Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Running a Maze

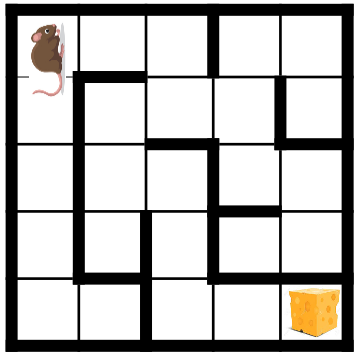
We present a systematic top-down development of an entire program to Run a Maze. We start from the beginning, but reference previous discussions from Chapters 1 and 4.

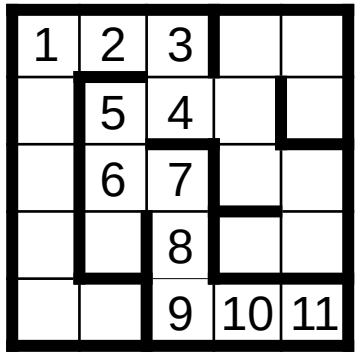
The main themes presented are:

- Use of a class to encapsulate a data representation.
- Consideration of alternative data representations.
- Structuring a program as two modules in a client/server relationship.
- The practice of information hiding.
- Incremental testing.
- Self-testing code.
- Exhaustive bounded testing of code.

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.

Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs “Unreachable” otherwise. A path is direct if it never visits any cell more than once.





Establish a framework:

```
class RunMaze:
    """
    Rat running.

    # Methods.
    main(cls) -> None

    # See also.
    Chapter 15 of Principled Programming
    """

    ________
```

☞ **Program top-down, outside-in.**

Establish a framework:

```
class RunMaze:
    """..."""

    @classmethod
    def main(cls) -> None:
        """Run a maze given as input, if possible."""
        _________
```

☞ **Program top-down, outside-in.**

Establish a framework:

```
class RunMaze:
    """..."""

    @classmethod
    def main(cls) -> None:
        """Run a maze given as input, if possible."""
        #.Input.
        #.Compute.
        #.Output.
```

☞ **Start by writing a top-level decomposition of the solution.**

Establish a framework:

```
class RunMaze:
    """..."""

    @classmethod
    def main(cls) -> None:
        """Run a maze given as input, if possible."""
        #.Input a maze of arbitrary size, or output “malformed input”
        #   and stop if the input is improper. Input format: TBD.
        #.Compute a direct path through the maze, if one exists.
        #.Output the direct path found, or “unreachable” if there is
        #   none. Output format: TBD.
```

☞ **Repeatedly improve comments by relentless copy editing.**

Top-level Code Structure

Establish a framework:

```
class RunMaze:
    """..."""

    @classmethod
    def main(cls) -> None:
            """Run a maze given as input, if possible."""
        # Input a maze of arbitrary size, or output “malformed input”
        #   and stop if the input is improper. Input format: TBD.
        RunMaze._input()

        # Compute a direct path through the maze, if one exists.
        RunMaze._solve()

        # Output the direct path found, or “unreachable” if there is
        #    none. Output format: TBD.
        RunMaze._output()
```

☞ **Many short procedures are better than large blocks of code.**

Stubs: Create stubs for the methods that have been introduced, which you can do mindlessly.

```
class RunMaze:
    ...

    @classmethod
    def _input(cls) -> None:
        """Input a maze of arbitrary size, or output “malformed input” and
           stop if the input is improper. Input format: TBD.
        """
        pass
    @classmethod
    def _solve(cls) -> None:
        """Compute a direct path through the maze, if one exists."""
        pass
    @classmethod
    def _output(cls) -> None:
        """Output the direct path found, or “unreachable” if there is none. Output format: TBD."""
        pass
```

☞ **Don’t type if you can avoid it; clone. Cut and paste, then adapt.**

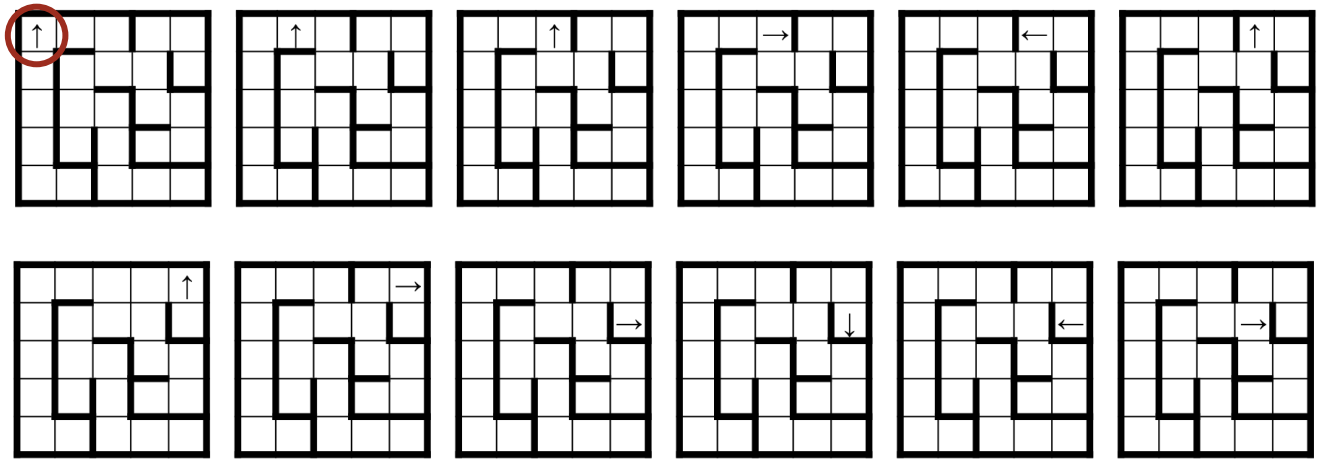
Names that begin with underscores are *protected* and internal to RunMaze; other classes should not access them. Although such access it is not denied, the underscores serve as a warning not to do so.

Stubs: Create stubs for the methods that have been introduced, which you can do mindlessly.

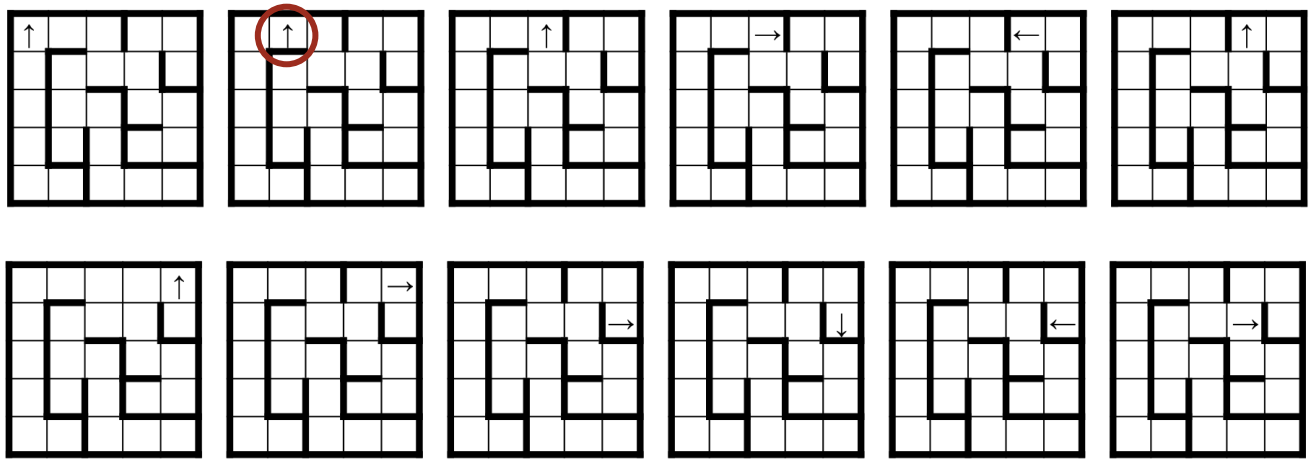
```
class RunMaze:
    ...

    @classmethod
    def _input(cls) -> None:
        """Input a maze of arbitrary size, or output "malformed input" and
           stop if the input is improper. Input format: TBD.
        """
        pass
    @classmethod
    def _solve(cls) -> None:
        """Compute a direct path through the maze, if one exists."""
        pass
    @classmethod
    def _output(cls) -> None:
        """Output the direct path found, or "unreachable" if there is none. Output format: TBD."""
        pass
```

☞ **Practice information hiding.**

Algorithm (from Chapter 4):



☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

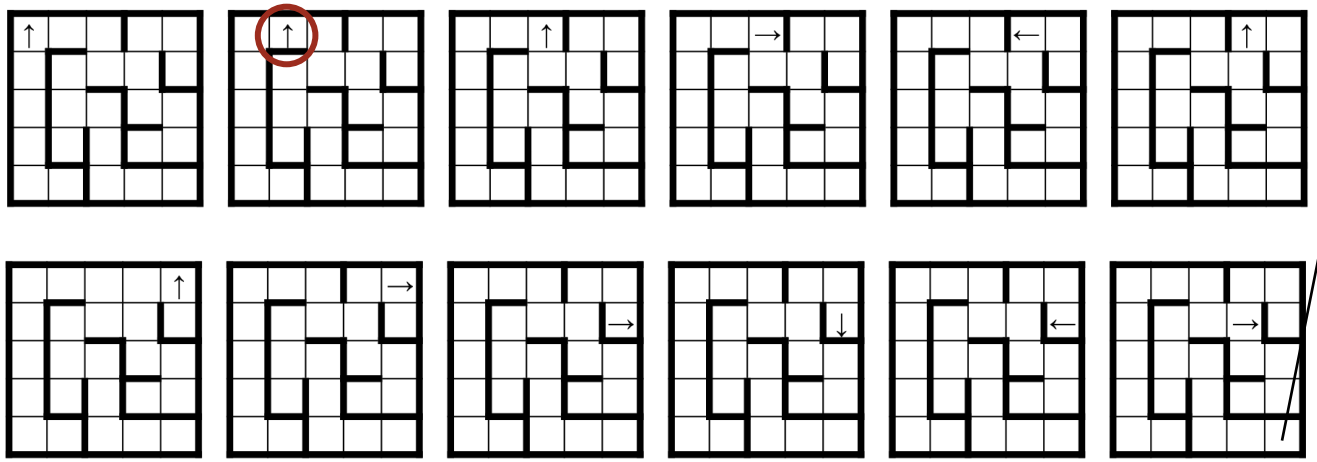


Sidestep

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

VARIANT:
Get closer to goal.

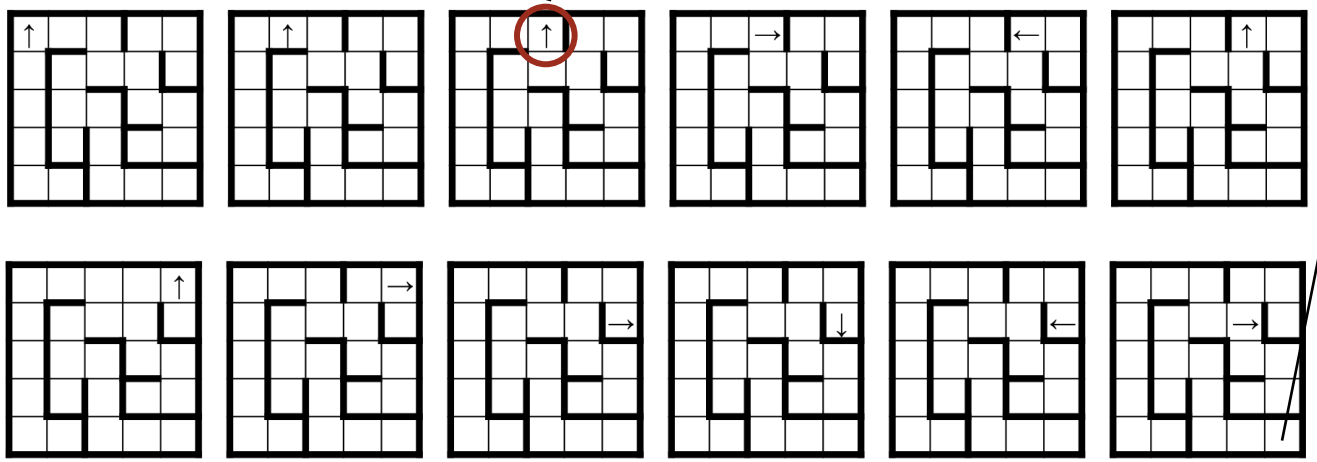


Sidestep

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

VARIANT:
Get closer to goal.

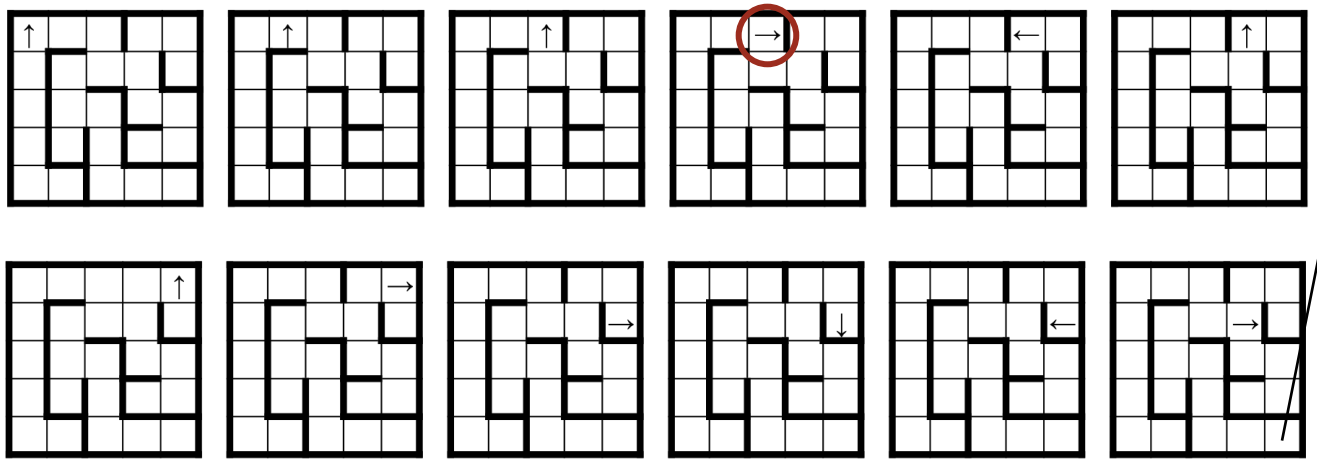


Sidestep

Algorithm

INVARIANT:
Left hand is on the interior surface of a peripheral wall.
"Peripheral" is not just "outer", but includes "attached" inner walls.

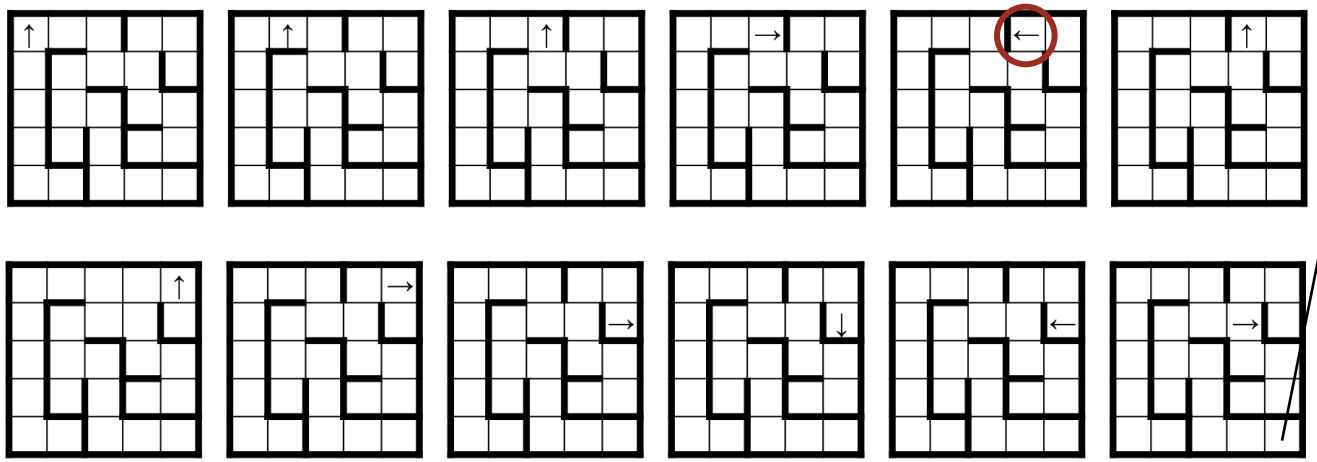
VARIANT:
Get closer to goal.



Turn convex corner

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

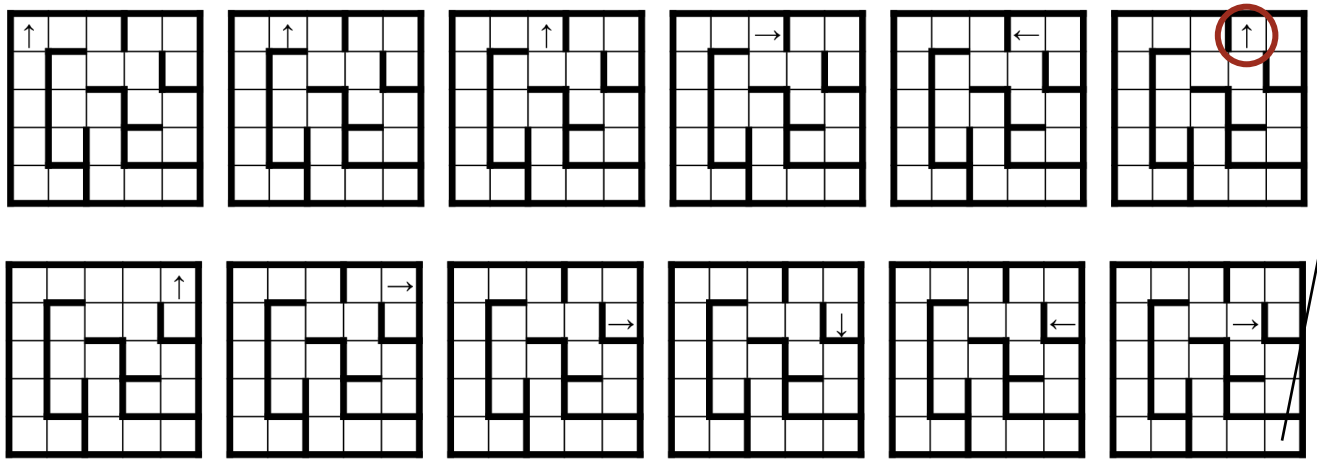
VARIANT:
Get closer to goal.



Pirouette to other side

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

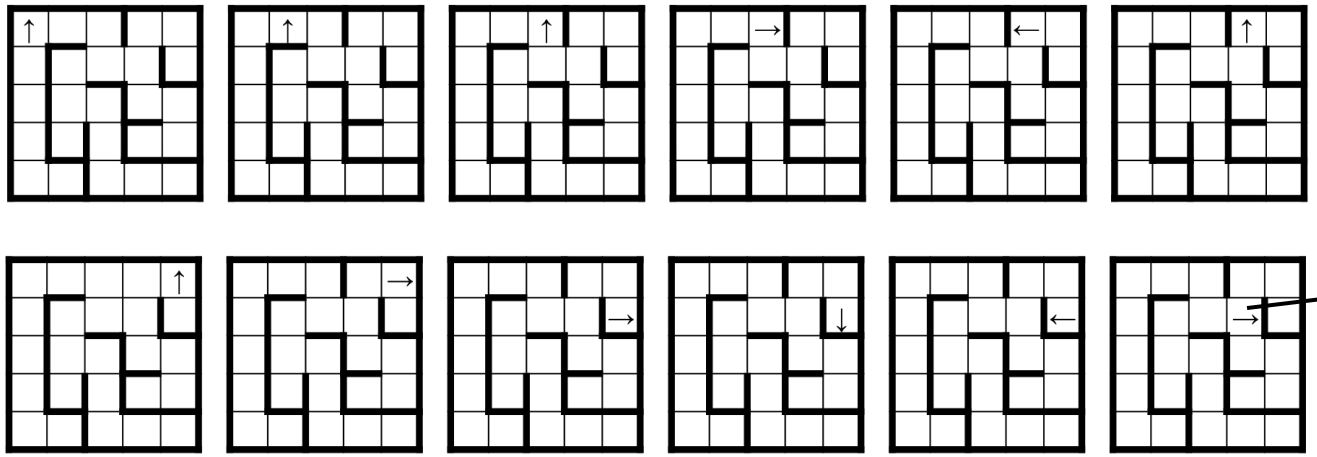
VARIANT:
Get closer to goal.



Turn convex corner

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

VARIANT:
Get closer to goal.

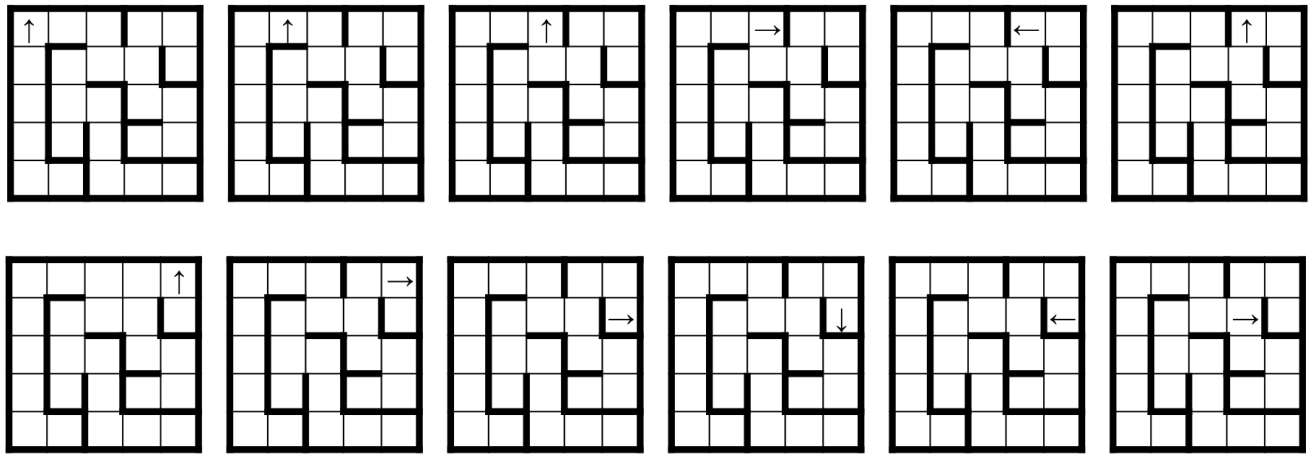


Actions:

- Sidestep
- Pirouette
- Turn convex corner
- (Turn concave corner)

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

VARIANT:
Get closer to goal.



Actions:

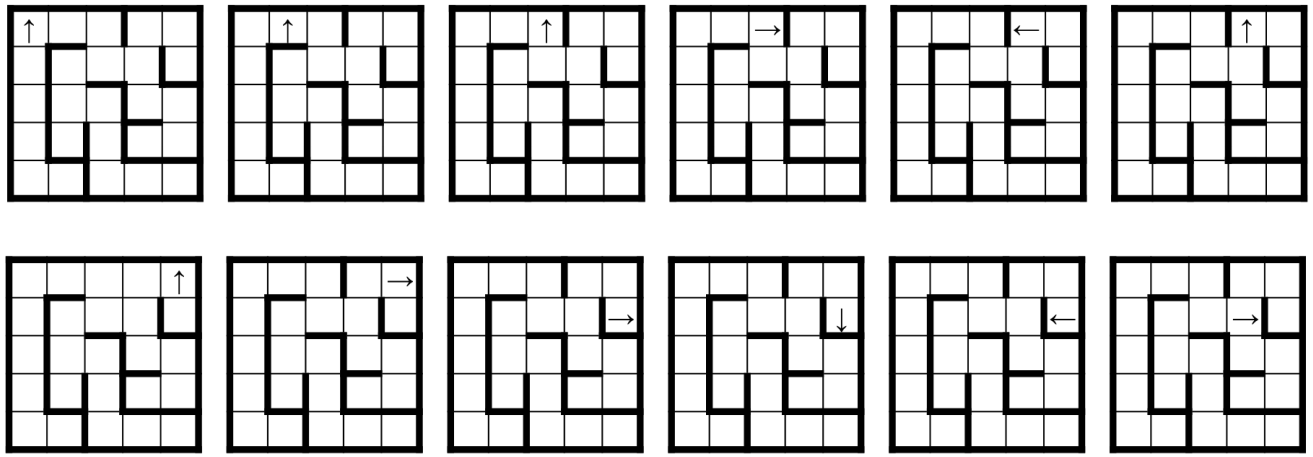
- Sidestep
- Pirouette
- Turn convex corner
- (Turn concave corner)

Query:

- What action to perform?

