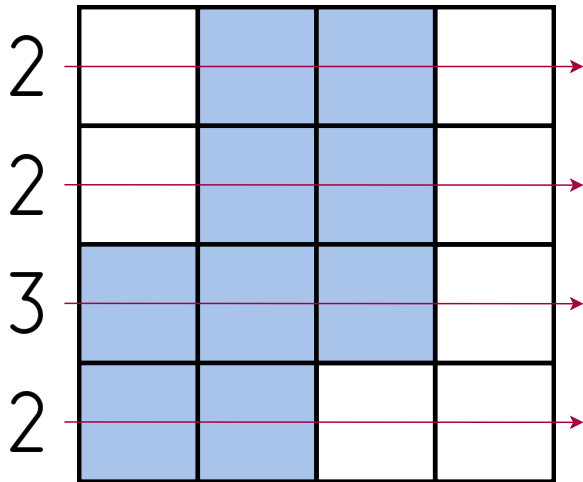

```
. # # .  
. # # .  
# # # .  
# # . .
```

This is our actual output.

But what is our input?

Getting the input

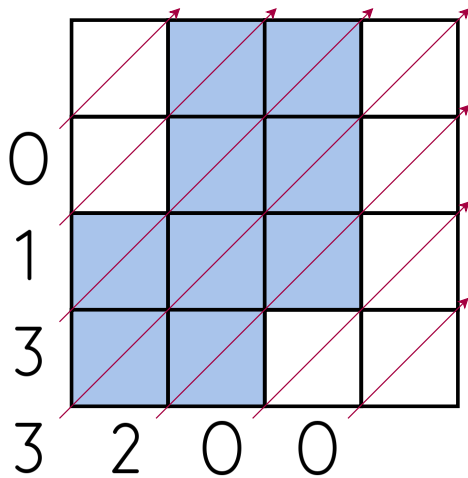
8 / 60



2 2 3 2

Getting the input

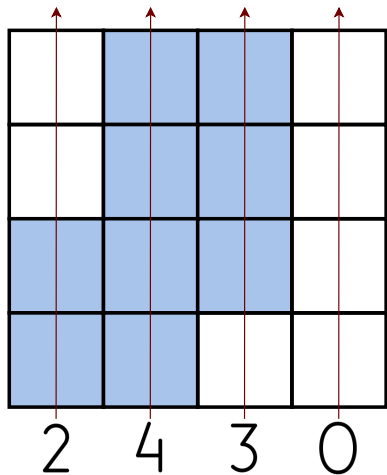
9 / 60



```
2 2 3 2
0 1 3 3 2 0 0
```

Getting the input

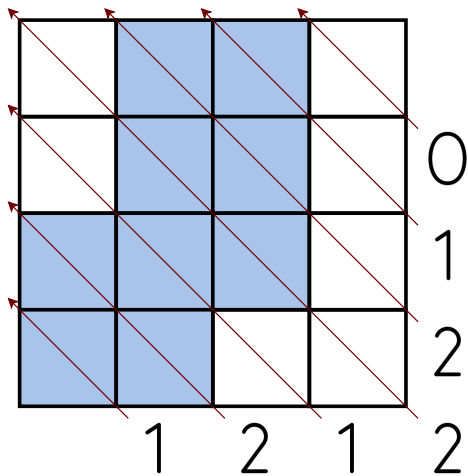
10 / 60



```
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
```

Getting the input

11 / 60



2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0

Getting the input

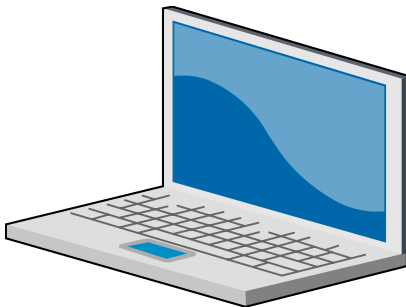
12 / 60

The number at the top represents the number of matrices that will follow.

