Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Running a Maze

Introduction

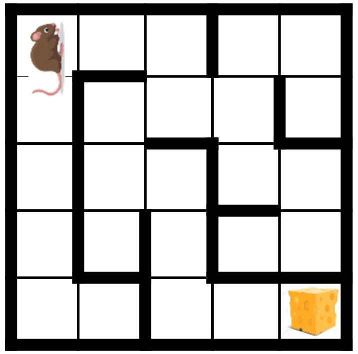
We present a systematic top-down development of an entire program to Run a Maze. We start from the beginning, but reference previous discussions from Chapters 1 and 4.

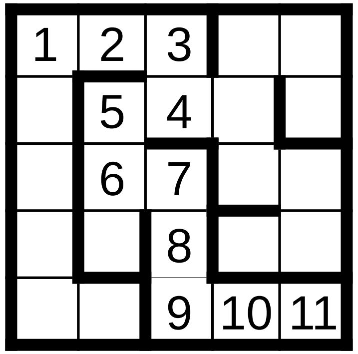
The main themes presented are:

- 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.





Top-level Code Structure

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   } /* RunMaze */
```

☞ **Program top-down, outside-in.**

Top-level Code Structure

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   /* Run maze. */
   public static void main() {
      } /* main */
   } /* RunMaze */
```

☞ **Program top-down, outside-in.**

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   /* Run maze. */
   public static void main() {
      /* Input. */
      /* Compute. */
      /* Output. */
      } /* main */
   } /* RunMaze */
```

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

Top-level Code Structure

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   /* Run a maze given as input, if possible. */
   public static void main() {
      /* 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. */
      } /* main */
   } /* RunMaze */
```

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

Top-level Code Structure

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   /* Run a maze given as input, if possible. */
   public static void main() {
      /* Input a maze of arbitrary size, or output “malformed input”
         and stop if the input is improper. Input format: TBD.*/
         Input();
      /* Compute a direct path through the maze, if one exists. */
         Solve();
      /* Output the direct path found, or “unreachable” if there is
         none. Output format: TBD. */
         Display();
      } /* main */
   } /* RunMaze */
```

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

Establish a framework:

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Input a maze of arbitrary size, or output “malformed input”
      and stop if the input is improper. Input format: TBD. */
   private static void Input() { } /* Input */
   /* Compute a direct path through the maze, if one exists. */
   private static void Solve() { } /* Solve */
   /* Output the direct path found, or “unreachable” if there is none.
      Output format: TBD. */
   private static void Output() { } /* Output */
   ...
 } /* RunMaze */
```

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

Top-level Code Structure

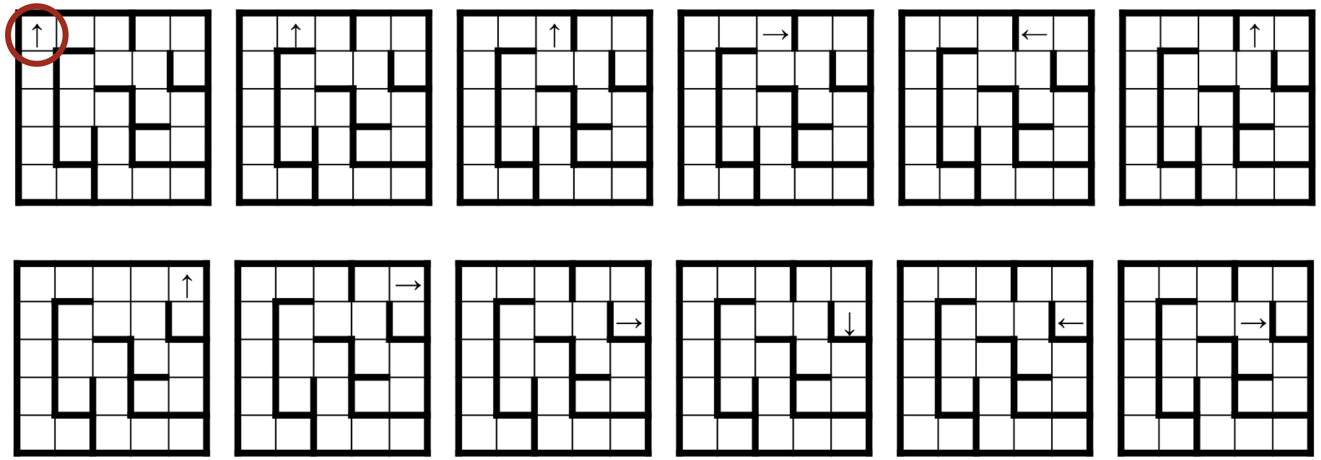
Private and internal to RunMaze. No other class needs to know about them.

Establish a framework:

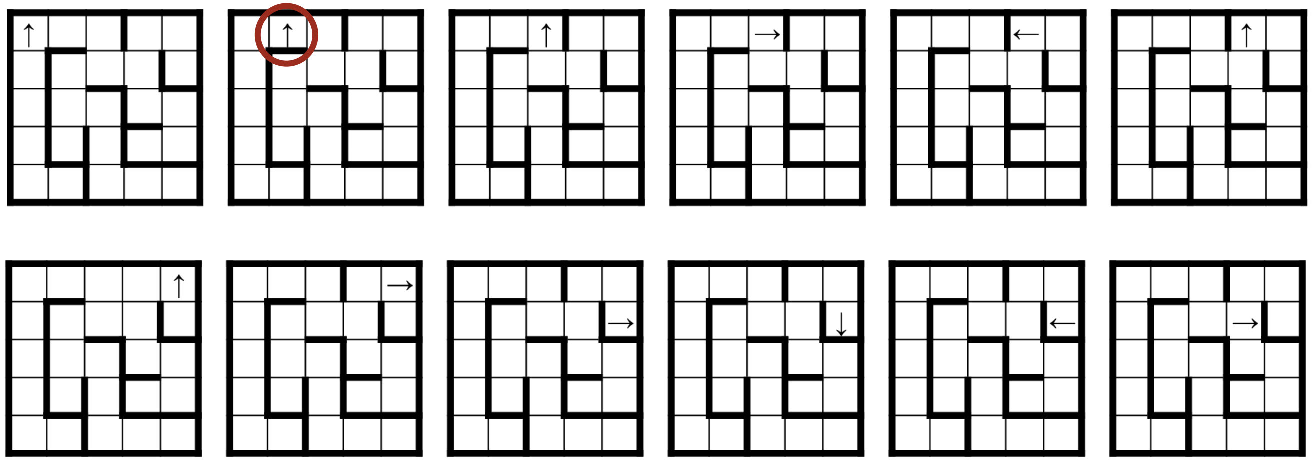
```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Input a maze of arbitrary size, or output “malformed input”
      and stop if the input is improper. Input format: TBD. */
   private static void Input() { } /* Input */
   /* Compute a direct path through the maze, if one exists. */
   private static void Solve() { } /* Solve */
   /* Output the direct path found, or “unreachable” if there is none.
      Output format: TBD. */
   private static void Output() { } /* Output */
   ...
 } /* RunMaze */
```

☞ **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?**

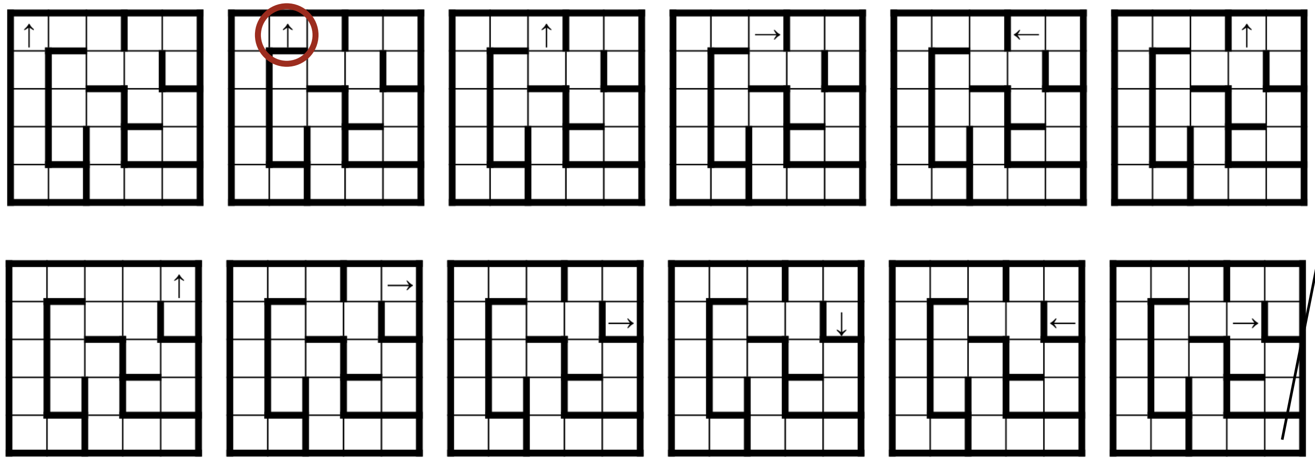


Sidestep

☞ **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.

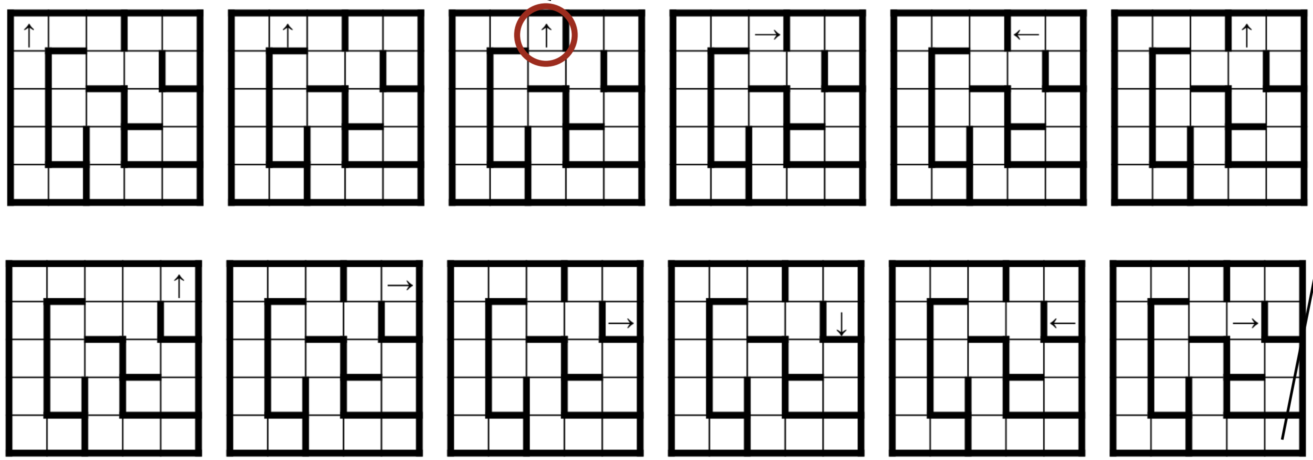


Sidestep

☞ **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.



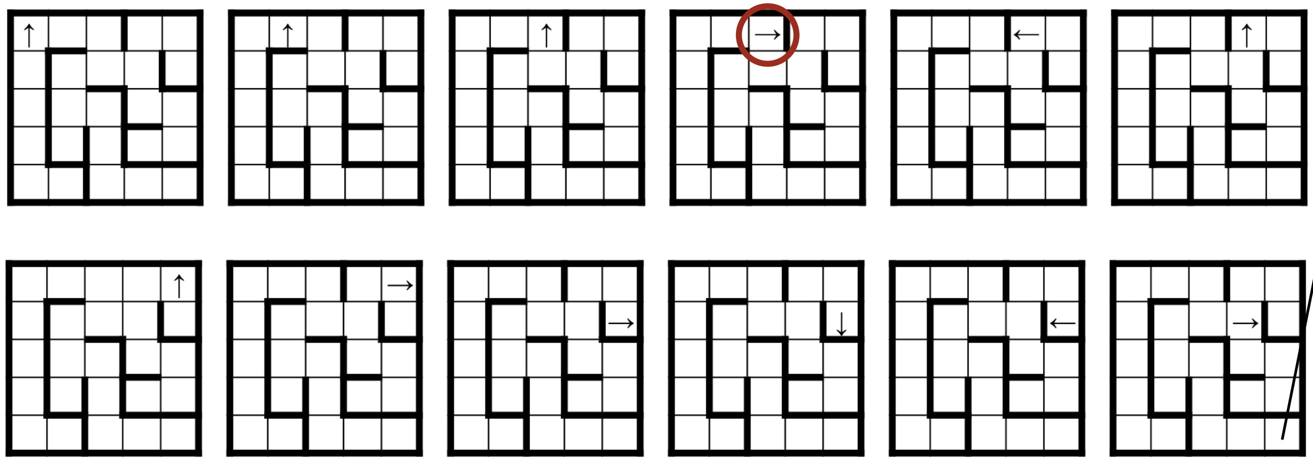
Sidestep

Algorithm

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.

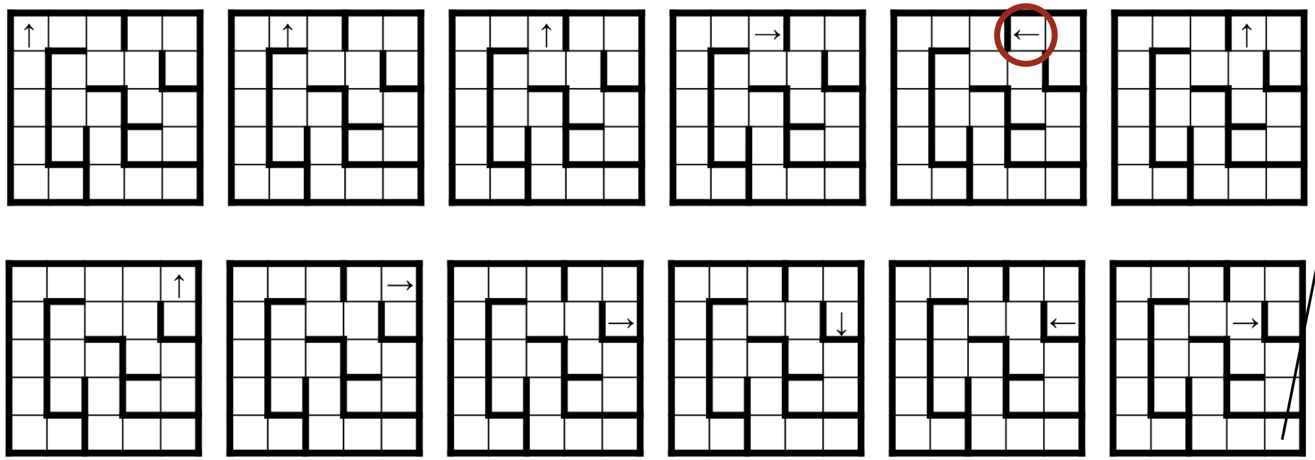


Turn convex corner

Algorithm

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

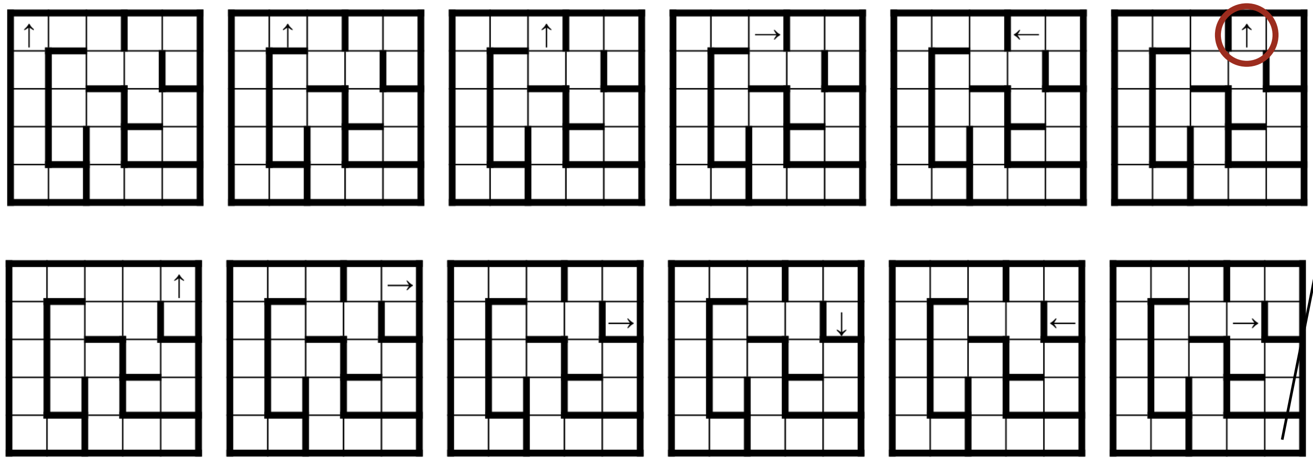
VARIANT:
Get closer to goal.



Pirouette to other side

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

VARIANT:
Get closer to goal.

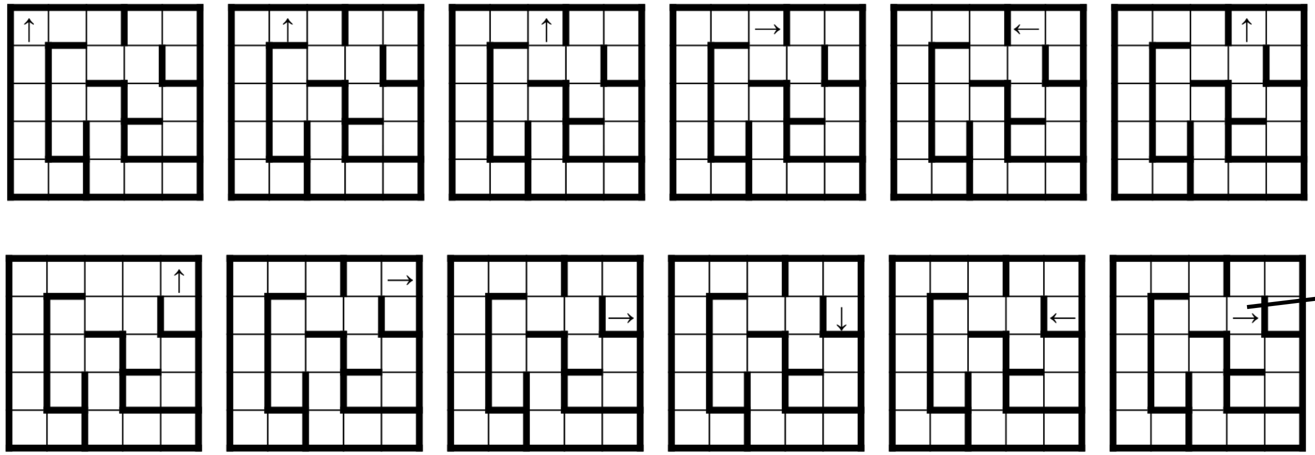


Turn convex corner

Algorithm

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

VARIANT:
Get closer to goal.