INVARIANT:
Left hand is on the interior surface of a peripheral wall.

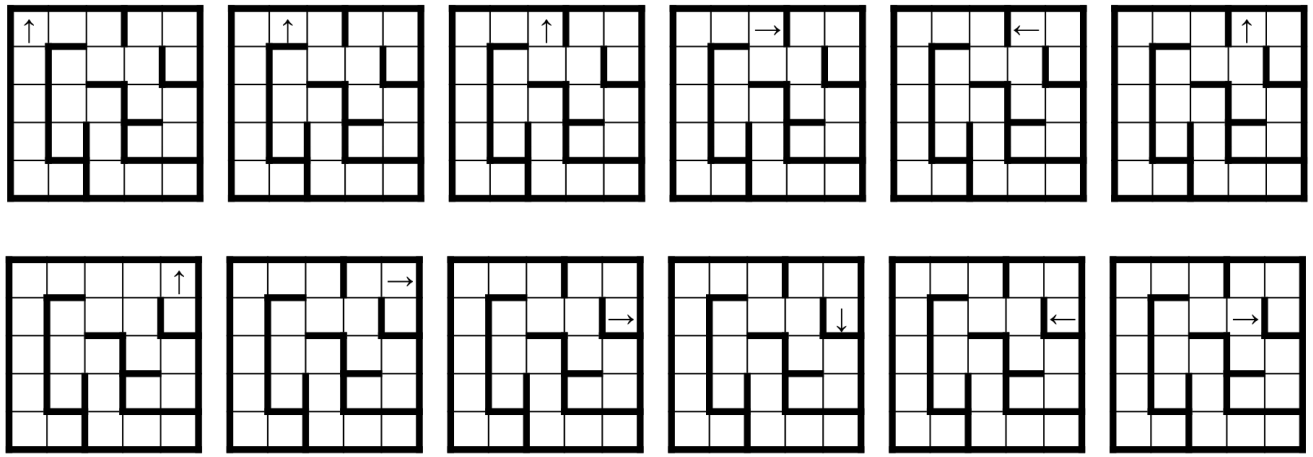
VARIANT:
Get closer to goal.



Actions:
- Sidestep
- Pirouette
- Turn convex corner
- (Turn concave corner)

Query:
- What action to perform?

Unit of progress:
- 1 wall-segment-surface



Physically, you don't need to distinguish cases, e.g., "just keep your hand on the wall and move to the right", but computationally, a case analysis must inspect the geometry, e.g.,

if _____ : Sidestep
elif _____ : Pirouette
elif _____ : Turn convex corner
else: Turn concave corner

Alternative Formulation: From Chapter 4.

(allow left-hand off wall if it is at a **door**)

INVARIANT:

Left hand is on the interior surface of a peripheral wall, or at a door.

Actions:

- Turn clockwise 90°
- Turn counterclockwise 90°
- Step forward

Query:

- Facing a wall?

Unit of progress:

- 1 wall-segment-surface-or-door

Alternative Formulation: From Chapter 4.

(allow left-hand off wall if it is at a **door**)

INVARIANT:

Left hand is on the interior surface of a peripheral wall, or at a door.

Actions:

- Turn clockwise 90°
- Turn counterclockwise 90°
- Step forward

} Finer-grained actions.

Query:

- Facing a wall?

Unit of progress:

- 1 wall-segment-surface-or-door

Alternative Formulation: From Chapter 4.

(allow left-hand off wall if it is at a **door**)

INVARIANT:

Left hand is on the interior surface of a peripheral wall, or at a door.

Actions:

- Turn clockwise 90°
- Turn counterclockwise 90°
- Step forward

Finer-grained actions.

Query:

- Facing a wall?

Local query.

Unit of progress:

- 1 wall-segment-surface-or-door

Alternative Formulation: From Chapter 4.

(allow left-hand off wall if it is at a **door**)

INVARIANT:

Left hand is on the interior surface of a peripheral wall, or at a door.

Actions:

- Turn clockwise 90°
- Turn counterclockwise 90°
- Step forward

Finer-grained actions.

Query:

- Facing a wall?

Local query.

Simpler to implement.

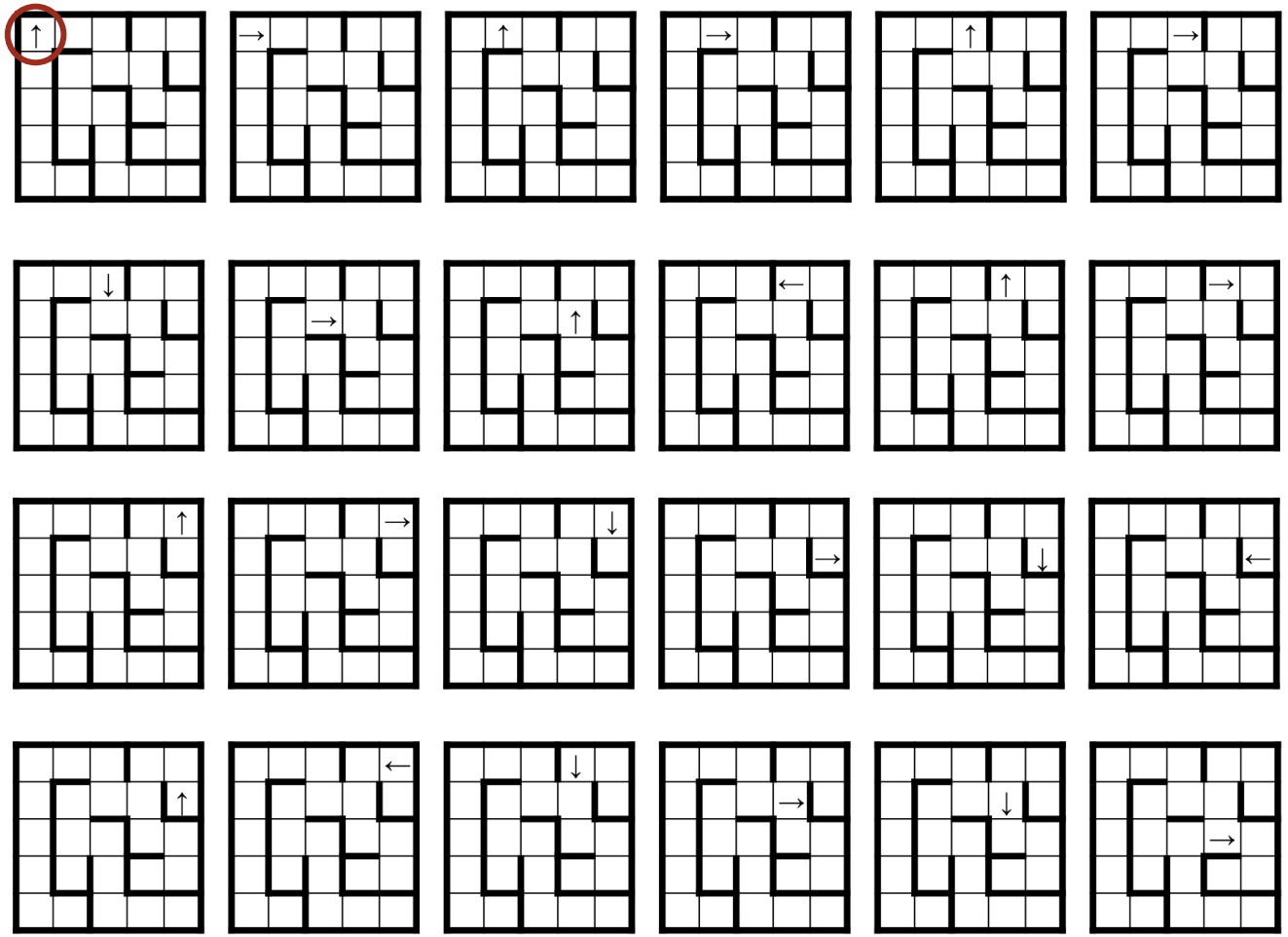
Unit of progress:

- 1 wall-segment-surface-or-door

Alternative Formulation: Pseudo-code, from Chapter 4.