In our case, it's just one.

```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

And that's our input!



2 Solution

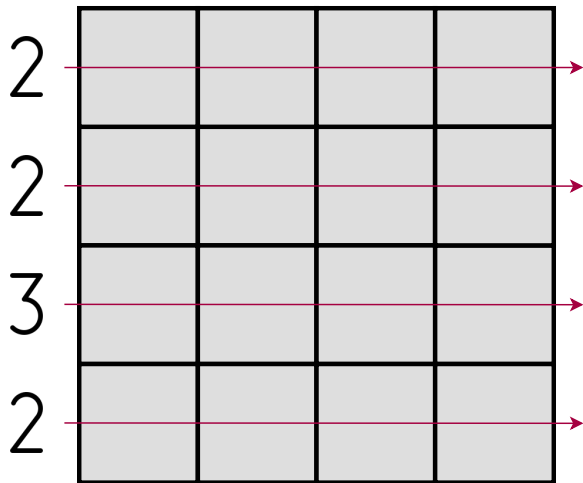
Tools

15 / 60

- Python
 - Personal experience
- Numpy
 - Convenient and efficient matrix operations
 - Personal experience

Example

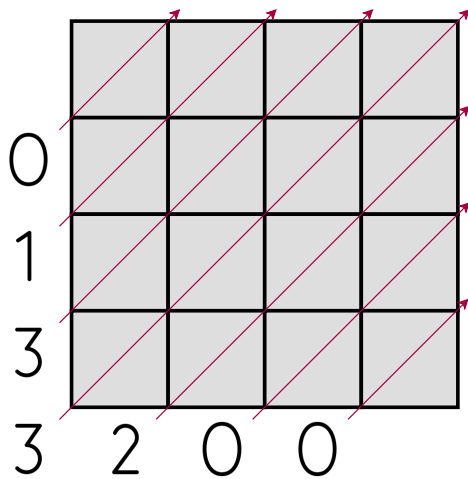
16 / 60



1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0

Example

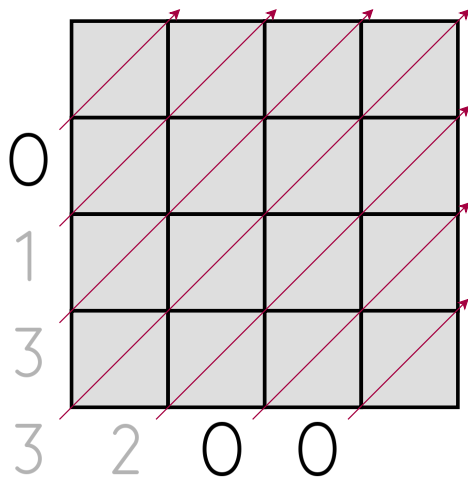
17 / 60