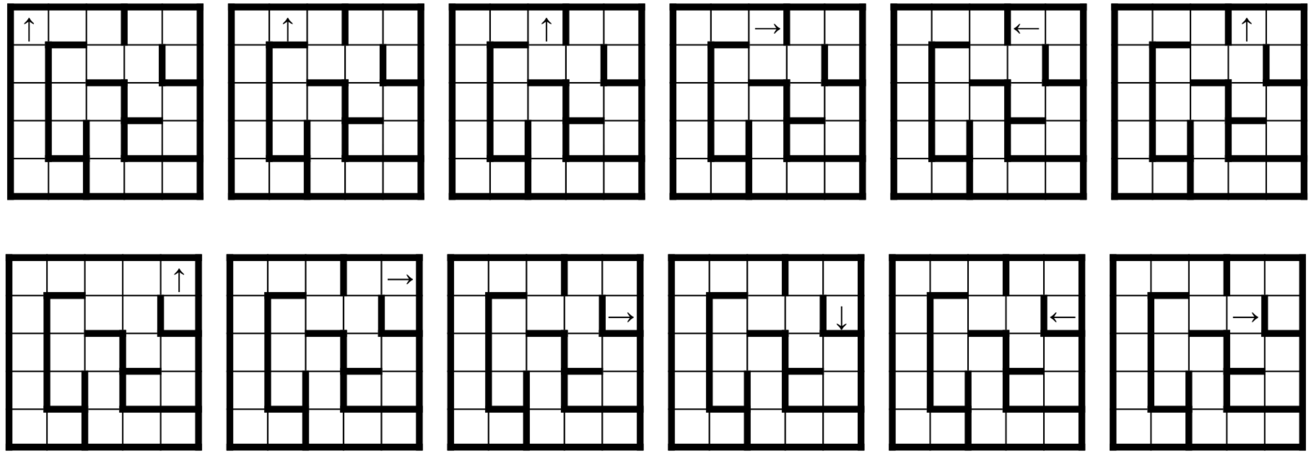


Actions:
- Sidestep
- Pirouette
- Turn convex corner
- (Turn concave corner)

Algorithm

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)

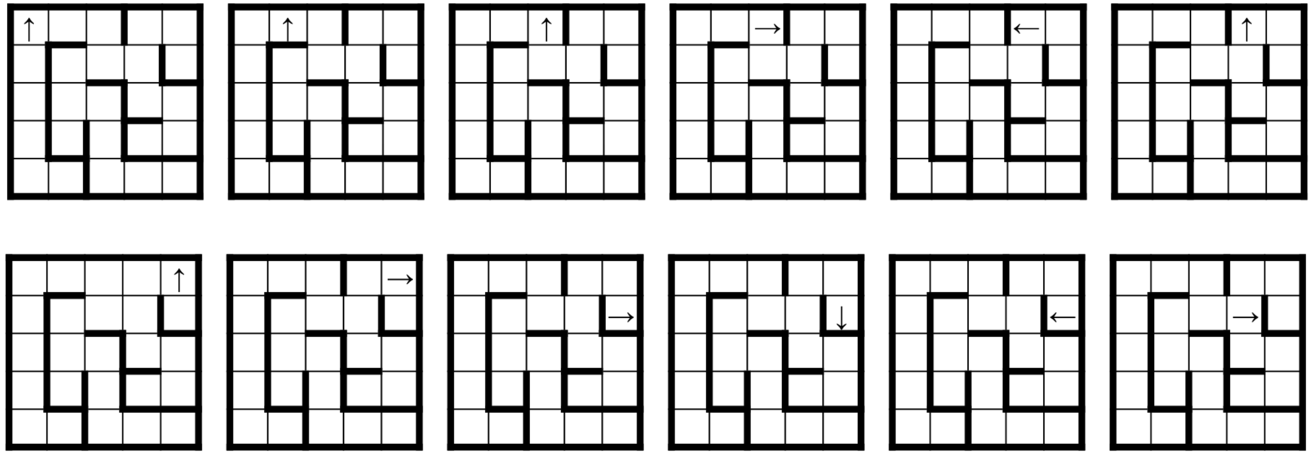
Query:

- What action to perform?

Algorithm

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

VARIANT:
Get closer to goal.

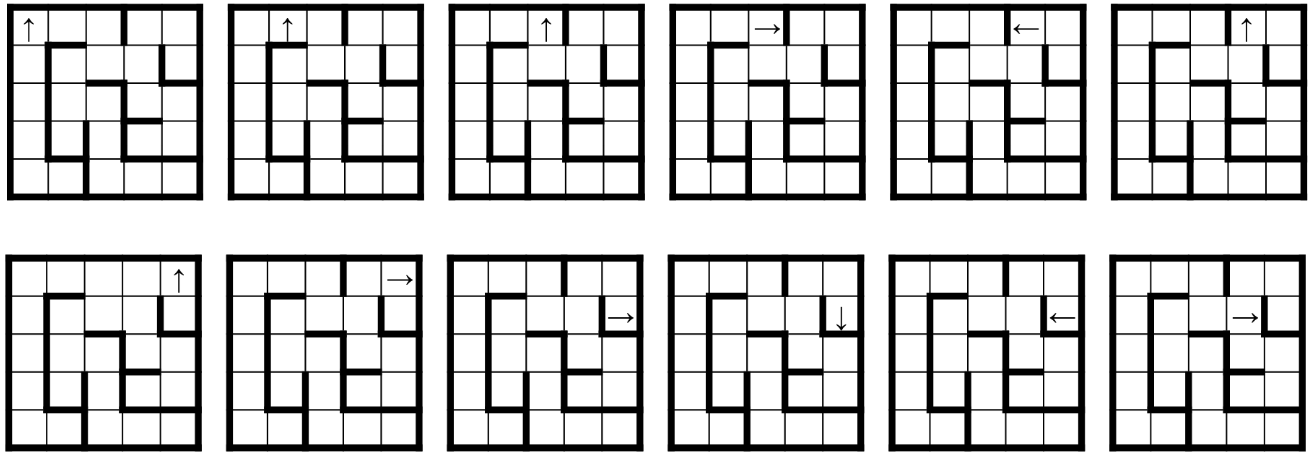


Actions:
- Sidestep
- Pirouette
- Turn convex corner
- (Turn concave corner)

Query:
- What action to perform?

Unit of progress:
- 1 wall-segment-surface

Algorithm



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
else if (___________) Pirouette
else if (___________) 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

Algorithm

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

Query:

- Facing a wall?

Unit of progress:

- 1 wall-segment-surface-or-door

Finer-grained actions.

Local query.

Simpler to implement.

