Scanner

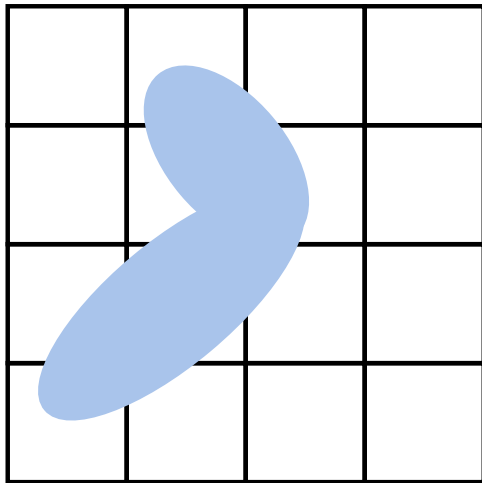**Selected Fun Problems of the ACM Programming Contest SS25**
04.06.2025

David Knöpp

About questions

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

1 Problem Definition

Result Matrix


```
.  #  #  .
.  #  #  .
#  #  #  .
#  #  .  .
```
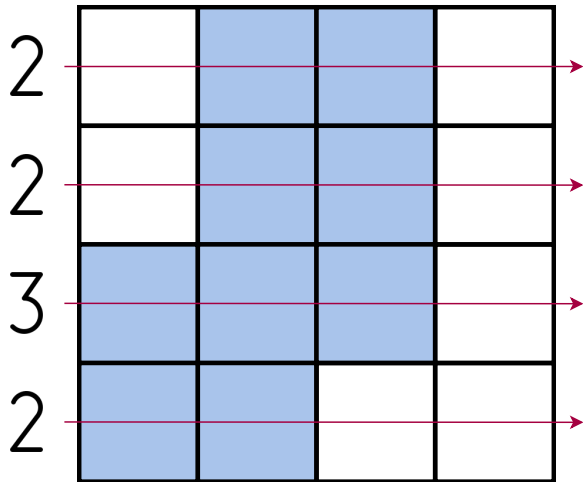
This is our actual output.
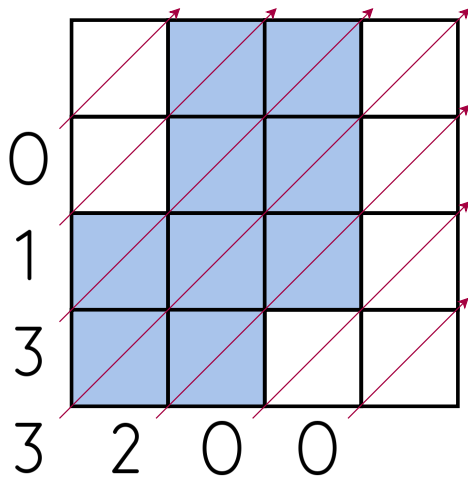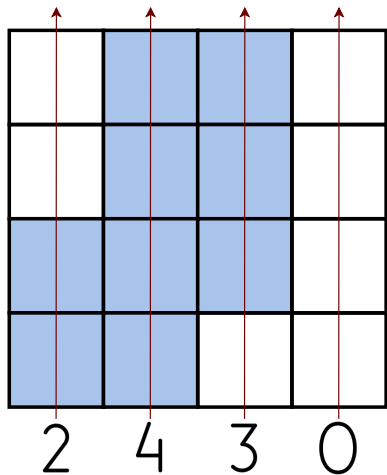
But what is our input?