```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Example

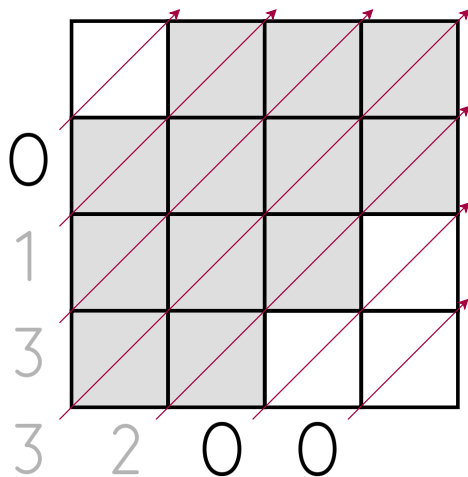
18 / 60



1						
2	2	3	2			
0	1	3	3	2	0	0
2	4	3	0			
1	2	1	2	2	1	0

Example

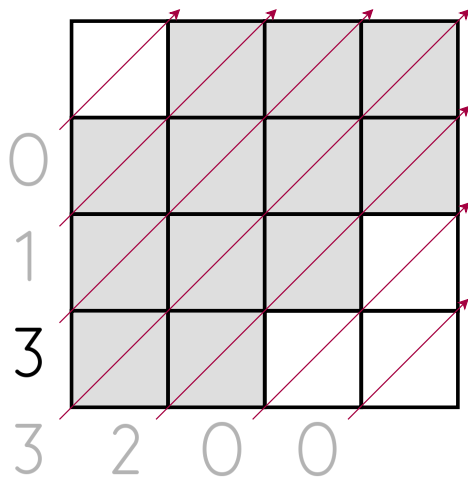
19 / 60



1						
2	2	3	2			
0	1	3	3	2	0	0
2	4	3	0			
1	2	1	2	2	1	0

Example

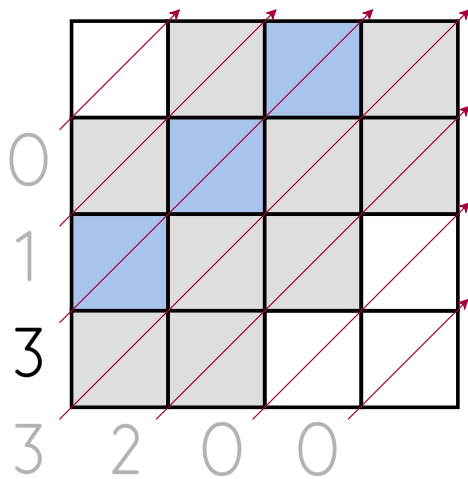
20 / 60



```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Example

21 / 60



```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

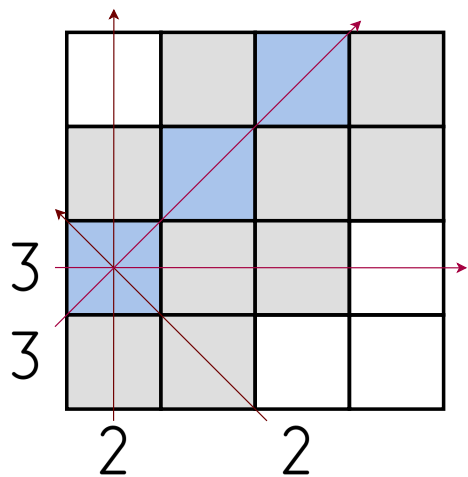
Example

22 / 60


```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Example

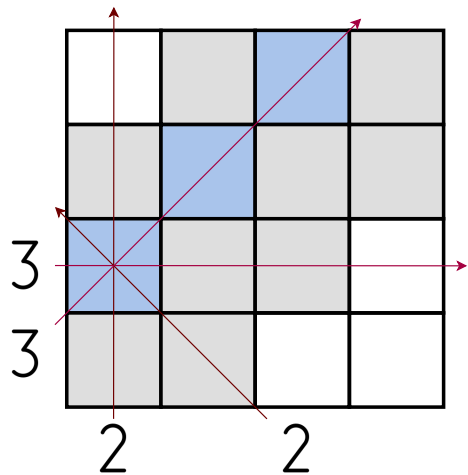
23 / 60