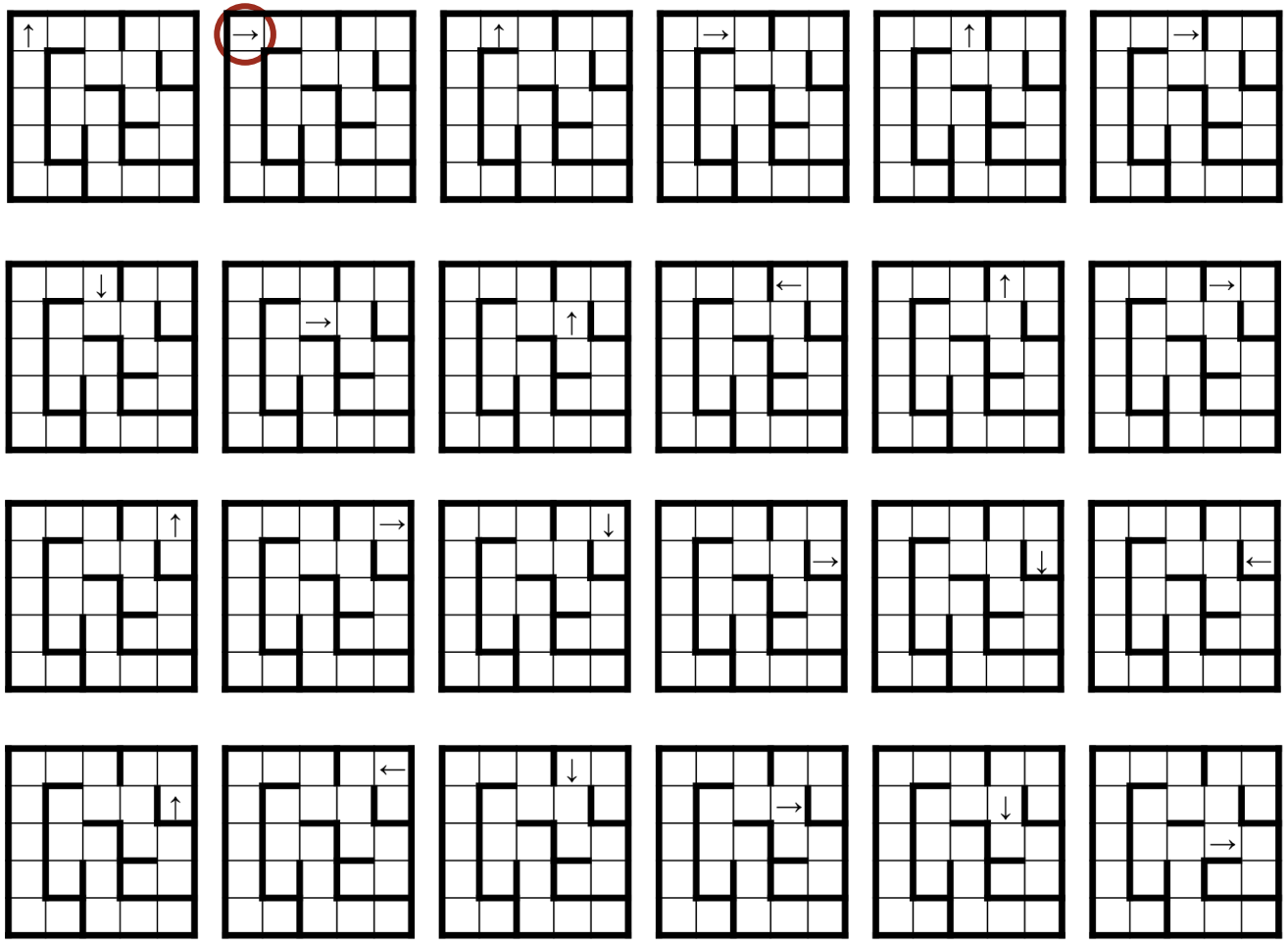
```
# Start in upper-left cell, facing up.
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```

INVARIANT:
Left hand is on the interior surface of a peripheral wall, or at a **door**.



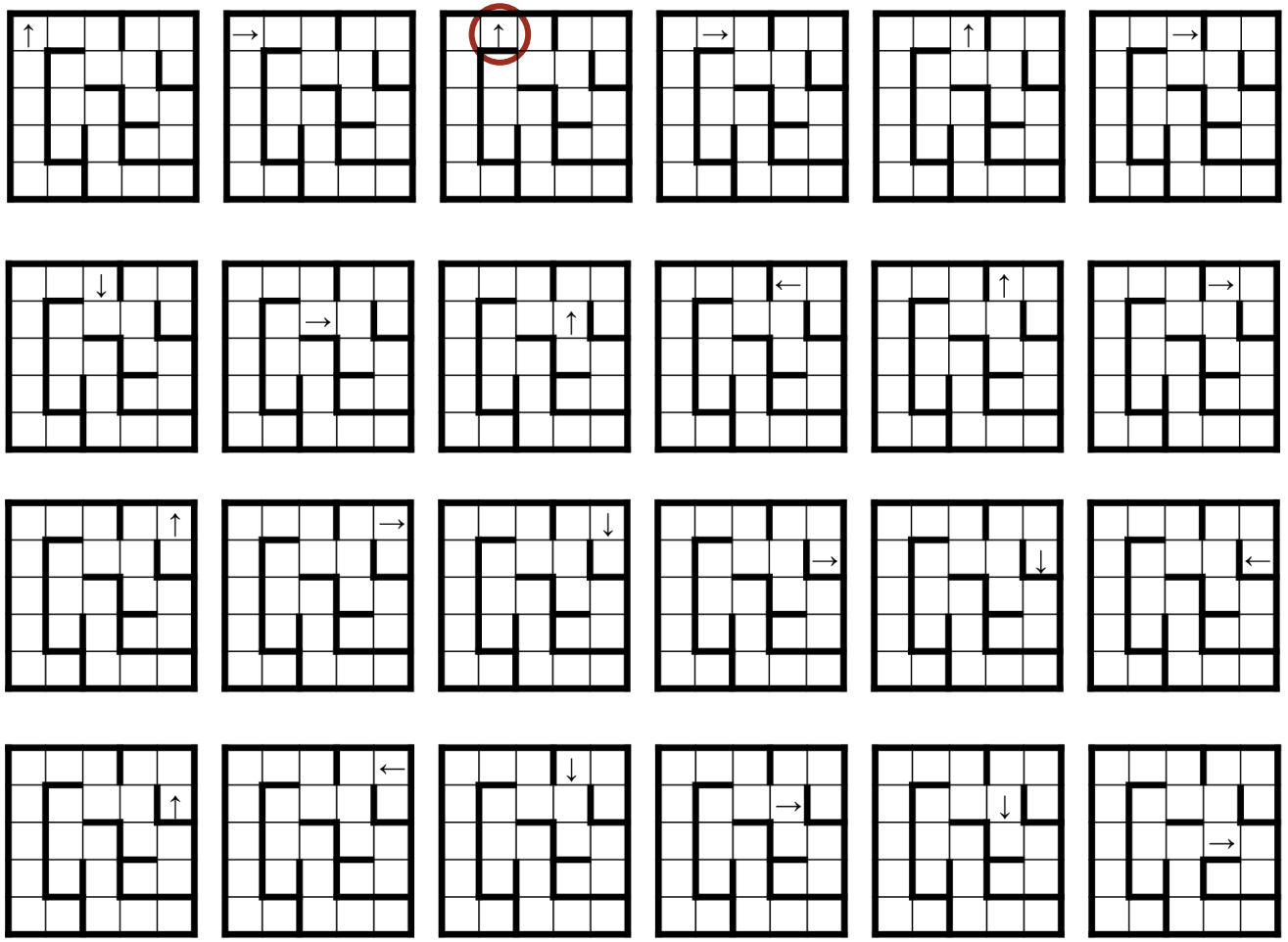
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



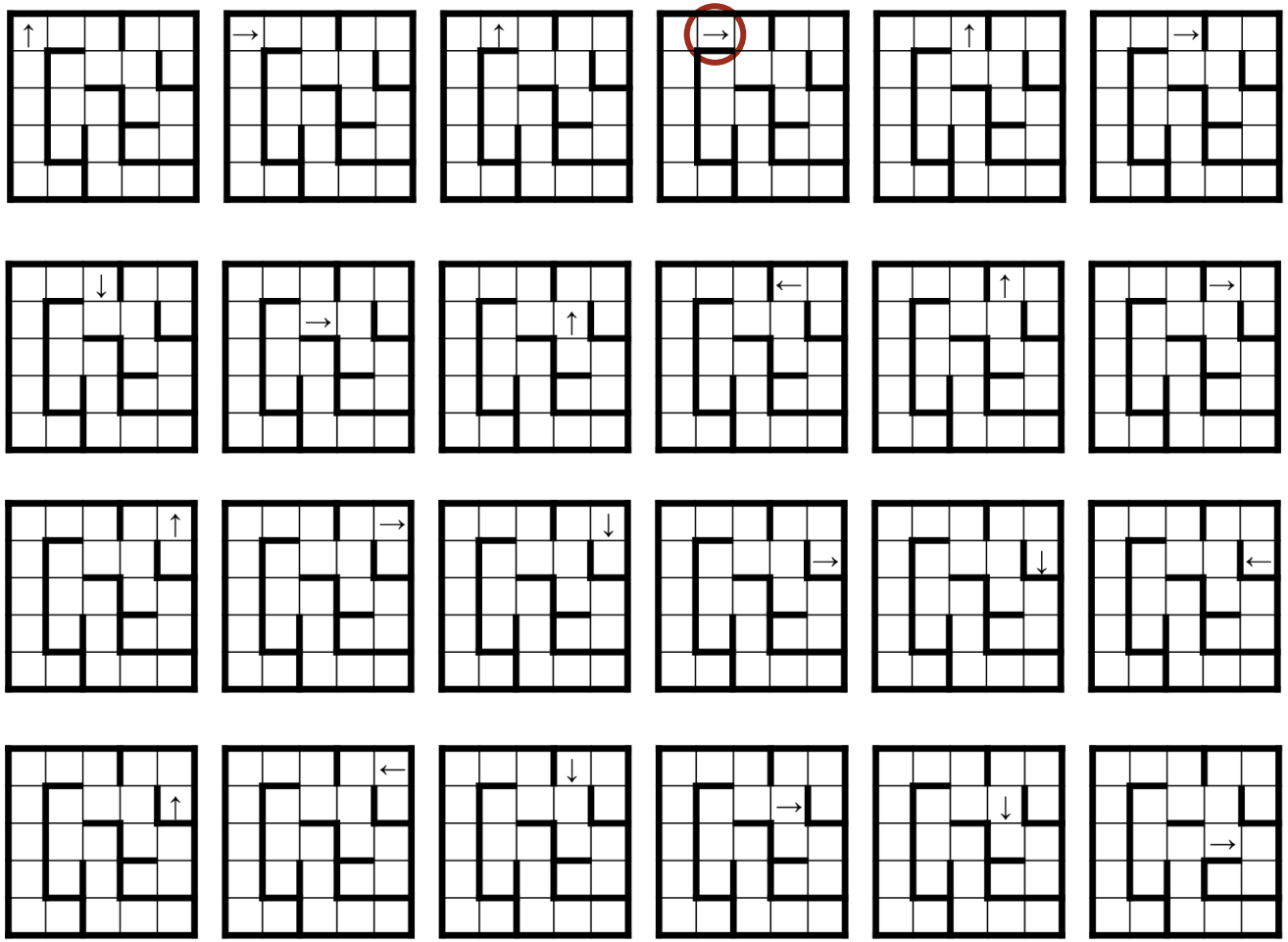
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



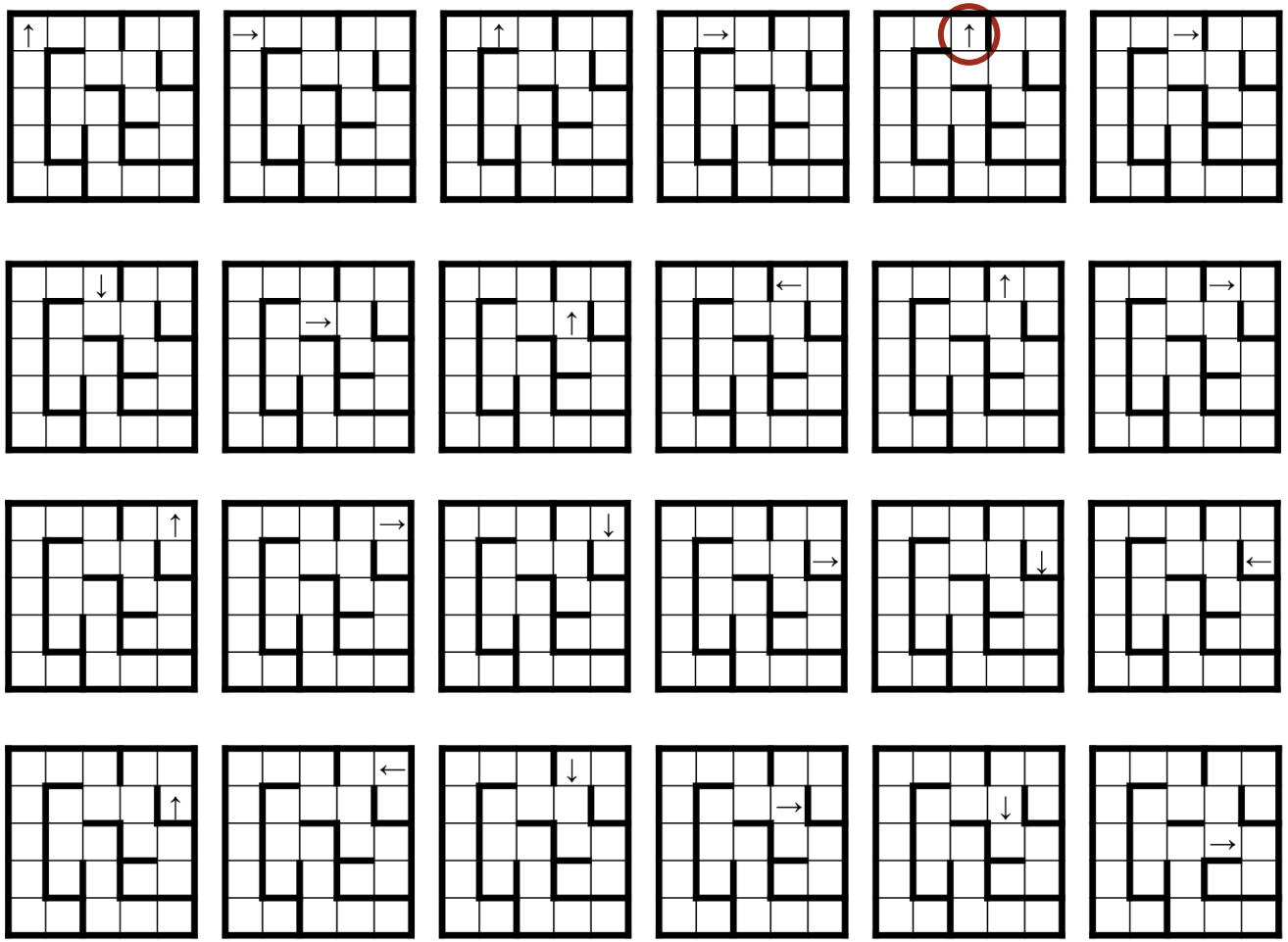
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



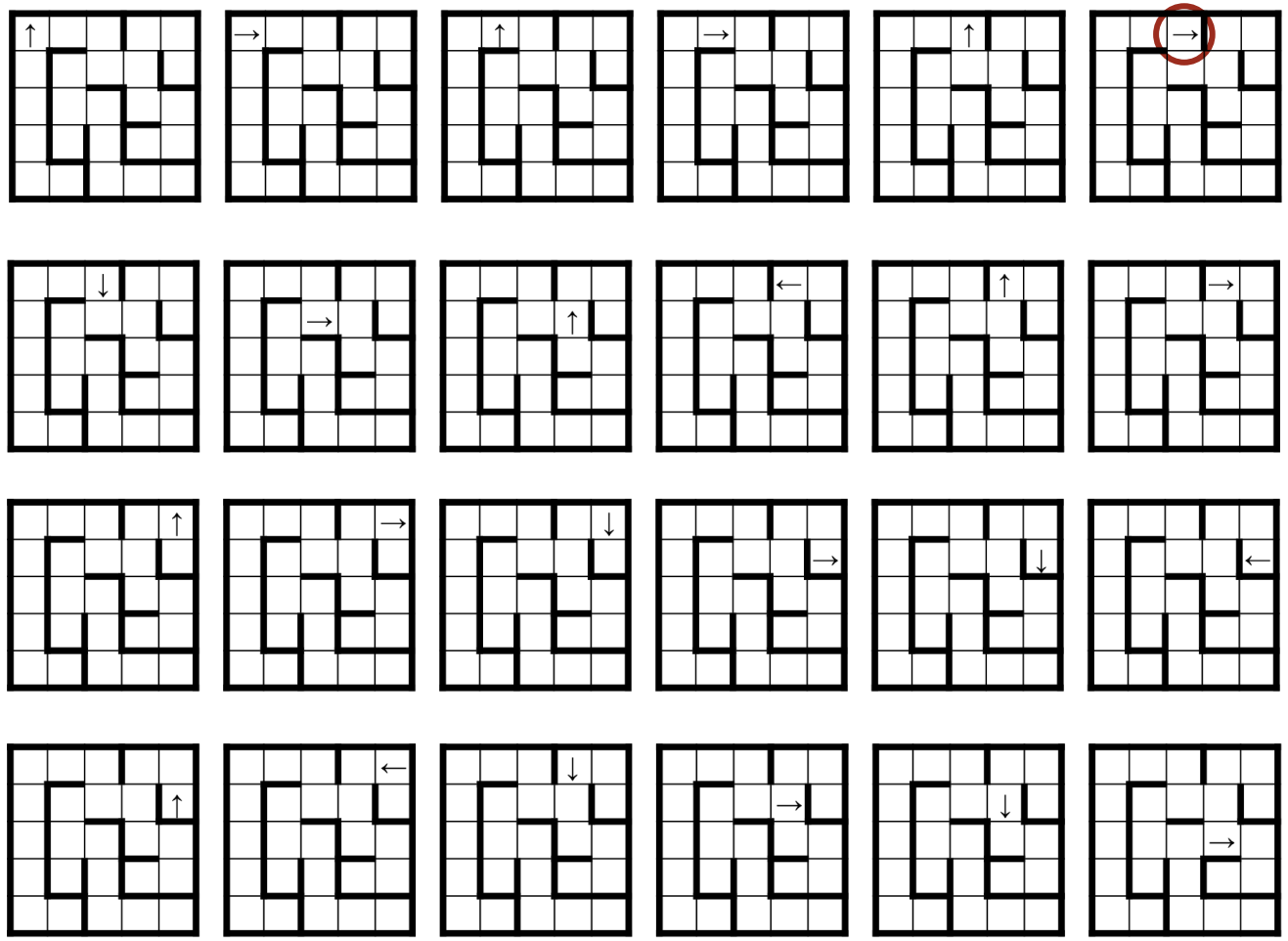
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



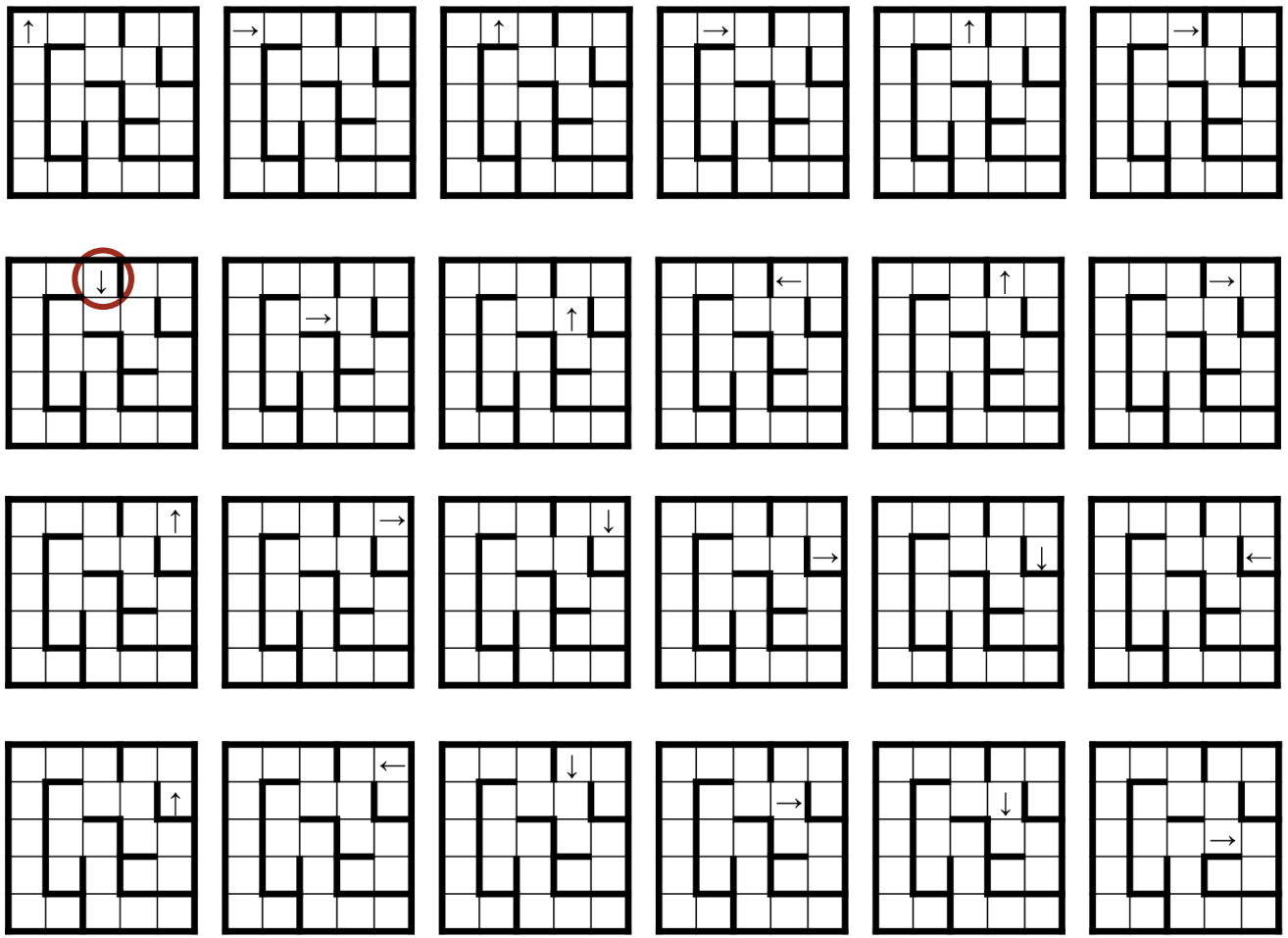
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



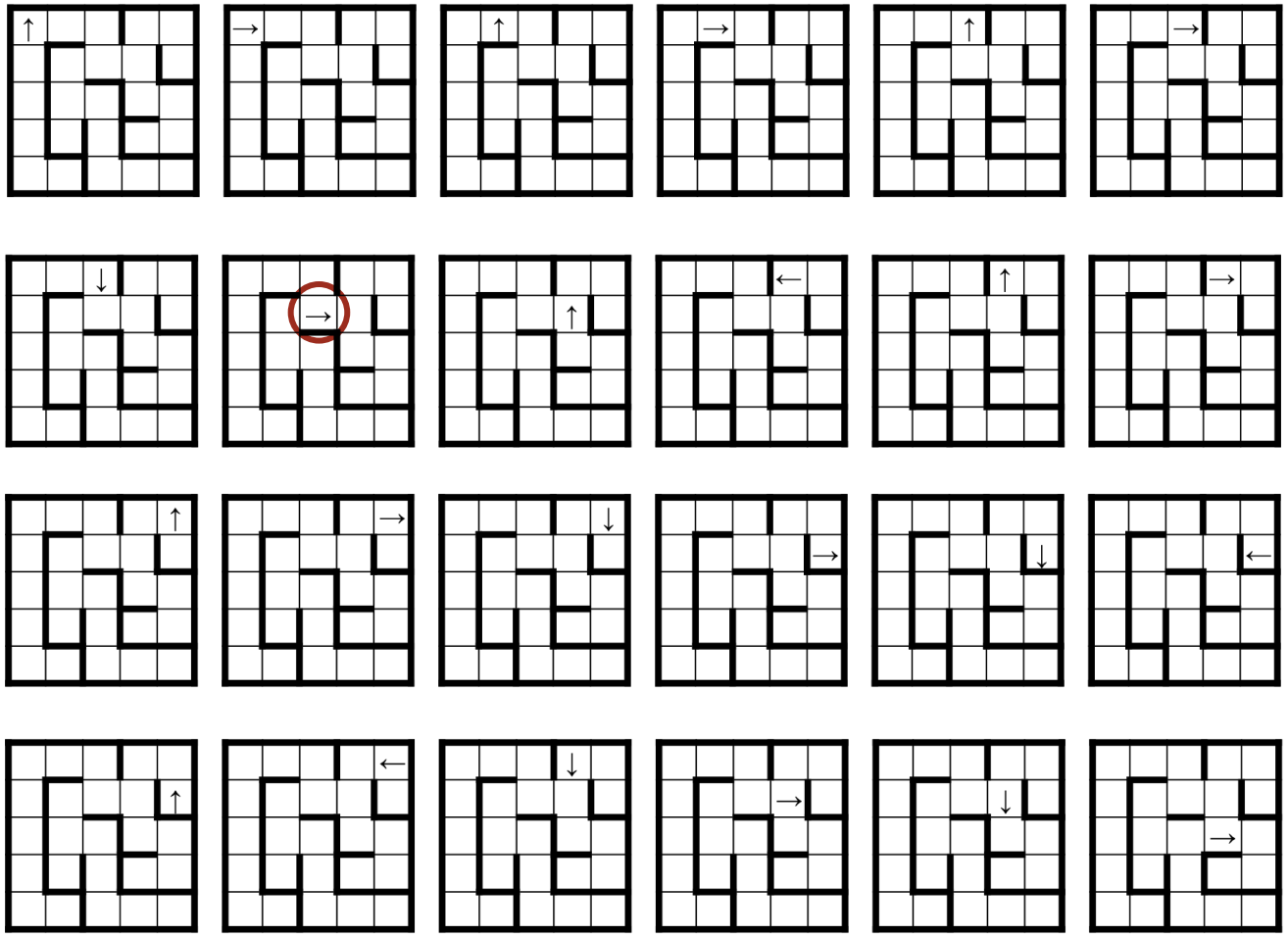
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



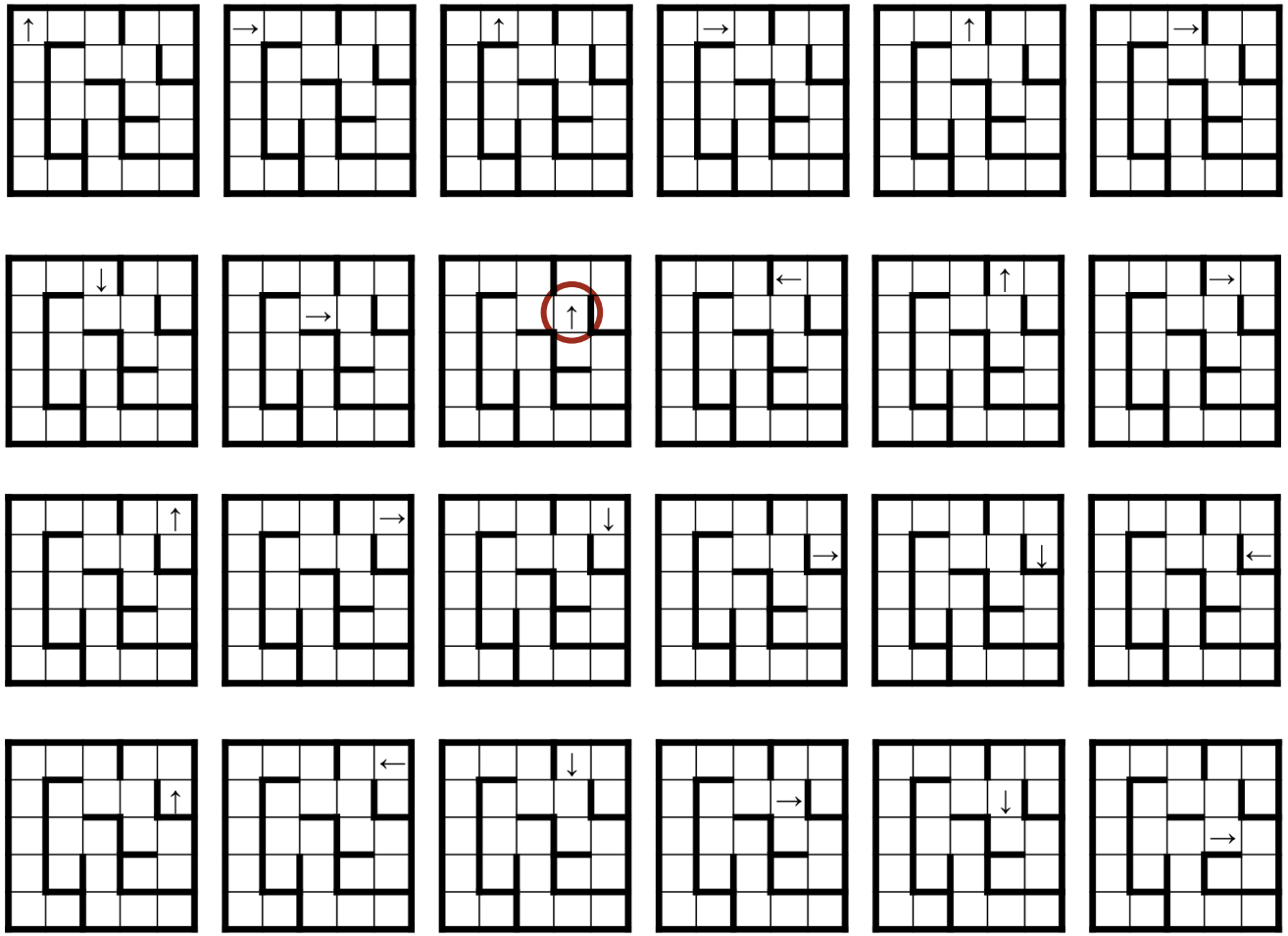
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



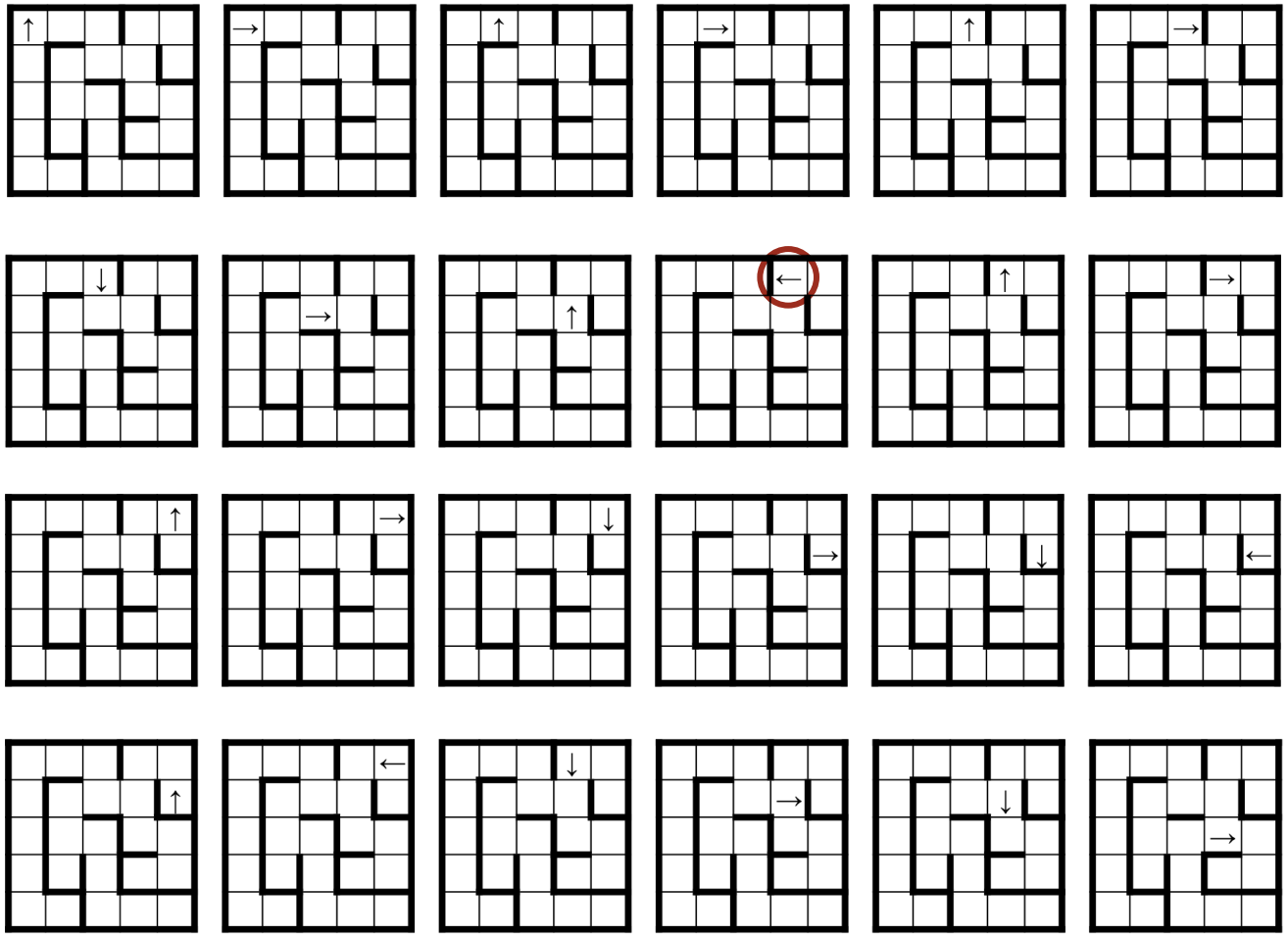
Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



Algorithm

```
while not(in-lower-right) and not(in-upper-left-about-to-cycle):
    if facing-wall :
        #.Turn 90° clockwise.
    else:
        #.Step forward.
        #.Turn 90° counterclockwise.
```



INVARIANT:
Left hand is on the interior surface of a peripheral wall, or at a **door**.
Establish INVARIANT as part of initialization of state.

Algorithm: Drop code into RunMaze.

```
class RunMaze:
    ...

    @classmethod
    def _input(cls) -> None:
        """
        Input a maze of arbitrary size, or output “malformed input” and stop if
        the input is improper. Input format: TBD.
        """
        pass
```

INVARIANT:
Left hand is on the interior surface of a peripheral wall, or at a **door**.
Establish INVARIANT as part of initialization of state.

Algorithm: Drop code into RunMaze.

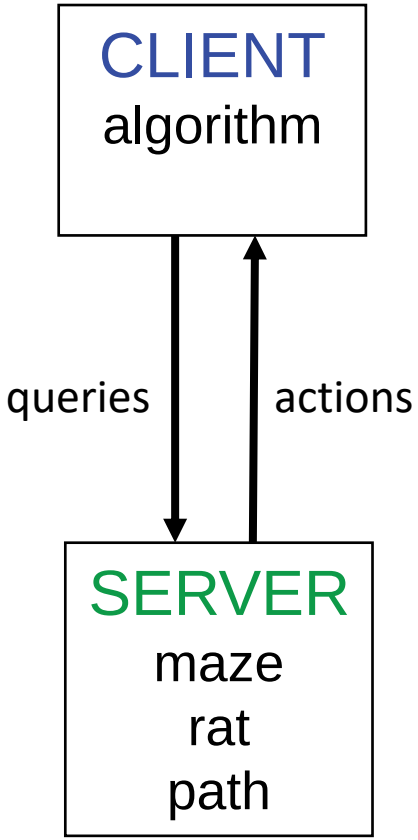
```
class RunMaze:
    ...

    @classmethod
    def _input(cls) -> None:
        """
        Input a maze of arbitrary size, or output “malformed input” and stop if
        the input is improper. Input format: TBD.
        """
        ⟨Obtain maze from input.⟩
        ⟨Start in upper-left cell, facing up.⟩
```

INVARIANT:
Left hand is on the interior surface of a peripheral wall, or at a **door**.
Maintain INVARIANT and make progress in `solve`.

Algorithm: Drop code into RunMaze, with pseudo-operations turned into method calls.

```
class RunMaze:
    ...

    @classmethod
    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists."""
        while not(is_at_cheese()) and not(is_about_to_repeat()):
            if is_facing_wall(): turn_clockwise()
            else:
                step_forward()
                turn_counter_clockwise()
    ...
```

Modular program structure: Separation of concerns.




```
class RunMaze:
    ...
    def main(self) -> None:
        """Run a maze given as input, if possible."""
        ...
    ...
```

```
class MRP:
    ...
    def turn_clockwise() -> None: ...
    def turn_counter_clockwise() -> None: ...
    ...
```

The algorithm is a client of services provided by class MRP.

Algorithm (from Chapter 4): Qualify names of methods of another class.

```
class RunMaze:
    ...

    @classmethod
    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists."""
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            else:
                MRP.step_forward()
                MRP.turn_counter_clockwise()

...
```

Procedure *stubs* for the services.

Operations:

```
class MRP:
    ...

    # Public Interface.
    @classmethod
    def turn_clockwise(cls) -> None: pass
    @classmethod
    def turn_counter_clockwise(cls) -> None: pass
    @classmethod
    def step_forward(cls) -> None: pass
    @classmethod
    def is_facing_wall(cls) -> bool: return ____
    @classmethod
    def is_at_cheese(cls) -> bool: return ____
    @classmethod
    def is_about_to_repeat(cls) -> bool: return ____
    ...
```

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Stubs provide *signatures*, i.e., names, types for return values, types for parameters (none), and visibility (underscores or not).

Operations:

```
class MRP:
    ...

    # Public Interface.
    @classmethod
    def turn_clockwise(cls) -> None: pass
    @classmethod
    def turn_counter_clockwise(cls) -> None: pass
    @classmethod
    def step_forward(cls) -> None: pass
    @classmethod
    def is_facing_wall(cls) -> bool: return ____
    @classmethod
    def is_at_cheese(cls) -> bool: return ____
    @classmethod
    def is_about_to_repeat(cls) -> bool: return ____
    ...
```

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Visible to client classes of MRP, e.g., RunMaze.

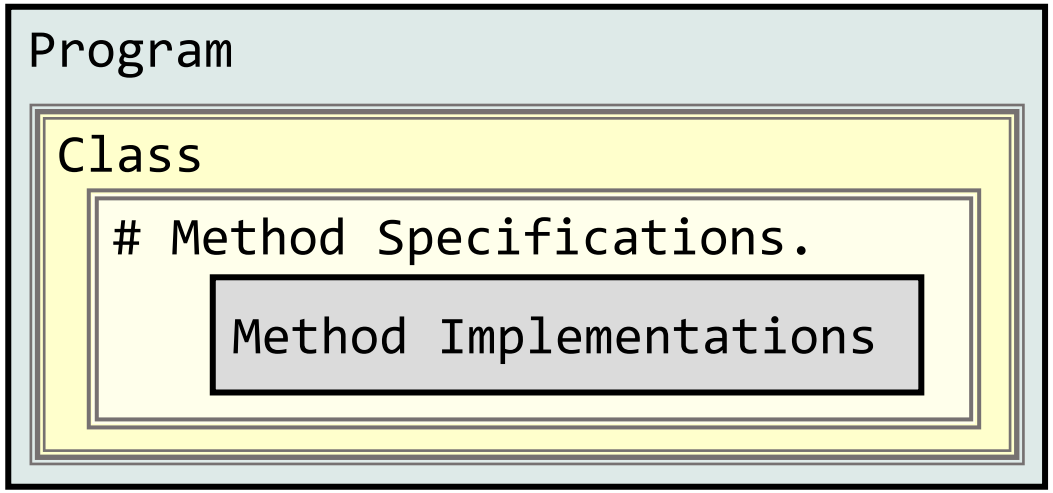
Operations:

```
class MRP:
    ...

