Principled Programming

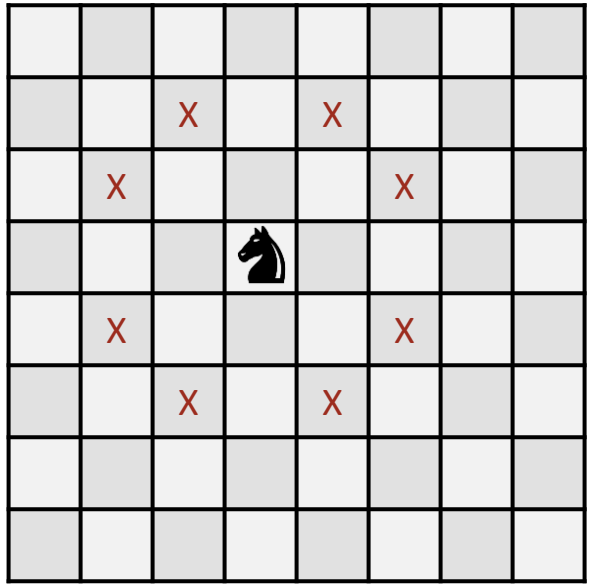
Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Knight's Tour

A Knight can move 2 squares in one direction, and 1 square in the perpendicular direction.



Can a Knight start in the upper left square, and visit every square of an 8-by-8 board exactly once?

1	10	23	42	7	4	13	18
24	41	8	3	12	17	6	15
9	2	11	22	5	14	19	32
0	25	40	35	20	31	16	0
0	36	21	0	39	0	33	30
26	0	38	0	34	29	0	0
37	0	0	28	0	0	0	0
0	27	0	0	0	0	0	0

This attempt failed after move 42, because the Knight got caught in a cul-de-sac.

We present a systematic top-down development of an entire program to find a Knight's Tour. The use of already-presented techniques includes:

- Sequential search.
- Sentinels.
- Find an integer argument at which a function value is minimal.

New techniques introduced include:

- Data representations, and their invariants.
- Use of symbolic constants, and tables of constants.
- Incremental testing.

Two new programming approaches that, while not guaranteed to solve a problem, may be effective, nonetheless:

- Use of heuristics.
- Use of randomness.

x

Where to begin: Get your feet wet.

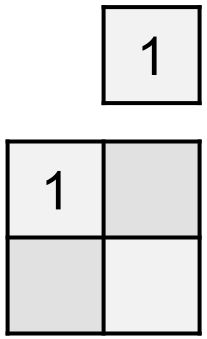
You can start by working the problem by hand, but may find it a bit overwhelming.

An alternative is to generalize to an N-by-N chess board, and then re-instantiate the problem for small values of N.

☞ **Make sure you understand the problem.**
☞ **Confirm your understanding with concrete examples.**

1

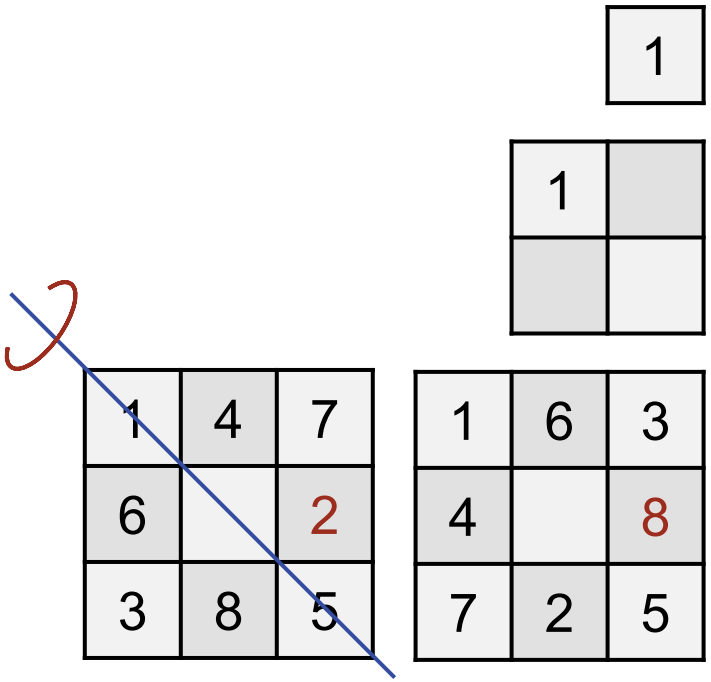
N=1. Solved from the get-go. So, the problem is solvable, in general.



N=1. Solved from the get-go. So, the problem is solvable, in general.

N=2. Impossible, and stuck from the get-go. So, the problem may be not solvable, in general.