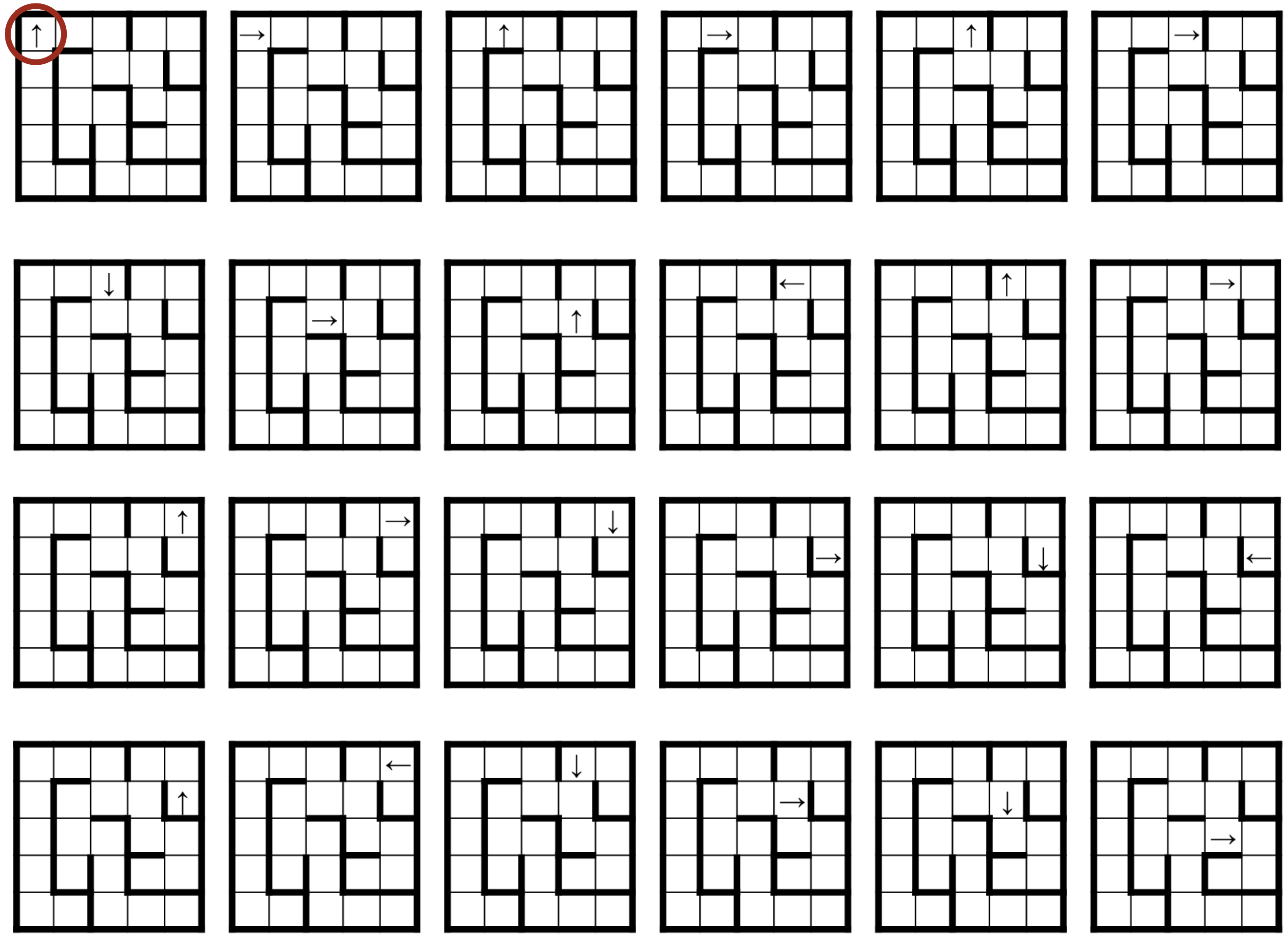
Alternative Formulation: Pseudo-code, from Chapter 4.

```
/* Start in upper-left cell, facing up. */
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```

Algorithm

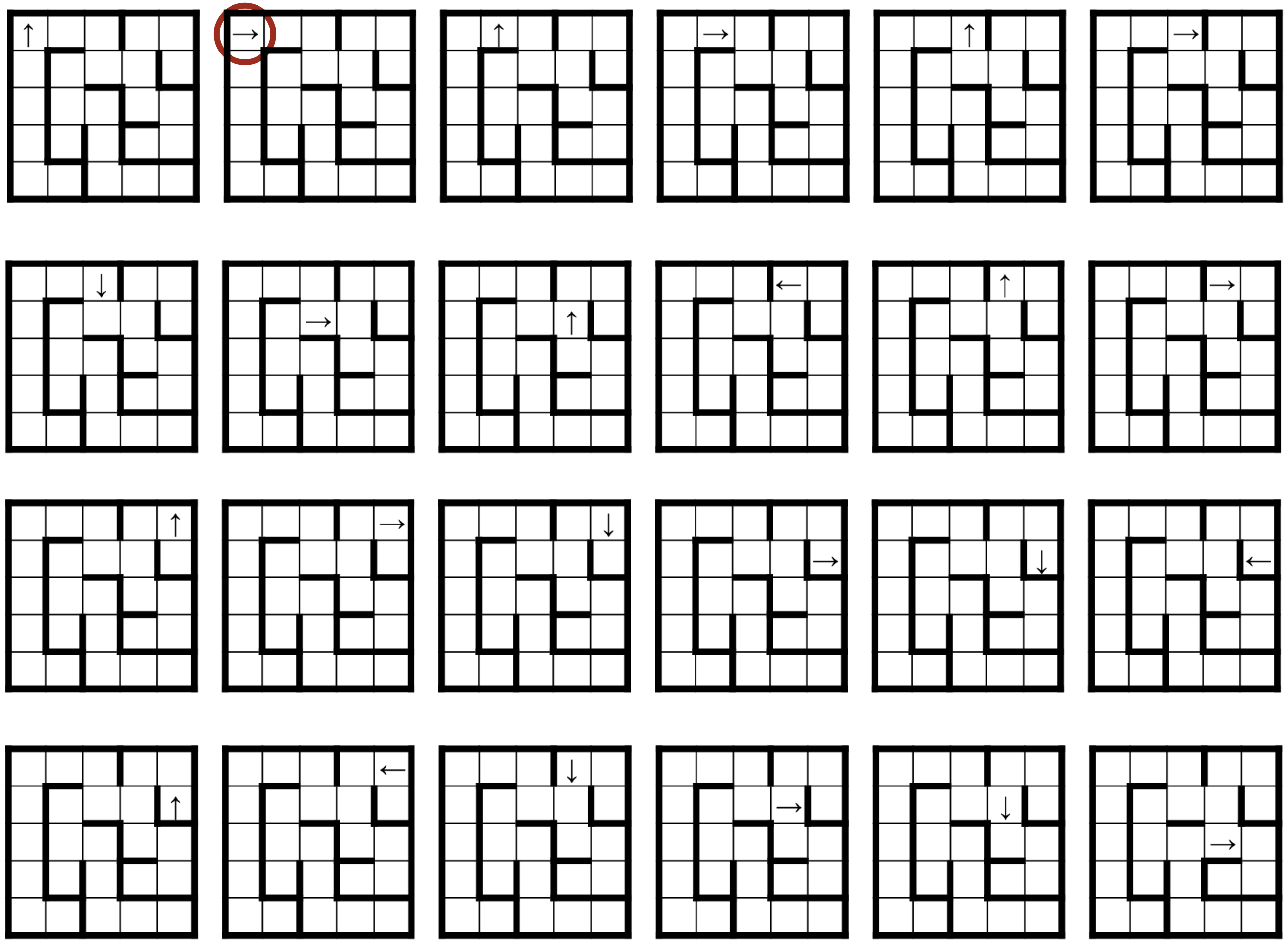
INVARIANT:

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



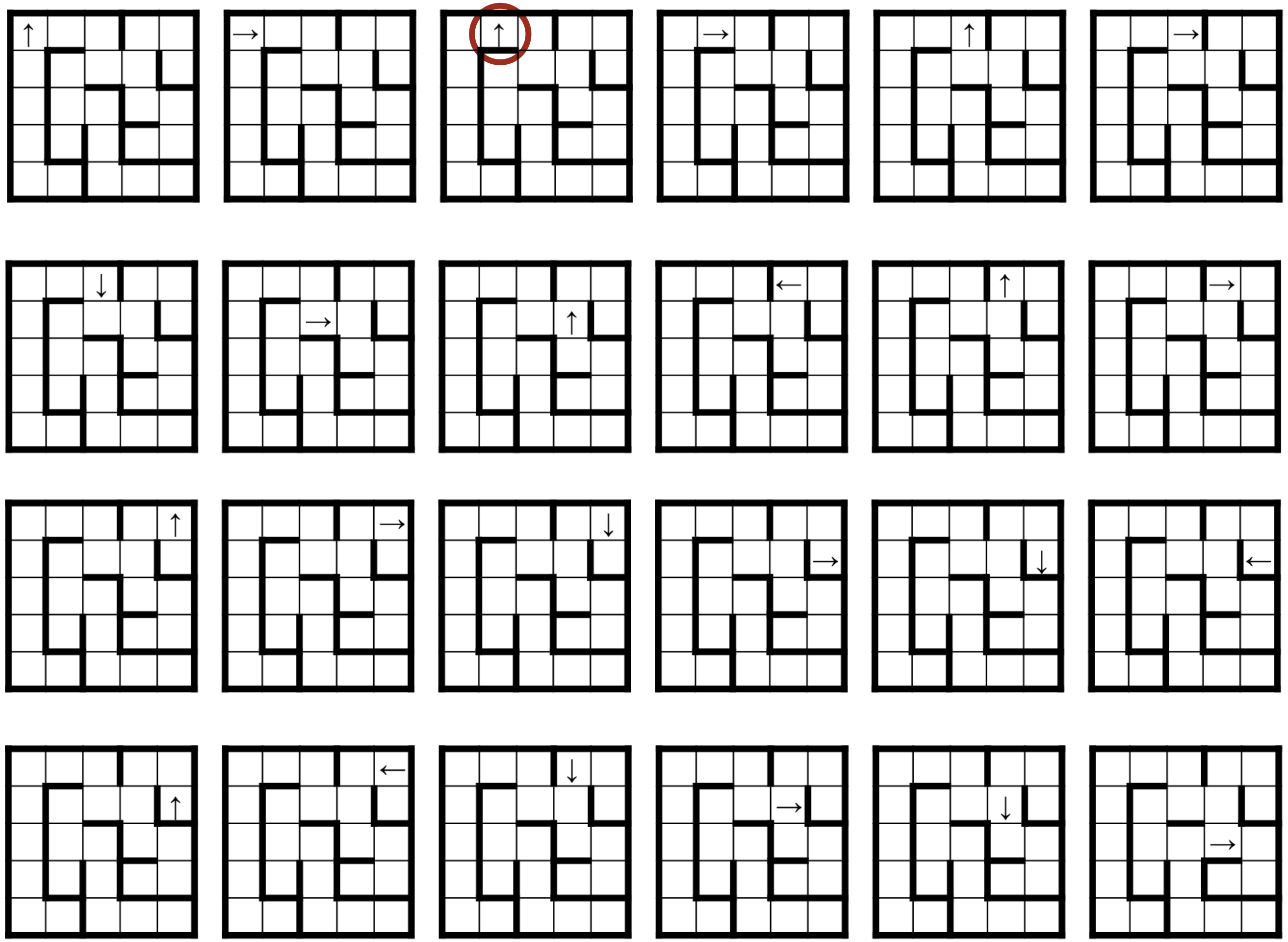
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



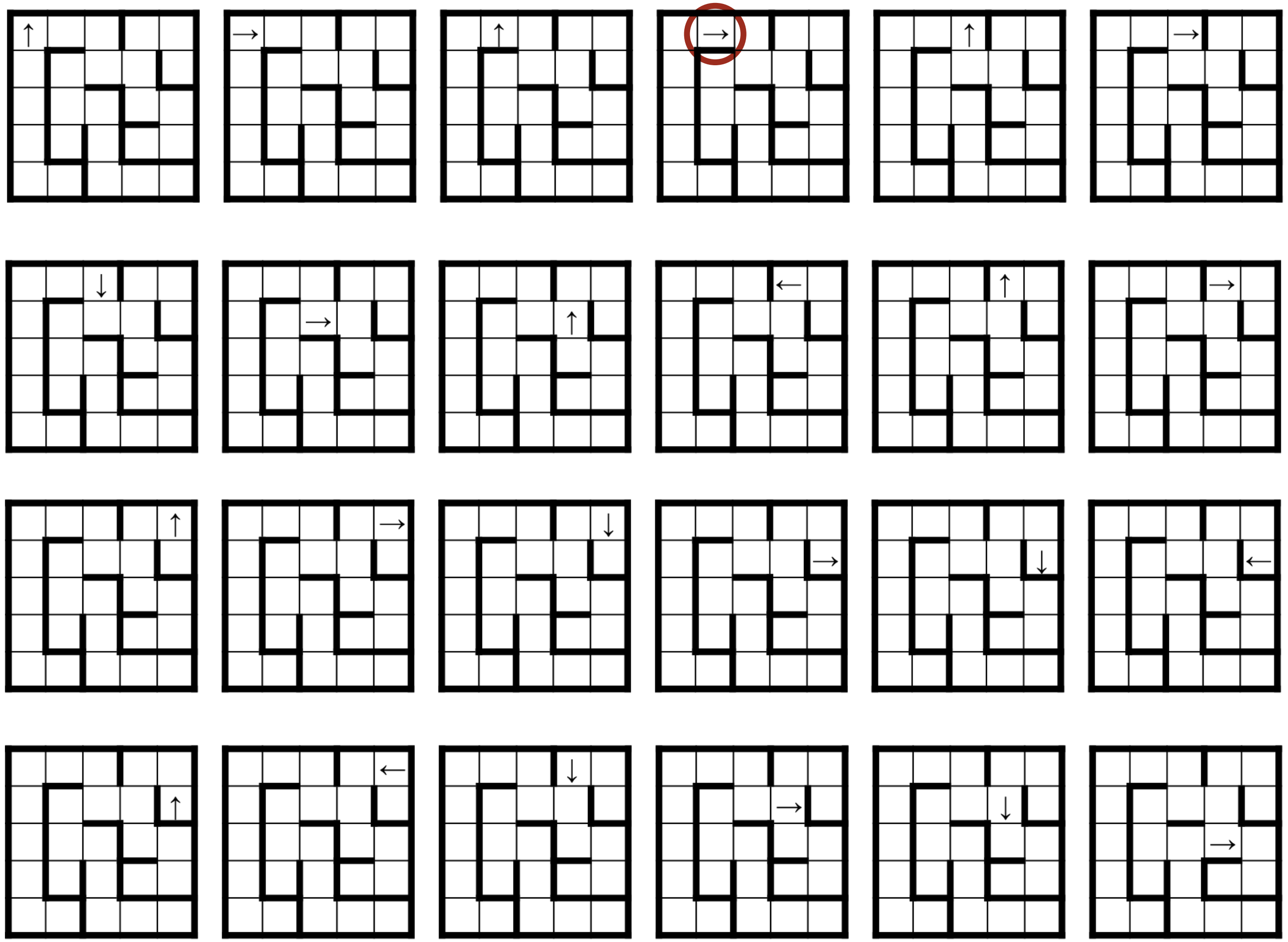
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



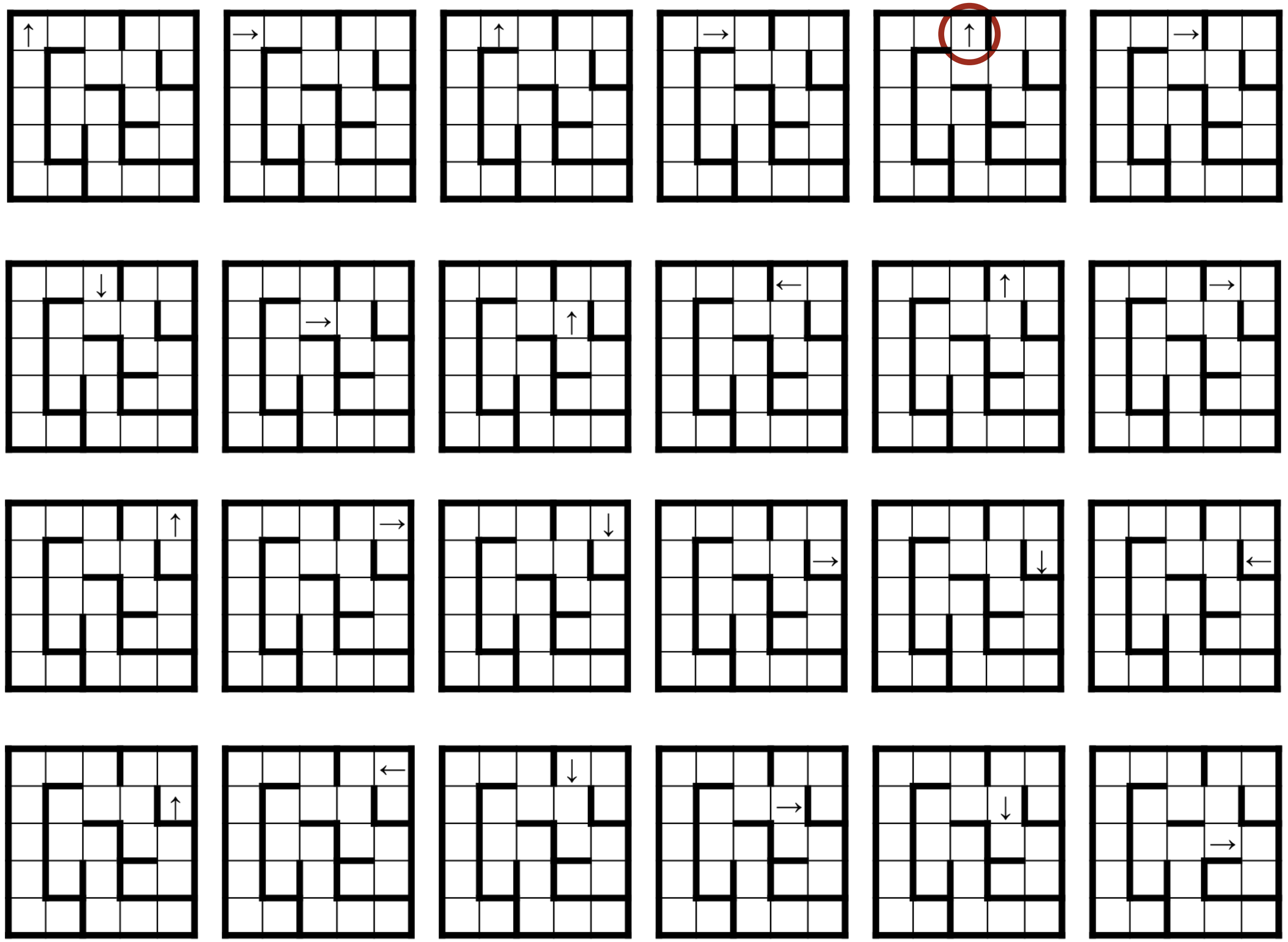
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



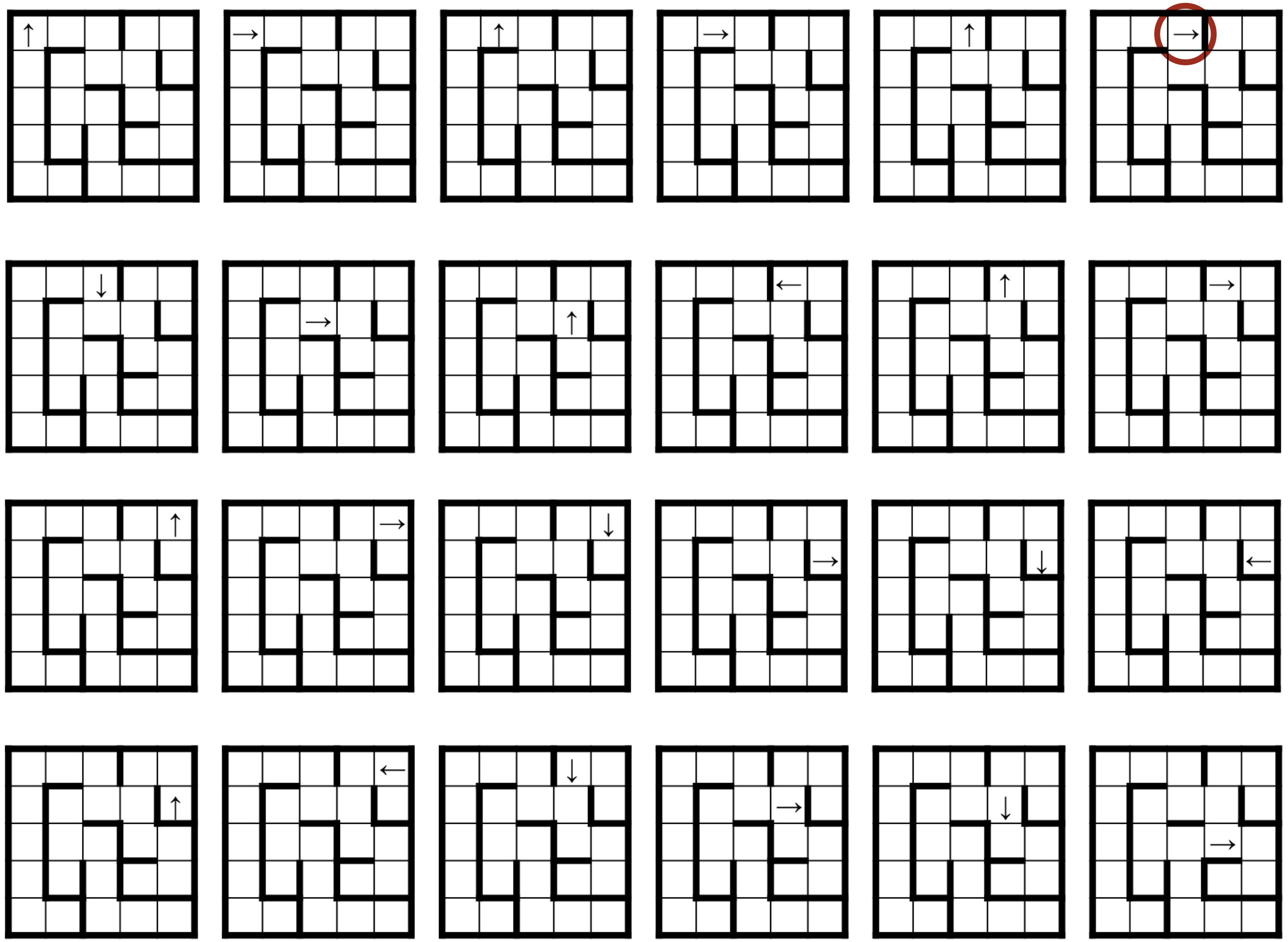
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



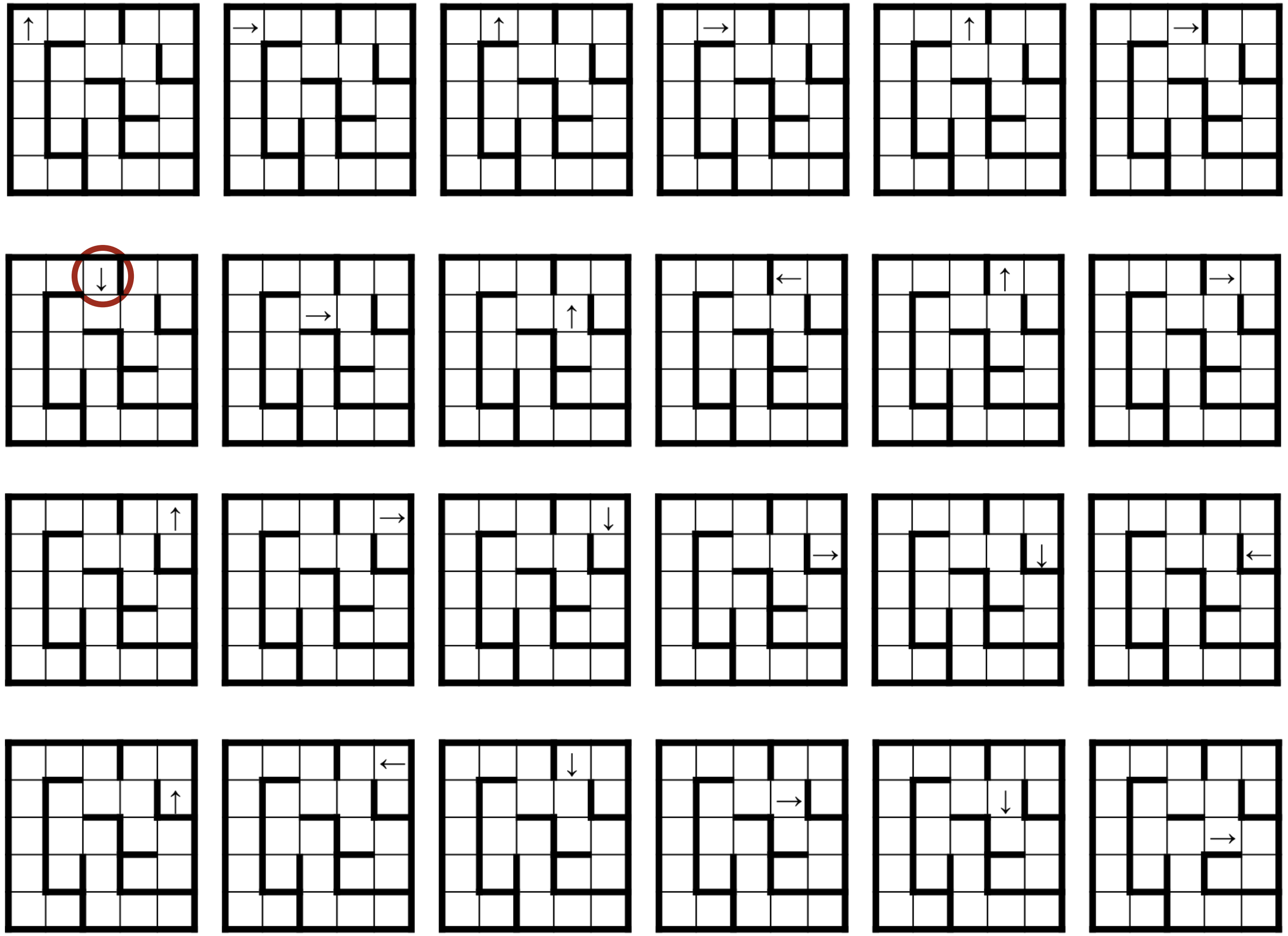
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



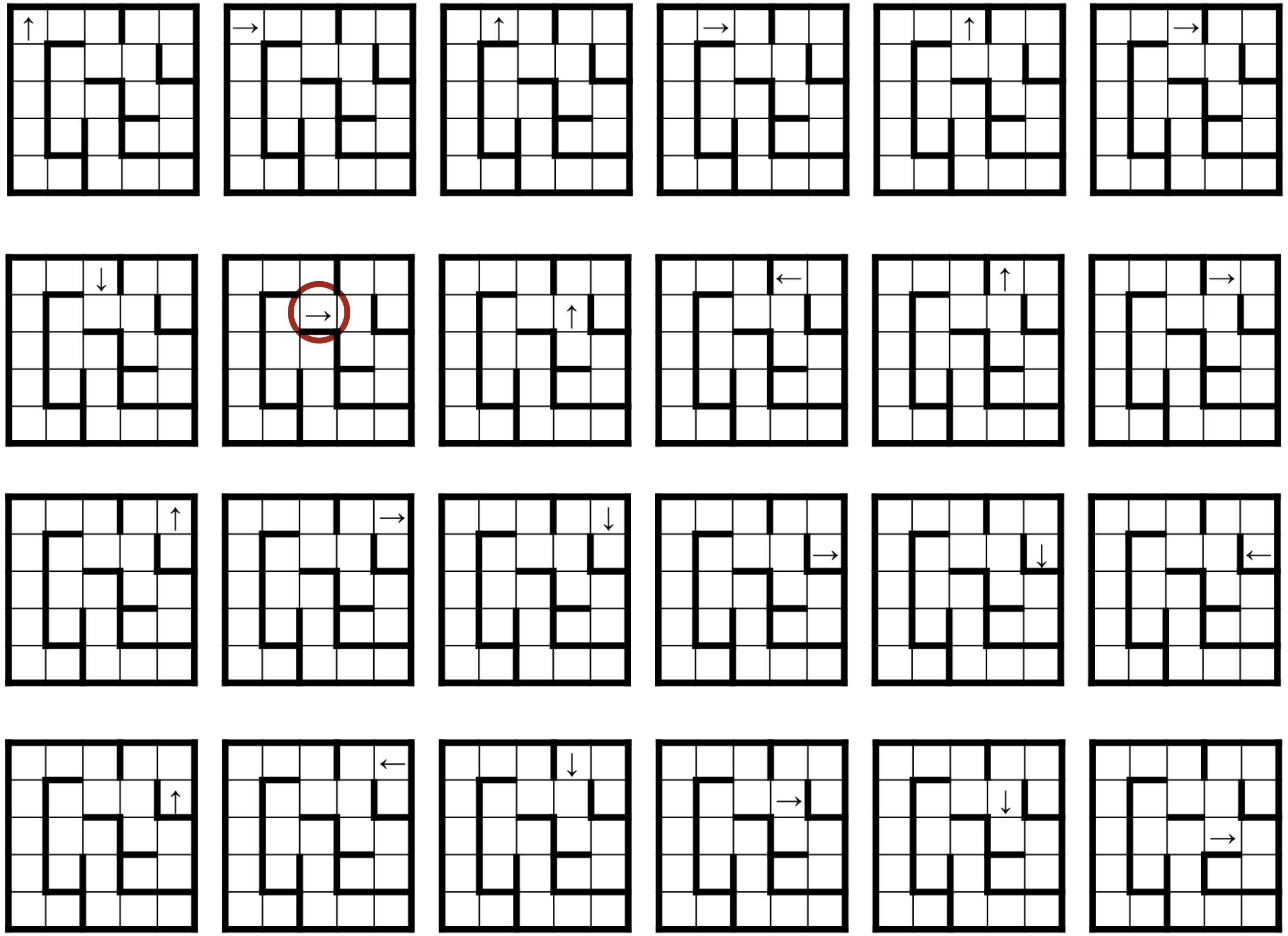
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



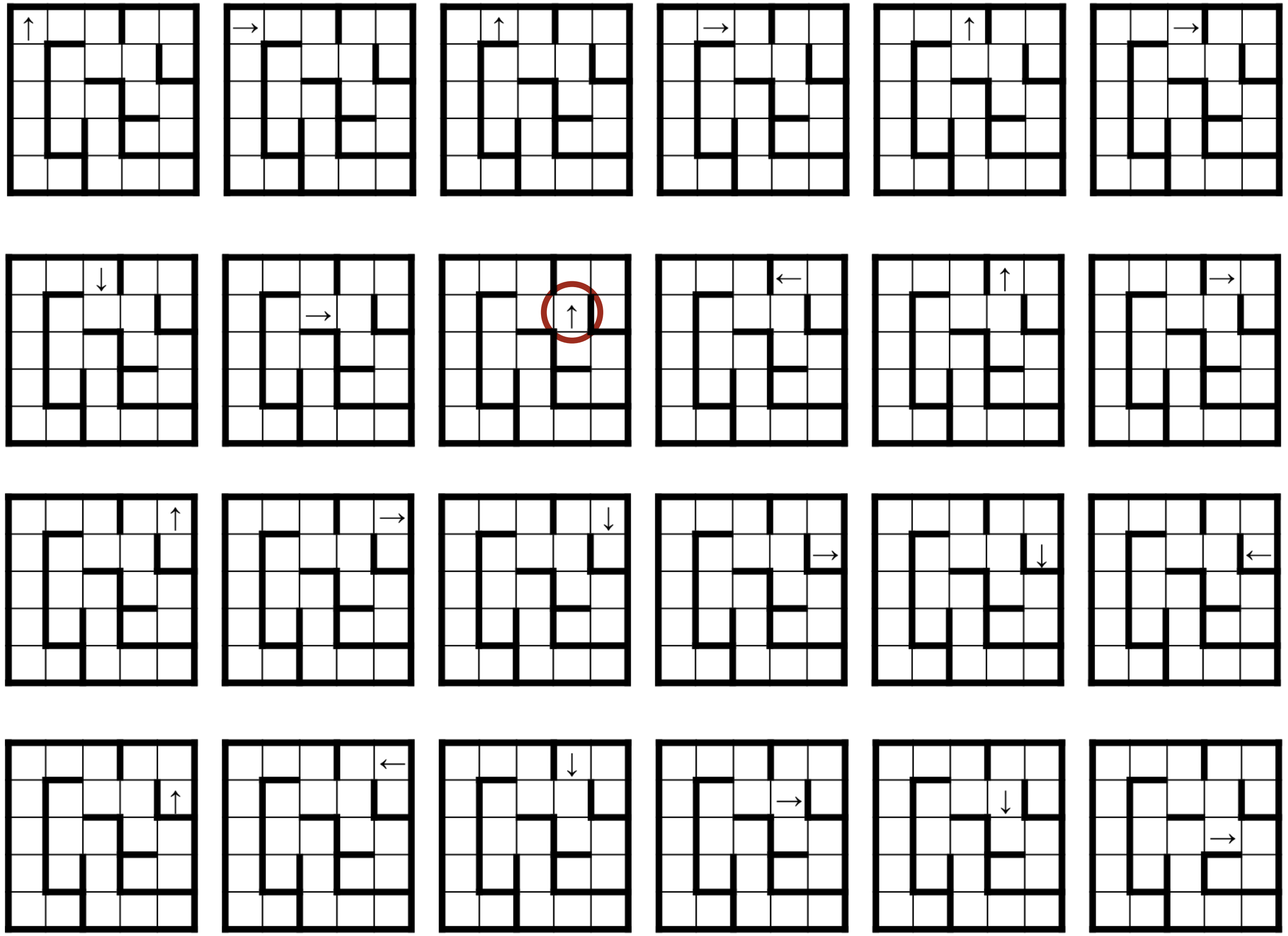
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



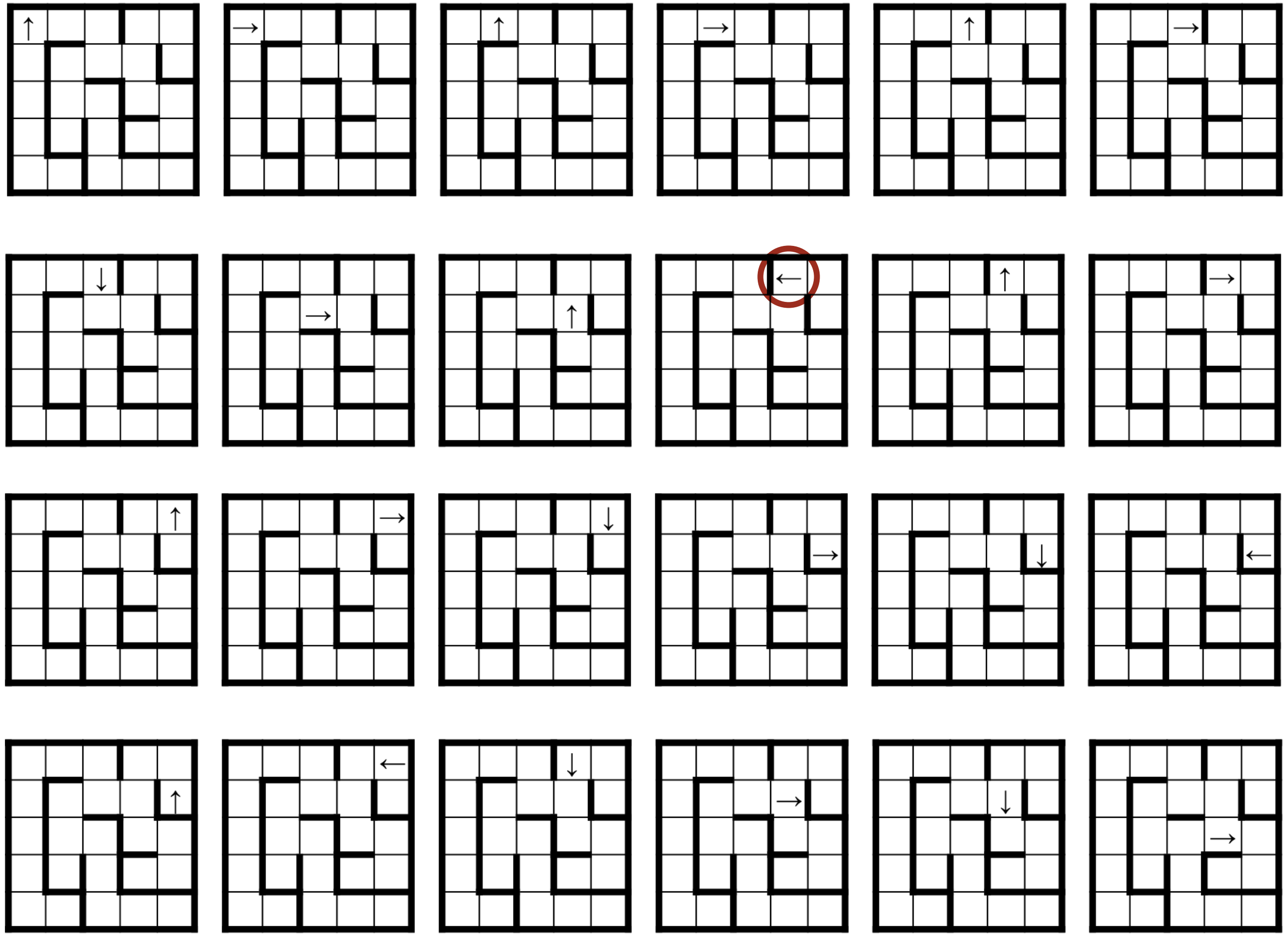
Algorithm