    # Public Interface.
    @classmethod
    def turn_clockwise(cls) -> None: pass
    @classmethod
    def turn_counter_clockwise(cls) -> None: pass
    @classmethod
    def step_forward(cls) -> None: pass
    @classmethod
    def is_facing_wall(cls) -> bool: return ____
    @classmethod
    def is_at_cheese(cls) -> bool: return ____
    @classmethod
    def is_about_to_repeat(cls) -> bool: return ____
    ...
```

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

State: The Maze, Rat, and Path data representations.

We (the implementers of MRP) design the data representation to record the state, and code the query and action operations to update it.

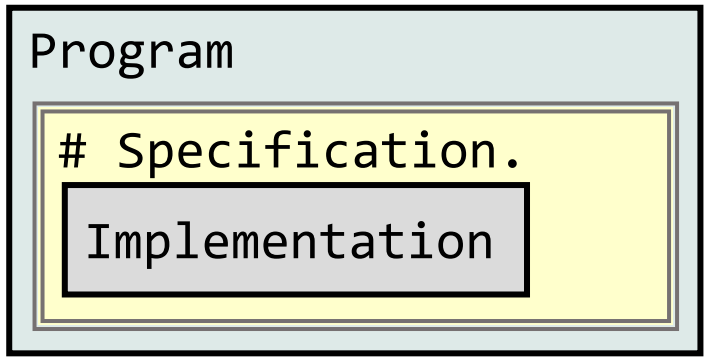


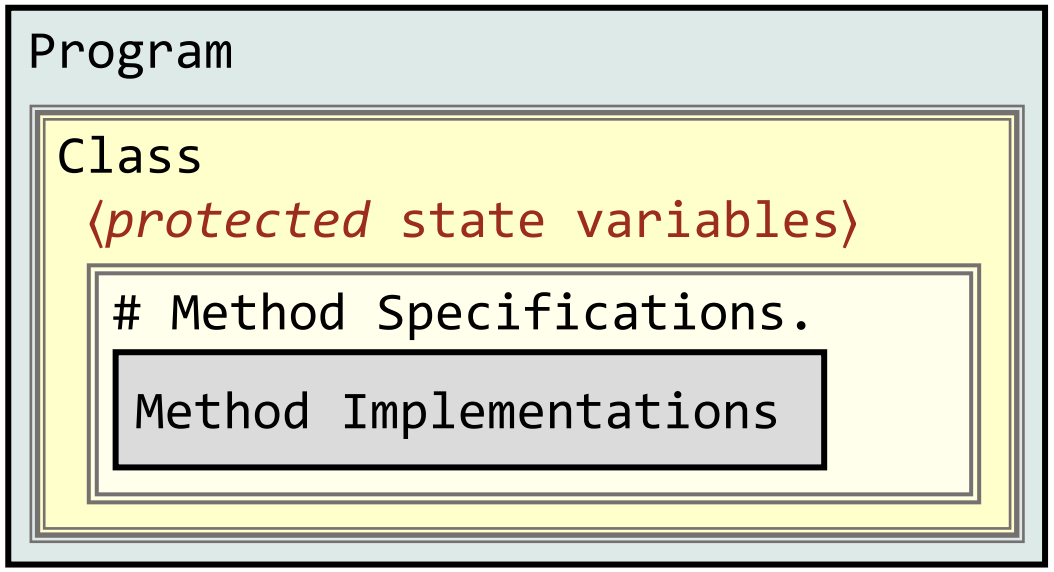
☞ **Practice information hiding.**

State: The Maze, Rat, and Path data representations.

We (the implementers of MRP) design the data representation to record the state, and code the operations to query and update it.

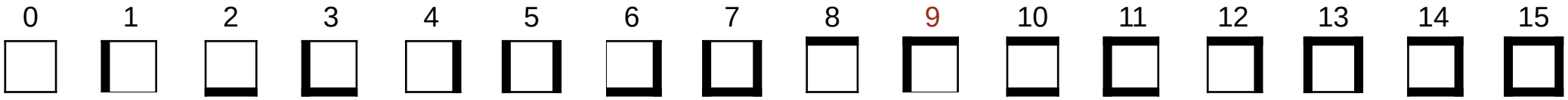
Clients of MRP will have no direct access to the state in MRP. Rather, they will only be able to interact with MRP via its operations, i.e., its interface. This is called an *abstract data type*, and generalizes our prior use of specifications for information hiding.

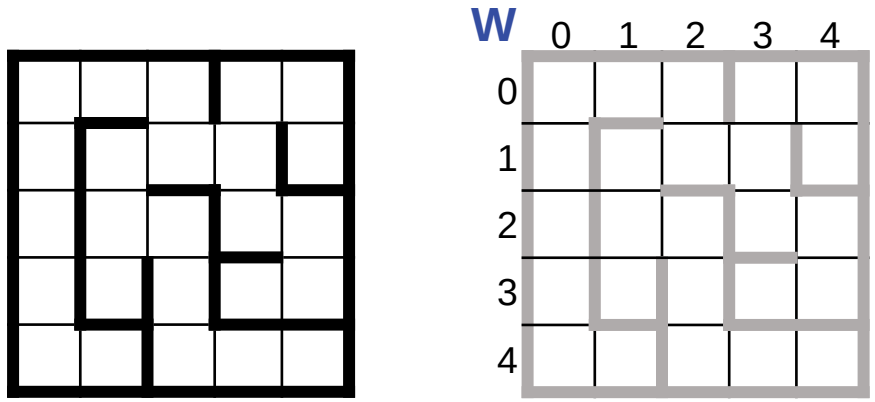




☞ **Practice information hiding.**

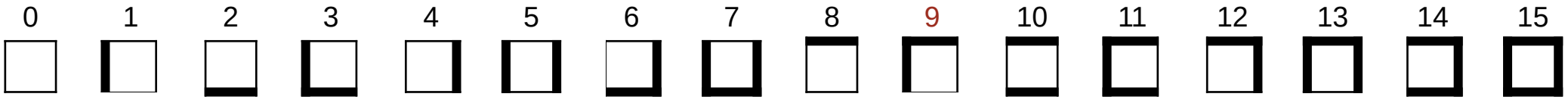
Maze Representation 1: N-by-N array W whose elements encode cell walls:

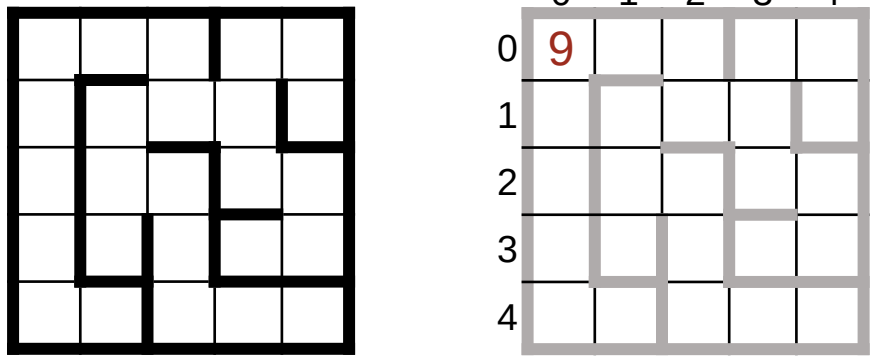




☞ **The touchstone of a data representation is its utility in performing the needed operations.**

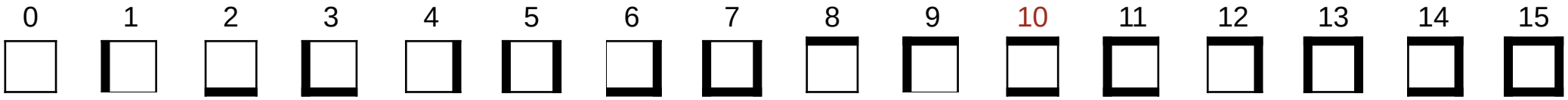
Maze Representation 1: N-by-N array W whose elements encode cell walls:

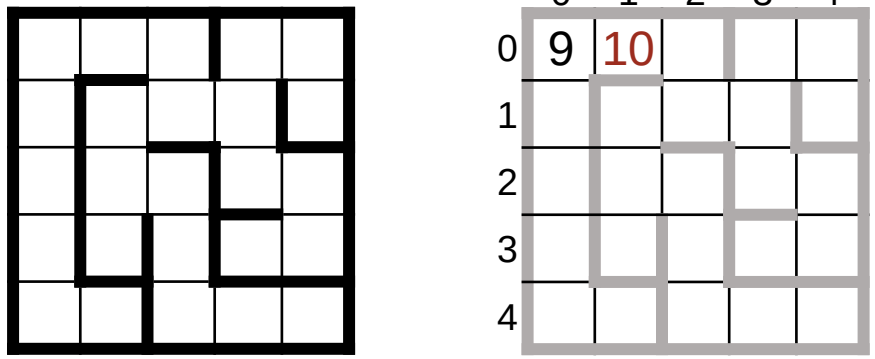




☞ **The touchstone of a data representation is its utility in performing the needed operations.**

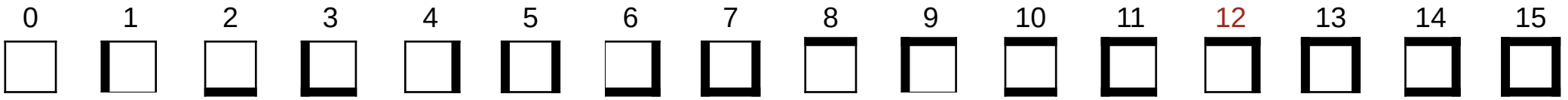
Maze Representation 1: N-by-N array W whose elements encode cell walls:

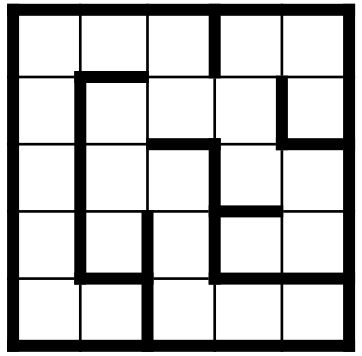




☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:



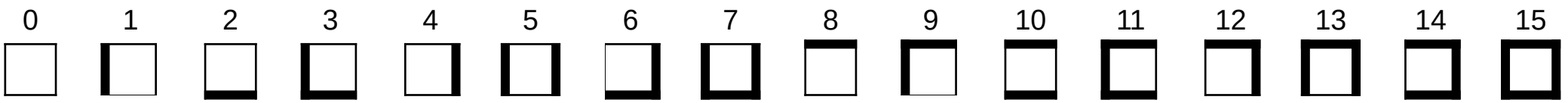


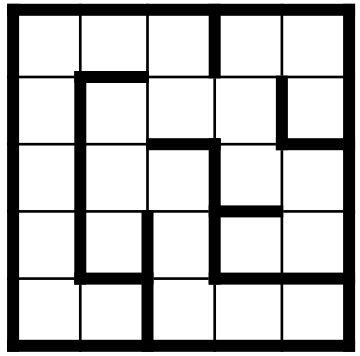
W

	0	1	2	3	4
0	9	10	12		
1					
2					
3					
4					

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:



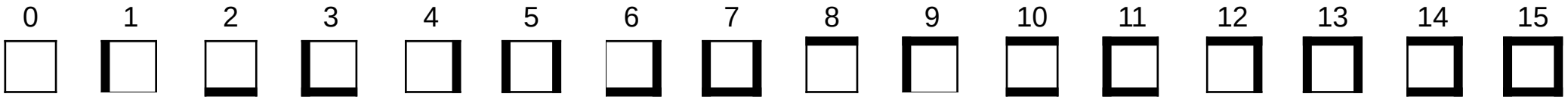


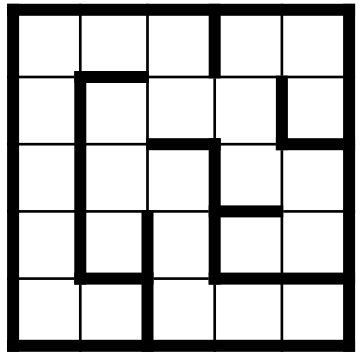
W

	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:





W

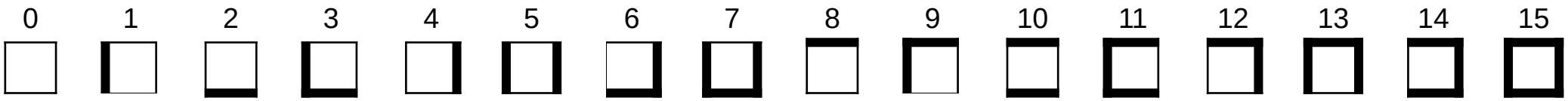
	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

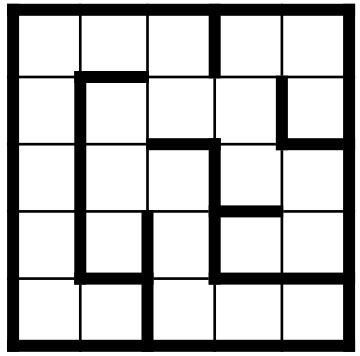
Anticipate

- Direction `d`, ⟨0,1,2,3⟩ = ⟨up,`right`,down,`left`⟩
- Decoder `isWall(r,c,d)`, **True** iff wall in direction d

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:





W

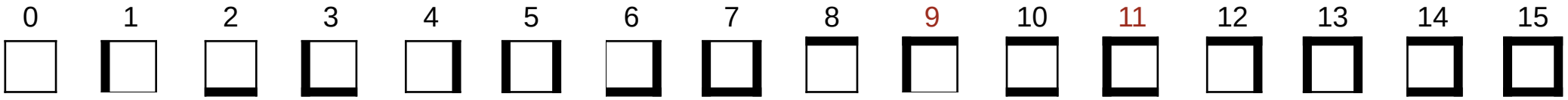
	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

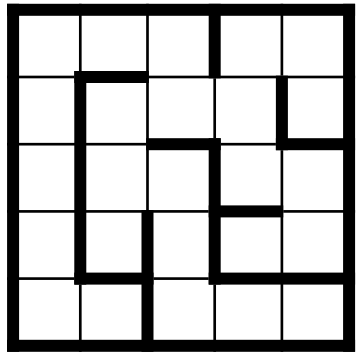
Positive

- Direct correspondence between physical maze and 2-D array **W**.

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:





W

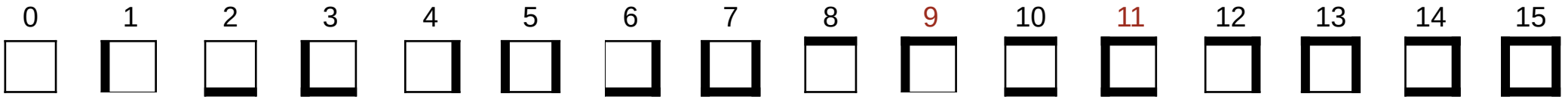
	0	1	2	3	4
0	9	11	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

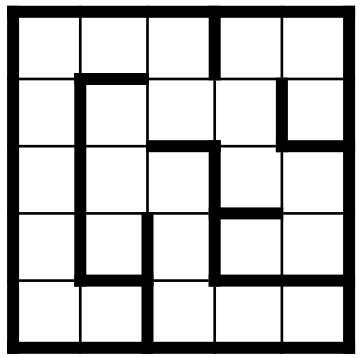
Negative

- Representation admits nonsensical data, e.g., 9 claims "there is no wall to the right", but 11 claims "there is a wall to the left".

☞ **Choose representations that by design do not have nonsensical configurations.**

Maze Representation 1: N-by-N array W whose elements encode cell walls:





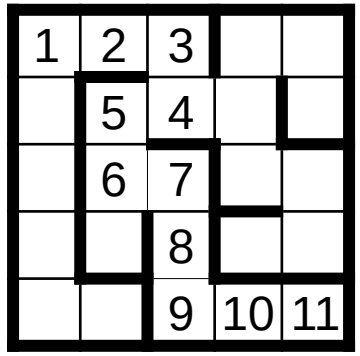
W

	0	1	2	3	4
0	9	11	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

Negatives

- Representation admits nonsensical data, e.g., 9 claims "there is no wall to the right", but 11 claims "there is a wall to the left".
- Decoder `isWall(r,c,d)` and corresponding encoder are somewhat fussy.

Path Representation 1: N-by-N array P whose elements are visit numbers or 0 (UNVISITED).



P

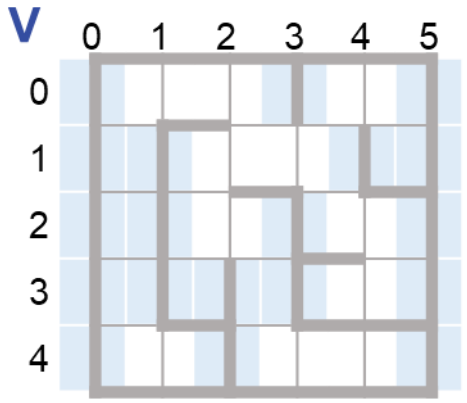
	0	1	2	3	4
0	1	2	3	0	0
1	0	5	4	0	0
2	0	6	7	0	0
3	0	0	8	0	0
4	0	0	9	10	11

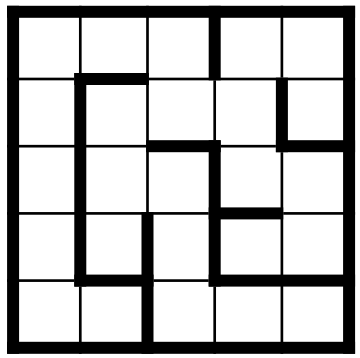
Positive

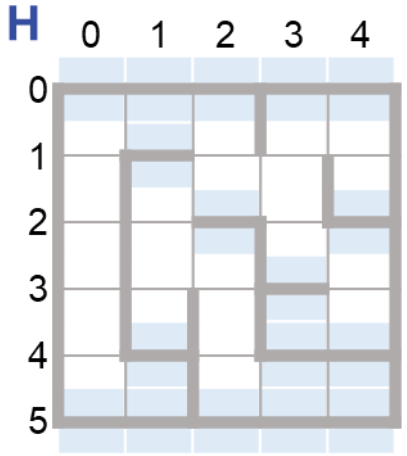
- Direct correspondence between physical maze and 2-D array **P**.

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 2: Separate **boolean** arrays, V and H, for **v**ertical and **h**orizontal walls.





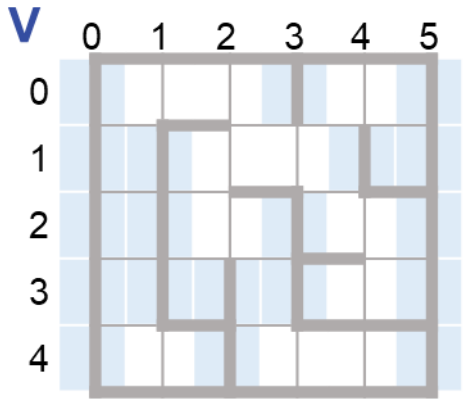


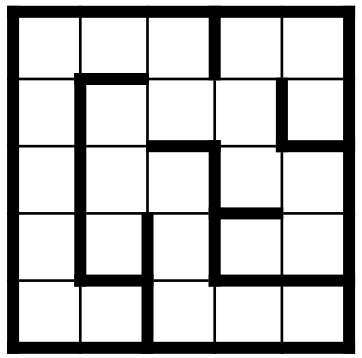
Eliminating Negatives of Representation 1

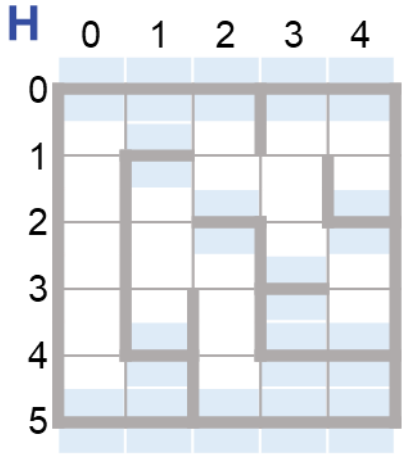
- Unique representation of each (possible) wall.
- Decoder and corresponding encoder are more straightforward.

☞ **Choose representations that by design do not have nonsensical configurations.**

Maze Representation 2: Separate **boolean** arrays, V and H, for **v**ertical and **h**orizontal walls.





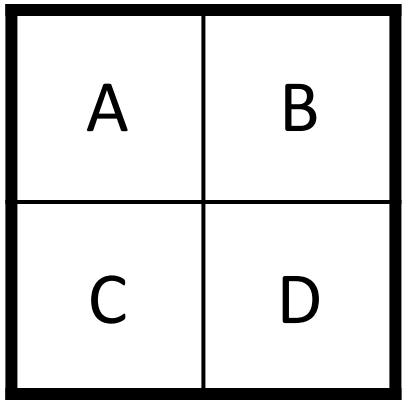


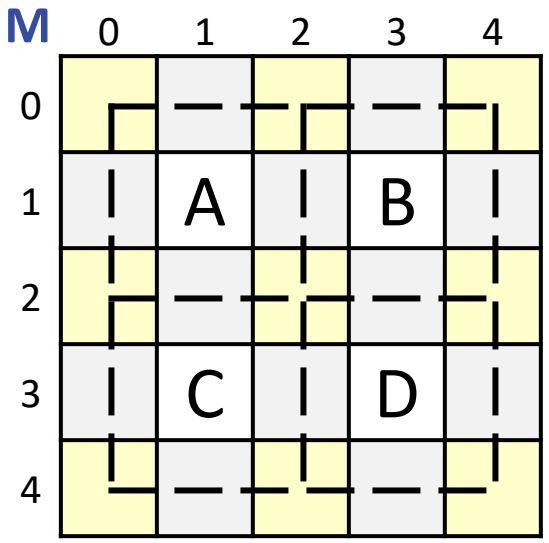
Negative of Representation 2

- Non-uniformity. Two arrays rather than one.

☞ **Choose data representations that are uniform, if possible.**

Maze Representation 3: (2·N+1)-by-(2·N+1) array M of of walls and path visit numbers.



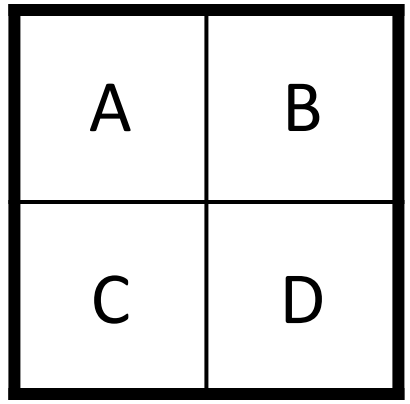


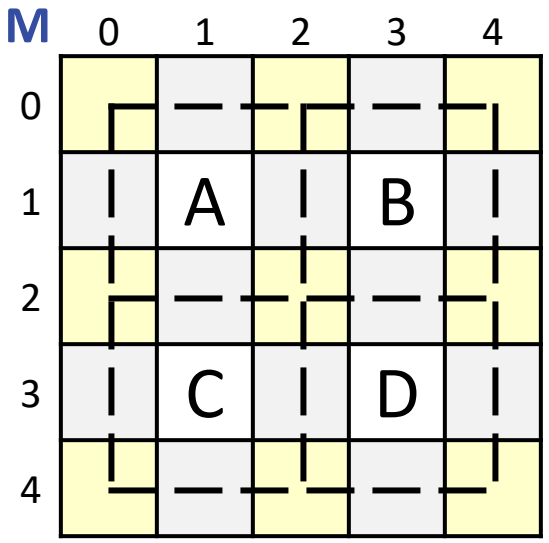
Positives

- Single 2-D array M for both walls and path.
- Unique array cell (gray) to represent each (possible) wall.
- Unique array cell (letters) for visit numbers.

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 3: (2·N+1)-by-(2·N+1) array M of of walls and path visit numbers.



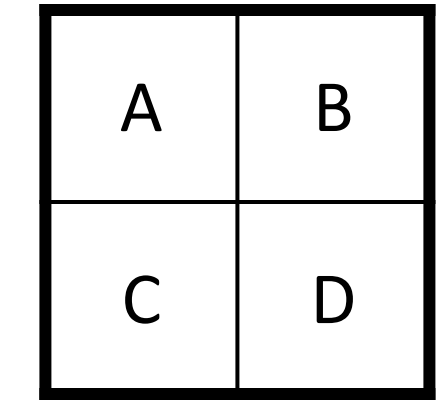


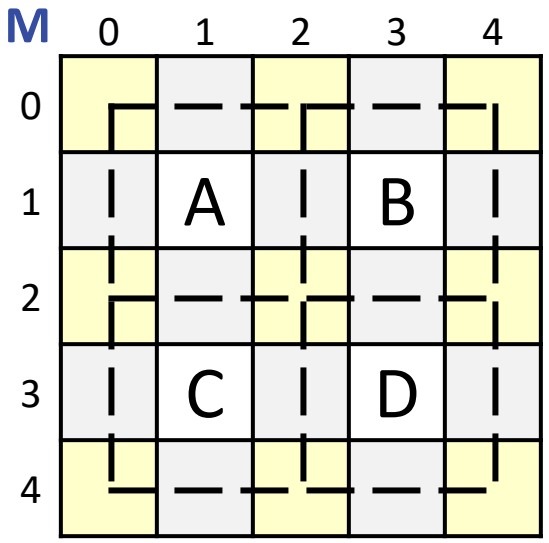
Negatives

- About ¼ of storage is wasted (yellow).
- Direct correspondence between maze coordinate system and 2-D array. indices lost.

☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Maze Representation 3: Adopt it.





☞ **Don't let the "perfect" be the enemy of the "good". Be prepared to compromise because there may be no perfect representation. Don't freeze.**

Data Representation Invariant:

```
class MRP:

    # Maze. Cells of an N-by-N maze are represented by elements of a
    #   (2*N+1)-by-(2*N+1) array M. Maze cell ⟨r,c⟩ is represented by array
    #   element M[2*r+1][2*c+1]. The possible walls ⟨top, right, bottom,
    #   left⟩ of the maze cell corresponding to ⟨r,c⟩ are represented by
    #   WALL or NO_WALL in ⟨M[r-1][c], M[r][c+1], M[r+1][c], M[r][c-1]⟩.
    #   The remaining elements of M are unused. lo is 1, and hi is 2*N-1.
    _N: int                     # _N is size of maze.
    _M: list[list[int]]         # _M is _N-by-_N maze, walls, and path.
    _WALL: int = -1             # _WALL encodes presence of a wall.
    _NO_WALL: int = 0           # _NO_WALL encodes absence of a wall.
    _lo: int                    # _lo is left and top maze indices.
    _hi: int                    # _hi is right and bottom maze indices.