2 2 3 2

Getting the input


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

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

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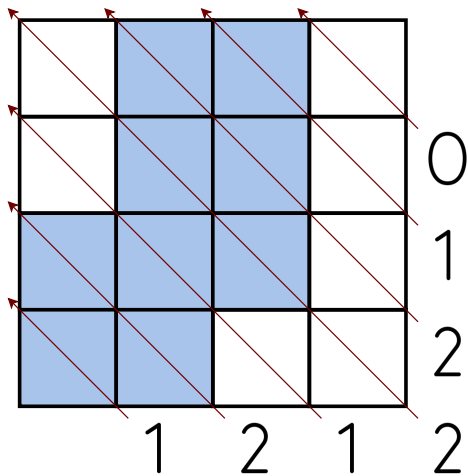
```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

2 Solution

Tools

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

Example

15 / 57


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

Example

16 / 57


```
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 / 57


```
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 / 57


```
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 / 57


```
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 / 57
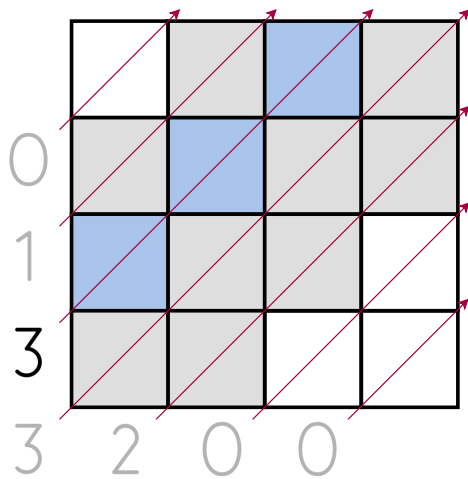
```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

Example **21 / 57**


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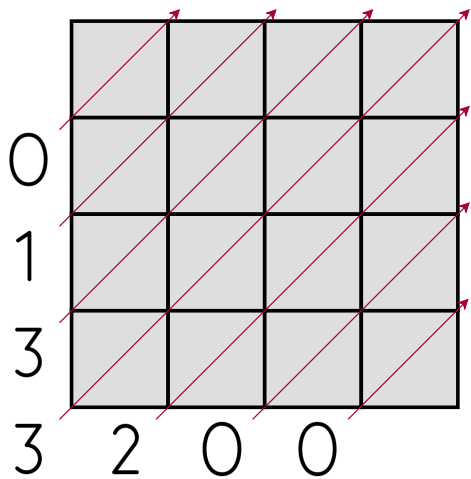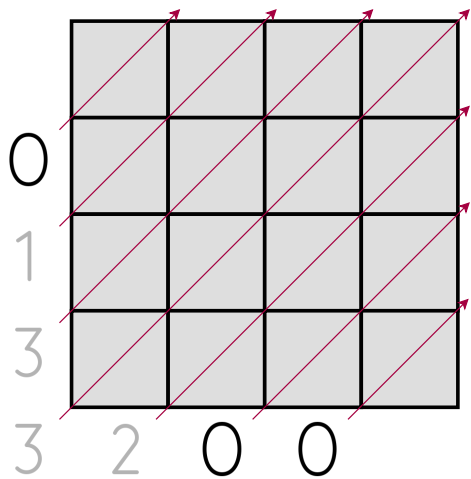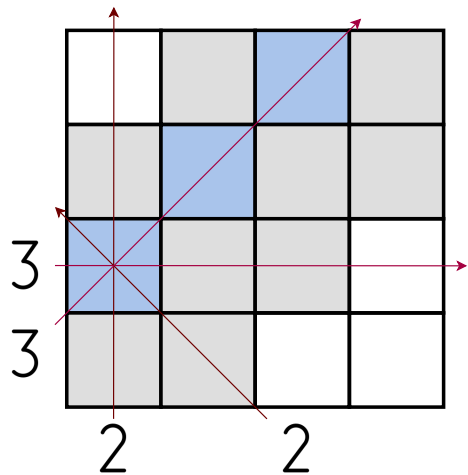
Example


```
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 / 57
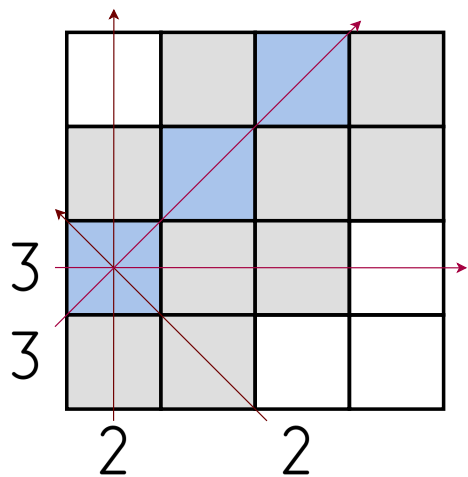