Understanding the Problem

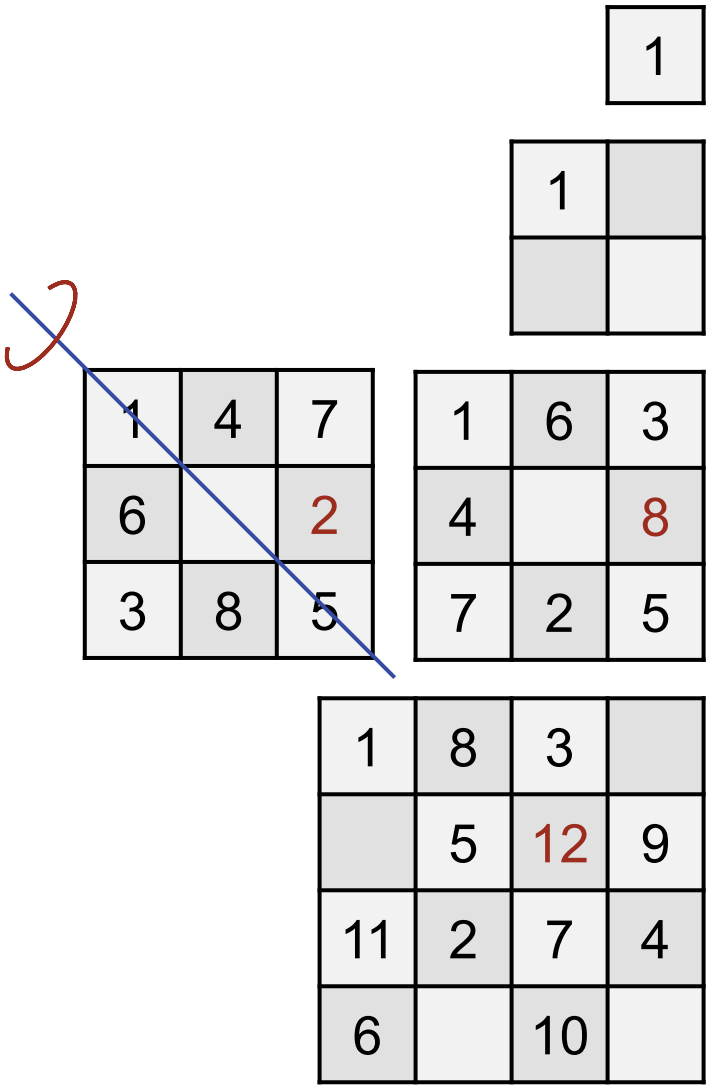


N=1. Solved from the get-go. So, the problem is solvable, in general.

N=2. Impossible, and stuck from the get-go. So, the problem may be not solvable, in general.

N=3. There is a choice at the begin, but thereafter the (partial) tour is proscribed. However, no tour can reach the central square. Taking symmetry into account, the initial choices were not fundamentally different. Might symmetry play a role?

Understanding the Problem



N=1. Solved from the get-go. So, the problem is solvable, in general.

N=2. Impossible, and stuck from the get-go. So, the problem may be not solvable, in general.

N=3. There is a choice at the begin, but thereafter the (partial) tour is proscribed. However, no tour can reach the central square. Taking symmetry into account, the initial choices were not fundamentally different. Might symmetry play a role?

N=4. Lots of choices. The tour shown is stuck in a cul-de-sac at move 12. No solution is readily found, and it is unclear whether there is one. The problem is already big enough to frustrate.

Establish a framework

```
class Tour {

   static void main() { } /* main */
   } /* Tour */
```

☞ **Aggregate the definitions of related variables and methods in a class.**

There are standards for header comments, but we will simplify.

Establish a framework

```
/* Knight's Tour. */
class Tour {

   static void main() { } /* main */
   } /* Tour */
```

☞ **A class header-comment is descriptive, and omits the details of the methods and variables of the class. Reference available auxiliary documentation.**

There are standards for header comments, but we will simplify.

Establish a framework

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() { } /* main */
   } /* Tour */
```

☞ **A method header-comment specifies the effect of invoking it, and (if the method has non-None type) the value returned. If the method has parameters, the specification is written in terms of those parameters.**

A standard pattern.

Establish a framework

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() {
      /* Initialize. */
      /* Compute. */
      /* Output. */
      } /* main */
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

A standard pattern, elaborated for the problem at hand.

Establish a framework

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() {
      /* Initialize: Establish a tour of length 1. */
      /* Compute: Extend the tour, if possible. */
      /* Output: Print the tour as numbered cells in an N-by-N grid of 0s. */
      } /* main */
   } /* Tour */
```

☞ **A statement-comment is written in terms of program variables, and assumes the representation invariants of those variables.**

Top-level Code Structure

Each pattern part to be implemented by a method of the class.

Code structure: Invoke separate methods to do the work.

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() {
      /* Initialize: Establish a tour of length 1. */
         Initialize();
      /* Compute: Extend the tour, if possible. */
         Solve();
      /* Output: Print the tour as numbered cells in an N-by-N grid of 0s. */
         Display();
      } /* main */
   } /* Tour */
```

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

Method stubs easily created by cut and paste, and light editing.

Code structure: Create stubs for methods.