```
while ( /* !in-lower-right && !in-upper-left-about-to-cycle */ )
   if ( /* facing-wall */ )
      /* Turn 90° clockwise. */
   else {
      /* Step forward. */
      /* Turn 90° counterclockwise. */
      }
```



Algorithm

```
while ( /* !in-lower-right && !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.

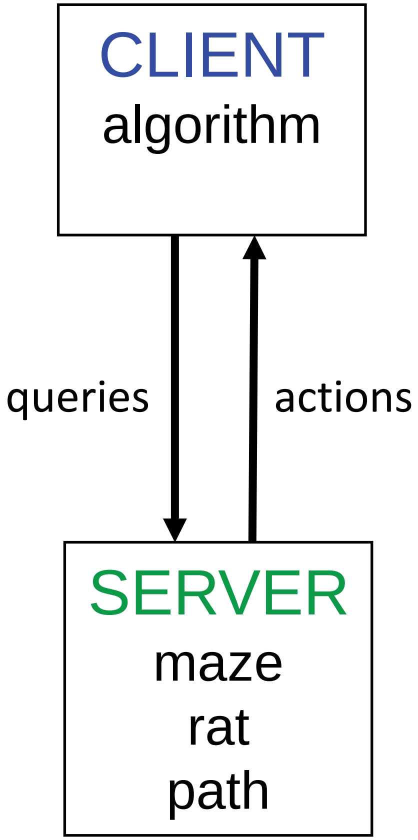
```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Input a maze of arbitrary size, or output “malformed input”
      and stop if the input is improper. Input format: TBD. */
   private static void Input() {
      ⟨Obtain maze from input.⟩
      ⟨Start in upper-left cell, facing up.⟩
      } /* Input */
   ...
   } /* RunMaze */
```

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.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Compute a ~~direct~~ path through the maze, if one exists. */
   private static void Solve() {
      while ( !isAtCheese() && !isAboutToRepeat() )
         if ( isFacingWall() ) TurnClockwise();
         else {
            StepForward();
            TurnCounterClockwise();
            }
      } /* Solve */
   ...
   } /* RunMaze */
```

Modular program structure: Separation of concerns.




```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Run a maze given as input, if possible. */
   public static void main() {
      ...
      } /* main *.
} /* RunMaze */

/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
    } /* MRP */
```

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

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

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Compute a direct path through the maze, if one exists. */
   private static void Solve() {
      while ( !MRP.isAtCheese() && !MRP.isAboutToRepeat() )
         if ( MRP.isFacingWall() ) MRP.TurnClockwise();
         else {
            MRP.StepForward();
            MRP.TurnCounterClockwise();
            }
      } /* Solve */
   ...
   } /* RunMaze */
```

Procedure stubs for the services.

Operations:

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   public static void TurnClockwise() { }
   public static void TurnCounterClockwise() { }
   public static void StepForward() { }
   public static boolean isFacingWall() { return ____; }
   public static boolean isAtCheese() { return ____; }
   public static boolean isAboutToRepeat() { return ____; }
   } /* MRP */
```

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

Stubs provide *signatures*, i.e., types for return types, types for parameters (none), and visibility.

Operations:

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   public static void TurnClockwise() { }
   public static void TurnCounterClockwise() { }
   public static void StepForward() { }
   public static boolean isFacingWall() { return ____; }
   public static boolean isAtCheese() { return ____; }
   public static boolean isAboutToRepeat() { return ____; }
   } /* MRP */
```

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

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

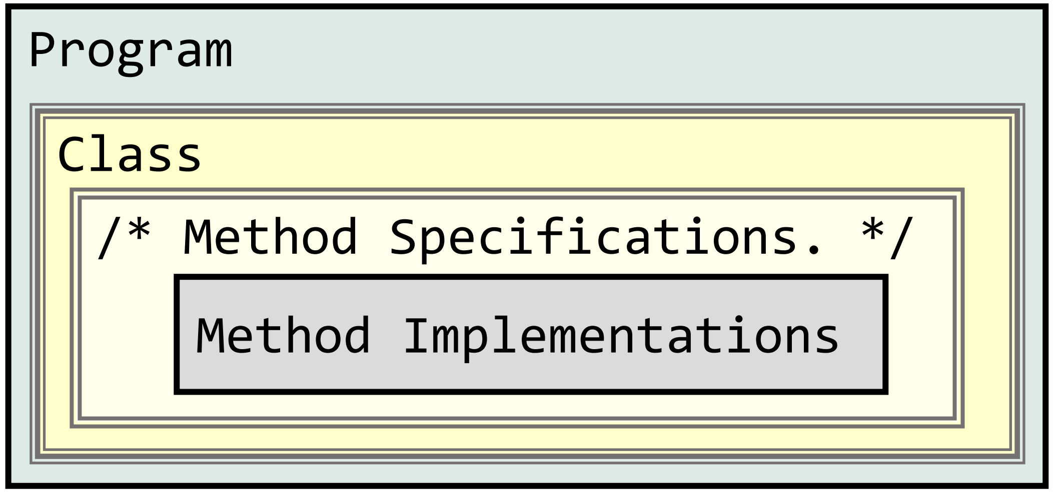
Operations:

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   public static void TurnClockwise() { }
   public static void TurnCounterClockwise() { }
   public static void StepForward() { }
   public static boolean isFacingWall() { return ____; }
   public static boolean isAtCheese() { return ____; }
   public static boolean isAboutToRepeat() { return ____; }
   } /* MRP */
```

☞ **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.

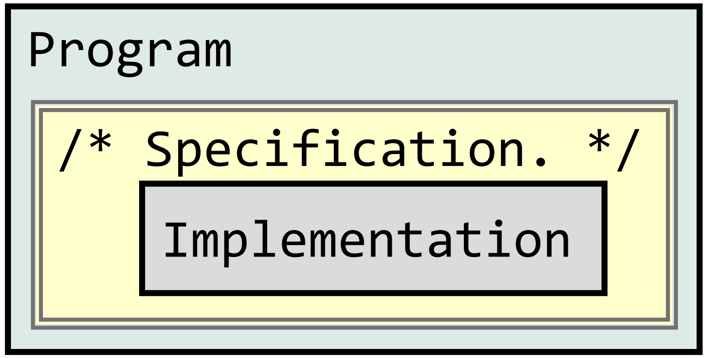


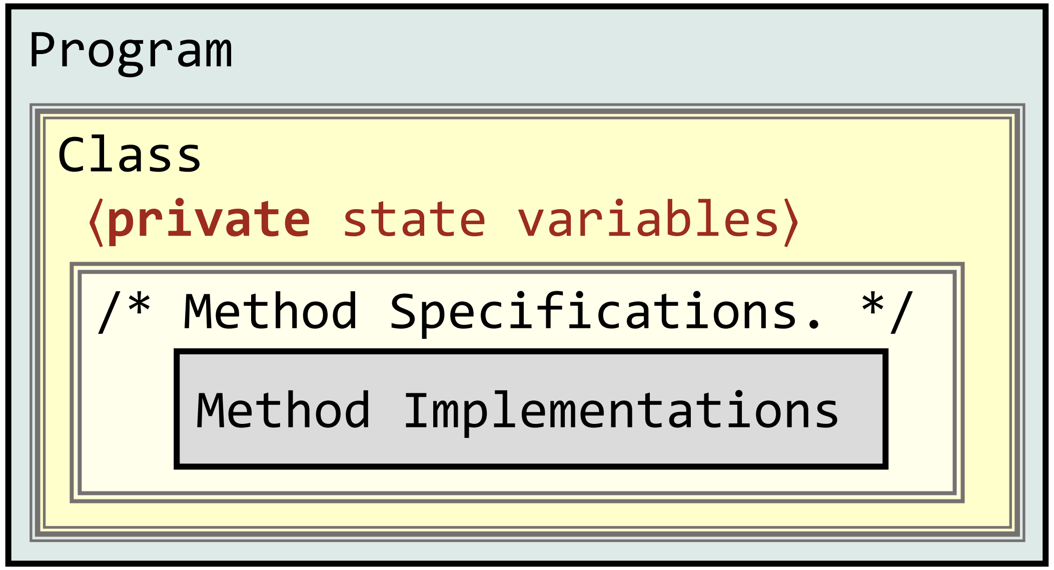
☞ **Practice information hiding.**

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.

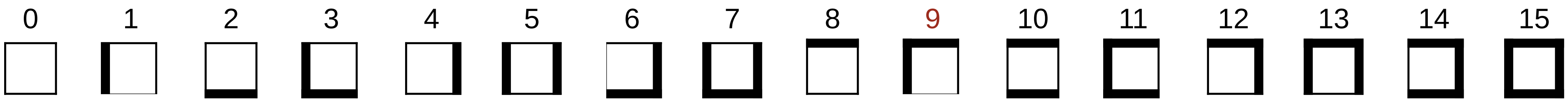
Clients of `MRP` will have no direct access to the state in `MRP`. Rather, they will only be able to interact with `MRP` via its operations, i.e., its interface. This is called an *abstract data type*, and generalizes our prior use of specifications for information hiding.

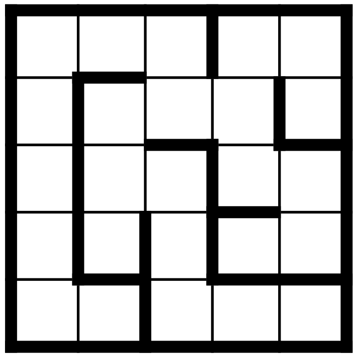


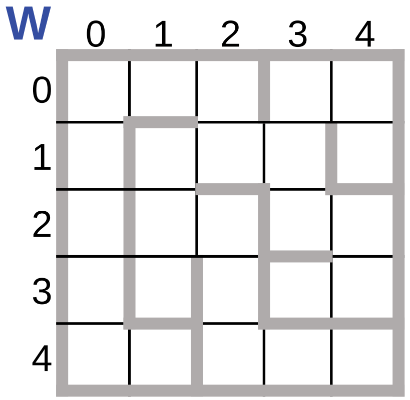


☞ **Practice information hiding.**

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

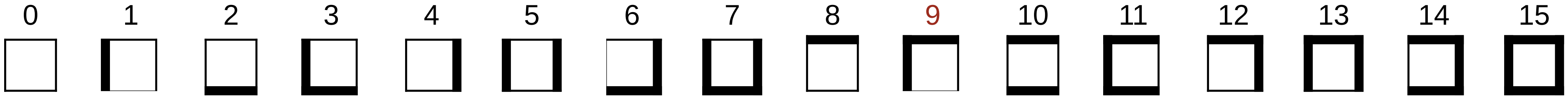


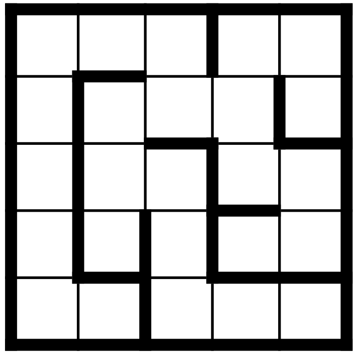


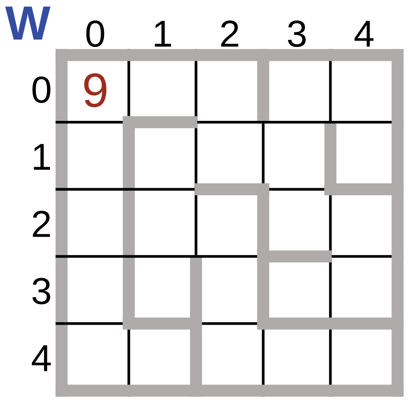


☞ **The touchstone of a data representation is its utility in performing the needed operations.**

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



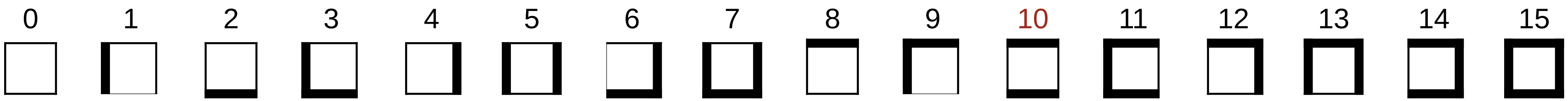


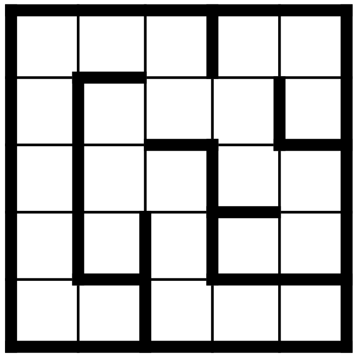


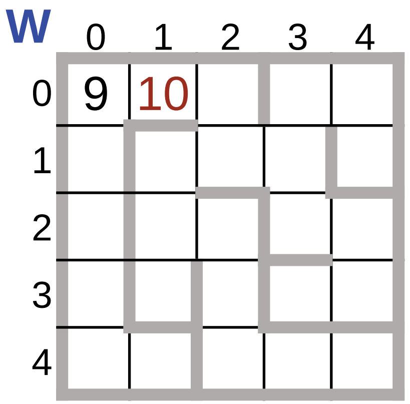
☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Data Representation

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

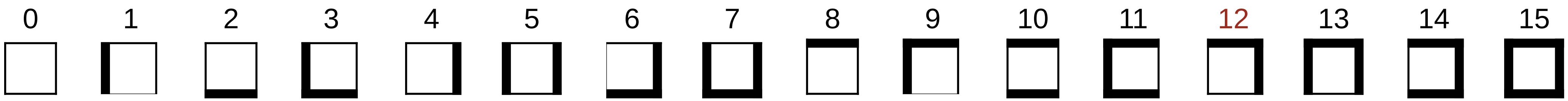


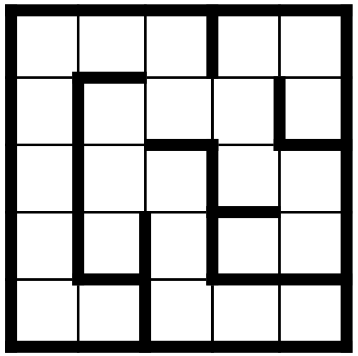


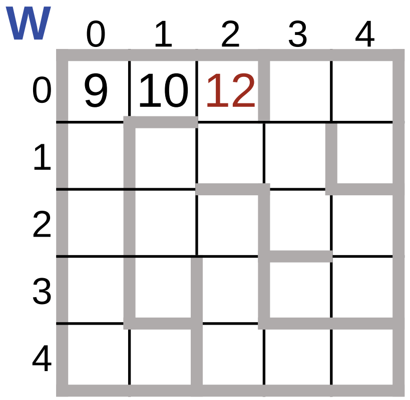


☞ **The touchstone of a data representation is its utility in performing the needed operations.**

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

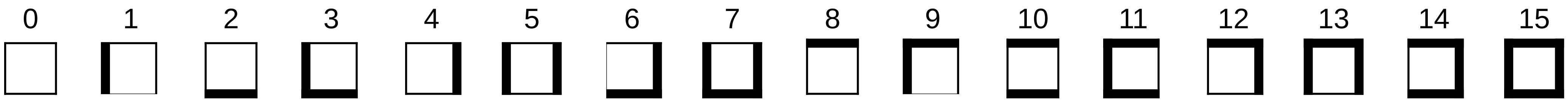


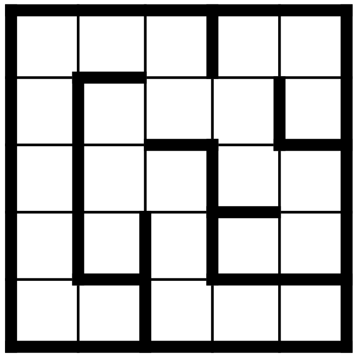




☞ **The touchstone of a data representation is its utility in performing the needed operations.**

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

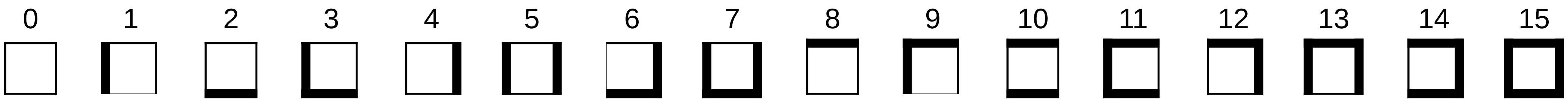


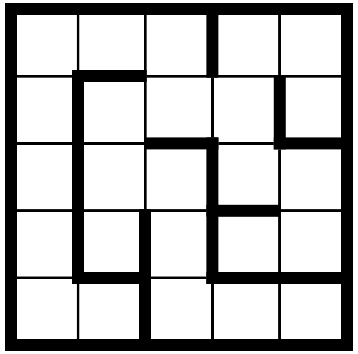


W	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

☞ **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

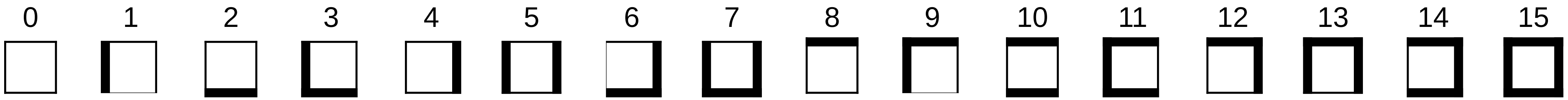
	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

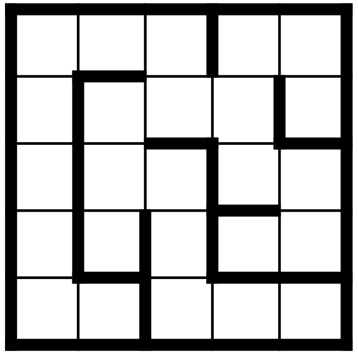
Anticipate

- 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:





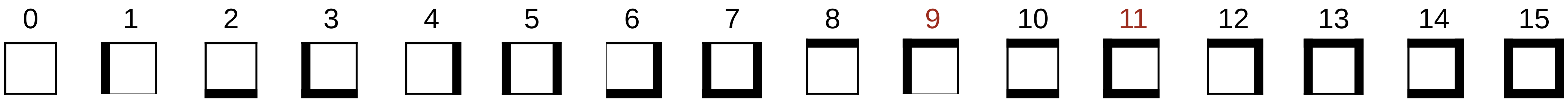
W	0	1	2	3	4
0	9	10	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

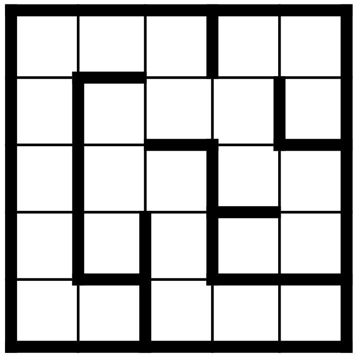
Positive

- 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

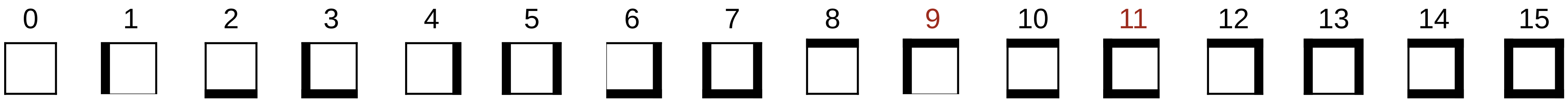
	0	1	2	3	4
0	9	11	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

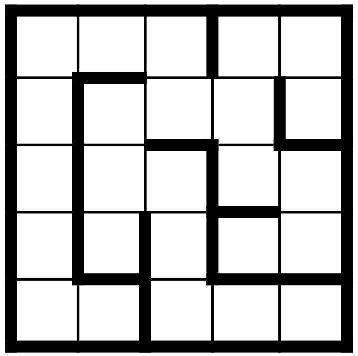
Negative

- 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:



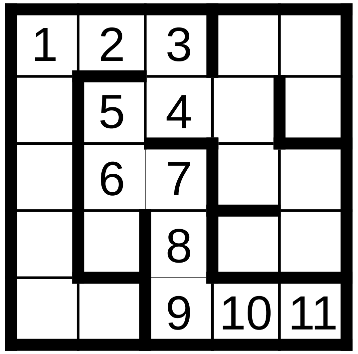


W	0	1	2	3	4
0	9	11	12	9	12
1	5	9	2	4	7
2	5	1	12	3	12
3	5	7	5	11	6
4	3	14	3	10	14

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`).



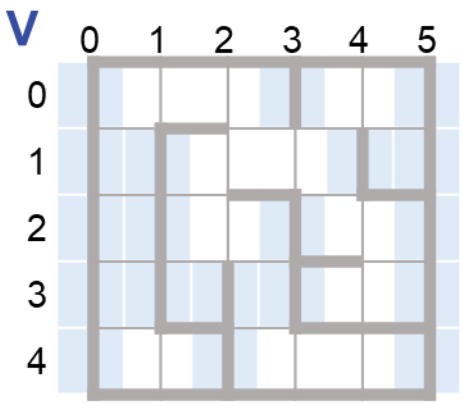
P	0	1	2	3	4
0	1	2	3	0	0
1	0	5	4	0	0
2	0	6	7	0	0
3	0	0	8	0	0
4	0	0	9	10	11

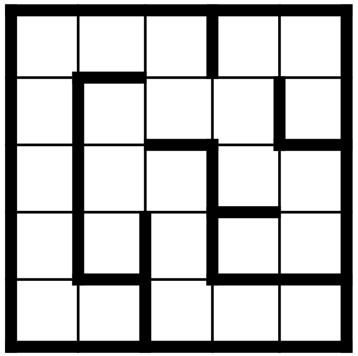
Positive

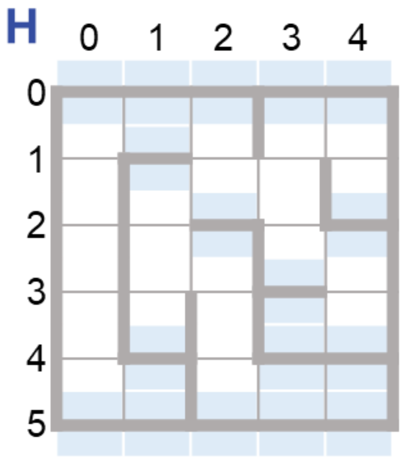
- Direct correspondence between physical maze and 2-D array **P**.

☞ **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.





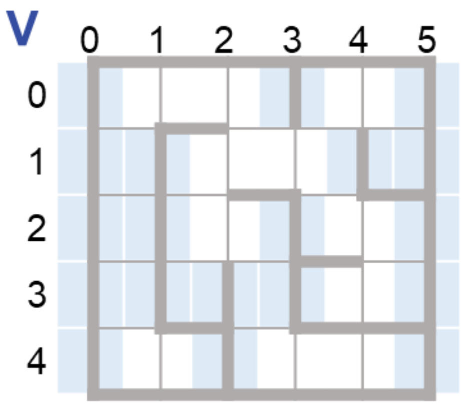


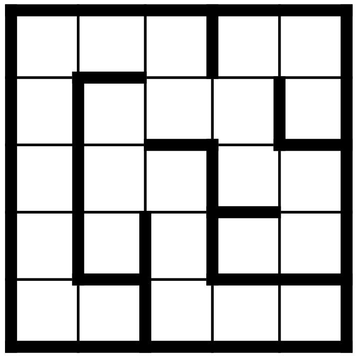
Eliminating Negatives of Representation 1

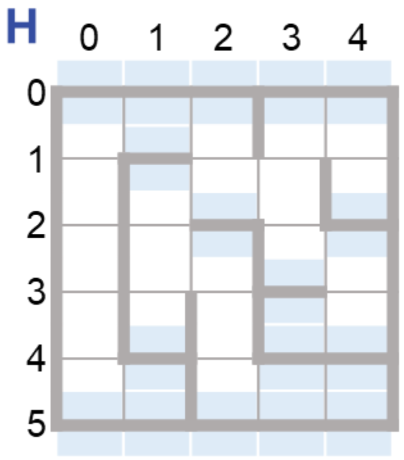
- Unique representation of each (possible) wall.
- Decoder and corresponding encoder are more straightforward.

☞ **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.





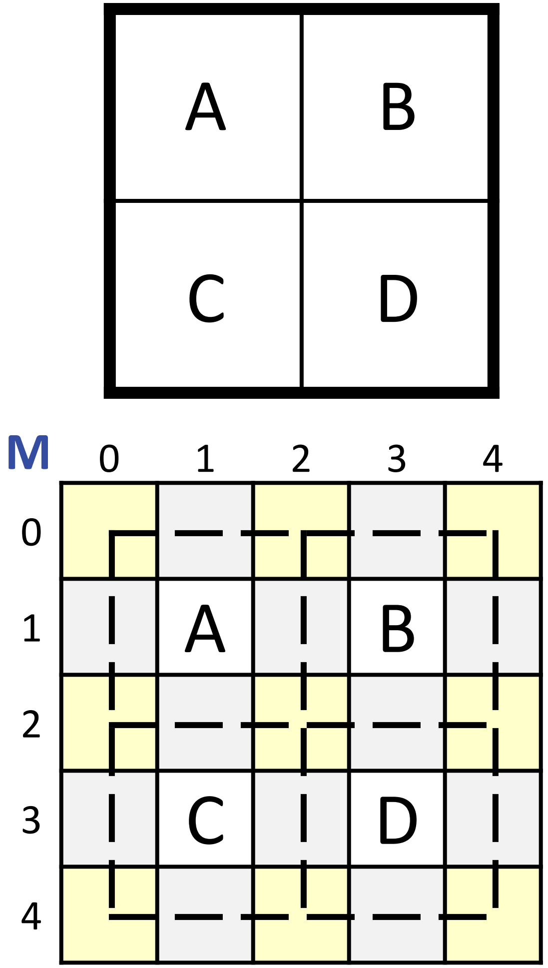


Negative of Representation 2

- 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.

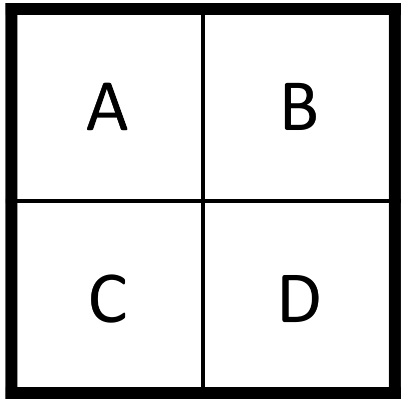


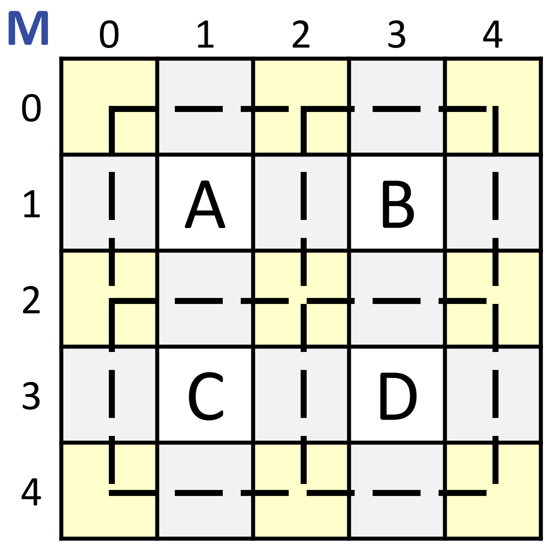
Positives

- 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.



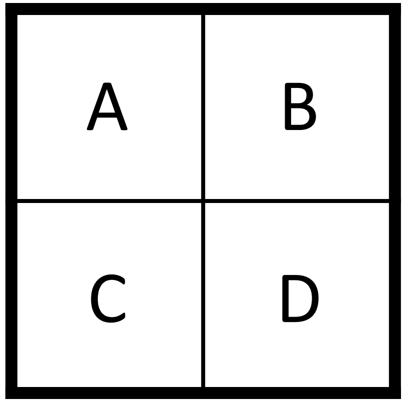


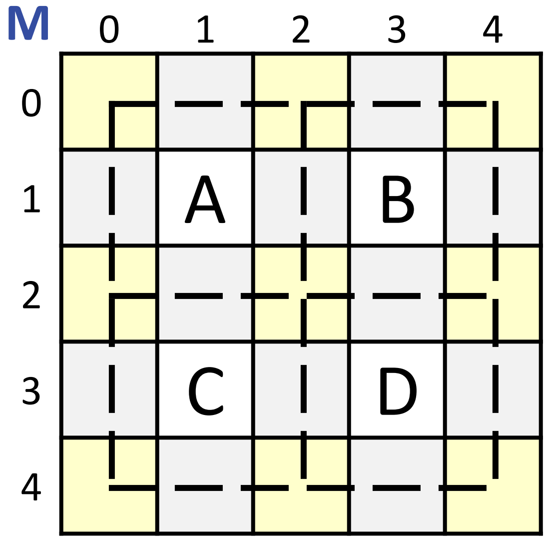
Negatives

- 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:

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   /* Maze. Cells of an N-by-N maze are represented by elements of
      array M[2*N+1][2*N+1]. 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 NoWall 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. */
      private static int N;      // Size of maze. */
      private static int M[][];  // Maze, walls, and path.
      private static final int Wall = -1;
      private static final int NoWall = 0;
      private static int lo, hi; // Left/top and right/bottom maze indices.
   ...
   } /* MRP */
```

☞ **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).**

Private and internal to MRP. No other class needs to know about them.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   /* Maze. Cells of an N-by-N maze are represented by elements of
      array M[2*N+1][2*N+1]. 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 NoWall 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. */
      private static int N;       // Size of maze. */
      private static int M[][];   // Maze, walls, and path.
      private static final int Wall = -1;
      private static final int NoWall = 0;
      private static int lo, hi; // Left/top and right/bottom maze indices.
   ...
   } /* MRP */
```

☞ **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:

```
/* Maze, Rat, and Path (MRP) Representations. */
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. */
      private static int r, c, d;
   ...
   } /* MRP */
```

☞ **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).**

Private and internal to MRP. No other class needs to know about them.