```
1
2 2 3 2
0 1 3 3 2 0 0
2 4 3 0
1 2 1 2 2 1 0
```

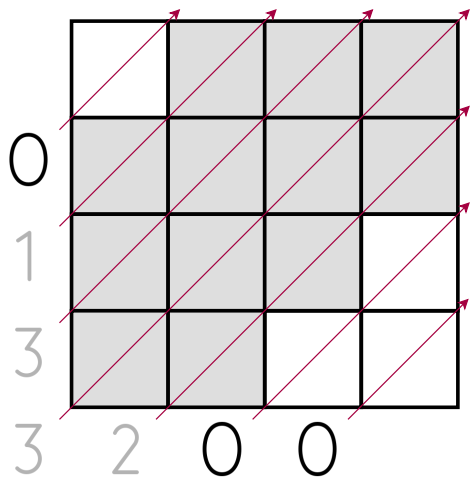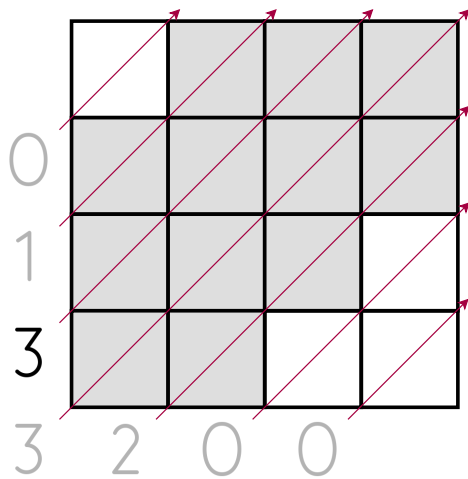# Example

24 / 57
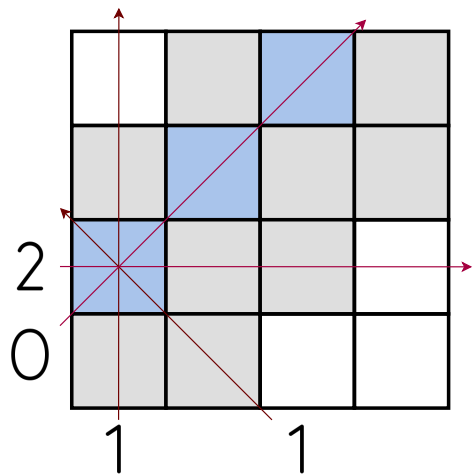

```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example 25 / 57
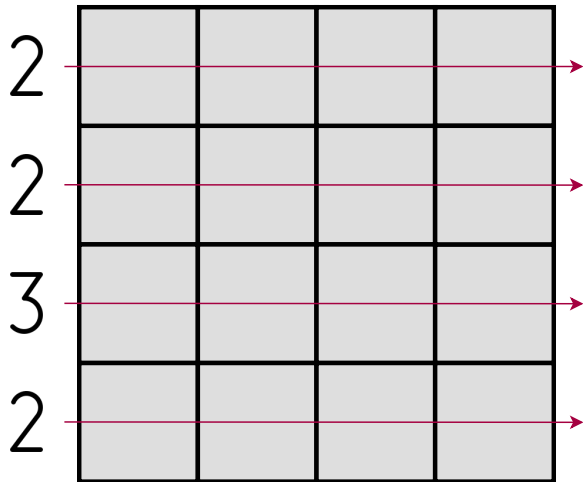
Example