```
/* Knight's Tour. */
class Tour {

   /* Establish a tour of length 1. */
   static void Initialize() { } /* Initialize */

   /* Extend the tour, if possible. */
   static void Solve() { } /* Solve */

   /* Print tour as numbered cells in N-by-N grid of 0s. */
   static void Display() { } /* Display */

   ...
   } /* Tour */
```

☞ **Write method stubs that allow partial programs to execute.**

Test early and often.

Add a temporary output statement to `Tour.main`

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() {
      /* Initialize: Establish a tour of length 1. */
         Initialize();
      /* Compute: Extend the tour, if possible. */
         Solve();
      /* Output: Print the tour as numbered cells in an N-by-N grid of 0s. */
         Display();
      /* Temporary output. */
         System.out.println("done");
      } /* main */
   } /* Tour */
```

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Test early and often.

Add a temporary output statement to `Tour.main` and invoke it from the interactive shell:

```
Tour.main()
```

```
/* Knight's Tour. */
class Tour {
   /* Output a (possibly partial) Knight's Tour. */
   static void main() {
      /* Initialize: Establish a tour of length 1. */
         Initialize();
      /* Compute: Extend the tour, if possible. */
         Solve();
      /* Output: Print the tour as numbered cells in an N-by-N grid of 0s. */
         Display();
      /* Temporary output. */
         System.out.println("done");
      } /* main */
   } /* Tour */
```

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Test early and often.

Add a temporary output statement to `Tour.main` and invoke it from the interactive shell:

```
Tour.main()
```

Incremental Testing:

Output:

```
done
```

What has been validated?

- Syntactic correctness of overall framework
- That the 3 methods were (presumably) executed in turn.

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Don’t go far before thinking about the (internal) data representation.

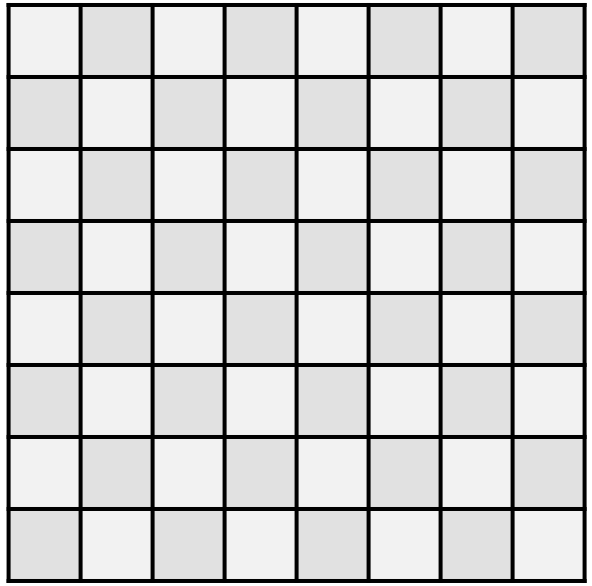
☞ **Dovetail thinking about code and data.**

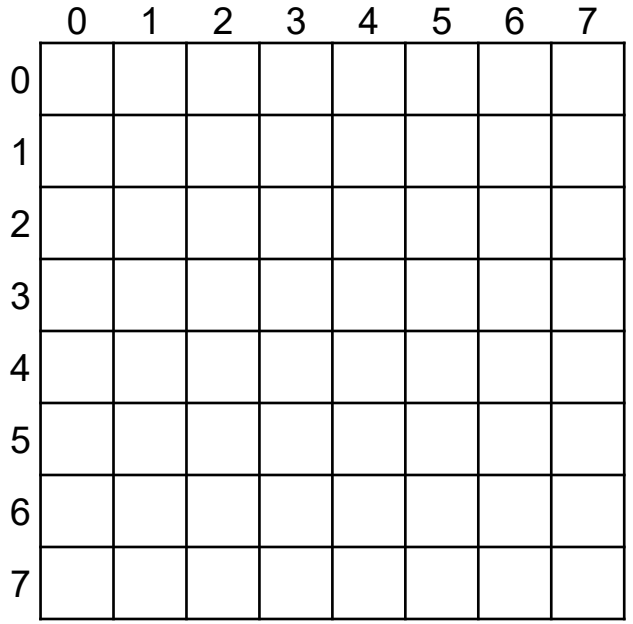
Data Representation:

We need representations of the board and a (partial) tour.

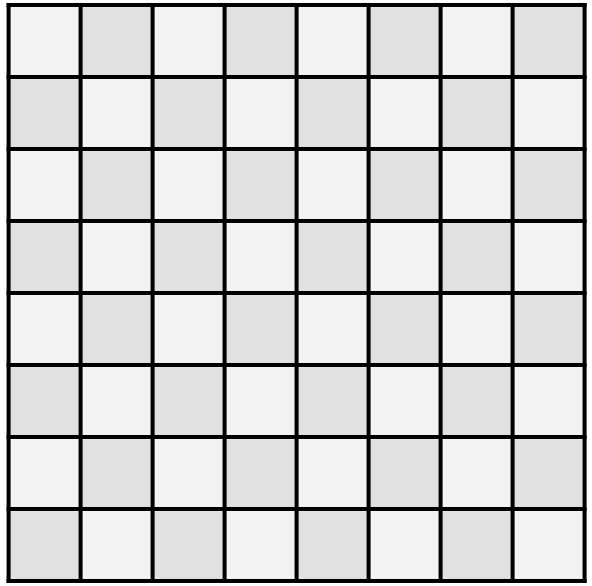
☞ **A program's internal data representation is central to the code; consider it early.**

Board Representation 1: The 2-D physical board can correspond directly to a 2-D array.