```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Example

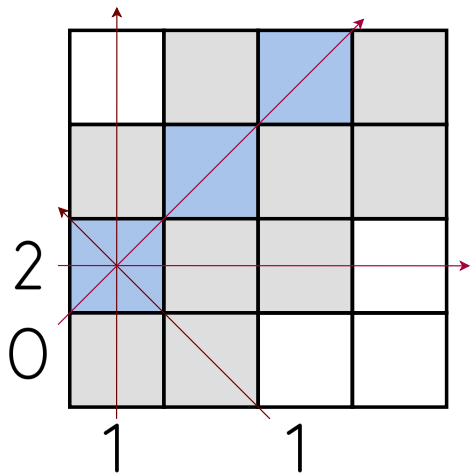
24 / 60



```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```


Example

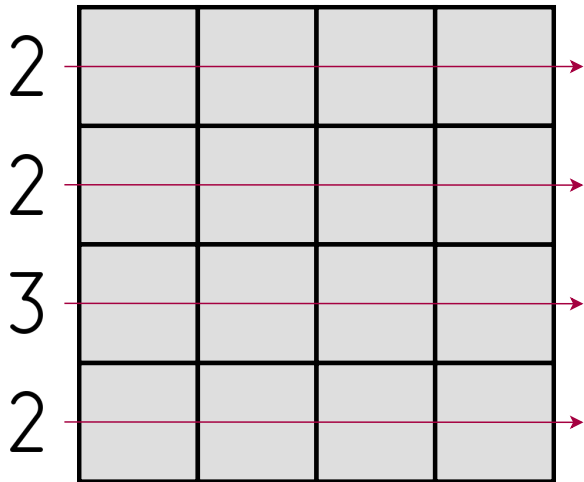
25 / 60



1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0

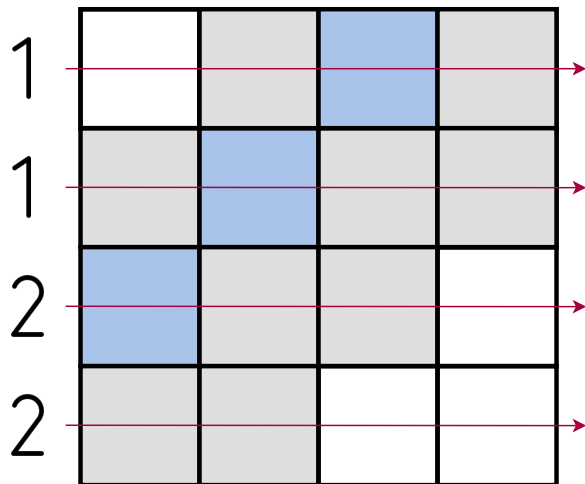
Example

26 / 60



Example

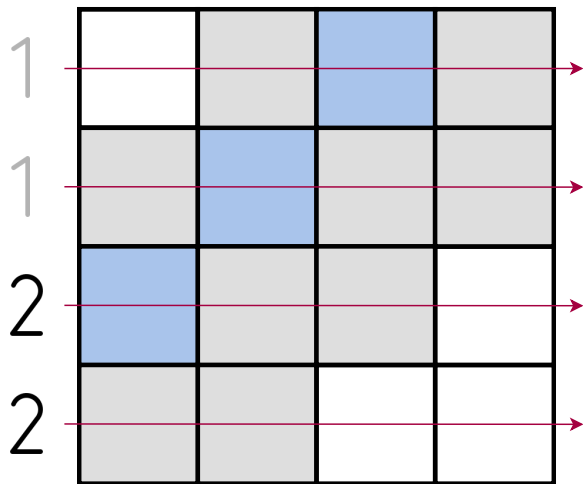
27 / 60



1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0

Example

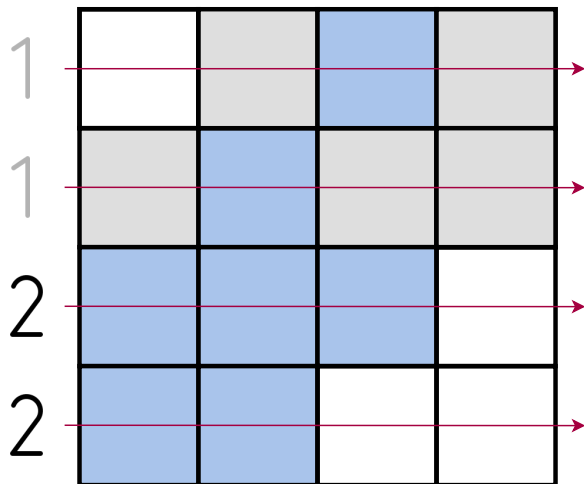
28 / 60