26 / 57
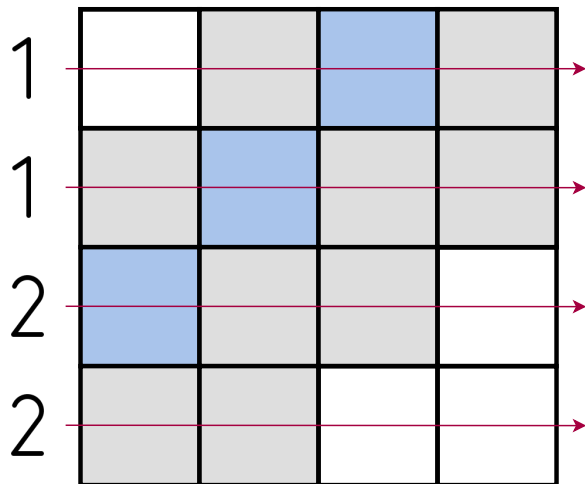

```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example 27 / 57
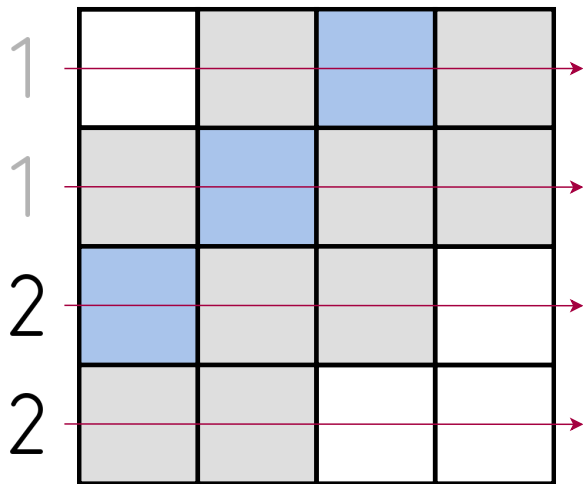

```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example

28 / 57
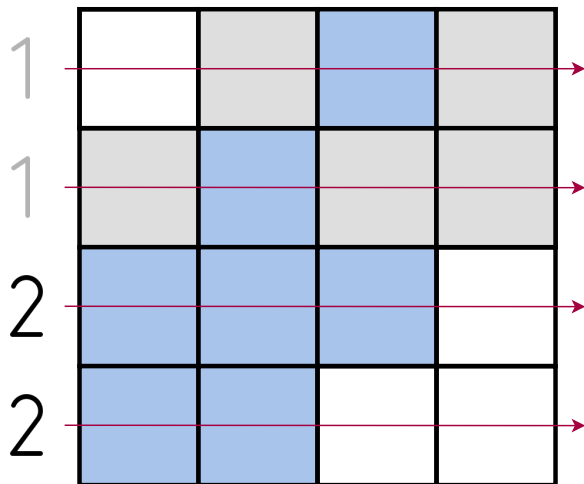

```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```

Example

29 / 57
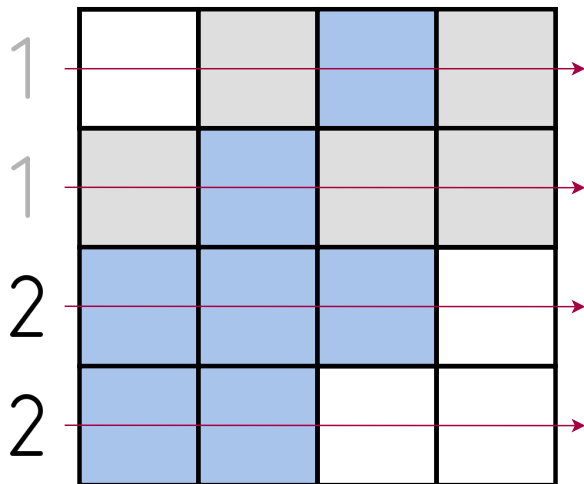