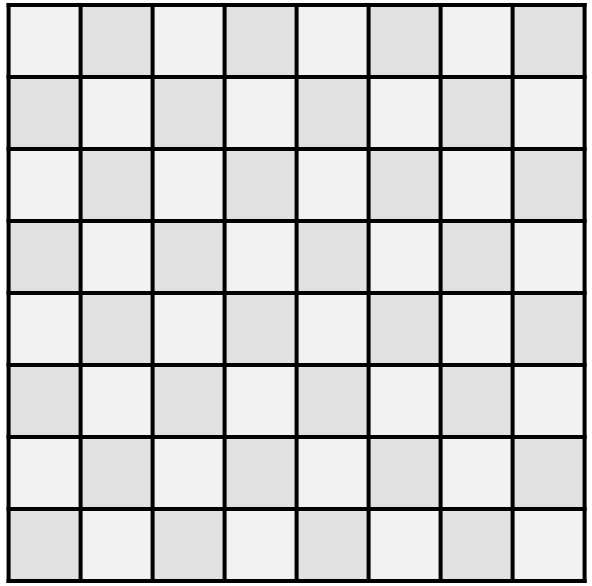


Tour Representation 1: The tour can be represented by visit numbers in array elements.



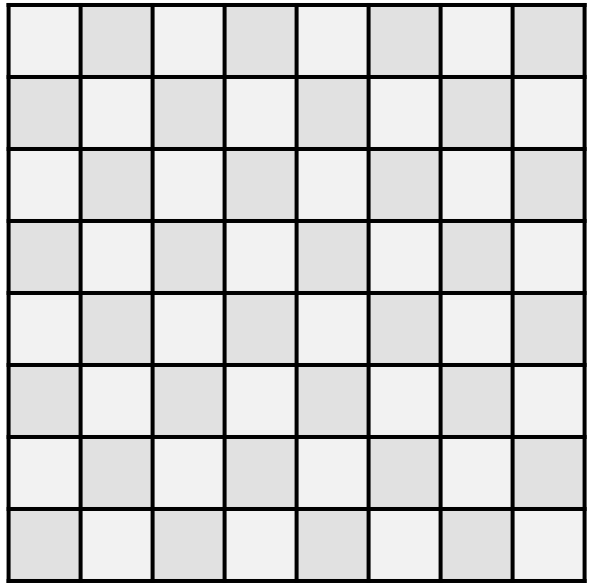
	0	1	2	3	4	5	6	7
0	1					4		
1				3				
2		2			5			
3								
4								
5								
6								
7								

Board Representation 1: A (currently) unvisited square can be 0.



	0	1	2	3	4	5	6	7
0	1	0	0	0	0	4	0	0
1	0	0	0	3	0	0	0	0
2	0	2	0	0	5	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

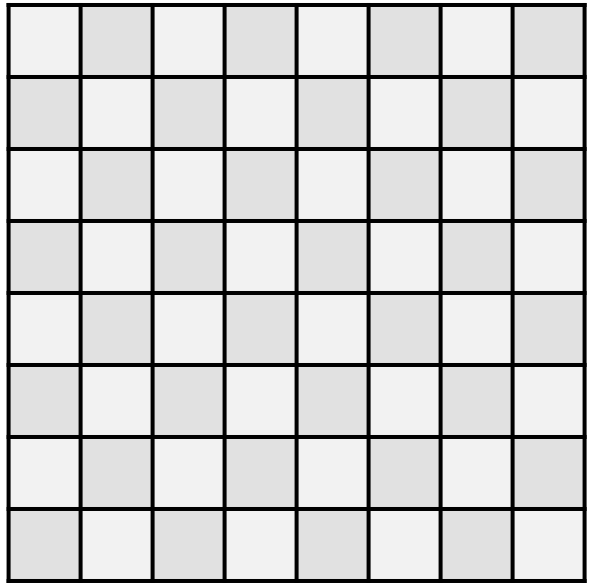
Board Representation 1: The array needs a name.