    ...
```

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Names that begin with _ are *protected* and internal to MRP. No other class needs to know about them.

```
class MRP:

    # Maze. Cells of an N-by-N maze are represented by elements of a
    #   (2*N+1)-by-(2*N+1) array M. Maze cell ⟨r,c⟩ is represented by array
    #   element M[2*r+1][2*c+1]. The possible walls ⟨top, right, bottom,
    #   left⟩ of the maze cell corresponding to ⟨r,c⟩ are represented by
    #   WALL or NO_WALL in ⟨M[r-1][c], M[r][c+1], M[r+1][c], M[r][c-1]⟩.
    #   The remaining elements of M are unused. lo is 1, and hi is 2*N-1.
    _N: int                     # _N is size of maze.
    _M: list[list[int]]         # _M is _N-by-_N maze, walls, and path.
    _WALL: int = -1             # _WALL encodes presence of a wall.
    _NO_WALL: int = 0           # _NO_WALL encodes absence of a wall.
    _lo: int                    # _lo is left and top maze indices.
    _hi: int                    # _hi is right and bottom maze indices.

    ...
```

☞ **Practice information hiding.**

Full-word names that are all capital letters, by convention, are intended to be constant throughout program execution.

```
class MRP:

    # Maze. Cells of an N-by-N maze are represented by elements of a
    #   (2*N+1)-by-(2*N+1) array M. Maze cell ⟨r,c⟩ is represented by array
    #   element M[2*r+1][2*c+1]. The possible walls ⟨top, right, bottom,
    #   left⟩ of the maze cell corresponding to ⟨r,c⟩ are represented by
    #   WALL or NO_WALL in ⟨M[r-1][c], M[r][c+1], M[r+1][c], M[r][c-1]⟩.
    #   The remaining elements of M are unused. lo is 1, and hi is 2*N-1.
    _N: int                         # _N is size of maze.
    _M: list[list[int]]             # _M is _N-by-_N maze, walls, and path.
    _WALL: int = -1                 # _WALL encodes presence of a wall.
    _NO_WALL: int = 0               # _NO_WALL encodes absence of a wall.
    _lo: int                        # _lo is left and top maze indices.
    _hi: int                        # _hi is right and bottom maze indices.

    ...
```

☞ **Minimize use of literal numerals in code; define and use symbolic constants.**

Data Representation Invariant:

```
class MRP:
    ...

    # Rat. The rat is located in cell M[r][c] facing direction d, where
    #   d=⟨0,1,2,3⟩ represents the orientation ⟨up,right,down,left⟩,
    #   respectively.
    _r: int
    _c: int
    _d: int

    ...
```

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Data Representation Invariant:

```
class MRP:
    ...

    # Path. When the rat has traveled to cell ⟨r,c⟩ via a given path
    #   through cells of the maze, the elements of M that correspond to
    #   those cells will be 1, 2, 3, etc., and all other elements of M
    #   that correspond to cells of the maze will be UNVISITED. The
    #   number of the last step in the path is move.
    _UNVISITED: int = 0
    _move: int

    ...
```

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Variables declared and initialized at the top-level of a class are called *class variables*, and are shared among all of the methods of the class.

```
class MRP:

    # Maze. Cells of an N-by-N maze are represented by elements of a
    #   (2*N+1)-by-(2*N+1) array M. Maze cell ⟨r,c⟩ is represented by array
    #   element M[2*r+1][2*c+1]. The possible walls ⟨top, right, bottom,
    #   left⟩ of the maze cell corresponding to ⟨r,c⟩ are represented by
    #   WALL or NO_WALL in ⟨M[r-1][c], M[r][c+1], M[r+1][c], M[r][c-1]⟩.
    #   The remaining elements of M are unused. lo is 1, and hi is 2*N-1.
    _N: int                     # _N is size of maze.
    _M: list[list[int]]         # _M is _N-by-_N maze, walls, and path.
    _WALL: int = -1             # _WALL encodes presence of a wall.
    _NO_WALL: int = 0           # _NO_WALL encodes absence of a wall.
    _lo: int = 1                # _lo is left and top maze indices.
    _hi: int = 2 * _N - 1       # _hi is right and bottom maze indices.

    ...
```

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Variables declared and initialized at the top-level of a class are called *class variables*, and are shared among all of the methods of the class.

```
class MRP:

    # Maze. Cells of an N-by-N maze are represented by elements of a
    #   (2*N+1)-by-(2*N+1) array M. Maze cell ⟨r,c⟩ is represented by array
    #   element M[2*r+1][2*c+1]. The possible walls ⟨top, right, bottom,
    #   left⟩ of the maze cell corresponding to ⟨r,c⟩ are represented by
    #   WALL or NO_WALL in ⟨M[r-1][c], M[r][c+1], M[r+1][c], M[r][c-1]⟩.
    #   The remaining elements of M are unused. lo is 1, and hi is 2*N-1.
    _N: int = 0                  # _N is size of maze.
    _M: list[list[int]] = []     # _M is _N-by-_N maze, walls, and path.
    _WALL: int = -1              # _WALL encodes presence of a wall.
    _NO_WALL: int = 0            # _NO_WALL encodes absence of a wall.
    _lo: int = 1                 # _lo is left and top maze indices.
    _hi: int = 2 * _N - 1        # _hi is right and bottom maze indices.

    ...
```

All class variables should be initialized even if the values will necessarily be updated later, e.g., _N and _M will be established by _input. Nonetheless, be consistent with the invariant.

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Data Representation Invariant:

```
class MRP:
    ...

    # Rat. The rat is located in cell M[r][c] facing direction d, where
    #   d=⟨0,1,2,3⟩ represents the orientation ⟨up,right,down,left⟩,
    #   respectively.
    _r: int = _lo
    _c: int = _lo
    _d: int = 0

    ...
```

All class variables should be initialized even if the values will necessarily be updated later, e.g., _N and _M will be established by _input. Nonetheless, be consistent with the invariant.

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Data Representation Invariant:

```
class MRP:
    ...

    # Path. When the rat has traveled to cell ⟨r,c⟩ via a given path
    #   through cells of the maze, the elements of M that correspond to
    #   those cells will be 1, 2, 3, etc., and all other elements of M
    #   that correspond to cells of the maze will be UNVISITED. The
    #   number of the last step in the path is move.
    _UNVISITED: int = 0
    _move: int = _lo

    ...
```

All class variables should be initialized even if the values will necessarily be updated later, e.g., _N and _M will be established by _input. Nonetheless, be consistent with the invariant.

☞ **A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

Auxiliary Data:

```
class MRP:
    ...

    # Unit vectors in direction d
    #                    d = 0,     1,    2,    3
    #                        up, right, down, left
    _deltaR: list[int] = [ -1,     0,    1,    0 ]
    _deltaC: list[int] = [  0,     1,    0,   -1 ]

   ...
```

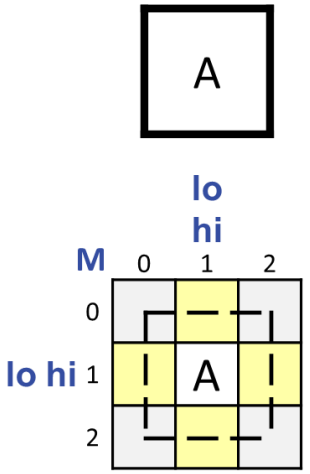
Operations: Complete the implementation

Note: `@classmethod` decorators have been omitted from this slide for brevity.

```
class MRP:
    ...

    # Public Interface.
    def turn_clockwise(cls) -> None:
        MRP._d = (MRP._d + 1) % 4
    def turn_counter_clockwise(cls) -> None:
        MRP._d = (MRP._d + 3) % 4
    def step_forward(cls) -> None:
        MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
        MRP._move += 1;  MRP._M[MRP._r][MRP._c] = MRP._move
    def is_facing_wall(cls) -> bool:
        return MRP._M[MRP._r + MRP._deltaR[MRP._d]
                      ][MRP._c + MRP._deltaC[MRP._d]] == MRP._WALL
    def is_at_cheese(cls) -> bool:
        return (MRP._r == MRP._hi) and (MRP._c == MRP._hi)
    def is_about_to_repeat(cls) -> bool:
        return (MRP._r == MRP._lo) and (MRP._c == MRP._lo) and (MRP._d==3)

    ...
```

Interface includes I/O: Only MRP knows the data representation, so it must do the I/O.

```
class MRP:
    ...

    @classmethod
    def input(cls) -> None:
        """Input N-by-N maze."""
        _______
    @classmethod
    def print_maze(cls) -> None:
        """Output N-by-N maze, with walls and path."""
        _______

    ...
```

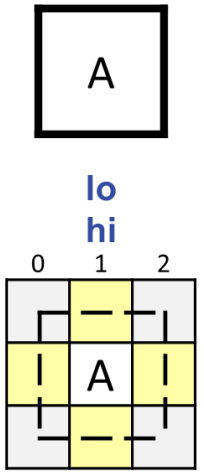


Input: Hard code a trivial initial example.

```
class MRP:
    ...

    @classmethod
    def input(cls) -> None:
        """Input N-by-N maze."""
        # Maze. As per representation invariant.
        MRP._N = 1
        MRP._lo = 1;  MRP._hi = 2 * MRP._N - 1
        MRP._M = [[0 for _ in range(2 * MRP._N + 1)] for _ in range(2 * MRP._N + 1)]
        MRP._M[0][1] = MRP._M[1][0] = MRP._M[1][2] = MRP._M[2][1] = MRP._WALL

        # Rat. Place rat in upper-left cell facing up.
        MRP._r = MRP._lo;  MRP._c = MRP._lo;  MRP._d = 0

        # Path. Establish the rat in the upper-left cell.
        MRP._move = lo;  MRP._M[MRP._r][MRP._c] = MRP._move
```

Use the representation invariants as a guide in helping to establish correct values. Don't worry about trying to avoid needless assignments; it's better to be complete than to risk missing something.

Slight language extension: Multiple lefthand sides for assignment statement.



Input: Hard code a trivial initial example.

```
class MRP:
    ...

    @classmethod
    def input(cls) -> None:
        """Input N-by-N maze."""
        # Maze. As per representation invariant.
        MRP._N = 1
        MRP._lo = 1;  MRP._hi = 2 * MRP._N - 1
        MRP._M = [[0 for _ in range(2 * MRP._N + 1)] for _ in range(2 * MRP._N + 1)]
        MRP._M[0][1] = MRP._M[1][0] = MRP._M[1][2] = MRP._M[2][1] = MRP._WALL

        # Rat. Place rat in upper-left cell facing up.
        MRP._r = MRP._lo;  MRP._c = MRP._lo;  MRP._d = 0

        # Path. Establish the rat in the upper-left cell.
        MRP._move = lo;  MRP._M[MRP._r][MRP._c] = MRP._move

    ...
```

Input: Invoke from the client.

```
class RunMaze:
    ...

    @classmethod
    def _input(cls) -> None:
        """
        Input a maze of arbitrary size, or output “malformed input” and stop
        if the input is improper. Input format: TBD.
        """

        MRP.input()
```

Output: Straightforward, so knock it off now, for the general case.

```
class MRP:
    ...

    @classmethod
    def print_maze(cls) -> None:
        """Output N-by-N maze, with walls and path."""
        for r in range(MRP._lo - 1, MRP._hi + 2):
            for c in range(MRP._lo - 1, MRP._hi + 2):
                if MRP._M[r][c] == MRP._WALL: s = "#"
                elif ((MRP._M[r][c] == MRP._NO_WALL) or
                            (MRP._M[r][c] == MRP._UNVISITED)): s = " "
                else: s = str(MRP._M[r][c]) + ""
                print((s + "  ")[0:3], end='')
            print()

    ...
```

Output: Invoke from the client.

```
class RunMaze:
    ...

    @classmethod
    def _output(cls) -> None:
        """Output the direct path found, or “unreachable” if there is none."""
        if not(MRP.is_at_cheese()): print ("Unreachable")
        else: MRP.print_maze()
    ...
```

Commentary : Design rules for abstract data types.

- Prefer fine-grained micro-operations over coarse-grained macro-operations.
  - E.g., `turn_clockwise` rather than `Pirouette`.
- It is better to support operations that are defined relative to the state than it is to reveal portions of the state itself. Avoid leaking details of any particular data representation.
  - E.g., `is_at_cheese` rather than `get_row` and `get_column`.
  - E.g., `turn_clockwise` rather than `get_direction` and `set_direction`.
- Avoid macro-operations that embody algorithmic details that belong in the client.
  - E.g., `RunMaze._solve` rather than `MRP.solve` .

File and Module Structure:

File run_maze.py

```
from mrp import MRP
class RunMaze:
    ⟨Declarations and definitions of class RunMaze⟩

RunMaze.main()   # Invoke the program
```

File mrp.py

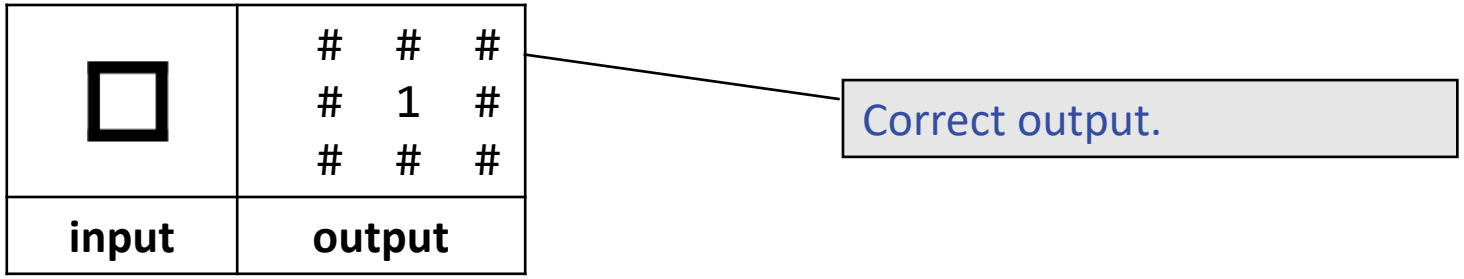
```
class MRP:
    ⟨Declarations and definitions of class MRP⟩
```

In a directory for the program, place each class (e.g., one with the camel-case name `MyFoo`) in its own file with a related lower-case name (e.g., `my_foo.py`).

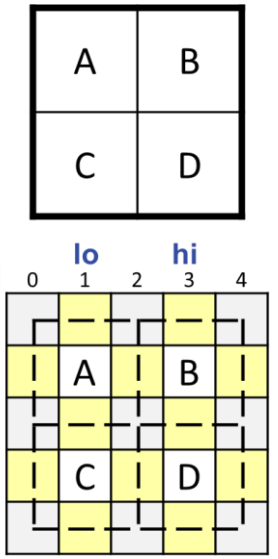
The file of the principle class imports classes it needs, as shown, and invokes its `main` method.

Controlled Testing: At first, use an empty stub for `solve`.

Test 1: Check for syntax errors, and check input/output framework.

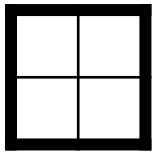


☞ **Test programs incrementally.**



Controlled Testing: Change `input` to hard-code a 2-by-2 maze, but still use an empty stub for `solve`.

Test 2: Check `output`.

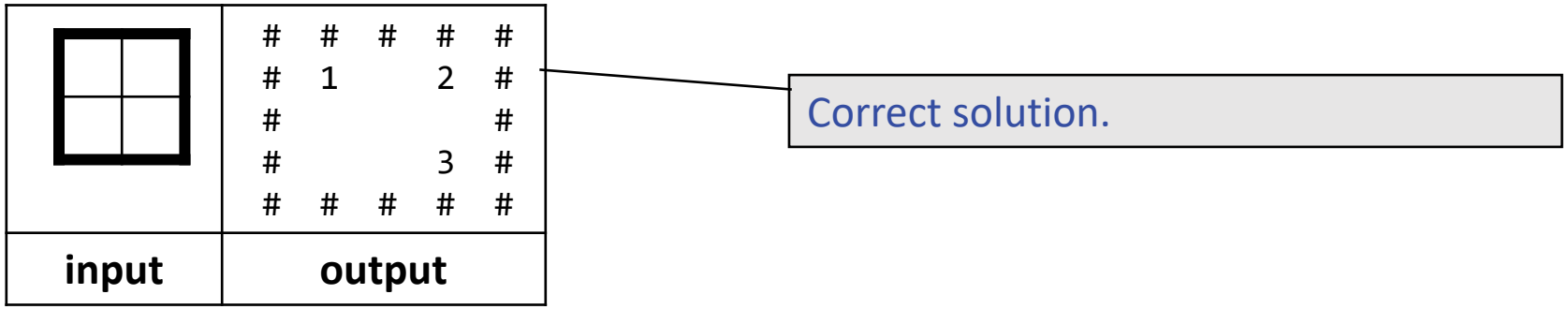
input	output
	Unreachable

Expected output since `solve` is just a stub. Validation of code for message.

☞ **Test programs incrementally.**

Controlled Testing: Now use the real code for `solve`.

Test 3: Further check of `output`, and check of `solve` for an empty 2-by-2 maze.



Controlled Testing: Change `input` to hard-code a 2-by-2, with an obstacle.

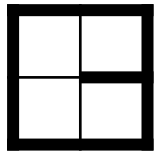
Test 4: Further check of `solve`.

input	output
	# # # # # # 1 # # # # # # # 2 3 # # # # # #

Correct solution. Appears to be going counter-clockwise, but this is an illusion: It is making its way around the obstacle clockwise when it stumbles into the cheese.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

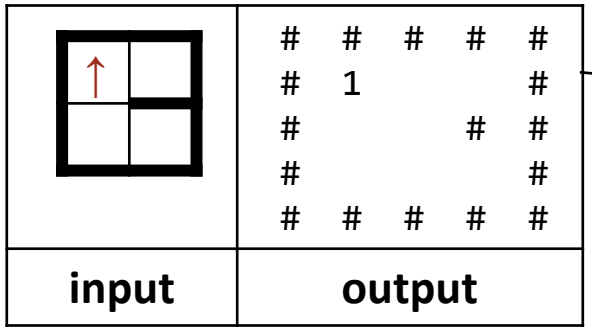
Test 5: Further check of `solve`.

input	output
	# # # # # # 3 2 # # # # # 4 5 # # # # # #

Anticipated incorrect solution. We are doing a complete exploration, and don't bother to detect the cul-de-sac. As a result, we overwrite the path, and leave a mess.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.

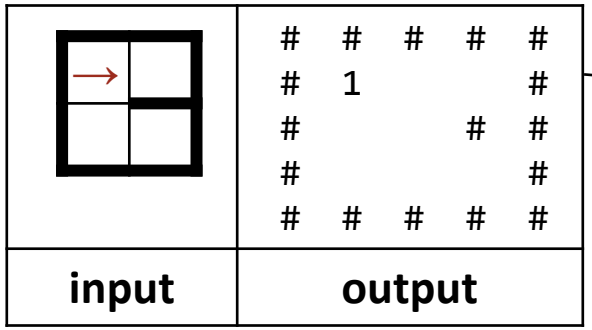


input	output
	# # # # # # 1 # # # # # # # # # # #

Replay.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.



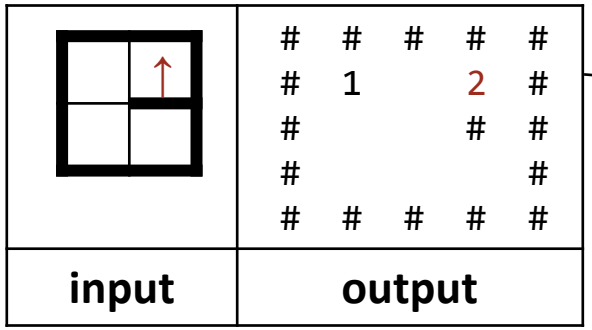
```
# # # # #
# 1     #
#     # #
#       #
# # # # #
```

input	output

Replay.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.



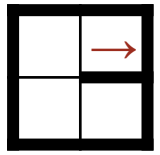
input	output
	# # # # # # 1 2 # # # # # # # # # # #

Replay.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.

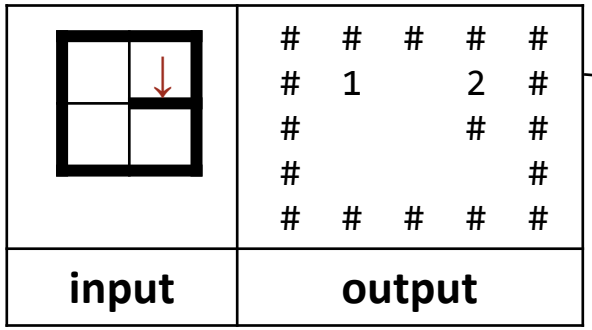
input	output

	`# # # # #`
	`# 1 2 #`
	`# # #`
	`# #`
	`# # # # #`

Replay.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.



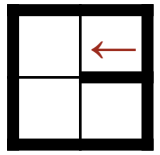
```
# # # # #
# 1     2 #
#       # #
#         #
# # # # #
```

input	output

Replay.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

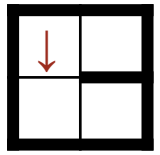
Test 5: Further check of `solve`.

input	output
	# # # # # # 1 2 # # # # # # # # # # #

Replay. This is the moment when we need to detect the imminent re-entry to a cell that is currently on the path.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

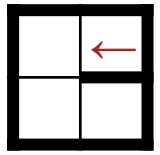
Test 5: Further check of `solve`.

input	output
	# # # # # # 3 2 # # # # # # # # # # #

We ignored the issue, and overwrote the 1 with a 3.

Controlled Testing: Change `input` to hard-code a 2-by-2, with a cul-de-sac.

Test 5: Further check of `solve`.

input	output
	# # # # # # 1 2 # # # # # # # # # # #

Backing up, we need to prevent this.

Algorithm (Revised)

Algorithm: Proceed only if about to enter a cell that is not on the current path.

```
class RunMaze:
    ...

    @classmethod
    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists."""
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()
    ...
```

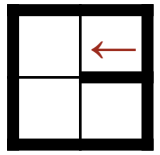
Add the check …

… and introduce `retract` to handle the cul-de-sac case.

Extend MRP: Add `is_facing_unvisited` to interface.

```
class MRP:
    ...

    @classmethod
    def is_facing_unvisited() -> bool:
        return MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                      ][MRP._c + 2 * MRP._deltaC[MRP._d]] == MRP._UNVISITED

     ...
```

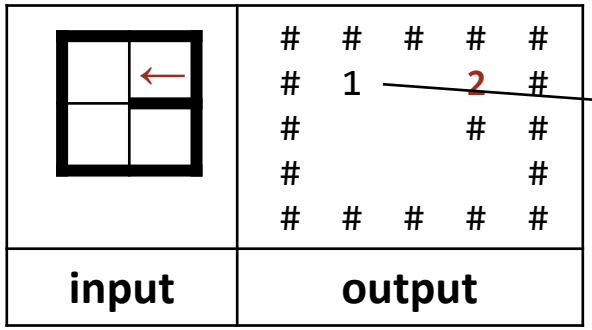
Retract:

input	output
	# # # # # # 1 2 # # # # # # # # # # #

The next step from here needed to detect the imminent re-entry to a cell that is currently on the path, but didn't bother.

Note: `@classmethod` decorators have been omitted from this slide for brevity.

Retract:



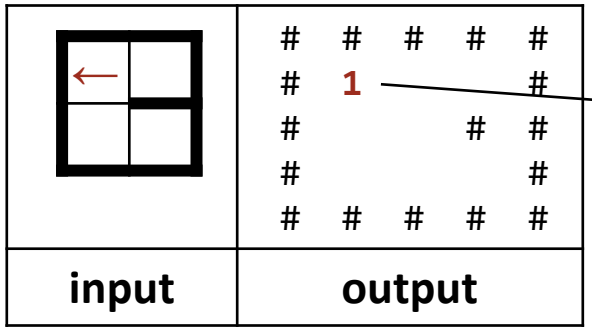
The next step from here needed to detect the imminent re-entry to a cell that is currently on the path, but didn't bother.

Need to undo the `step_forward` that took us into the cul-de-sac.