```
/* Maze, Rat, and Path (MRP) Representations. */
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. */
      private static int r, c, d;
   ...
   } /* MRP */
```

☞ **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:

```
/* Maze, Rat, and Path (MRP) Representations. */
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. */
      private static final int Unvisited = 0;
      private static int move;
   ...
 } /* MRP */
```

☞ **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).**

Private and internal to MRP. No other class needs to know about them.

```
/* Maze, Rat, and Path (MRP) Representations. */
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. */
      private static final int Unvisited = 0;
      private static int move;
   ...
 } /* MRP */
```

☞ **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).**

Interface Implementation:

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   // Unit vectors in direction d =          0,     1,    2,    3
   //                                       up, right, down, left
      private static final int deltaR[] = { -1,     0,    1,    0 };
      private static final int deltaC[] = {  0,     1,    0,   -1 };
   public static TurnClockwise()
      { d = (d+1)%4; }
   public static TurnCounterClockwise()
      { d = (d+3)%4; }
   public static StepForward()
      { r = r+2*deltaR[d]; c = c+2*deltaC[d]; move++; M[r][c] = move; }
   public staticboolean isFacingWall()
      { return M[r+deltaR[d]][c+deltaC[d]]==Wall; }
   public staticboolean isAtCheese()
      { return (r==hi)&&(c==hi); }
   public static boolean isAboutToRepeat()
      { return (r==lo)&&(c==lo)&&(d==3); }
 } /* MRP */
```

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

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Input N-by-N maze. */
   public static void Input() {
      } /* Input */
   /* Output N-by-N maze, with walls and path. */
   public static void PrintMaze() {
      } /* PrintMaze */
   } /* MRP */
```

Slight language extension: Multiple lefthand sides for assignment statement.

Input: Hard code an initial example.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Input N-by-N maze. */
   public static void Input() {
      /* Maze. As per representation invariant. */
         N = 1;                        // Size of maze.
         lo= 1; hi = 2*N-1;            // First and last edges of maze.
         M = new int[2*N+1][2*N+1]; // Maze, walls, and path.
         M[0][1] = M[1][0] = M[1][2] = M[2][1] = Wall;
      /* Rat. Place rat in upper-left cell facing up. */
         r = lo; c = lo; d = 0;
      /* Path. Establish the rat in the upper-left cell. */
         move = 1; M[r][c] = move;
      } /* Input */
   } /* MRP */
```

Input: Invoke from the client.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Input a maze, or reject the input as malformed. */
   private static void Input() { MRP.Input(); } /* Input */
   ...
   } /* RunMaze */
```

Output: Straightforward, so knock it off, in general.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Output N-by-N maze, with walls and path. */
   public static void PrintMaze() {
      for (int r = lo-1; r<=hi+1; r++) {
         for (int c = lo-1; c<=hi+1; c++) {
            String s;
            if (M[r][c]==Wall) s = "#";
            else if (M[r][c]==NoWall || M[r][c]==Unvisited) s = " ";
            else s = M[r][c]+"";
            System.out.print((s+" ").substring(0,3));
            }
         System.out.println();
         }
      } /* PrintMaze */
   } /* MRP */
```

Output: Invoke from the client.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Output the direct path found, or “unreachable” if there is none. */
   private static void Output() {
      if (!MRP.isAtCheese()) System.out.println("Unreachable");
      else MRP.PrintMaze();
      } /* Output */
   ...
   } /* RunMaze */
```

Commentary : Design rules for abstract data types.

- Prefer fine-grained micro-operations over coarse-grained macro-operations.
  - E.g., `TurnClockwise` 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., `isAtCheese` rather than `getRow` and `getColumn`.
  - E.g., `TurnClockwise` rather than `getDirection` and `SetDirection`.
- Avoid macro-operations that embody algorithmic details that belong in the client.
  - E.g., `RunMaze.Solve` rather than `MRP.Solve` .

Initial Tests

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

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

	`#`
	`# 1 #`
	`#`

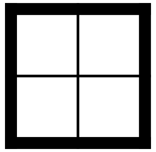
input	**output**

Correct output.

☞ **Test programs incrementally.**

Controlled Testing: Still use an empty stub for `Solve`, but change Input to hard-code a 2-by-2.

Test 2: Check `Output`.

	Unreachable
input	**output**

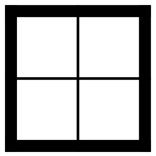
Expected output since `Solve` is just a stub. Validation of code for message.

☞ **Test programs incrementally.**

Initial Tests

Controlled Testing: Now use real code for `Solve`.

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

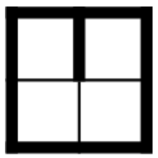
<table>
<tr><td></td><td><pre>
  # #
1 2

3
  # #
</pre></td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

Correct solution.

Controlled Testing: Change `Input` to hard-code a 2-by2, with an obstacle.

Test 4: Further check of `Solve`.



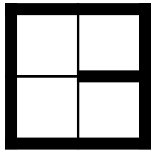
```
  #  #
# 1    #

# 2  3 #
  #  #
```

input	output

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-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.

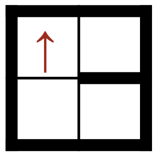
	# # # 3 2 # # 4 5 # # #
input	**output**

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.

Direct Paths

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.



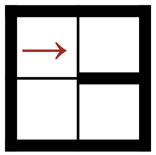
```
  #  #
# 1    #

#      #
  #  #
```

input	output

Replay.

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.

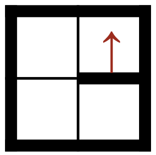
<table>
<tr><td></td><td># #
1

# #
#</td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

Replay.

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.



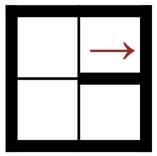
```
  #  #
# 1  2 #

#      #
  #  #
```

input	output

Replay.

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.

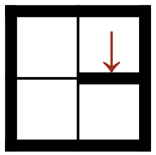
<table>
<tr><td></td><td><pre>
   #   #
1   2

#   #
   #   #
</pre></td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

Replay.

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.

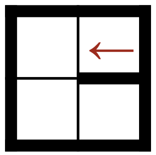
<table>
<tr><td></td><td><pre>
  # #
1 2

#   #
  # #
</pre></td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

Replay.

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.



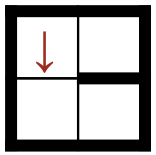
```
   #   #
#  1   2  #

#         #
   #   #
```

input	output

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-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.

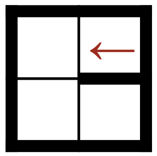
<table>
<tr><td></td><td><pre>
  #   #
3   2

#   #
  #   #
</pre></td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

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

Controlled Testing: Change `Input` to hard-code a 2-by2, with a cul-de-sac.

Test 5: Further check of `Solve`.




```
   #   #
#  1   2  #


#         #
   #   #
```

input	output

Backing up, we need to prevent this.

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

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Compute a direct path through the maze, if one exists. */
   private static void Solve() {
      while ( !MRP.isAtCheese() && !MRP.isAboutToRepeat() )
         if ( MRP.isFacingWall() ) MRP.TurnClockwise();
         else if ( MRP.isFacingUnvisited() ) {
            MRP.StepForward();
            MRP.TurnCounterClockwise();
            }
         else Retract();
      } /* Solve */
   ...
   } /* RunMaze */
```

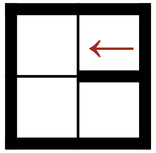
Add the check …

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

Extend MRP: Add `isFacingUnvisited` to interface.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   public static boolean isFacingUnvisited()
      { return M[r+2*deltaR[d]][c+2*deltaC[d]] == Unvisited; }
   ...
 } /* MRP */
```

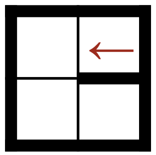
Retract:

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

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.

Direct Paths

Retract:



```
   #   #
#  1   2  #


#         #
   #   #
```

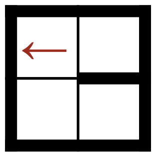
input	output

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 `StepForward` that took us into the cul-de-sac.

```
public static StepForward()
  { r = r+2*deltaR[d]; c = c+2*deltaC[d]; move++; M[r][c] = move; }
```

Retract:



```
   #  #
 # 1    #

 #      #
   #  #
```

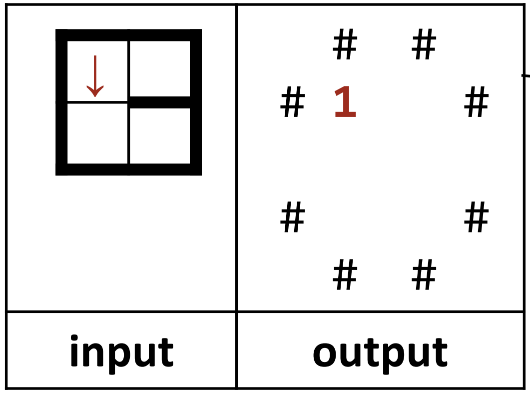
input	output

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 `StepForward` that took us into the cul-de-sac.

```
public static StepForward()
  { r = r+2*deltaR[d]; c = c+2*deltaC[d]; move++; M[r][c] = move; }
public static void StepBackward()
  { M[r][c] = Unvisited; move--; r = r+2*deltaR[d]; c = c+2*deltaC[d]; }
```

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

```
public static StepForward()
  { r = r+2*deltaR[d]; c = c+2*deltaC[d]; move++; M[r][c] = move; }
public static void StepBackward()
  { M[r][c] = Unvisited; move--; r = r+2*deltaR[d]; c = c+2*deltaC[d]; }
```

Retract: Implemented as follows.

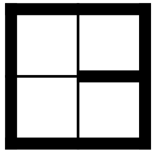
```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Unwind abortive exploration. */
   private static void Retract () {
      MRP.StepBackward();
      MRP.TurnCounterClockwise();
      } /* Retract */
   ...
   } /* RunMaze */
```

Retract: Implemented as follows.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Unwind abortive exploration. */
   private static void Retract () {
      MRP.StepBackward();
      MRP.TurnCounterClockwise();
      } /* Retract */
   ...
   } /* RunMaze */
```

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

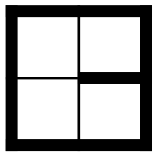
Test 6: Redo Test 5.

<table>
<tr><td></td><td><pre>
   #   #
1

2   3
   #   #
</pre></td></tr>
<tr><td>input</td><td>output</td></tr>
</table>

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

Test 6: Redo Test 5.



```
   #   #
#  1       #

#  2   3   #
   #   #
```

input	output

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 hardcoded initial example.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Input N-by-N maze. */
   public static void Input() {
      /* Maze. As per representation invariant. */
         N = 1;                          // Size of maze.
         lo= 1; hi = 2*N-1;              // First and last indices of maze.
         M = new int[2*N+1][2*N+1]; // Maze, walls, and path.
         M[0][1] = M[1][0] = M[1][2] = M[2][1] = Wall;
      /* Rat. Place rat in upper-left cell facing up. */
         r = lo; c = lo; d = 0;
      /* Path. Establish the rat in the upper-left cell. */
         move = 1; M[r][c] = move;
      } /* Input */
   } /* MRP */
```

Input: Identify places to change, and code.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Input N, and (2N+1)-by-(2N+1) values; non-blanks are walls. */
   public static void Input() {
      /* Maze. */
         Scanner in = new Scanner(System.in);
         N = ⟨value for N⟩;
         M = new int[2*N+1][2*N+1]; // Maze, walls, and path.
         ⟨Define each element of M⟩
      /* Rat. */
         r = lo; c = lo; d = 0;
      /* Path. */
         move = 1; M[r][c] = move;
      } /* Input */
   } /* MRP */
```

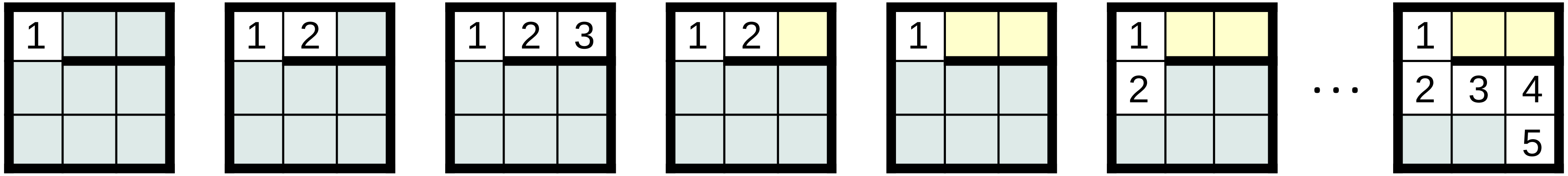
Input

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
   /* Input N, and (2N+1)-by-(2N+1) values; non-blanks are walls. */
   public static void Input() {
      /* Maze. */
         Scanner in = new Scanner(System.in);
         N = in.nextInt(); in.nextLine();
         lo = 1; hi = 2*N-1;         // Left and right edges of maze.
         M = new int[2*N+1][2*N+1]; // Maze, walls, and path.
         for (int r=lo-1; r<=hi+1; r++) {
            String line = in.nextLine();
            for (int c=lo-1; c<=hi+1; c++)
               if ((r%2==1) && (c%2==1)) M[r][c] = Unvisited;
               else if (line.substring(c,c+1).equals(" "))
                  M[r][c] = NoWall;
               else M[r][c] = Wall;
            }
      /* Rat. */
         r = lo; c = lo; d = 0;
      /* Path. */
         move = 1; M[r][c] = move;
      } /* Input */
   } /* MRP */
```

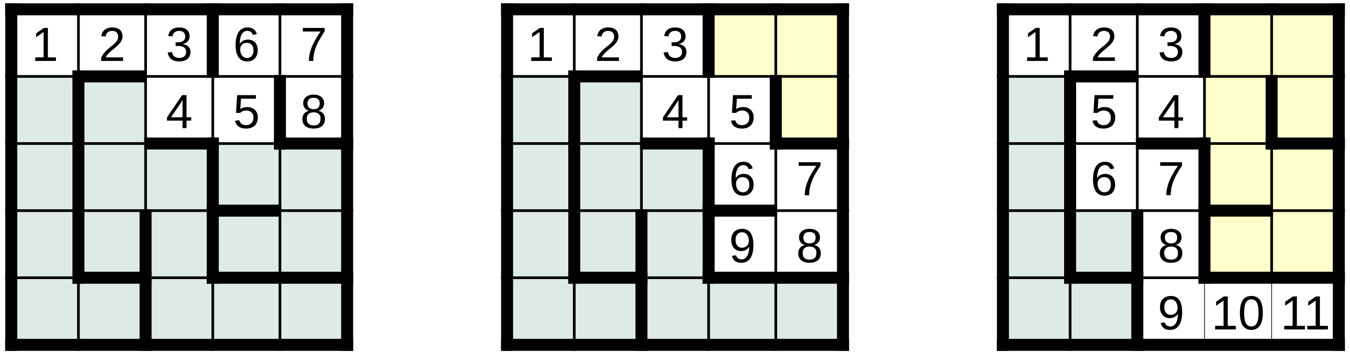
Fussy detail that we shall skip over.

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

Deeper cul-de-sacs



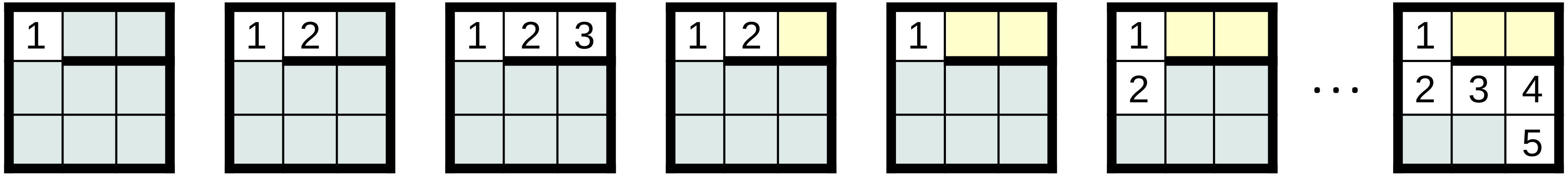
Higgledy-piggledy cul-de-sacs



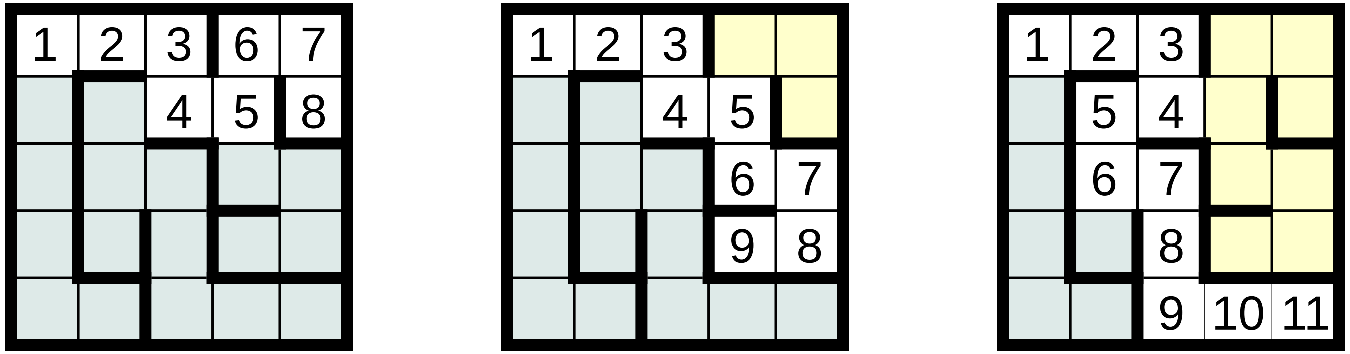
☞ **Test programs thoroughly.**

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.**

Testing, revisited

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

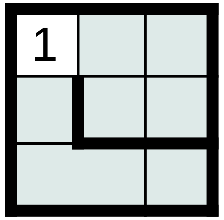
Review Code:

- 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?

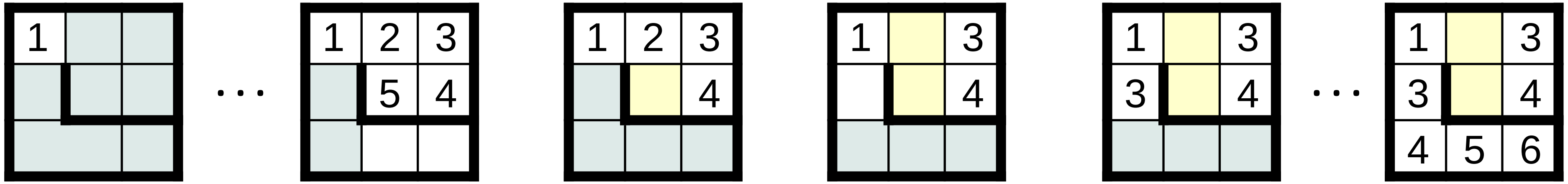
Review Code:

- 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?

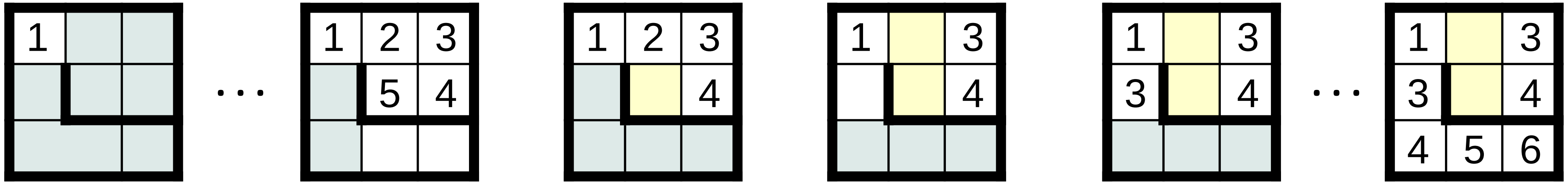
Do you have to just keep trying until you think of a room-shaped cul-de-sac?



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

☞ **Test programs thoroughly.**

Direct Paths, revisited



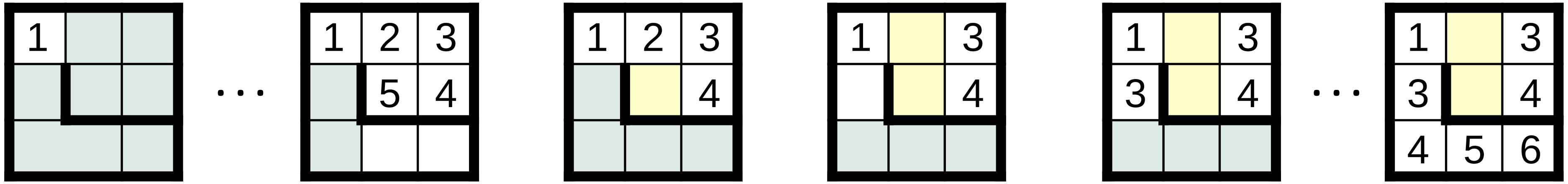
Retract: Implemented as follows.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Unwind abortive exploration. */
   private static void Retract () {
      MRP.StepBackward();
      MRP.TurnCounterClockwise();
      } /* Retract */
   ...
   } /* RunMaze */
```