B	0	1	2	3	4	5	6	7
0	1	0	0	0	0	4	0	0
1	0	0	0	3	0	0	0	0
2	0	2	0	0	5	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

- ☞ **Aspire to making code self-documenting by choosing descriptive names.**
- ☞ **Use single-letter variable names when it makes code more understandable.**

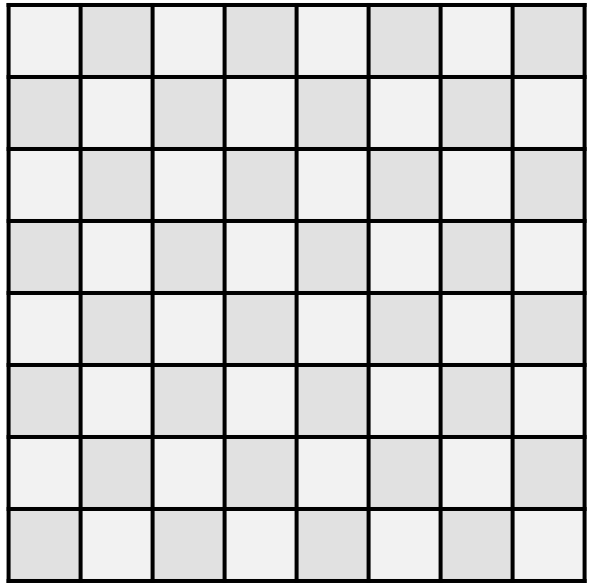
Board Representation 1: Plan for generality by representing the problem size as N.



B	0	1	2	3	4	5	6	7	N
0	1	0	0	0	0	4	0	0	
1	0	0	0	3	0	0	0	0	
2	0	2	0	0	5	0	0	0	
3	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	
N									

- ☞ **Minimize use of literal numerals in code; define and use symbolic constants.**
- ☞ **Aim for single-point-of-definition.**

Board Representation 1: To allow for future flexibility, use symbolic constants for index limits.



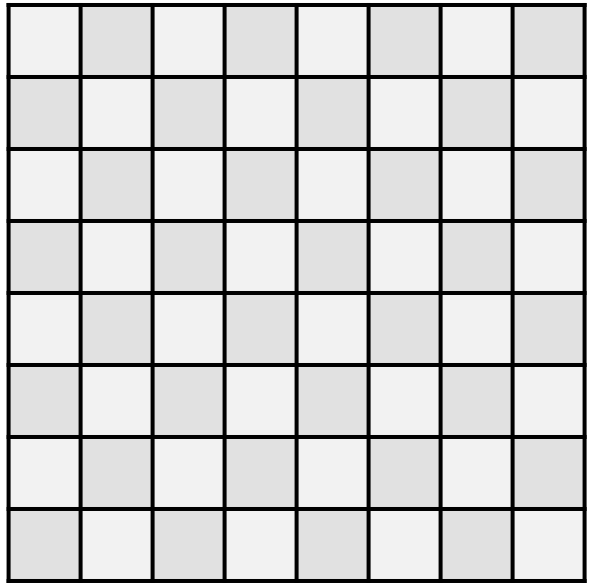
B \ N	0	1	2	3	4	5	6	7
lo 0	1	0	0	0	0	4	0	0
1	0	0	0	3	0	0	0	0
2	0	2	0	0	5	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
hi 7	0	0	0	0	0	0	0	0

(lo = 0, hi = 7; B = row index, N = column index)

BLANK 0

- ☞ **Minimize use of literal numerals in code; define and use symbolic constants.**
- ☞ **Aim for single-point-of-definition.**

Board Representation 1: Keep track of state in redundant variables.



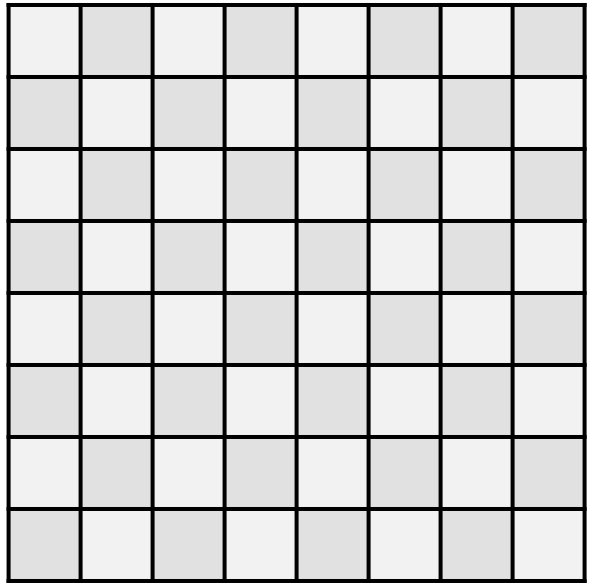
		lo				c			hi	
	B	0	1	2	3	4	5	6	7	N
lo	0	1	0	0	0	0	4	0	0	
	1	0	0	0	3	0	0	0	0	
r	2	0	2	0	0	5	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
hi	7	0	0	0	0	0	0	0	0	
	N									

BLANK 0

move 5

☞ **Introduce redundant variables in a representation to simplify code, or make it more efficient.**

Board Representation 1: Write invariants for the data representations as specifications.



		lo				c			hi	
	B	0	1	2	3	4	5	6	7	N
lo	0	1	0	0	0	0	4	0	0	
	1	0	0	0	3	0	0	0	0	
r	2	0	2	0	0	5	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
hi	7	0	0	0	0	0	0	0	0	
	N									

BLANK 0

move 5

☞ **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).**

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **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 at the top-level of a class are called *class* (or *static*) *variables*, and are shared among all of the methods of the class.