```
def step_forward(cls) -> None:
    MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
    MRP._move += 1;  MRP._M[MRP._r][MRP._c] = MRP._move
```

Note: `@classmethod` decorators have been omitted from this slide for brevity.

Retract:



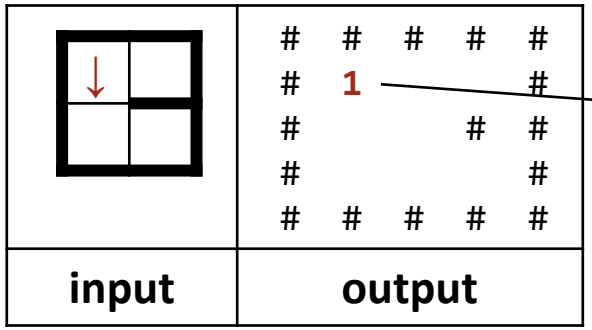
The next step from here needed to detect the imminent re-entry to a cell that is currently on the path, but didn't bother.

Need to undo the `step_forward` that took us into the cul-de-sac.

```
def step_forward(cls) -> None:
    MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
    MRP._move += 1;  MRP._M[MRP._r][MRP._c] = MRP._move
def step_backward(cls) -> None:
    MRP._M[MRP._r][MRP._c] = MRP._UNVISITED;  MRP._move -= 1
    MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
```

Note: `@classmethod` decorators have been omitted from this slide for brevity.

Retract:



The next step from here needed to detect the imminent re-entry to a cell that is currently on the path, but didn't bother.

Need to undo the `step_forward` that took us into the cul-de-sac, and turn as if it had been skipped.

```
def step_forward(cls) -> None:
    MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
    MRP._move += 1;  MRP._M[MRP._r][MRP._c] = MRP._move
def step_backward(cls) -> None:
    MRP._M[MRP._r][MRP._c] = MRP._UNVISITED;  MRP._move -= 1
    MRP._r += 2 * MRP._deltaR[MRP._d];  MRP._c += 2 * MRP._deltaC[MRP._d]
```

Retract: Implemented as follows.

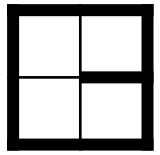
```
class RunMaze:
    ...

    @classmethod
    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.step_backward()
        MRP.turn_counter_clockwise()

    ...
```

Retract: Implemented as follows.

```
class RunMaze:
    ...

    @classmethod
    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.step_backward()
        MRP.turn_counter_clockwise()

    ...
```

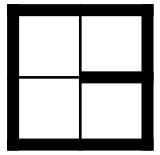
Marker: You have just been deliberately led astray, but we will keep going.

Test 6: Redo Test 5.

input	output
	# # # # # # 1 # # # # # 2 3 # # # # # #

Correct solution. We backed out of the cul-de-sac, and proceeded to the lower-right cell.

Test 6: Redo Test 5.

input	output
	# # # # # # 1 # # # # # # # 2 3 # # # # # #

Correct solution. We backed out of the cul-de-sac, and proceeded to the lower-right cell.

Could we be done? Perhaps, but we will need to test on bigger mazes. It's time to code the general-purpose `input` method.

Input: Start with the hardcoded initial example.

```
class MRP:
    ...

    @classmethod
    def input(cls) -> None:
        """Input N-by-N maze."""“
        # Maze. As per representation invariant.
        MRP._N = 1
        MRP._lo = 1;  MRP._hi = 2 * MRP._N - 1
        MRP._M = [[0 for _ in range(2 * MRP._N + 1)] for _ in range(2 * MRP._N + 1)]
        MRP._M[0][1] = MRP._M[1][0] = MRP._M[1][2] = MRP._M[2][1] = MRP._WALL

        # Rat. Place rat in upper-left cell facing up.
        MRP._r = MRP._lo;  MRP._c = MRP._lo;  MRP._d = 0

        # Path. Establish the rat in the upper-left cell.
        MRP._move = 1;  MRP._M[MRP._r][MRP._c] = MRP._move

    ...
```

Input: Identify places to generalize.

```
class MRP:
    ...

    @classmethod
    def input(cls) -> None:
        """Input N-by-N maze."""
        # Maze. As per representation invariant.
☞       MRP._N = ⟨value for _N⟩
        MRP._lo = 1;  MRP._hi = 2 * MRP._N - 1
        MRP._M = [[0 for _ in range(2 * MRP._N + 1)] for _ in range(2 * MRP._N + 1)]
☞       ⟨Define each element of _M⟩

        # Rat. Place rat in upper-left cell facing up.
        MRP._r = MRP._lo;  MRP._c = MRP._lo;  MRP._d = 0

        # Path. Establish the rat in the upper-left cell.
        MRP._move = 1;  MRP._M[MRP._r][MRP._c] = MRP._move

    ...
```

Input: Create a class for rapid prototyping.

```
# Rapid prototyping harness
class RapidPrototype:
```

```
    # Simplified relevant code from input().
☞   _N = ⟨value for _N⟩
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
☞   ⟨Define each element of _M⟩
```

Input: Provide needed context.

```
# Rapid prototyping harness
class RapidPrototype:
    # Relevant constants.
    _UNVISITED = 0;  _WALL = -1;  _NO_WALL = 0
```

```
    # Simplified relevant code from input().
☞   _N = ⟨value for _N⟩
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
☞   ⟨Define each element of _M⟩
```

Input: Simulate the input file in a string variable, and split it into lines.

```
# Rapid prototyping harness
class RapidPrototype:
    # Relevant constants.
    _UNVISITED = 0;  _WALL = -1;  _NO_WALL = 0

    # Input file split into lines.
    _file = "2\nxxxxx\nx   x\nx   x\nx   x\nxxxxx"
    _lines = _file.split("\n")

    # Simplified relevant code from input().
☞   _N = ⟨value for _N⟩
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
☞   ⟨Define each element of _M⟩
```

Input: Simulate per-line input

```
# Rapid prototyping harness
class RapidPrototype:
    # Relevant constants.
    _UNVISITED = 0;  _WALL = -1;  _NO_WALL = 0

    # Input file split into lines.
    _file = "2\nxxxxx\nx   x\nx   x\nx   x\nxxxxx"
    _lines = _file.split("\n")

    # Simplified relevant code from input().
☞   _N = int(_lines[0]); del _lines[0]
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
☞   for r in range(_lo - 1, _hi + 2):
        _line = _lines[0]; del _lines[0]
        ⟨Define each element of the r-th row of _M from the _line⟩
```

```
xxxxx
x   x
x   x
x   x
xxxxx
['2', 'xxxxx', 'x   x', 'x   x', 'x   x', 'xxxxx']
2
```

```
# Rapid prototyping harness
class RapidPrototype:
    # Relevant constants.
    _UNVISITED = 0;  _WALL = -1;  _NO_WALL = 0

    # Input file split into lines.
    _file = "2\nxxxxx\nx   x\nx   x\nx   x\nxxxxx"; print(_file)
    _lines = _file.split("\n"); print(_lines)

    # Simplified relevant code from input().
    _N = int(_lines[0]); del _lines[0]; print(_N)
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
    for r in range(_lo - 1, _hi + 2):
        _line = _lines[0]; del _lines[0]
        ⟨Define each element of the r-th row of _M from the _line⟩
```

Use print statements, as needed, for diagnostic output

Input: Define elements of array M.

```
# Rapid prototyping harness
class RapidPrototype:
    # Relevant constants.
    _UNVISITED = 0;  _WALL = -1;  _NO_WALL = 0

    # Input file split into lines.
    _file = "2\nxxxxx\nx   x\nx   x\nx   x\nxxxxx"
    _lines = _file.split("\n")

    # Simplified relevant code from input().
☞   _N = int(_lines[0]); del _lines[0]
    _lo = 1;  _hi = 2 * _N - 1
    _M = [[0 for _ in range(2 * _N + 1)] for _ in range(2 * _N + 1)]
☞   for r in range(_lo - 1, _hi + 2):
        _line = _lines[0]; del _lines[0]
        for c in range(_lo - 1, _hi + 2):
            if (r % 2 == 1) and (c % 2 == 1): _M[r][c] = _UNVISITED
            elif line[c:c+1] == " ": _M[r][c] = _NO_WALL
            else: _M[r][c] = _WALL
```

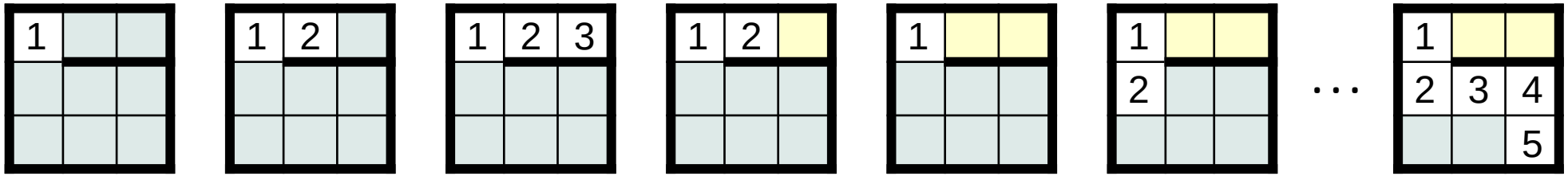
Note: `@classmethod` decorators have been omitted from this slide for brevity.

Input: Retrofit the prototype into `MRP.input()`, including the file simulated in a string variable.

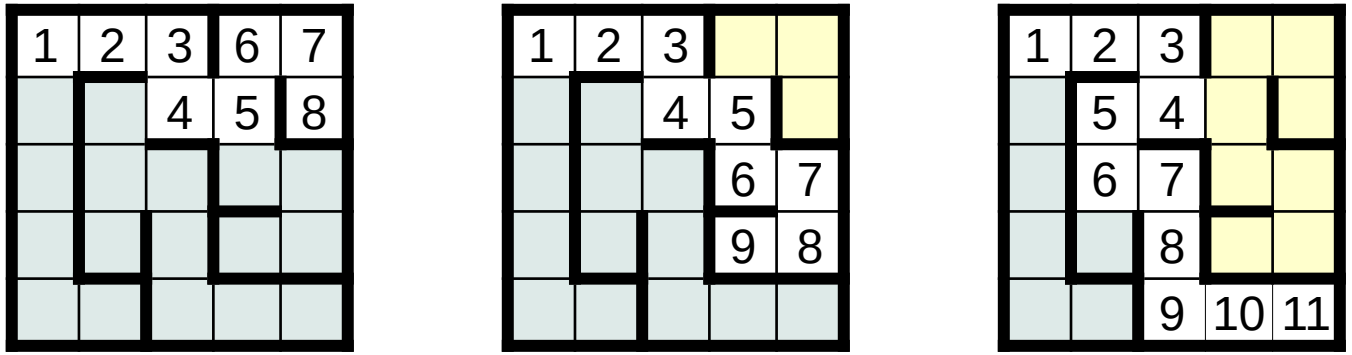
```
class MRP:
    ...

    def input(cls) -> None:
        """Input N-by-N maze."""
        # Input file split into lines.
        file = "2\nxxxxx\nx   x\nx   x\nxxxxx"
☞       lines = file.split("\n")

        # Maze. As per representation invariant.
        MRP._N = int(lines[0]); del lines[0]
        MRP._lo= 1; MRP._hi = 2 * MRP._N - 1
☞       MRP._M = [[0 for _ in range(2 * MRP._N + 1)] for _ in range(2 * MRP._N + 1)]
        for r in range(MRP._lo - 1, MRP._hi + 2):
            line = lines[0]; del lines[0]
☞           for c in range(MRP._lo - 1, MRP._hi + 2):
                if (r % 2 == 1) and (c % 2 == 1): MRP._M[r][c] = MRP._UNVISITED
                elif line[c:c+1] == " ":
                    MRP._M[r][c] = MRP._NO_WALL
                else: MRP._M[r][c] = MRP._WALL

        # Rat. Place rat in upper-left cell facing up.
            ...
    ...
```

Controlled Testing: Try every sort of maze you can think of.

Deeper cul-de-sacs



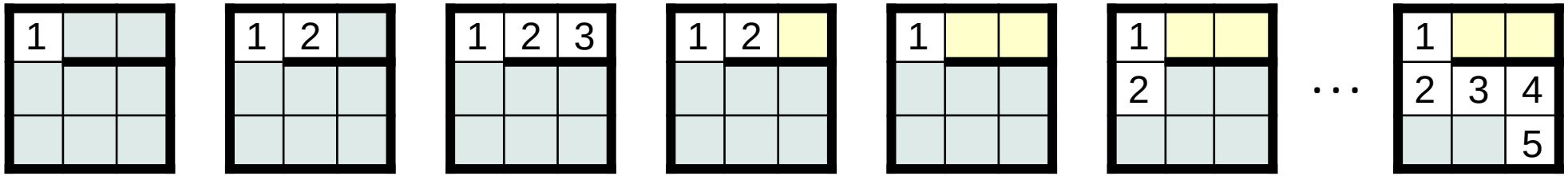
Higgledy-piggledy cul-de-sacs



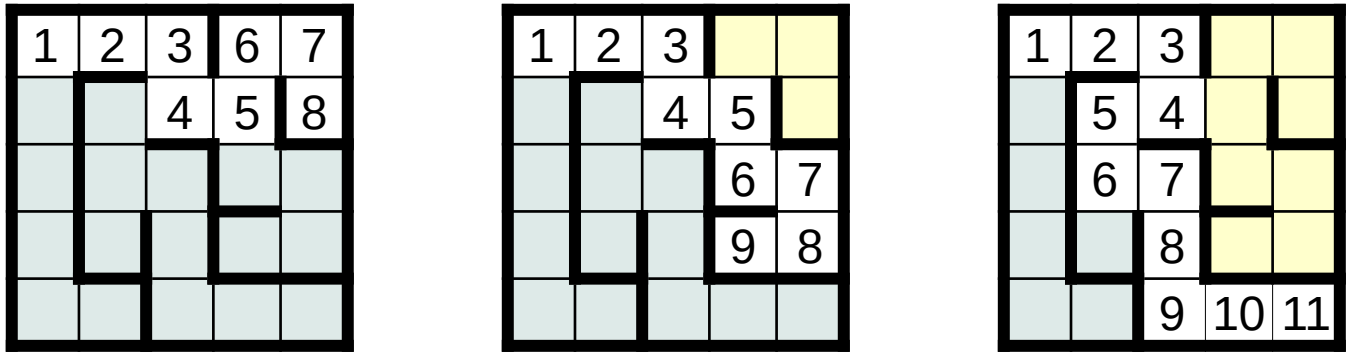
☞ **Test programs thoroughly.**

Controlled Testing: But how can you know when you are done?

Deeper cul-de-sacs



Higgledy-piggledy cul-de-sacs



☞ **Beware of premature self-satisfaction.**

Controlled Testing: But how can you know when you are done?

Review Code:

- You were supposed to be very systematic, but did you consider every case?

Review Test data:

- You were supposed to be very systematic, but did you consider every case?

☞ **Test programs thoroughly.**

Controlled Testing: But how can you know when you are done?

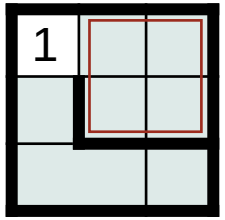
Review Code:

- You were supposed to be very systematic, but did you consider every case?

Review Test data:

- You were supposed to be very systematic, but did you consider every case?

Do you have to just keep trying until you think of a room-shaped cul-de-sac?



. . .

☞ **Test programs thoroughly.**

Controlled Testing: But how can you know when you are done?

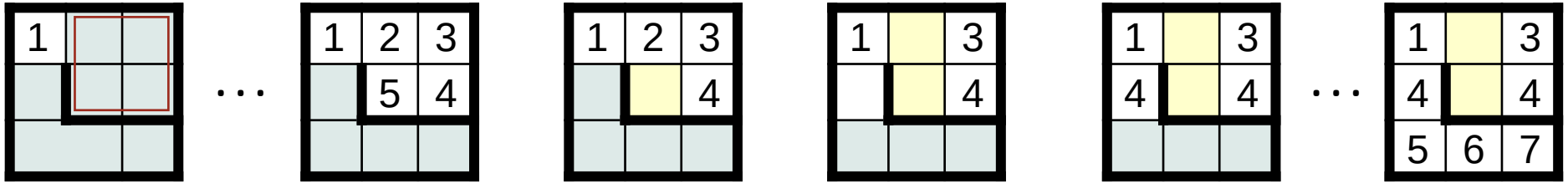
Review Code:

- You were supposed to be very systematic, but did you consider every case?

Review Test data:

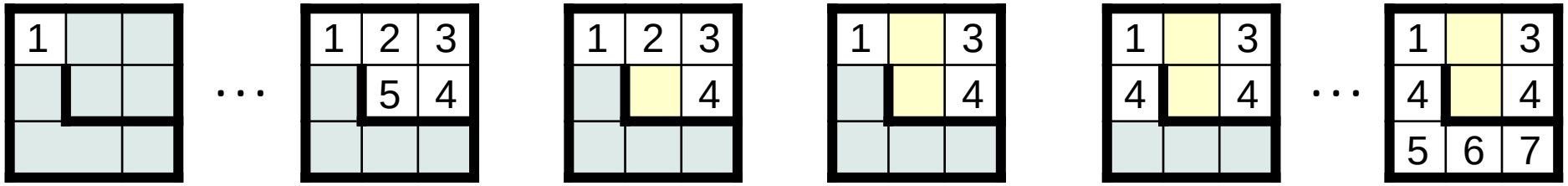
- You were supposed to be very systematic, but did you consider every case?

Do you have to just keep trying until you think of a room-shaped cul-de-sac?



Aargh! We only considered corridor-shaped cul-de-sacs.

☞ **Test programs thoroughly.**



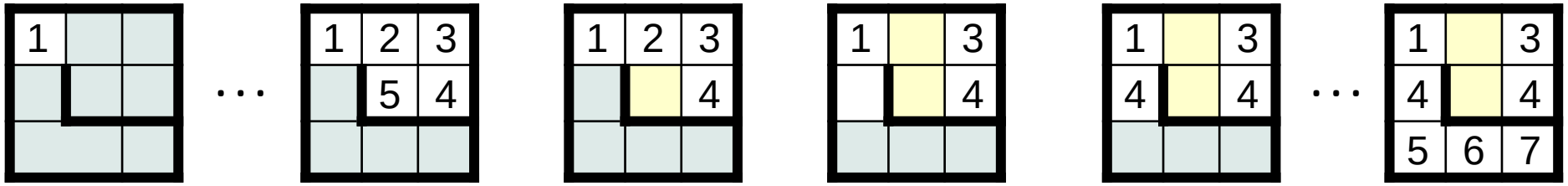
Retract: Recall that the implementation was as follows.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.step_backward()
        MRP.turn_counter_clockwise()

    ...
```

Recall: We deliberately led you astray, but we kept going.

This didn't *unwind* the traversal of the cul-de-sac; it only undid the first step *into* the cul-de-sac. This worked fine even for deep corridor-shaped cul-de-sacs (which could be backed out of one "first-step" at a time).

Note: `@classmethod` decorators have omitted from this slide forward for brevity.

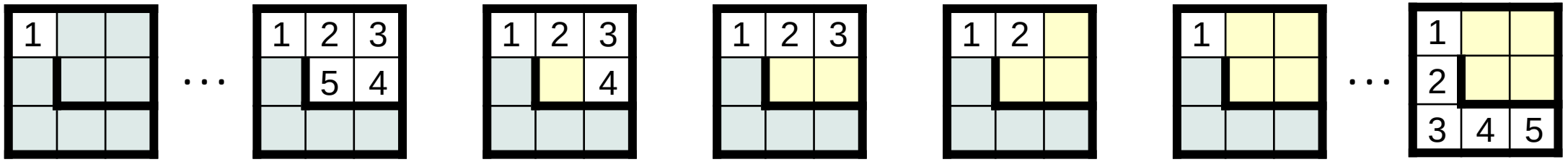


Retract: To be implemented now as follows.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        while not-unwound:
            MRP.face_previous()
            MRP.step_backward()
        MRP.turn_counter_clockwise()