```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example

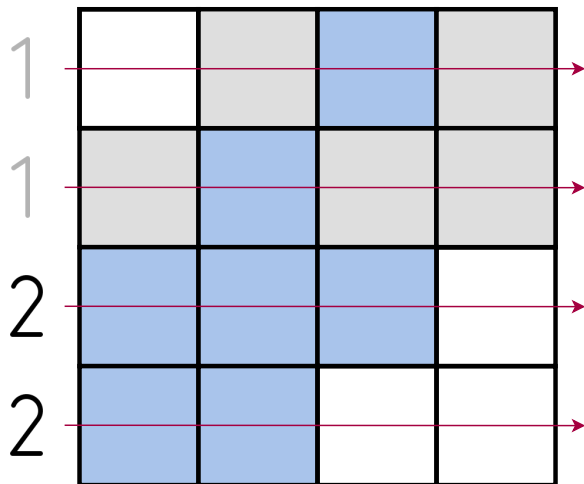
29 / 60



```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example

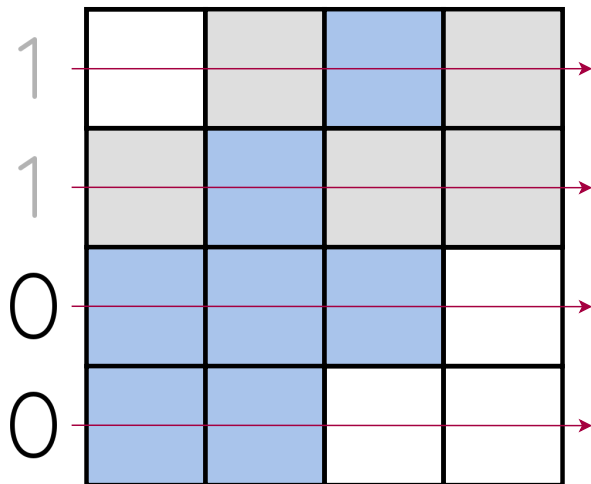
30 / 60



```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example

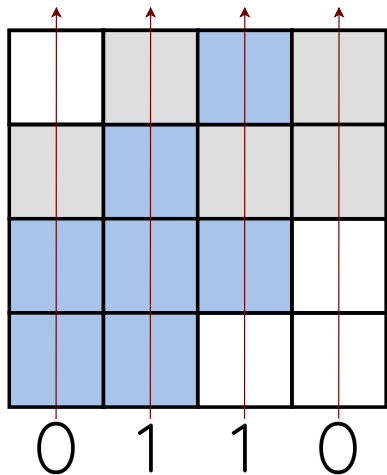
31 / 60



```
1
1 1 0 0
0 1 0 1 0 0 0
0 1 1 0
0 0 0 0 2 0 0
```

Example

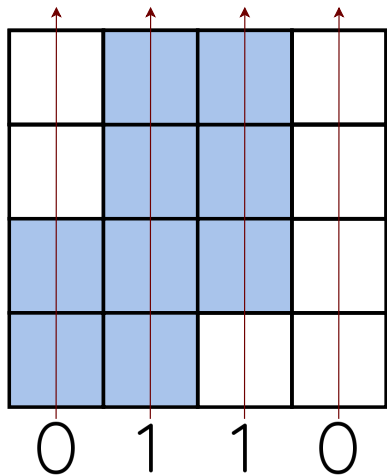
32 / 60



```
1
1 1 0 0
0 1 0 1 0 0 0
0 1 1 0
0 0 0 0 2 0 0
```


Example

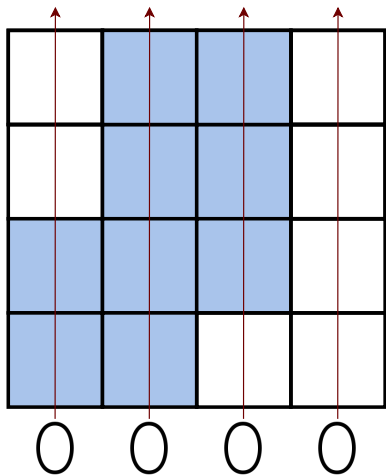
33 / 60



```
1
1 1 0 0
0 1 0 1 0 0 0
0 1 1 0
0 0 0 0 2 0 0
```