```
1
1 1 2 2
0 1 0 3 2 0 0
1 3 2 0
1 1 1 1 2 0 0
```
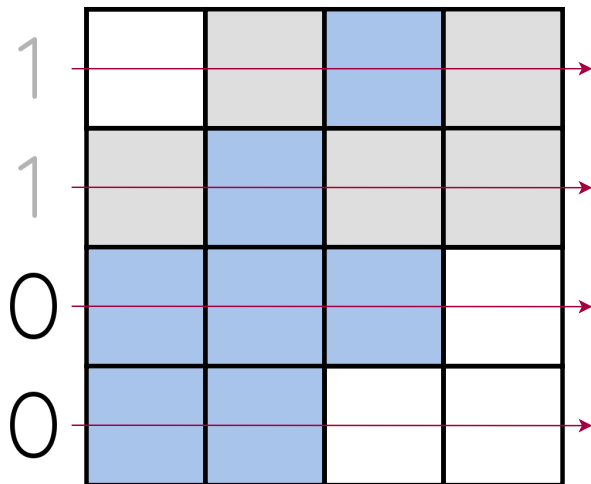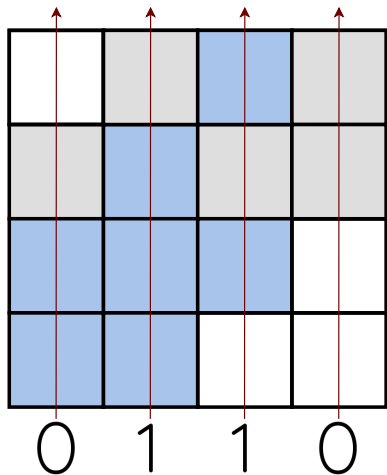
Example


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

Example


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

Example


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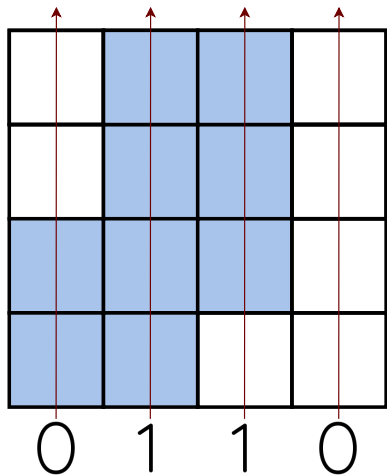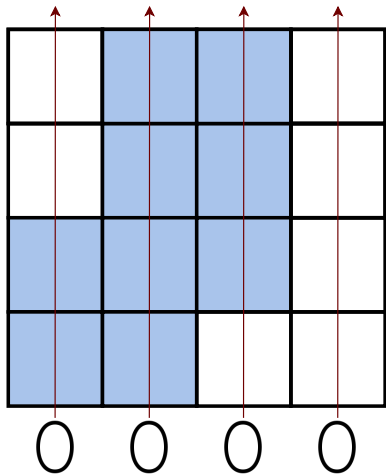
Example


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

Example


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

```python
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)
```

```python
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 `compare_and_fill` always find a solution?
- What if there is no solution?
- What if there are multiple solutions?

Are we done?

- Does `compare_and_fill` 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?
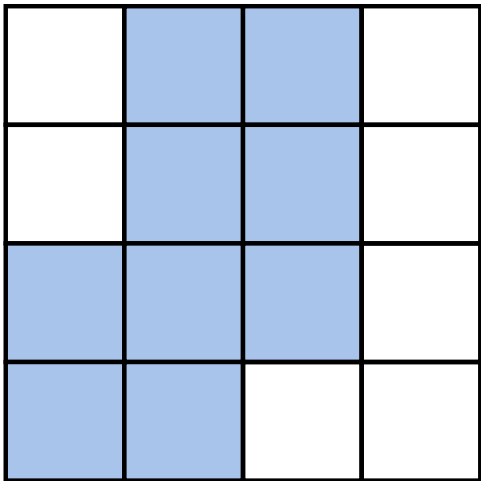
```
1
0 0 0 0
0 0 0 0 0 0 0
0 0 0 0
0 0 0 0 0 0 0
```