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

Direct Paths, revisited

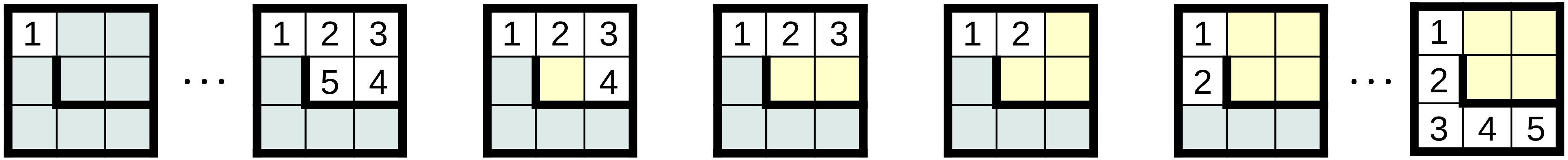


Retract: Implemented as follows.

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Unwind abortive exploration. */
   private static void Retract () {
      while ( /* not unwound */ ) {
         MRP.FacePrevious();
         MRP.StepBackward();
         }
      TurnCounterClockwise();
      } /* Retract */
   } /* RunMaze */
```

Picking up the “bread crumbs”.

Correction: Now we are going in the right direction, truly unwinding the path.



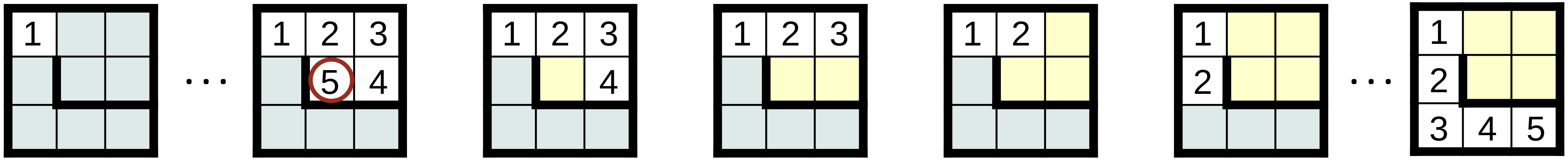
Direct Paths, revisited

Code as if in class MRP, where there is access to the data representation.

⟨r,c⟩

↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]]
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

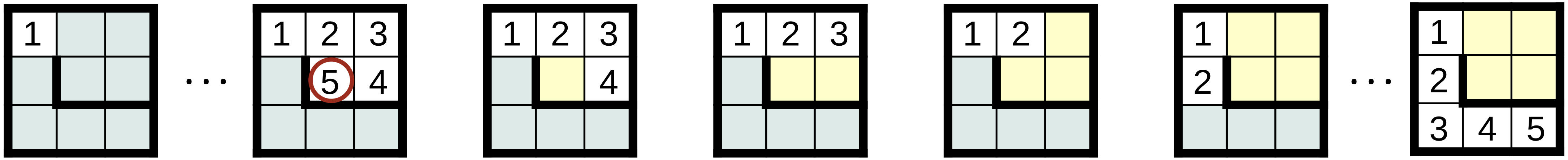


Direct Paths, revisited

Trace: There are actually two separate cul-de-sacs, 2-5 and 1-2.

○ ⟨r,c⟩
↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
☞ public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]]
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

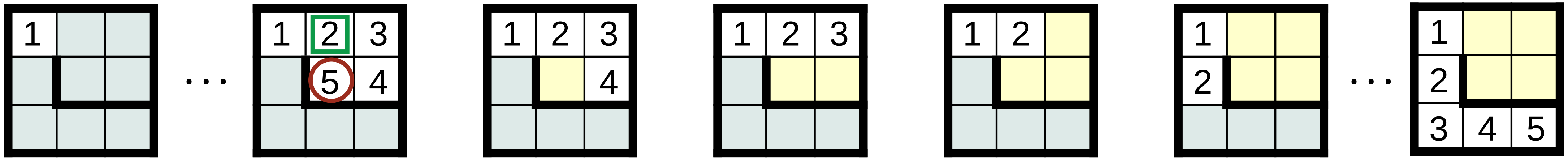


Direct Paths, revisited

⟨r,c⟩

↑ d

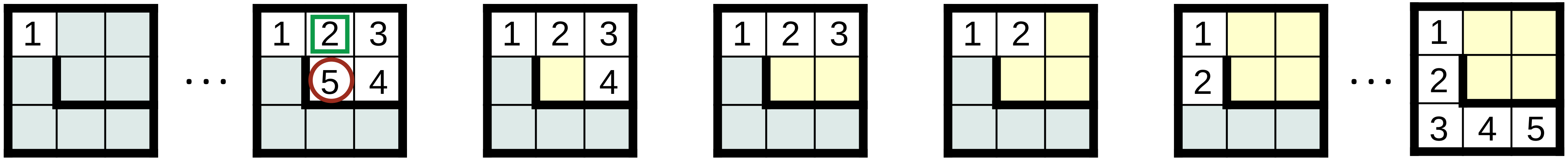
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.              ↑
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

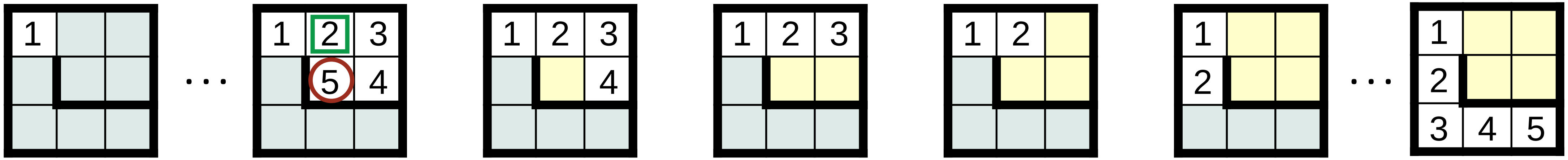


Direct Paths, revisited

⟨r,c⟩

↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.              ↑
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

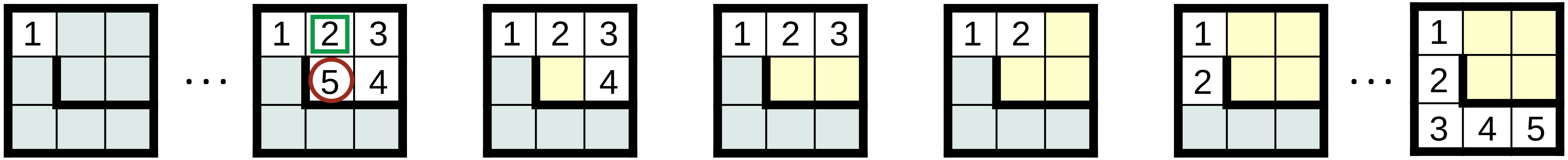


Direct Paths, revisited

⟨r,c⟩

→ d

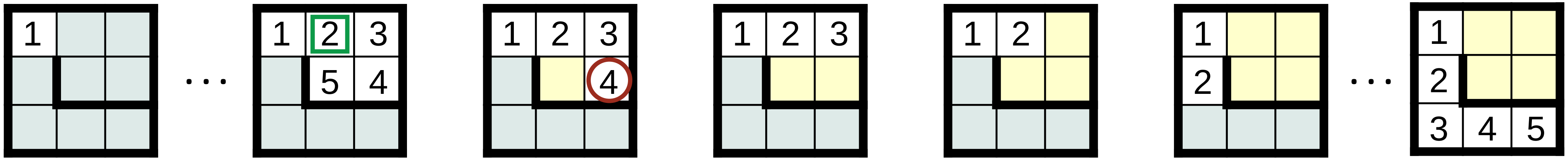
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.              ↑
      while ( M[r][c] != neighborNumber ) {
☞        FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
→ d

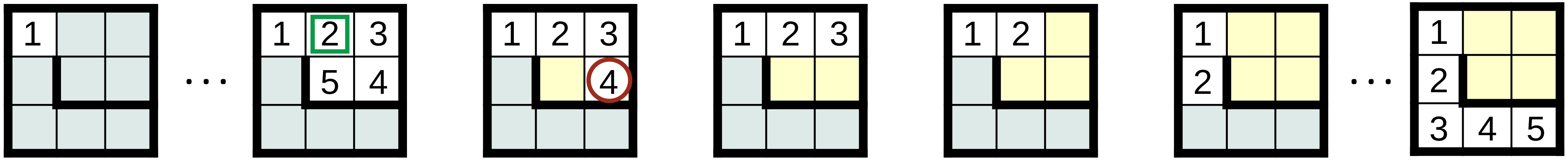
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

○ ⟨r,c⟩
→ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

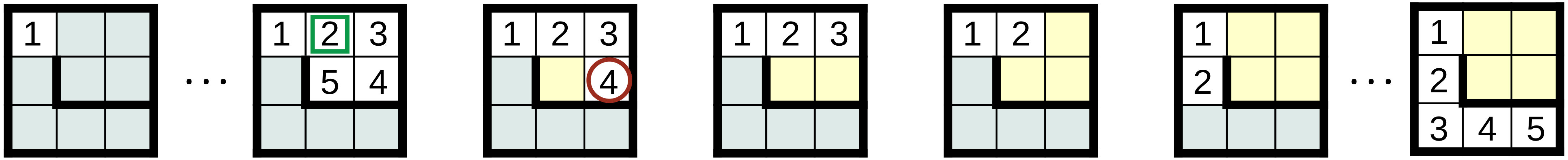


Direct Paths, revisited

⟨r,c⟩

↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

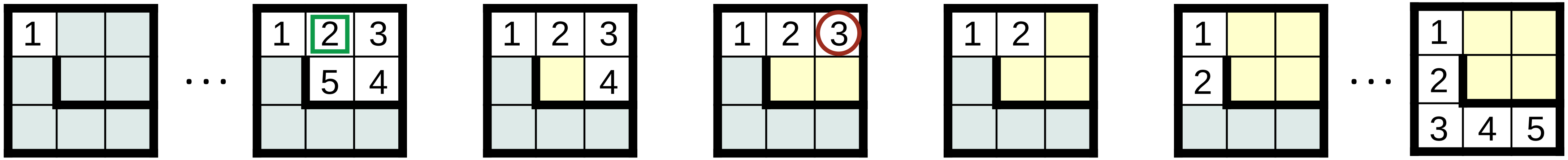


Direct Paths, revisited

○ ⟨r,c⟩

↑ d

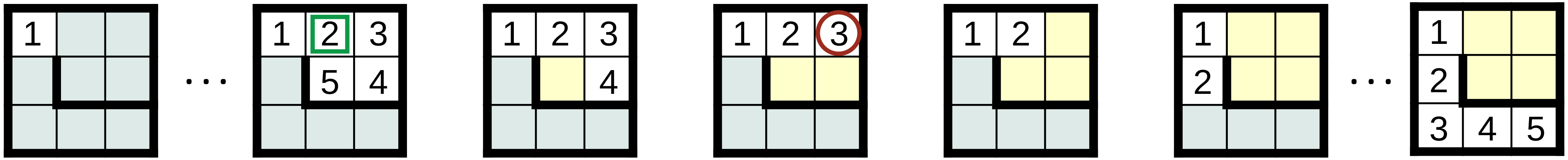
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;      // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
↑ d

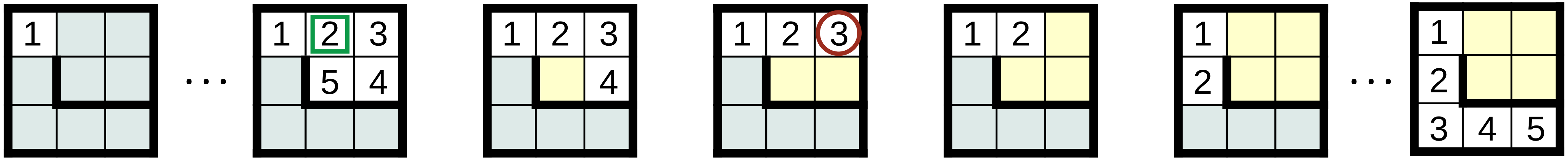
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.             ↑
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

◯ ⟨r,c⟩
← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.             ↑
      while ( M[r][c] != neighborNumber ) {
☞        FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

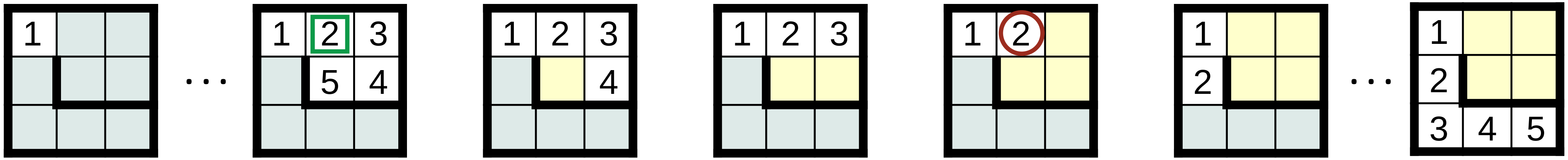


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.             ↑
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

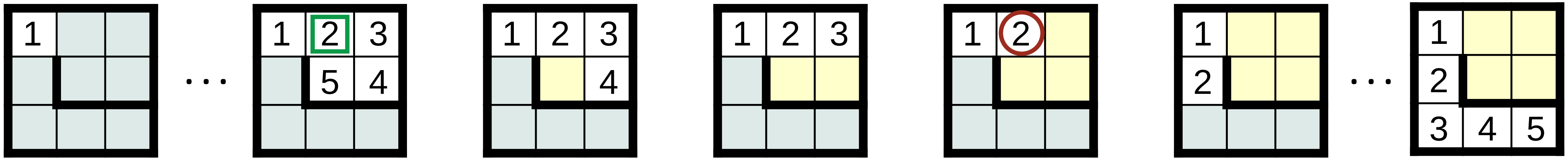


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

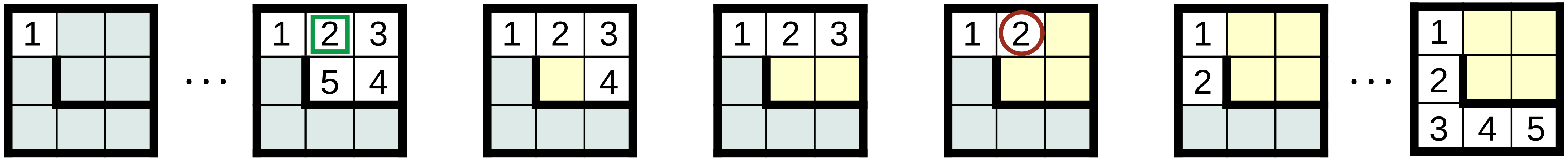


Direct Paths, revisited

⟨r,c⟩

↑ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   2
      int neighborDirection = d;  // Save direction.              ↑
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.           ↑
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

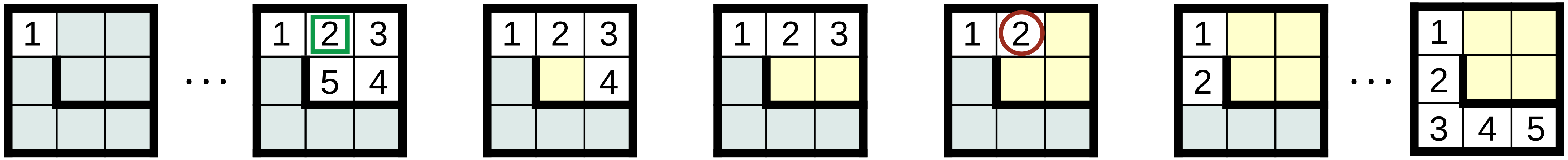


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

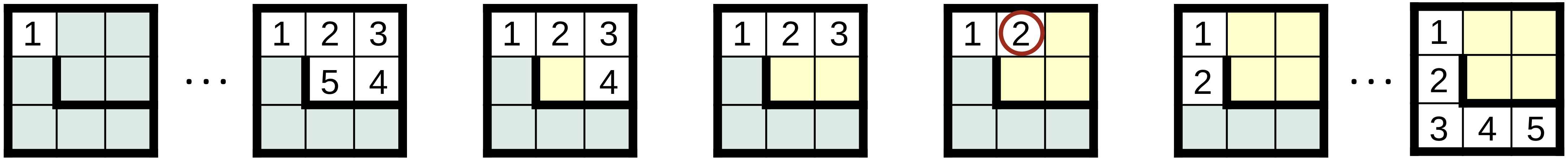


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.          ←
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



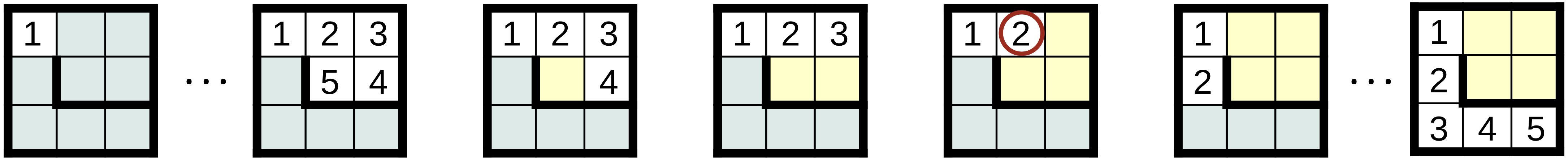
Direct Paths, revisited

⟨r,c⟩

← d

Second call to Retract.

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]]
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

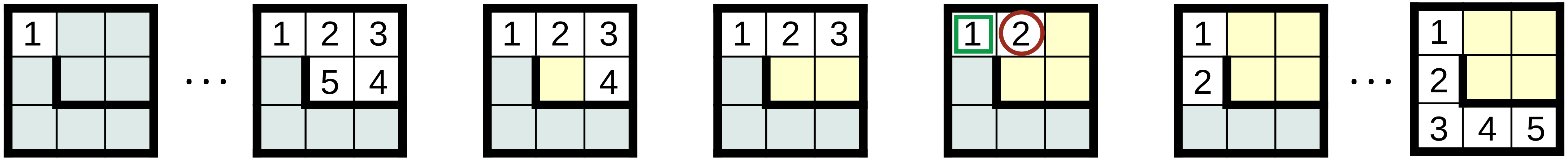


Direct Paths, revisited

⟨r,c⟩

← d

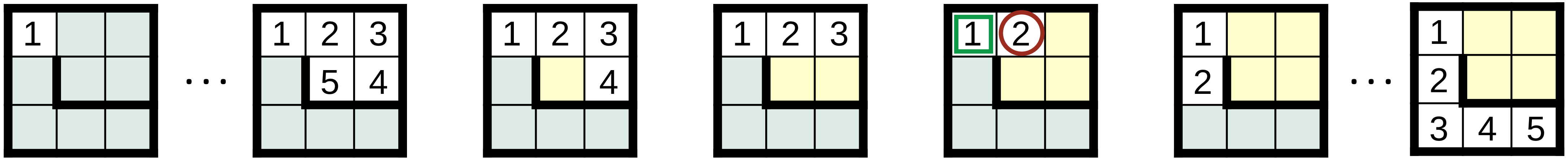
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   1
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   1
      int neighborDirection = d;  // Save direction.               ←
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

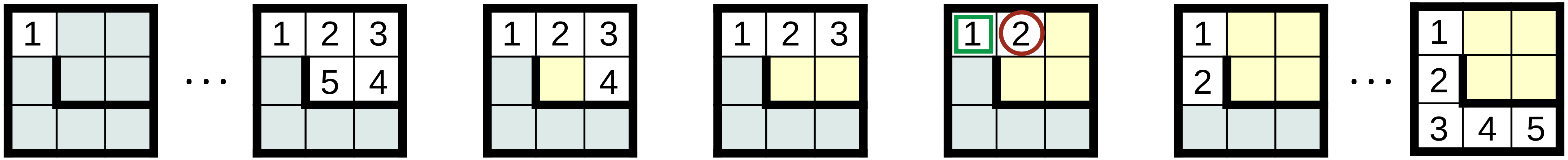


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   1
      int neighborDirection = d;  // Save direction.              ←
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

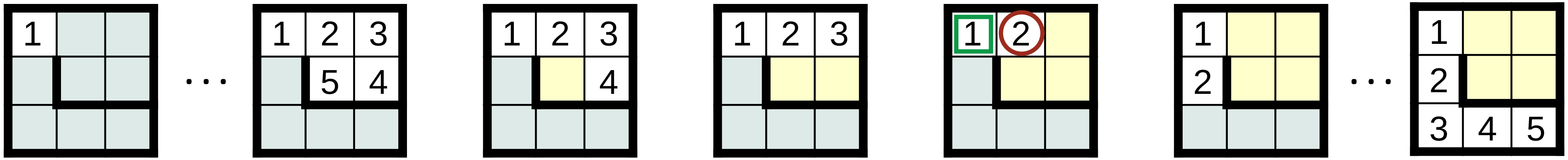


Direct Paths, revisited

⟨r,c⟩

← d

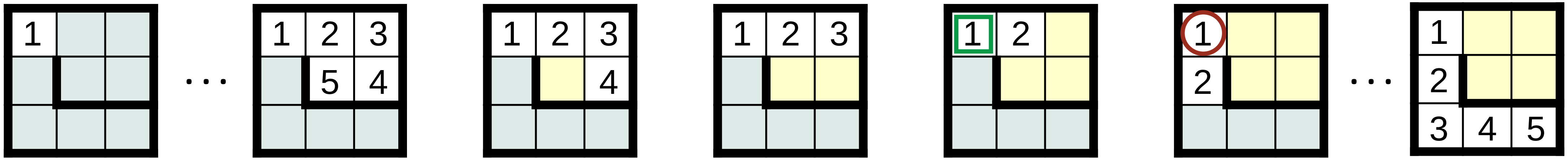
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   1
      int neighborDirection = d;  // Save direction.             ←
      while ( M[r][c] != neighborNumber ) {
☞        FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];  1
      int neighborDirection = d;  // Save direction.            ←
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
☞        StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

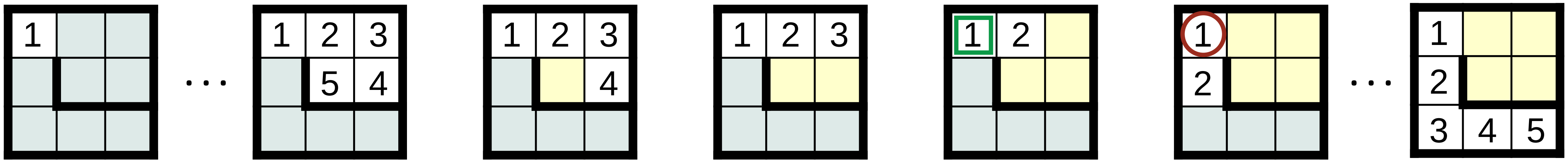


Direct Paths, revisited

⟨r,c⟩

← d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```

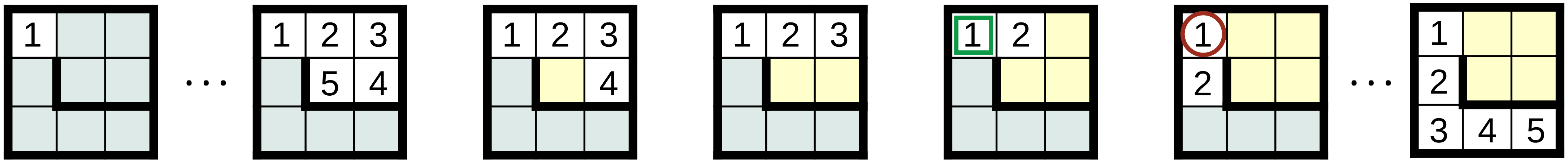


Direct Paths, revisited

⟨r,c⟩

← d

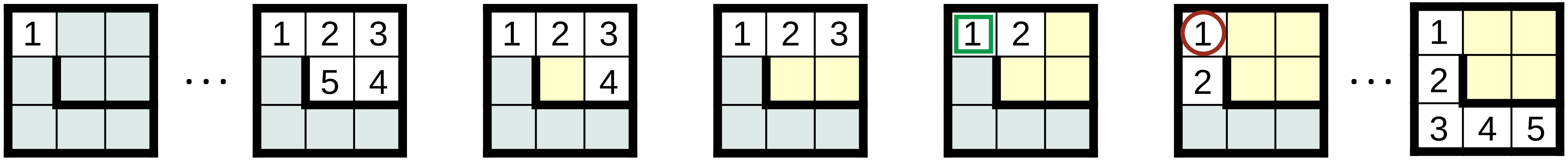
```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];
      int neighborDirection = d;  // Save direction.
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.
      TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