Example

34 / 60



1

0 0 0 0

0 0 0 0 0 0 0

0 0 0 0

0 0 0 0 0 0 0

Example

35 / 60

1
0 0 0 0
0 0 0 0 0 0 0
0 0 0 0
0 0 0 0 0 0 0

Code Snippets

36 / 60

- Snippets of the actual Python-program
- Highly simplified - treat it like pseudo-code

```
def compare_and_fill(sensor_data_point, arr):  
    # number of unassigned elements  
    n_of_unassigned = n_of_unassigned(arr)  
  
    # Declare all unassigned as empty  
    if sensor_data_point == 0:  
        for cell in arr:  
            if cell == UNASSIGNED:  
                cell = EMPTY  
  
    # Declare all unassigned as full  
    elif sensor_data_point == n_of_unassigned:  
        for cell in arr:  
            if cell == UNASSIGNED:  
                cell = FULL  
                update_sensor_data(cell.x, cell.y)
```

```
while(not is_done()):  
    diag_lr = get_diagonal_lr(matrix)  
    diag_rl = get_diagonal_rl(matrix)  
  
    # Horizontals  
    for i in range(height):  
        compare_and_fill(sensor_data_horizontal[i], matrix[i,:])  
  
    # LR-Diags  
    for i in range(height + width - 1):  
        compare_and_fill(sensor_data_diagonal_lr[i], diag_lr[i])  
  
    # Verticals  
    for i in range(width):  
        compare_and_fill(sensor_data_vertical[i], matrix[:,i])  
  
    # RL-Diags  
    for i in range(height + width - 1):  
        compare_and_fill(sensor_data_diagonal_rl[i], diag_rl[i])
```

Are we done?

- Does the algorithm always find a solution?
- What if there is no solution?
- What if there are multiple solutions?

Are we done?

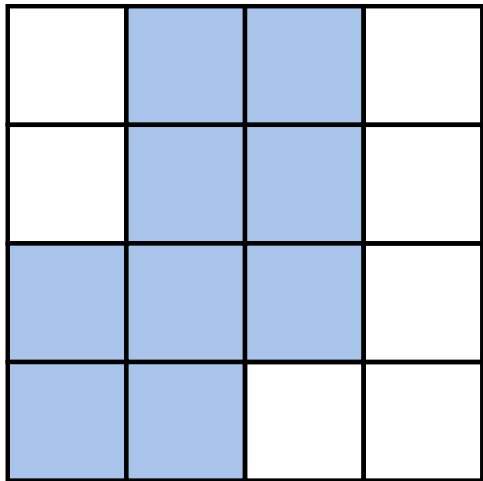
- Does the algorithm always find a solution?
- What if there is no solution?
- What if there are multiple solutions?

Let's first answer a different question:

- How do we know whether we found a solution?

How do we know whether we found a solution?

41 / 60



```
1
0 0 0 0
0 0 0 0 0 0 0
0 0 0 0
0 0 0 0 0 0 0
```

All input_datas-entries are zero

Invalid state

42 / 60


```
1
0 0 0 0
0 0 0 0 0 0 0
0 1 0 0
0 0 0 0 0 0 0
```

Every cell is assigned, but there is one input_data-entry left!

⇒ contradiction

Termination Condition

1. Check whether all `input_data` is zero

```
def is_data_used(sensor_data):  
    return not np.any(sensor_data != 0)
```

2. Check whether all cells are assigned

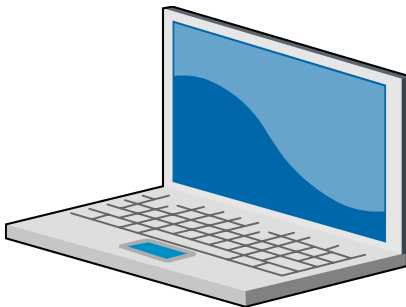
```
def is_all_assigned(matrix):  
    return not np.any(matrix.cell == UNASSIGNED)
```

3. Check whether we're done

```
def is_done():  
    return is_data_used(sensor_data) or is_all_assigned(matrix)
```

Demo: No Solution

44 / 60



Multiple solutions

- We always only assign `EMPTY` or `FULL` to a cell if we know it's state for certain
- Therefore, with our current method, we can only detect certain solutions
- If multiple solutions exist for a given input, we should get stuck