All input_datas-entries are zero

Invalid state


```
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

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

2. Check whether all cells are assigned

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

3. Check whether we are done

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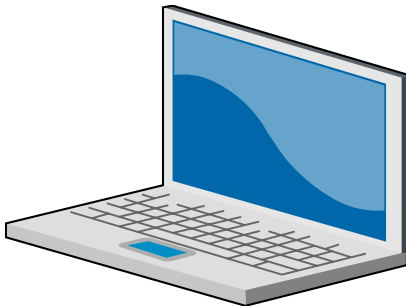
Multiple solutions

- We assign `EMPTY` or `FULL` **only if** we know a state for certain
- $\Rightarrow$ with `compare_and_fill`, we can only detect **single** solutions
- If multiple solutions exist for a given input $\Rightarrow$ get stuck

```python
# 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
```

```python
# ... 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
```
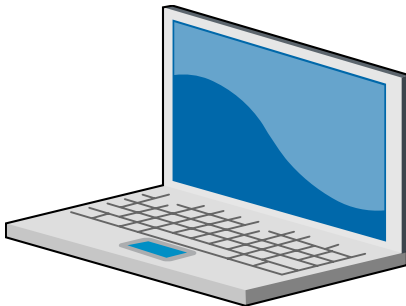
Conclusion

Does the algorithm always find a solution?
Answer: If it exists: Yes! But...

• ... 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

Time Complexity

- Take a matrix of dimension $(n \times m)$
- Worst case: Local search from the beginning
- Worst case time complexity: $\mathcal{O}(2^{n \cdot m})$

- Problem Definition: **Body** Scanner
- ⇒ Matrix property: Most FULL cells are neighbored

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

- Problem Definition: **Body** Scanner
- ⇒ Matrix property: Most FULL cells are neighbored

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

Time Complexity: Using matrix property

- Problem Definition: **Body** Scanner
- ⇒ Matrix property: Most FULL cells are neighbored

- compare_and_fill uses this property.
- Reduces search-space significantly.

```
. . . . . . . . . # # . . . . .
. . . . . . . . # # # # # # .
. . . . . . . # # # # # . . .
. . . . . . . # # # # . . # .
. . . . . . # # # # # # # # #
. . . . . # # # # # # # # # #
. . . . . # # # # # # # # # #
. . . . . # . . # # # # # # #
. . . . . . . . . # # # . . .
. . . . . . . . . . # . . . .
```
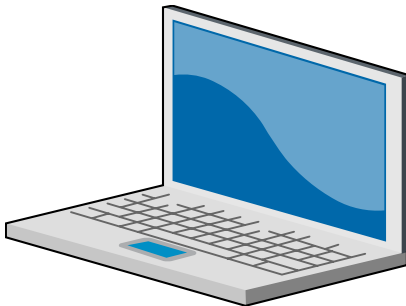
Algorithm Justification

Experiment:

1. Generate 10000 random inputs of dimension (10×15)
2. Run scanner.py: Abort after $t_{\max} = 0.01s$
3. Measure number of timeouts

Algorithm Justification

Experiment:

1. Generate 10000 random inputs of dimension (10×15)
2. Run `scanner.py`: Abort after $t_{\max} = 0.01s$
3. Measure number of timeouts

Results:

matrix type	timeout-rate
random	99.69%
chunk	1.47%

Algorithm Justification

Experiment:

1. Generate 10000 random inputs of dimension (10×15)
2. Run `scanner.py`: Abort after $t_{\max} = 0.01s$
3. Measure number of timeouts

Results:

matrix type	timeout-rate
random	99.69%
chunk	1.47%

$\Rightarrow$ Most inputs can be solved in sub-exponential time!

Summary

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:


```python
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)
```