    ...
```

Picking up the "bread crumbs".

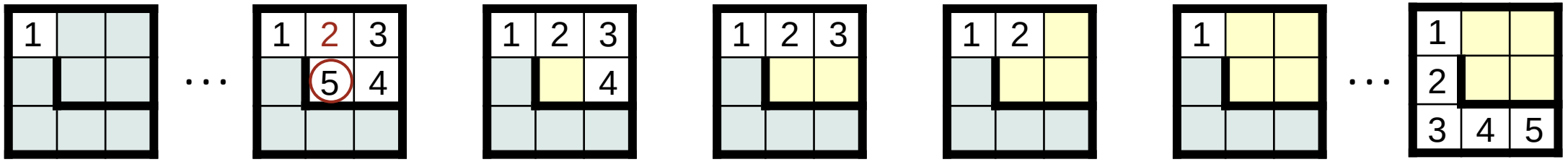
Correction: Now we will truly unwind the path.



Algorithm (Revised)

`retract` is coded in class MRP in order to have direct access to the data representation.

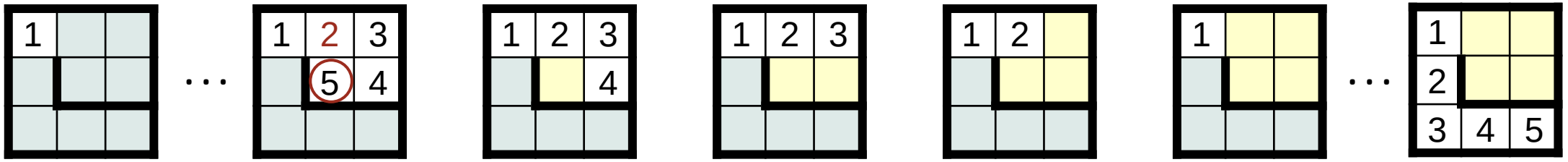
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d      # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction      # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

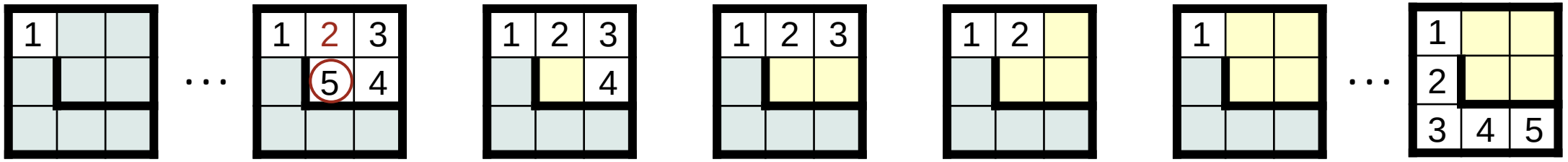
We record the identity of the about-to-be-revisited neighbor



Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

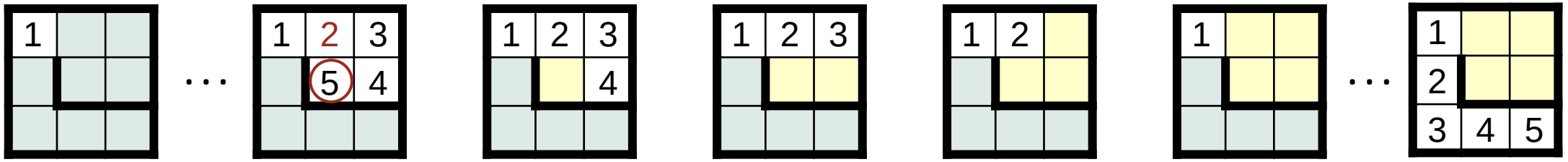
We record the identity of the about-to-be-revisited neighbor, and the direction we were facing when we detected it.



Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

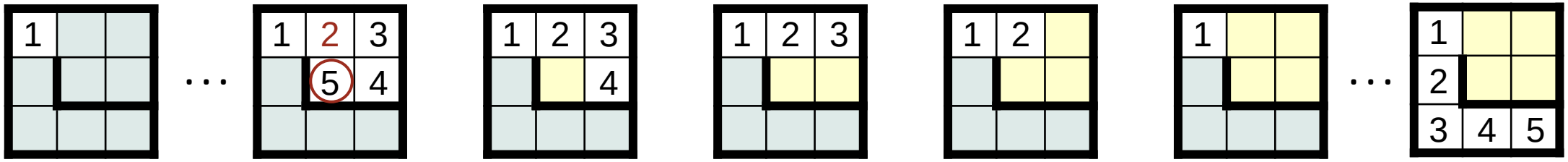
We record the identity of the about-to-be-revisited neighbor, and the direction we were facing when we detected it. We stop unwinding when we get to it



Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

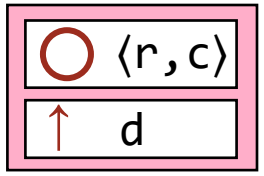
We record the identity of the about-to-be-revisited neighbor, and the direction we were facing when we detected it. We stop unwinding when we get to it, and restore the direction in which we were facing.

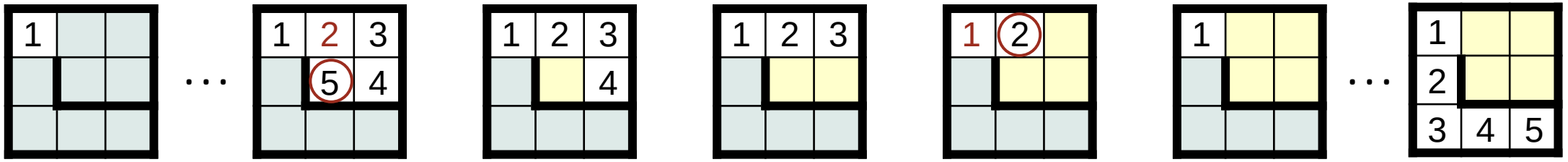


Algorithm (Revised)

Trace: There are actually two separate cul-de-sacs: one detected from 5 (facing 2), and the other from 2 (facing 1).

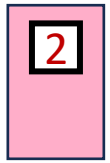
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

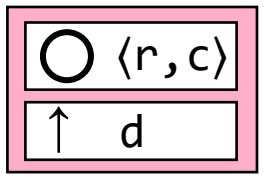


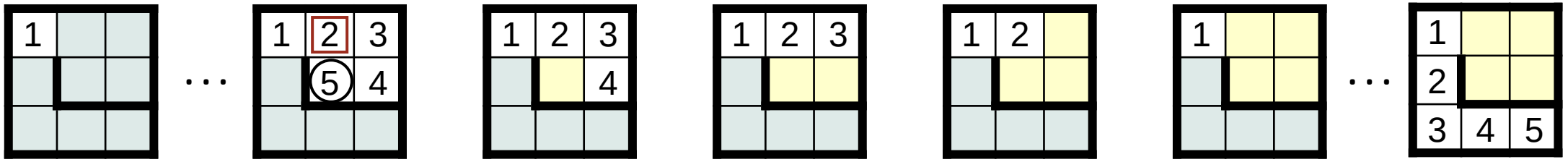


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

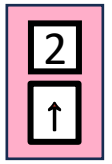



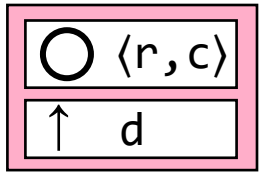


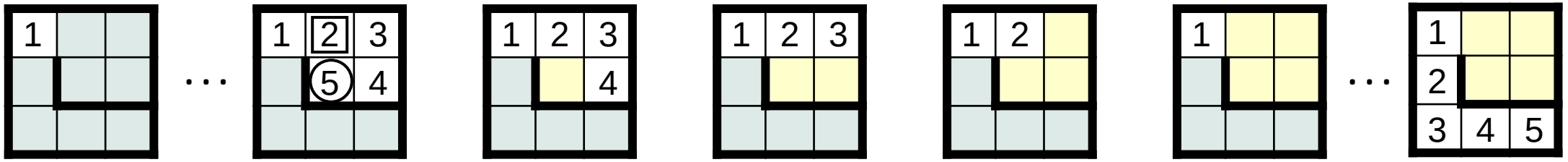


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

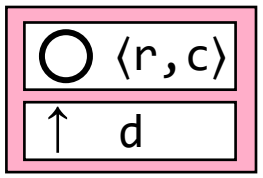


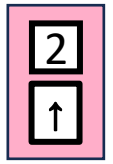


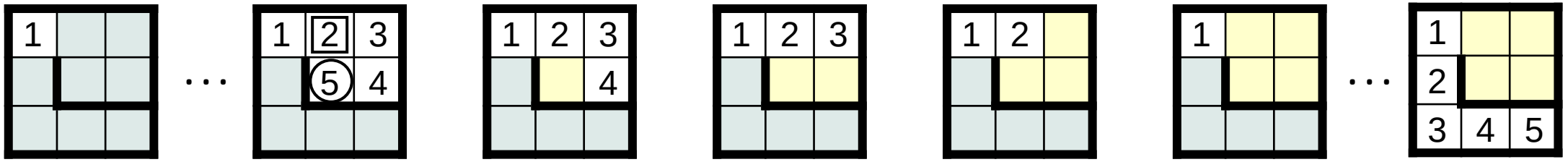


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

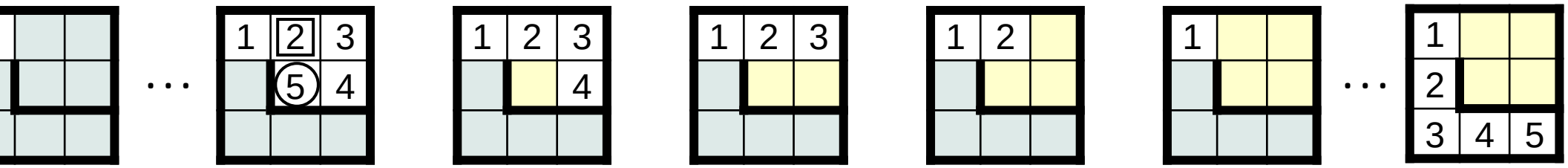






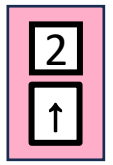
Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



Algorithm (Revised)

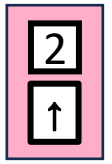
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

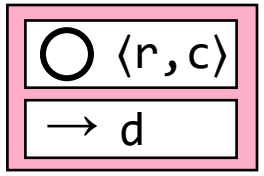


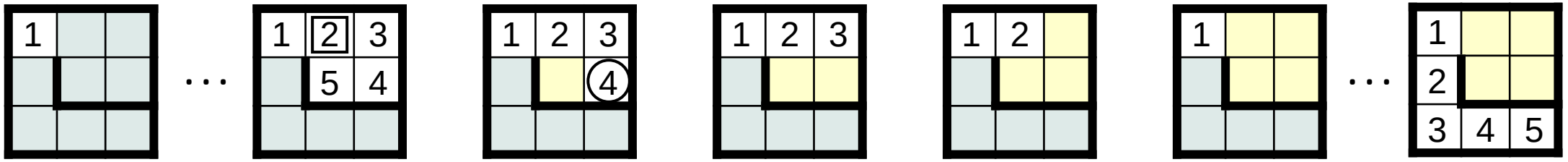
time →

Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

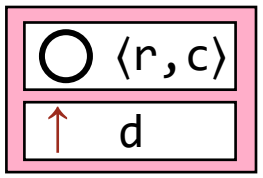


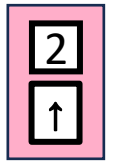


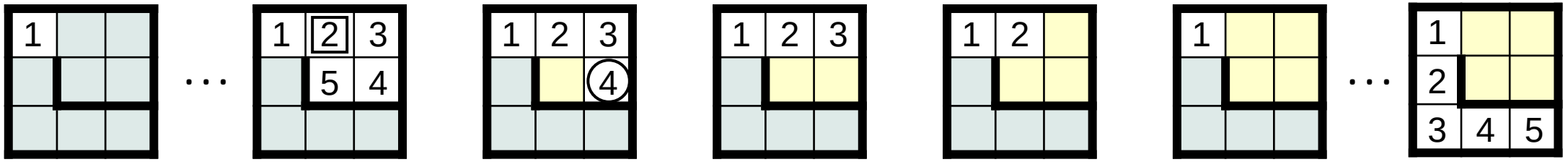


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

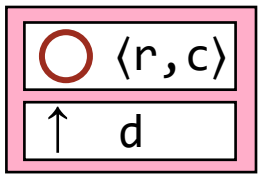


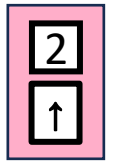




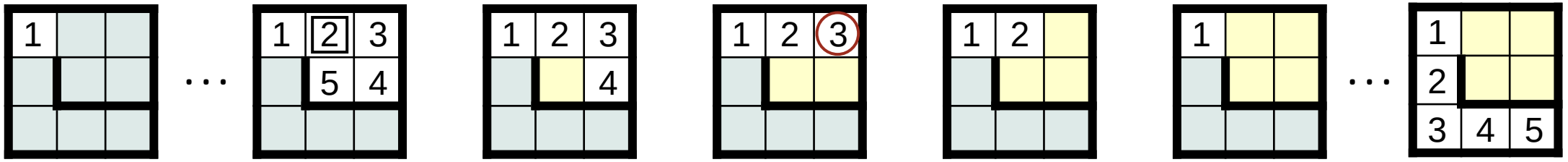
Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



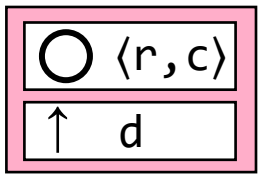


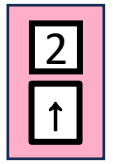
time →

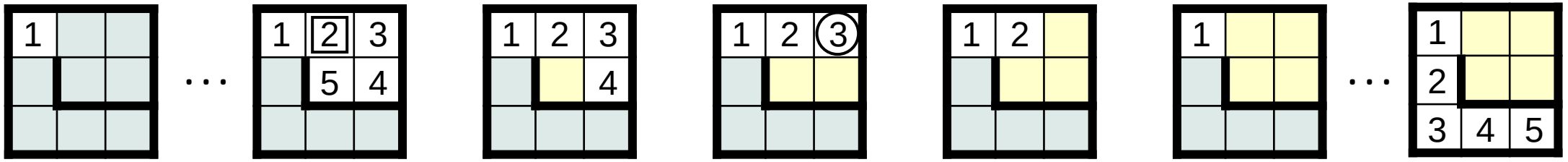


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

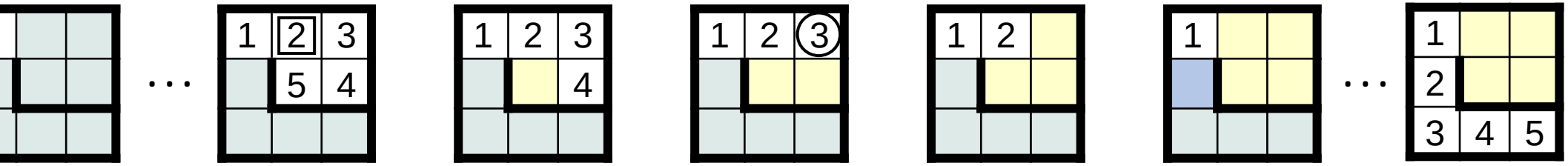






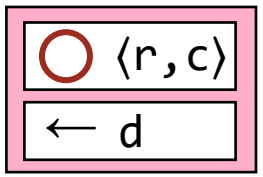
Algorithm (Revised)

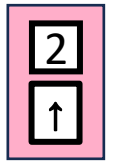
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



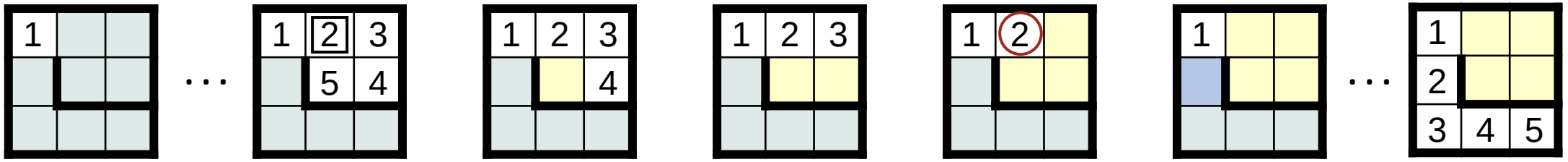
Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



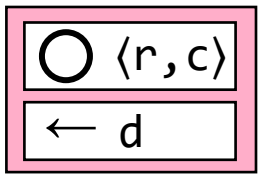


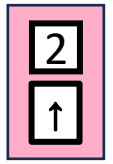
time →

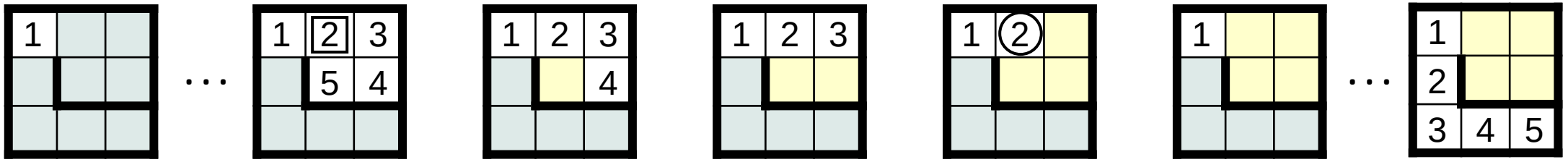


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

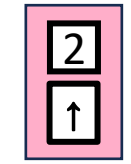


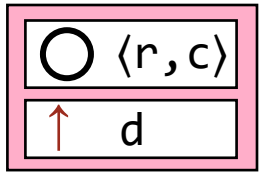


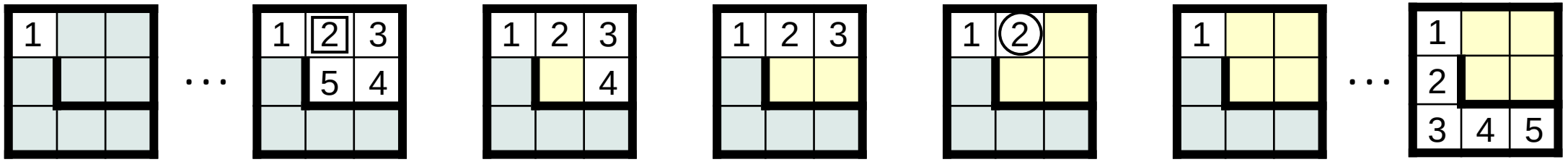


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

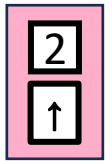


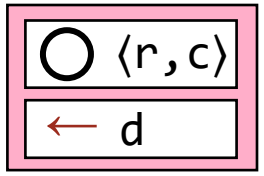


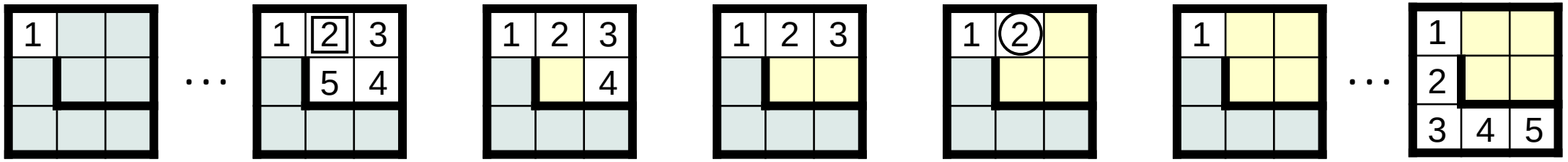


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

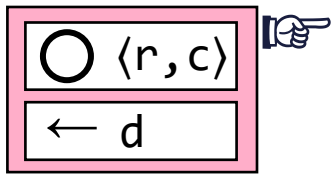


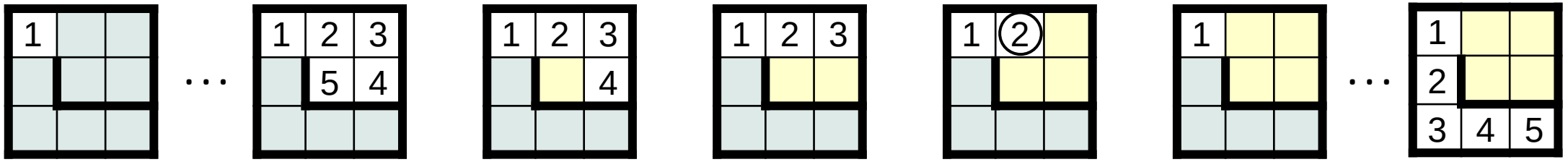




Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

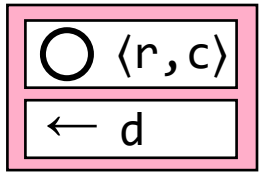


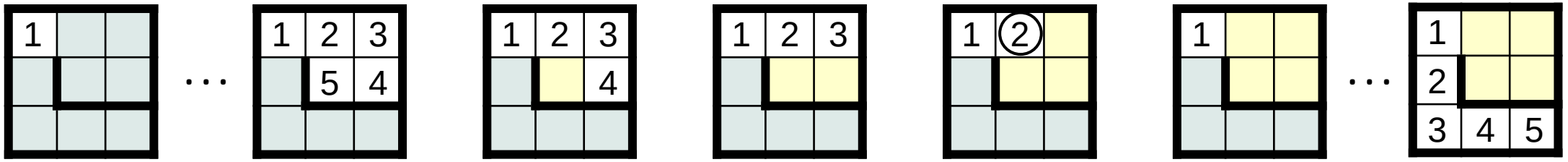


Algorithm (Revised)

Second call to `retract`.

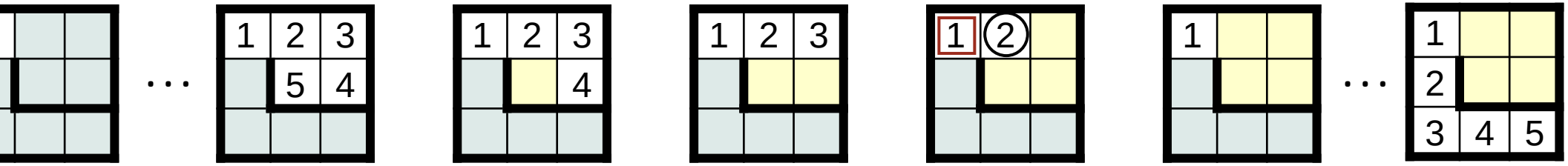
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

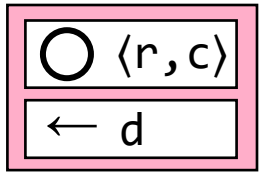




Algorithm (Revised)

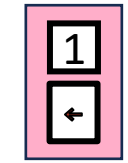
```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

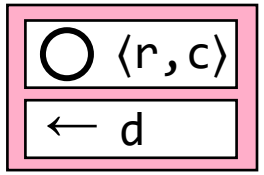


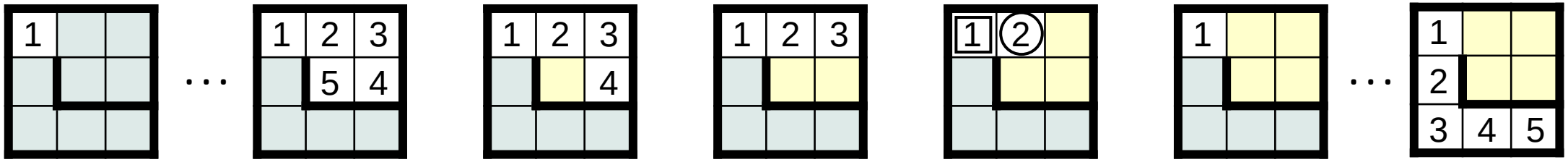


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

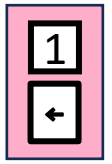


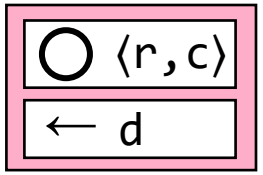


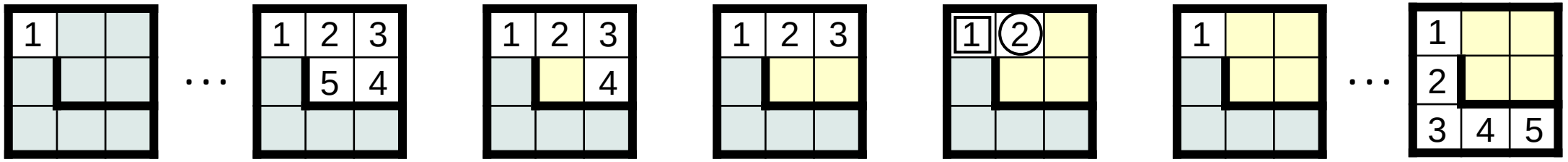


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

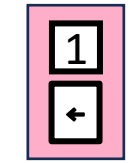


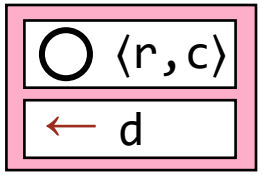


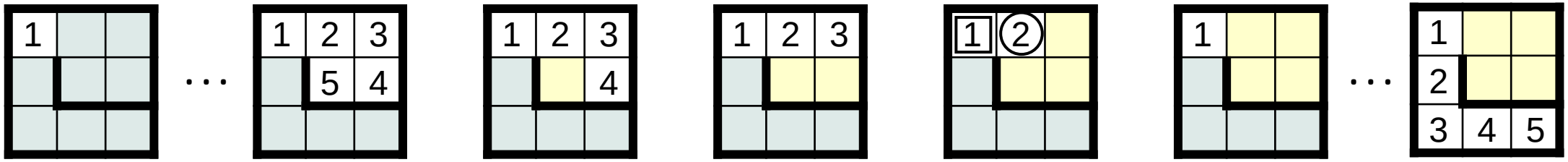


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

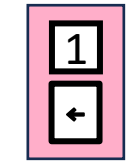


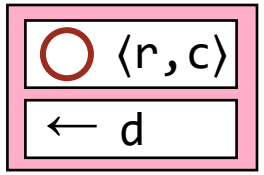




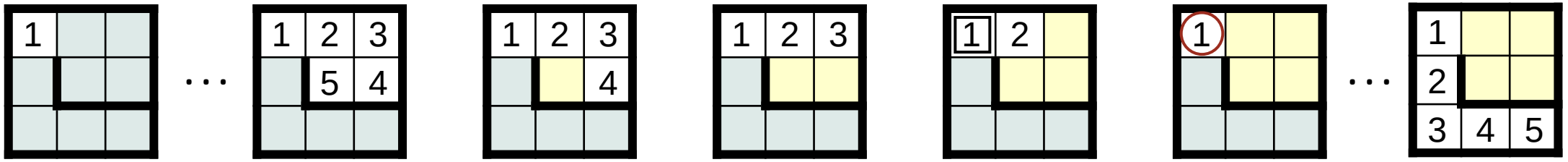
Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```



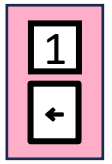


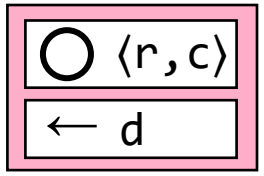
time →

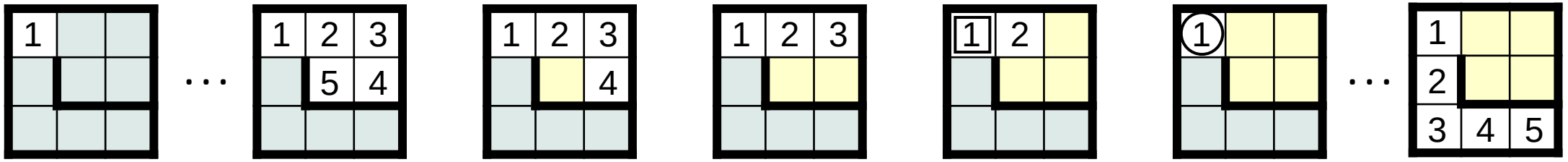


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

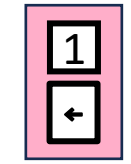


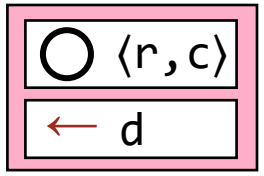


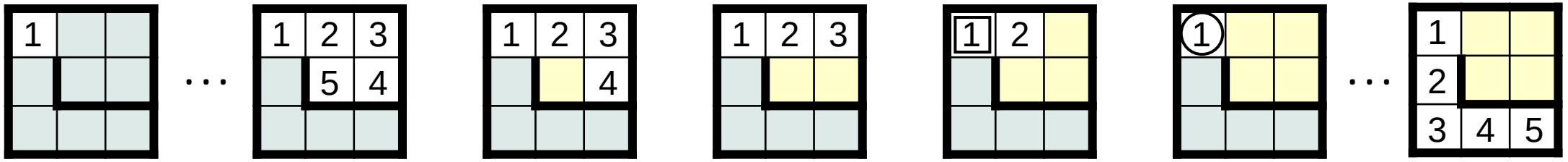


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

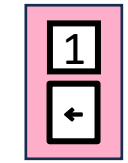


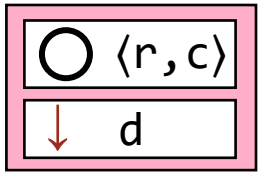


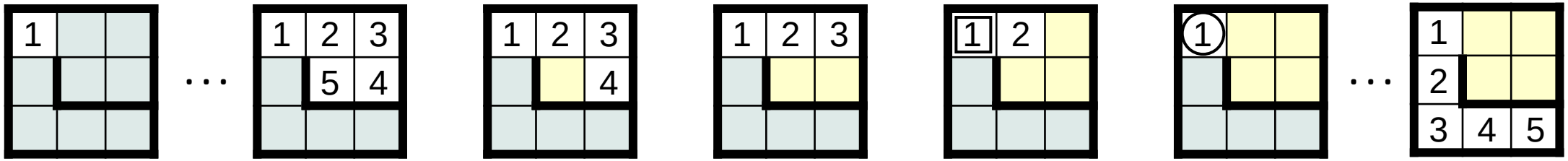


Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

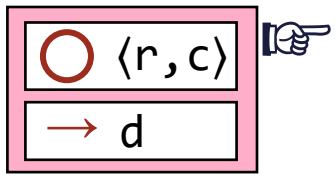






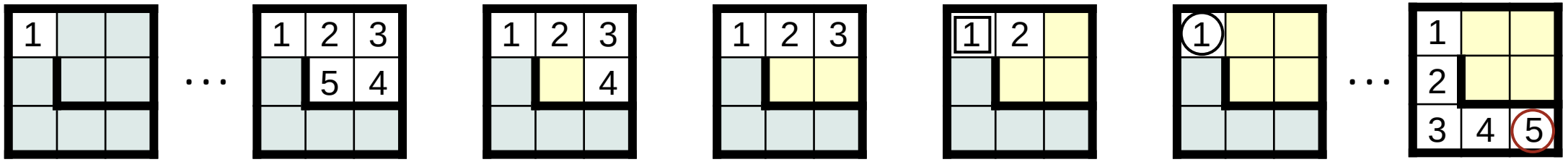
Algorithm (Revised)

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d       # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction       # Restore direction.
        MRP.turn_counter_clockwise()
```




```
    ...
```

time →



Recall that `retract` was coded in class `MRP` in order to have direct access to the data representation.