Direct Paths, revisited

⟨r,c⟩
↓ d

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      int neighborNumber    = M[r+2*deltaR[d]][c+2*deltaC[d]];   1
      int neighborDirection = d;  // Save direction.             ←
      while ( M[r][c] != neighborNumber ) {
         FacePrevious();
         StepBackward();
         }
      d = neighborDirection;     // Restore direction.           ↓
☞     TurnCounterClockwise();
      } /* Retract */
   ...
   } /* MRP */
```



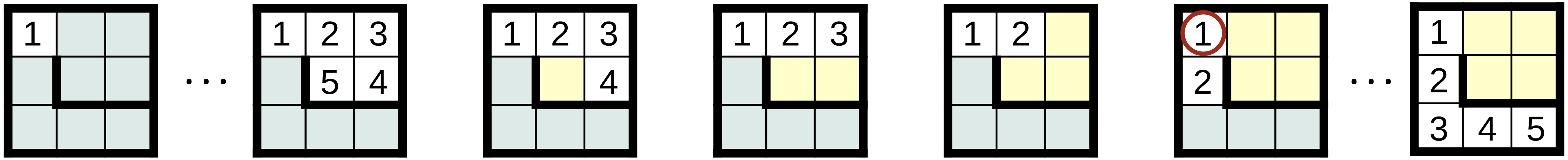
Direct Paths, revisited

```
/* Rat running. See Chapter 15 of text. */
class RunMaze {
   ...
   /* Unwind abortive exploration. */
   public static void Retract() {
      MRP.RecordNeighborAndDirection();

      while ( !MRP.isAtNeighbor() ) {
         MRP.FacePrevious();
         MRP.StepBackward();
         }
      MRP.RestoreDirection();
      MRP.TurnCounterClockwise();
      } /* Retract */
   ...
   } /* RunMaze */
```

But Retract is too algorithmic for MRP. It really belongs in RunMaze.

MRP can provide three helpful primitives:

- `RecordNeighborAndDirection`
- `isAtNeighbor`
- `RestoreDirection`



Direct Paths, revisited

MRP state variables: Support the notion of an “arrow in a cell”.

```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   ...
  private static int neighborNumber;      // Recorded visit #.
  private static int neighborDirection;   // Dir. at time of recording.

  public static void RecordNeighborAndDirection ()
    { neighborNumber = M[r+2*deltaR[d]][c+2*deltaC[d]]; neighborDirection = d; }

  public static boolean isAtNeighbor()  { return M[r][c]==neighborNumber; }

  public static void RestoreDirection() { d = neighborDirection; }
  ...
  } /* MRP */
```

MRP operations (colloquially):

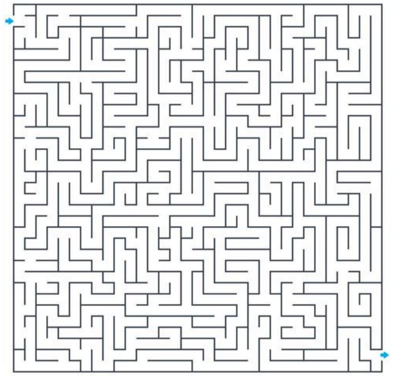
- “Toss an arrow into a neighbor”,
- “Detect being in that neighbor”, and
- “Align direction with the arrow”.

Remaining Implementation: FacePrevious, just a Sequential Search.

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   public static void FacePrevious() {
      int d = 0;
      while ( isFacingWall() || M[r][c]-1 != M[r+2*deltaR[d]][c+2*deltaC[d]] )
         d++;
      } /* FacePrevious */
   ...
   } /* MRP */
```

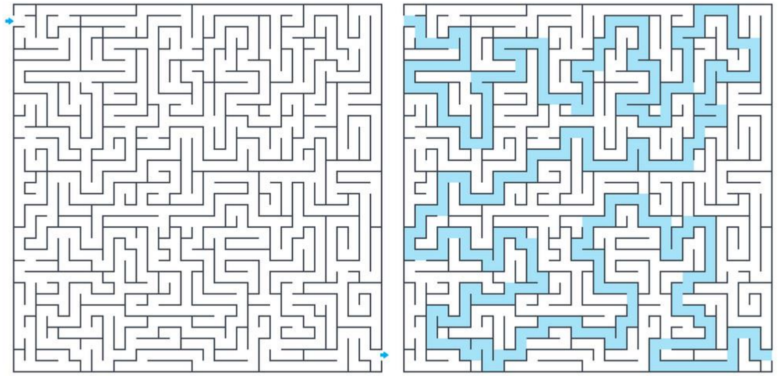
Self-checking: The setting.

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.

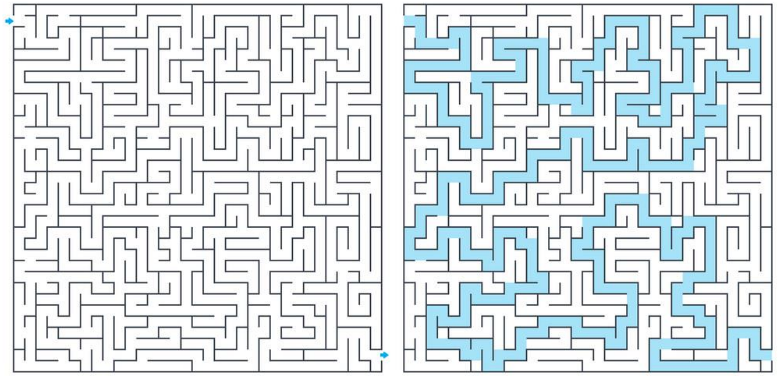


Self-checking: The setting.

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.



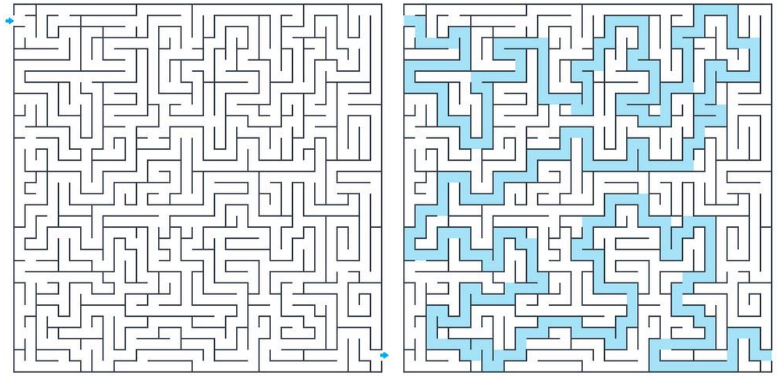
Self-checking: The setting.

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.

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Return false iff rat reached cell ⟨r,c⟩ via an invalid path. */
   public static boolean isValidPath(int r, int c) {
      if ( M[r][c]==Unvisited ) return true; // No claim if Unvisited.
      else
         while ( !((r==lo)&&(c==lo)) ) {
            /* Go to any valid predecessor; return false if there is none. */
               int d = 0;
               while ( d<4 && (M[r+deltaR[d]][c+deltaC[d]] == Wall ||
                       M[r+2*deltaR[d]][c+2*deltaC[d]] != M[r][c]-1 ) ) d++;
               if (d==4) return false;
               r = r+2*deltaR[d]; c = c+2*deltaC[d];
            }
      return true; // Reached upper-left cell.
      } /* isValidPath */
   ...
   } /* MRP */
```

Self-checking: The checking code.

```
/* Rat running. See Chapter 15 of text. */
class MRP {
   ...
   /* Return false iff rat reached lower-right cell via an invalid path. */
   public static boolean isSolution () {
      return isValidePath(hi,hi);
      } /* isSolution */
   ...
   } /* MRP */
```

Self-checking:. Making the assertion as a last step in RunMaze.

```
assert MRP.isSolution(): "internal program error";
```

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

Testing, revisited yet again

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.

```
/* Create an N-by-N maze with walls given by the bits of w. */
public static void GenerateInput(int N, int w) {
   /* Maze. */
      M = new int[2*N+1][2*N+1];
      lo = 1; hi = 2*N-1;
   /* Set boundary walls. */
      for (int i=0; i<=hi+1; i++)
         M[lo-1][i] = M[hi+1][i] = M[i][lo-1] = M[i][hi+1] = Wall;
   /* Set 2*n*(n-1) interior walls to the corresponding bits of w. */
      for (int r=lo; r<=hi; r++)
         for (int c=lo; c<=hi; c++)
            if ( (r%2==0 && c%2==1)||(r%2==1 && c%2==0) ) {
               if ( w%2==1 ) M[r][c] = Wall; else M[r][c] = NoWall;
               w = w/2;
               }
   /* Rat. */
      r = lo; c = lo; d = 0;
   /* Path. */
      move = 1; M[r][c] = move;
 } /* GenerateInput */
```

Exhaustive Bounded Testing: Iterating through mazes.

```
/* Generate/solve all mazes of sizes up through 4, and validate paths found. */
public static void test() {
   for (int N = 1; N<=4; N++)
      for (int i=0; i<Math.pow(2,2*N*(N-1)); i++) {
         MRP.GenerateInput(N,i);
         Solve();
         assert MRP.isValidPath(): "internal program error";
         }
   System.out.println( "passed" );
   } /* test */
```

Random Testing: For larger mazes.

Can't test them all, but can generate and test random mazes of a given size (for as long as you want), and validate solutions. This is called *fuzz testing.*

N.B. Given the way wall configurations are expressed above, you will have to use Java's **`long`** or BigInteger integers for big values of N.