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **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).**

Define `hi` in terms of `lo` and `N` to facilitate possible future changes.

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **Avoid rigid code. Anticipate change. Parameterize**

Define initial values for variables, as much as possible in terms of one another.

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **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 denoted as **`final`** are constants.

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **Leverage features of the programming language and its compiler that protect you from mistakes.**

The default value for integers is 0, which we have chosen as the value of BLANK.

Board Representation 1: Specify the data representation.

```
/* Knight's Tour. */
class Tour {

   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;            // Size of the chess board.
      static final int BLANK = 0;        // Square not on the current tour.
      static final int lo = 0;           // Row/column of upper-left square.
      static final int hi = lo+N-1;      // Row/column of lower-right square.
      static int B[][] = new int[N][N]; // Chess board, initially 0s.

   } /* Tour */
```

☞ **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).**

Tour Representation 1: Write invariants for the data representations as specifications.

```
/* Knight's Tour. */
class Tour {
   ...
   /* A Tour of length move is given by elements of B numbered 1
       to move. Squares numbered consecutively go from ⟨0,0⟩ to ⟨r,c⟩,
       and correspond to legal moves for a Knight. */
       static int r = lo    // Row coordinate of Knight.
       static int c = lo;   // Column coordinate of Knight.
       static int move= 1;  // Length of tour.
       // B[lo][lo] = move;  // Part of the tour invariant includes the
                             // Knight being in the upper-left square
                             // when move=1. It is not possible to do
                             // that here (in the middle of declarations)
                             // so we defer it until Initialize.

   } /* Tour */
```

☞ **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).**

Define initial values for variables, as much as possible in terms of one another.

Tour Representation 1: Write invariants for the data representations as specifications.

```
/* Knight's Tour. */
class Tour {
   ...
   /* A Tour of length move is given by elements of B numbered 1
       to move. Squares numbered consecutively go from ⟨0,0⟩ to ⟨r,c⟩,
       and correspond to legal moves for a Knight. */
       static int r = lo    // Row coordinate of Knight.
       static int c = lo;   // Column coordinate of Knight.
       static int move= 1;  // Length of tour.
       // B[lo][lo] = move;  // Part of the tour invariant includes the
                             // Knight being in the upper-left square
                             // when move=1. It is not possible to do
                             // that here (in the middle of declarations)
                             // so we defer it until Initialize.

   } /* Tour */
```

☞ **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).**

Comment out part of the invariant that cannot be established until later.

Tour Representation 1: Write invariants for the data representations as specifications.

```
/* Knight's Tour. */
class Tour {
   ...
   /* A Tour of length move is given by elements of B numbered 1
       to move. Squares numbered consecutively go from ⟨0,0⟩ to ⟨r,c⟩,
       and correspond to legal moves for a Knight. */
       static int r = lo    // Row coordinate of Knight.
       static int c = lo;   // Column coordinate of Knight.
       static int move= 1;  // Length of tour.
       // B[lo][lo] = move;  // Part of the tour invariant includes the
                             // Knight being in the upper-left square
                             // when move=1. It is not possible to do
                             // that here (in the middle of declarations)
                             // so we defer it until Initialize.

   } /* Tour */
```

☞ **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).**

Assess the Representation: What operations are needed, and what is the utility of the representation proposed?

- **Plan**, as appropriate.
- **Stop** at a cul-de-sac, either on the 64th move or earlier.
- **Extend** the tour, if possible.
- **Retract** the tour, if the strategy calls for backtracking.

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

Assess the Representation: What operations are needed, and what is the utility of the representation proposed?

- **Plan**, as appropriate.
  - Access to full board B could provide any information needed.
- **Stop** at a cul-de-sac, either on the 64th move or earlier.
- **Extend** the tour, if possible.
- **Retract** the tour, if the strategy calls for backtracking.

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

Assess the Representation: What operations are needed, and what is the utility of the representation proposed?

- **Plan**, as appropriate.
  - Access to full board B could provide any information needed.
- **Stop** at a cul-de-sac, either on the 64th move or earlier.
  - Access to full board B will provide visibility of available neighbors.
- **Extend** the tour, if possible.
- **Retract** the tour, if the strategy calls for backtracking.

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

Assess the Representation: What operations are needed, and what is the utility of the representation proposed?

- **Plan**, as appropriate.
  - Access to full board B could provide any information needed.
- **Stop** at a cul-de-sac, either on the 64th move or earlier.
  - Access to full board B will provide visibility of available neighbors.
- **Extend** the tour, if possible.
  - To advance from `B[r][c]` to the neighbor `B[r'][c']`, set ⟨r,c⟩ to ⟨r', c'⟩, increment `move`, and store `move` in `B[r'][c'].`
- **Retract** the tour, if the strategy calls for backtracking.