Multiple Solutions: Local Search

46 / 60

```
# value is either FULL or EMPTY
def search_in_branch(idx, value, matrix, sensor_data):
    # save old data
    old_matrix = matrix.copy()
    old_sensor_data = sensor_data.copy()

    # assign variable
    matrix[idx] = value
    if value == FULL:
        update_sensor_data(idx[1], idx[0])
        fill_loop()

    # re-assign old data
    matrix = old_matrix
    sensor_data = old_sensor_data
```

Multiple Solutions: Local Search

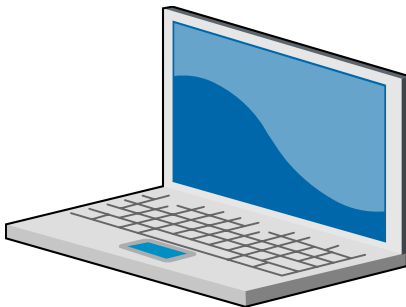
47 / 60

```
# ... inside fill_loop()
if not has_change_occured:
    # indices of unassigned fields
    indices_of_unassigned = np.argwhere(matrix.cell == UNASSIGNED)

    for idx in indices_of_unassigned:
        # recursive calls
        for assignment in [EMPTY, FULL]:
            search_in_branch(idx, value, matrix, sensor_data)
            if solutions_found > 1:
                # the solution is ambiguous -> leave loop
                return
```

Demo: Local Search

48 / 60



Are we done?

Does the algorithm always find a solution?

Answer: No!

- A full assignment is valid only if all entries of `sensor_data` become zero
- We can get stuck \Rightarrow perform local search

What if there is no solution?

Answer: Data will be contradictory

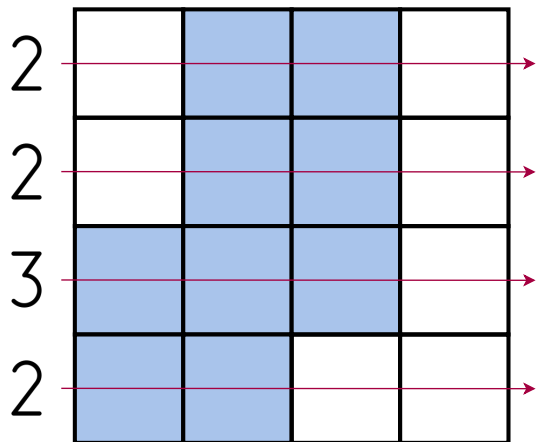
What if there are multiple solutions?

Answer: We get stuck \Rightarrow perform local search

3 Discussion

Interpretation as Linear System of Equations

51 / 60



→

$$2 = 0 + 1 + 1 + 0$$

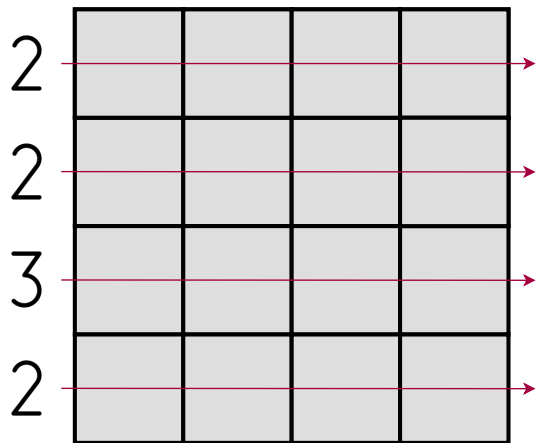
$$2 = 0 + 1 + 1 + 0$$

$$3 = 1 + 1 + 1 + 0$$

$$2 = 1 + 1 + 0 + 0$$

Interpretation as Linear System of Equations

52 / 60



→

More general:

$$2 = x_0 + x_1 + x_2 + x_3$$

$$2 = x_4 + x_5 + x_6 + x_7$$

$$3 = x_8 + x_9 + x_{10} + x_{11}$$

$$2 = x_{12} + x_{13} + x_{14} + x_{15}$$

Interpretation as Linear System of Equations

For a matrix of dimension $(n \times n)$, we get

- 2^n variables $x_0, \dots, x_{2^n-1} \in \{0, 1\}$
- $n + n + (2n - 1) + (2n - 1) = 6n - 2$ linearly independent equations