```
class MRP:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d      # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction      # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

Algorithm (Revised)

Recall that `retract` was coded in class MRP in order to have direct access to the data representation. But it really is too algorithmic for MRP, and more properly belongs in RunMaze.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d      # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction      # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

Algorithm (Revised)

Recall that `retract` was coded in class MRP in order to have direct access to the data representation.
But it really is too algorithmic for MRP, and more properly belongs in RunMaze.
But then it wouldn't have access to the data representation, which is *protected* in MRP.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d      # Save direction.
        while MRP._M[MRP._r][MRP._c] != MRP._neighbor_number:
            MRP.face_previous()
            MRP.step_backward()
        MRP._d = MRP._neighbor_direction      # Restore direction.
        MRP.turn_counter_clockwise()

    ...
```

Algorithm (Revised)

Algorithm (Revised)

Recall that `retract` was coded in class MRP in order to have direct access to the data representation.
But it really is too algorithmic for MRP, and more properly belongs in RunMaze.
But then it wouldn't have access to the data representation, which is *protected* in MRP.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

First call to `retract`.

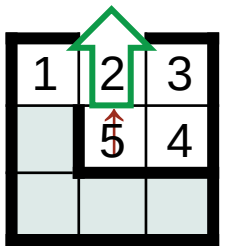
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"



Algorithm (Revised)

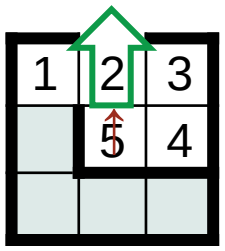
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

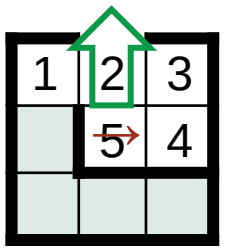
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
    ☞       MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

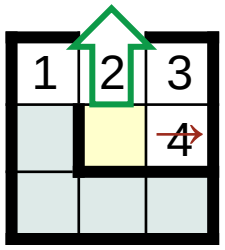
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

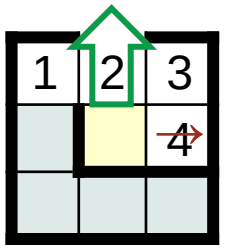
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

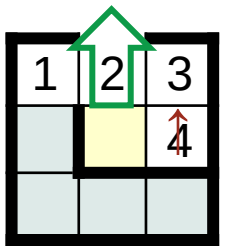
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
    ☞       MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

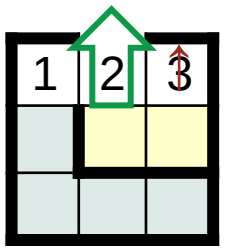
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

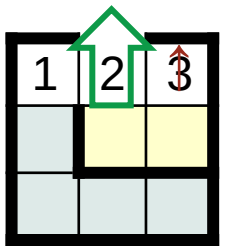
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

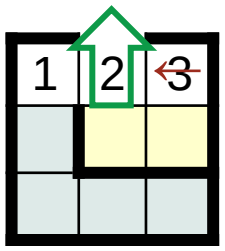
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
    ☞       MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

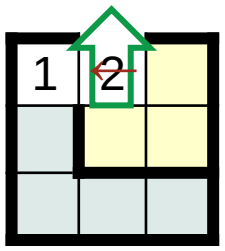
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

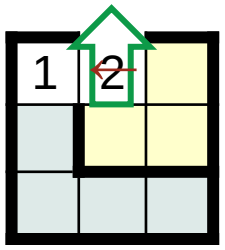
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



Algorithm (Revised)

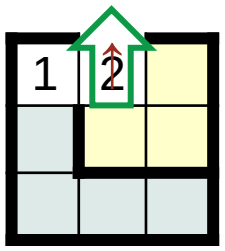
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"
- "Align direction with the arrow"



Algorithm (Revised)

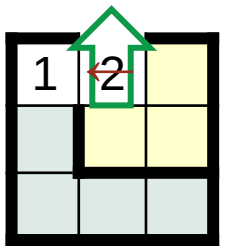
```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
    ☞ MRP.turn_counter_clockwise()

    ...
```

The solution is for MRP to encapsulate the needed code as an extension of its services:

- `record_neighbor_and_direction`
- `is_at_neighbor`
- `restore_direction`

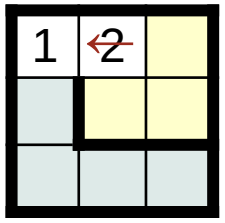
The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"
- "Align direction with the arrow"



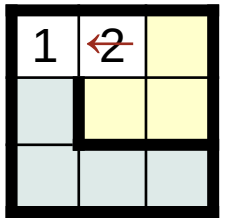
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
    ☞   while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



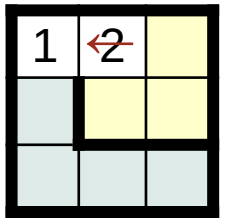
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



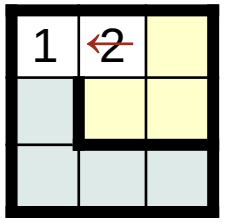
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
        ☞ else: RunMaze.retract()

    ...
```



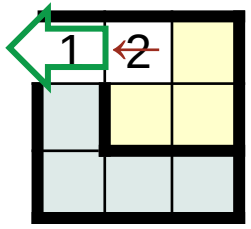
Algorithm (Revised)

Second call to `retract`.

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"

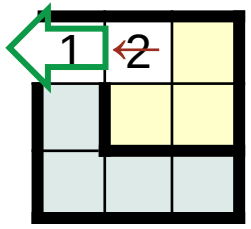


Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"

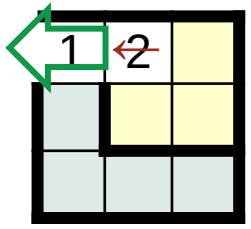


Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
☞           MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"



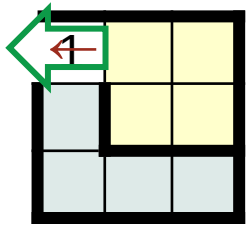
No change

Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"

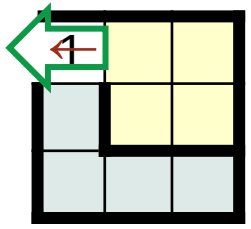


Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
    ☞   while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"

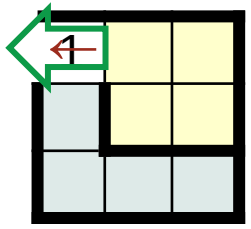


Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
    ☞   MRP.restore_direction()
        MRP.turn_counter_clockwise()

    ...
```

The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"
- "Align direction with the arrow"



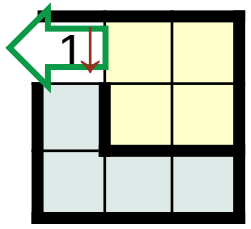
No change

Algorithm (Revised)

```
class RunMaze:
    ...

    def retract(cls) -> None:
        """Unwind abortive exploration."""
        MRP.record_neighbor_and_direction()
        while not(MRP.is_at_neighbor()):
            MRP.face_previous()
            MRP.step_backward()
        MRP.restore_direction()
    ☞ MRP.turn_counter_clockwise()

    ...
```

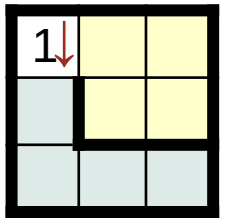
The MRP operations (colloquially) are:

- "Toss an arrow into a neighbor"
- "Detect being in that neighbor"
- "Align direction with the arrow"



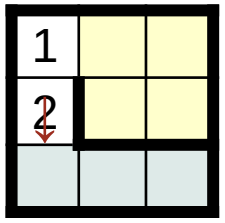
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
    ☞   while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



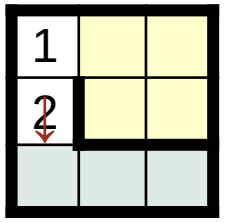
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



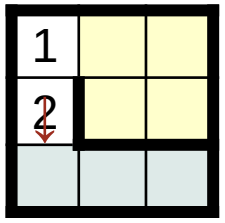
Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



Algorithm (Revised)

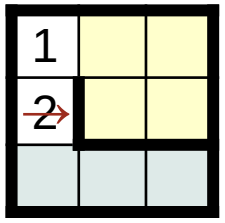
```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists.
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



We're on our way.

Algorithm (Revised)

```
class RunMaze:
    ...

    def _solve(cls) -> None:
        """Compute a ~~direct~~ path through the maze, if one exists."""
        while not(MRP.is_at_cheese()) and not(MRP.is_about_to_repeat()):
            if MRP.is_facing_wall(): MRP.turn_clockwise()
            elif MRP.is_facing_unvisited():
                MRP.step_forward()
                MRP.turn_counter_clockwise()
            else: RunMaze.retract()

    ...
```



We're on our way.

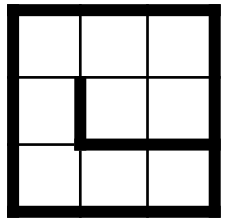
State variables of MRP supporting the notion of an “arrow in a cell”.

```
class MRP:
    ...

    # Recorded state.
    _neighbor_number: int      # Visit number of cell into which the arrow was tossed.
    _neighbor_direction: int   # Direction when the arrow was tossed.

    def record_neighbor_and_direction(cls) -> None:
        """Toss an arrow into the neighboring cell in the direction faced."""
        MRP._neighbor_number = MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                      ][MRP._c + 2 * MRP._deltaC[MRP._d]]
        MRP._neighbor_direction = MRP._d

    def is_at_neighbor(cls) -> bool:
        """Detect being in the same cell as the arrow."""
        return MRP._M[MRP._r][MRP._c] == MRP._neighbor_number

    def restore_direction(cls) -> None:
        """Align direction with the arrow."""
        MRP._d = MRP._neighbor_direction

    ...
```

Remaining Implementation: face_previous, just a Sequential Search.

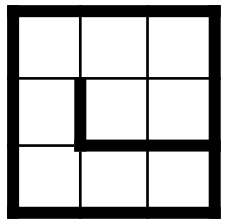
```
class MRP:
    ...

    @classmethod
    def face_previous(cls) -> None:
        MRP._d = 0
        while MRP.is_facing_wall() or (MRP._M[MRP._r][MRP._c] - 1 !=
                                       MRP._M[MRP._r + 2 * MRP._deltaR[MRP._d]
                                              ][MRP._c + 2 * MRP._deltaC[MRP._d]]
            ): MRP._d += 1

    ...
```

Test 7:

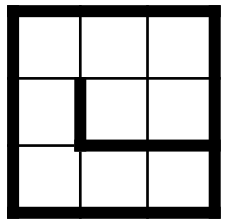
input	output
	# # # # # # # # 1 # # # # # 2 # # # # # # # # # 3 4 5 # # # # # # # #

Testing (Revisited)

Test 7:

input	output
	# # # # # # # # 1 # # # # # 2 # # # # # # # # # 3 4 5 # # # # # # # #

Test 7:

input	output
	# # # # # # # # 1 # # # # # 2 # # # # # # # # # 3 4 5 # # # # # # # #

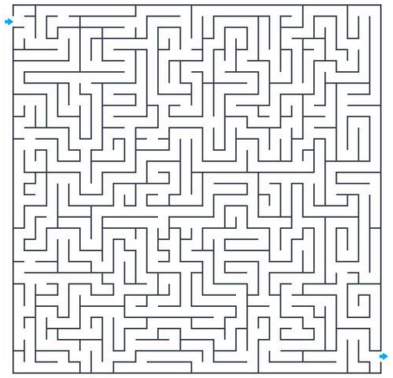
But how can we know there isn't yet another lingering bug?

"Program testing can be used to show the presence of bugs, but never to show their absence!"

— Edsger W. Dijkstra

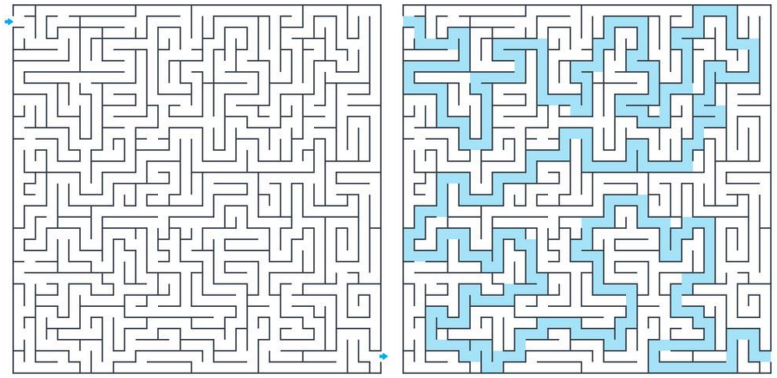
Self-checking: The setting.

It is often easier to automatically check the correctness of a problem solution than it is to find the solution in the first place.



Self-checking: The setting.

It is often easier to automatically check the correctness of a problem solution than it is to find the solution in the first place.

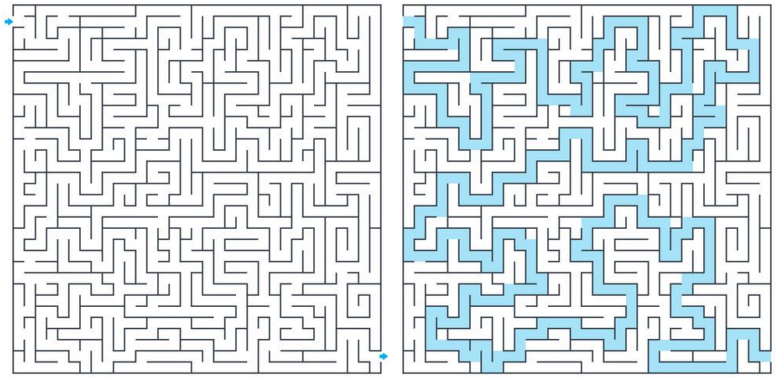


Self-checking: The setting.

It is often easier to automatically check the correctness of a problem solution than it is to find the solution in the first place.

Running a Maze can be viewed as a search problem that either succeeds (by finding a path), or that announces “unreachable”.

Checking the answer “unreachable” is no easier than the original problem because it involves discovering a path that contradicts the unreachability claim.



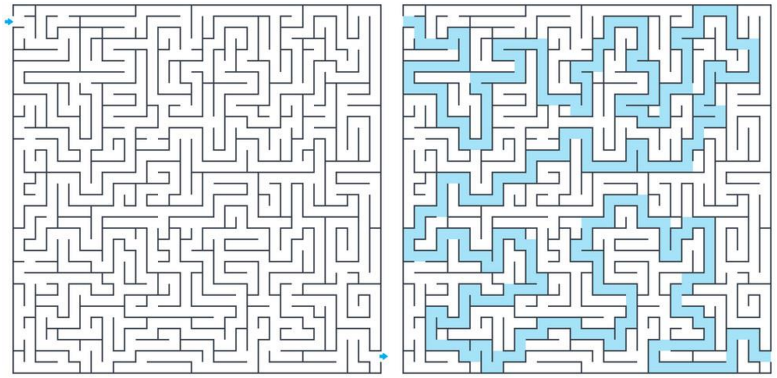
Self-checking: The setting.

It is often easier to automatically check the correctness of a problem solution than it is to find the solution in the first place.

Running a Maze can be viewed as a search problem that either succeeds (by finding a path), or that announces “unreachable”.

Checking the answer “unreachable” is no easier than the original problem because it involves discovering a path that contradicts the unreachability claim.

But if the program claims a path, it can be checked for correctness.



Self-checking: The checking code.

```
class MRP:
    ...
    @classmethod
    def _is_valid_path(cls, r: int, c: int) -> bool:
        """Return False iff rat reached cell ⟨r,c⟩ via an invalid path."""
        if MRP._M[r][c] == MRP._UNVISITED: return True # No claim if UNVISITED.
        else:
            while not((r == MRP._lo) and (c == MRP._lo)):
                # Go to any valid predecessor; return False if there is none.
                d = 0
                while (d < 4) and ((
                        MRP._M[r + MRP._deltaR[d]][c + MRP._deltaC[d]]
                            == MRP._WALL
                        ) or (MRP._M[r + 2 * MRP._deltaR[d]][c + 2 * MRP._deltaC[d]]
                            != (MRP._M[r][c] - 1)
                    )): d += 1
                if d == 4: return False
                r += 2 * MRP._deltaR[d]; c += 2 * MRP._deltaC[d]
            return True # Reached upper-left cell.
    ...
```

Self-checking: The checking code.

```
class MRP:
    ...

    @classmethod
    def is_solution(cls) -> bool:
        """Return False iff rat reached lower-right cell via an invalid path."""
        return MRP._is_valid_path(MRP._hi, MRP._hi)

    ...
```

Self-checking:. Make the assertion the last step in `RunMaze.main`.

```
# Stop execution if path found is not a solution.
    assert MRP.is_solution(), "internal program error"
```

N.B. No warning from the **assert** "confirms" that the solution is correct, provided, of course, that `MRP.is_solution()` does not itself contain a bug. We should test that it does actually return **False** for (some) bad paths, (say) by wantonly buggering paths in `MRP.print_maze()`.

N.B. The code in `MRP.is_valid_path()` is missing a check for the absence of noise off the path.

Exhaustive Bounded Testing:

There are an infinite number of mazes, so exhaustive testing is not possible.

For given N, there are a finite number of N-by-N mazes, so exhaustive testing of up to size N is feasible, in principle. How many are there?

Answer: 2^w, where *w* is the number of places where a wall can either exist or not exist:

- Outer walls must exist.
- Each of N rows of cells has N-1 interior vertical-wall positions.
- Each of N columns of cells has N-1 inerior horizontal-wall positions.

So w = 2*N*(N-1).

Feasible up through N=4.

N	$2^{2 \cdot N \cdot (N-1)}$
1	$2^0 = 1$
2	$2^4 = 16$
3	$2^{12} = 4{,}096$
4	$2^{24} = 16{,}777{,}216$
5	2^{90}

Exhaustive Bounded Testing: Maze generation.

```
class MRP:
    ...
     @classmethod
    def generate_input(cls, N: int, w: int) -> None:
        """Create an N-by-N maze with walls given by the bits of w."""
        # Maze.
        MRP._N = N;  lo = MRP._lo = 1;  hi = MRP._hi = 2 * N - 1
        MRP._M = [[0 for _ in range(2 * N + 1)] for _ in range(2 * N + 1)]
        # ================================================================
        # Set boundary walls.
        for i in range(0, hi + 2):
            MRP._M[lo - 1][i] = MRP._M[hi + 1][i] = MRP._WALL
            MRP._M[i][lo - 1] = MRP._M[i][hi + 1] = MRP._WALL

        # Set 2*n*(n-1) interior walls to the corresponding bits of w.
        for r in range(lo, hi + 1):
            for c in range(lo, hi + 1):
                if (r % 2 == 0 and c % 2 == 1) or (r % 2 == 1 and c % 2 == 0):
                    if w % 2 == 1: MRP._M[r][c] = MRP._WALL
                    else: MRP._M[r][c] = MRP._NO_WALL
                    w = w // 2

        # Rat.
        MRP._r = lo;  MRP._c = lo;  MRP._d = 0

        # Path.
        MRP._move = 1;  MRP._M[lo][lo] = MRP._move
```

Exhaustive Bounded Testing: Iterating through mazes.

```
class RunMaze:
    ...

    @classmethod
    def exhaustive_test(cls) -> None:
        """Generate/solve all mazes of sizes up through 3, and validate paths found."""
        for N in range(1, 4):
            for w in range(0, 2 ** (2 * N *(N - 1))):
                MRP.generate_input(N, w)
                RunMaze._solve()
                assert MRP.is_solution(), "internal program error"
                if (w > 0) and (w % 100000 == 0): print(w) # Heartbeat.
            print("passed for size", N)
        print("passed")

    ...
```

Random Testing: Generation and testing with random input data is called “fuzz testing”.

```
class RunMaze:
    ...

    @classmethod
    def fuzz_test(cls, f: int, n: int) -> None:
        """Fuzz f mazes of size n."""
        for k in range(0, f): RunMaze.random_test(n)

    @classmethod
    def random_test(cls, n: int) -> None:
        """Create a random maze of size n."""
        # Let w be random walls for the n-by-n maze.
        w = random.randint(0, 2 ** (2 * n *(n - 1)))

        # Generate maze of size n with walls w, solve, and validate (or abort).
        print("Size:", n, "Walls:", w) # (Comment out for serious test.)
        MRP.generate_input(n, w)       # Create n-by-n maze with walls w.
        RunMaze._solve()               # Attempt a solution.
        assert MRP.is_solution()       # Validate solution (or abort).
        MRP.print_maze()               # (Comment out for serious test.)

    ...
```

Random Testing: Sampling of output (most are unreachable).

```
RunMaze.Fuzz(3, 4)

Size: 4 Walls: 7345699
#  #  #  #  #  #  #  #  #
#  1  #     #           #
#              #        #
#  2     3        #     #
#  #           #        #
#        4     5        #
#                    #  #
#     #     #  6     7  #
#  #  #  #  #  #  #  #  #
Size: 4 Walls: 5141212
#  #  #  #  #  #  #  #  #
#  1     2     3  #     #
#  #     #           #  #
#     #  5     4  #     #
#              #     #  #
#     #  6     7     8  #
#  #     #     #        #
#           #        9  #
#  #  #  #  #  #  #  #  #
Size: 4 Walls: 6884164
#  #  #  #  #  #  #  #  #
#  1              #     #
#                    #  #
#           #     #     #
#        #              #
#              #        #
#              #        #
#     #     #           #
#  #  #  #  #  #  #  #  #
```

Solution validated.

Solution validated.

Claim of unreachability not validated.