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

Assess the Representation: What operations are needed, and what is the utility of the representation proposed?

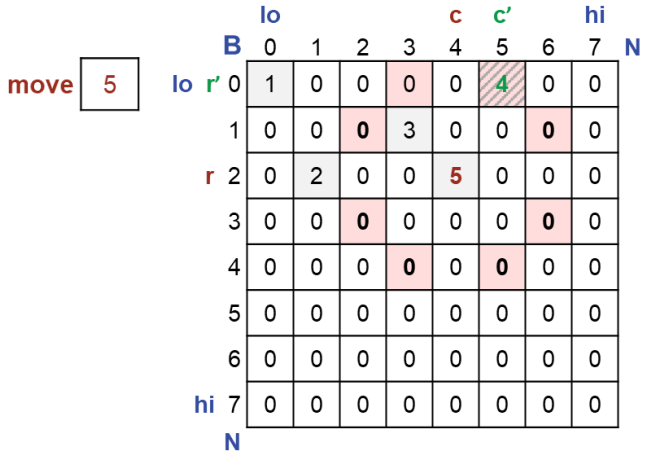
- **Plan**, as appropriate.
  - Access to full board B could provide any information needed.
- **Stop** at a cul-de-sac, either on the 64th move or earlier.
  - Access to full board B will provide visibility of available neighbors.
- **Extend** the tour, if possible.
  - To advance from `B[r][c]` to the neighbor `B[r'][c']`, set ⟨r,c⟩ to ⟨r', c'⟩, increment `move`, and store `move` in `B[r'][c'].`
- **Retract** the tour, if the strategy calls for backtracking.
  - To undo the most recent **extend**, store BLANK in `B[r][c]`, locate previous square ⟨r',c'⟩, set ⟨r,c⟩ to ⟨r',c'⟩, and decrement `move`.

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

Alternative Representation: Address a shortcoming of Representation 1.

- **Retract** the tour, if the strategy calls for backtracking.
  - To undo the most recent **extend**, store BLANK in `B[r][c]`, locate previous square ⟨`r'`,`c'`⟩, set ⟨`r`,`c`⟩ to ⟨`r'`,`c'`⟩, and decrement `move`.

  For Representation 1, a **search** would be required to find ⟨r′, c′⟩.



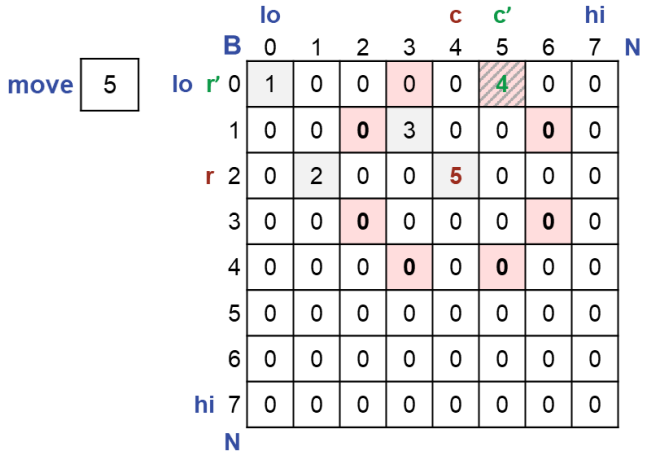
Such a **search** would inspect the eight neighbors of ⟨`r`,`c`⟩ to find which `B[r'][c']` was `move-1`.

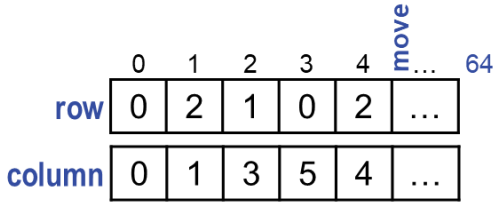
☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Alternative Representation:

- **Retract** the tour, if the strategy calls for backtracking.
  - To undo previous **extend**, locate previous square ⟨r′, c′⟩, set ⟨r,c⟩ to ⟨r′, c′⟩, and decrement move.

  For Representation 1, a **search** would be required to find ⟨r′,c′⟩.