Interpretation as Linear System of Equations

For a matrix of dimension $(n \times n)$, we get

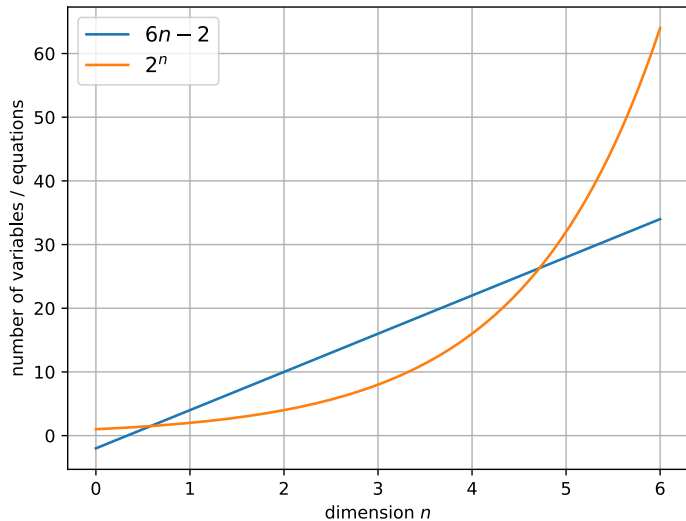
- 2^n variables $x_0, \dots, x_{2^n-1} \in \{0, 1\}$
- $n + n + (2n - 1) + (2n - 1) = 6n - 2$ linearly independent equations

Question: For which $n \in \mathbb{N}$ is the linear system of equations underdetermined?

\Rightarrow Solve $2^n > 6n - 2$

Interpretation as Linear System of Equations

55 / 60



Interpretation as Linear System of Equations

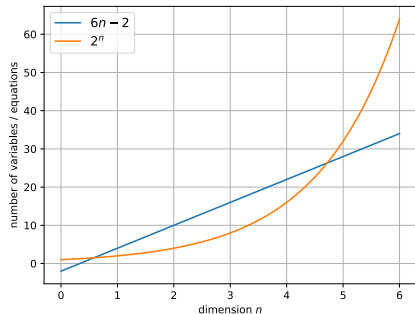
56 / 60

Theory indicates:

- For all $n \geq 5$, the LSE is underdetermined \Rightarrow Possibly no solution

Experiment confirms this!

- For all valid inputs of $n < 5$, a solution is found
- For some $n \geq 5$, the algorithm gets stuck (we have seen an example) \Rightarrow perform local search



Why it's a good algorithm for this problem

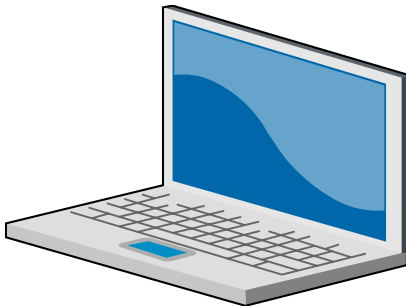
The problem definition states that we are working with a “body” scanner

- Matrix property: Most FULL cells are located next to each other
- Our algorithm uses this property

⇒ Most inputs can be solved in sub-exponential time

Demo: Chunk Inputs

58 / 60



4 Summary

Summary

60 / 60

Input:

```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Output:


```
def compare_and_fill(sensor_data_point, arr):
    # number of unassigned elements
    n_of_unassigned = n_of_unassigned(arr)
    # Declare all unassigned as empty
    if sensor_data_point == 0:
        for cell in arr:
            if cell == UNASSIGNED:
                cell = EMPTY

    # Declare all unassigned as full
    elif sensor_data_point == n_of_unassigned:
        for cell in arr:
            if cell == UNASSIGNED:
                cell = FULL
                update_sensor_data(cell.x, cell.y)
```