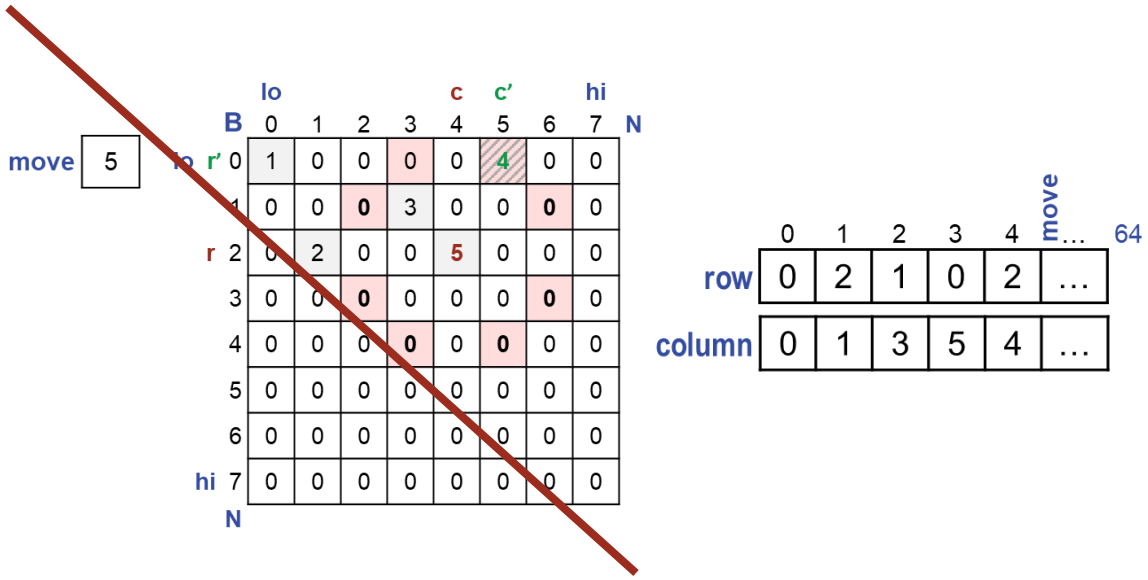


But if the coordinates of tour squares were represented as ordered collections, `row` and `column`, **retract** could be implemented by just decrementing `move`. No **search** would be required.

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

Alternative Representation: Why do we need the board B at all?

Why not just represent the tour by the two ordered collections, `row` and `column`?

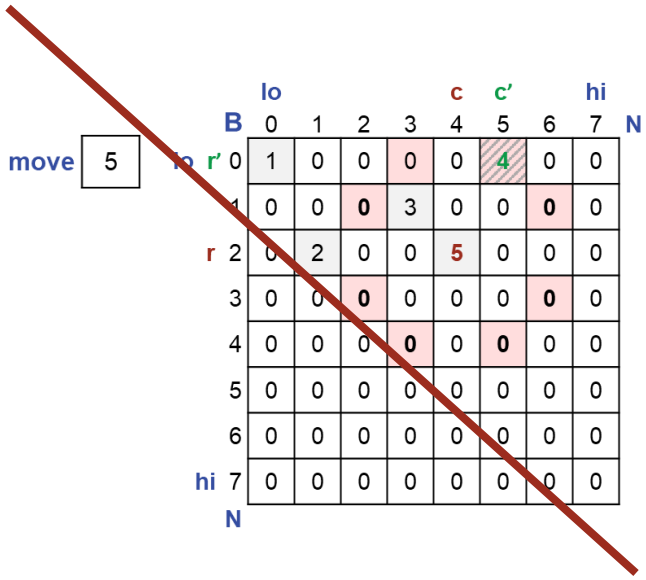


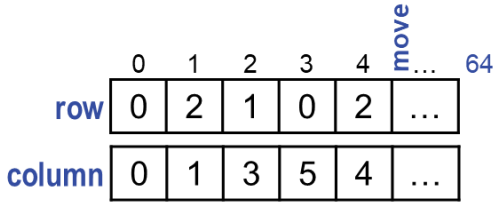
☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Alternative Representation: Why do we need the board B at all?

Why not just represent the tour by the two ordered collections, `row` and `column`?

- **Extend** the tour, if possible.





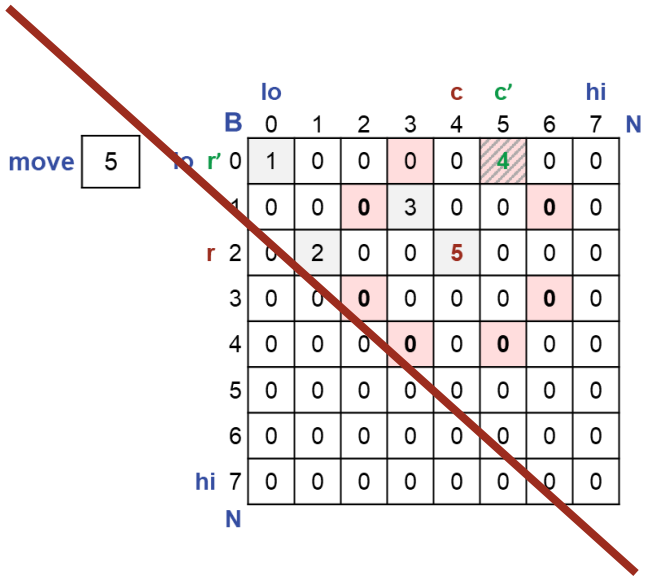
Without the board B, testing whether an ⟨r′,c′⟩ is "unvisited" would require determining whether it is on the current tour, which would require a **search** of the tour in `row` and `column`.

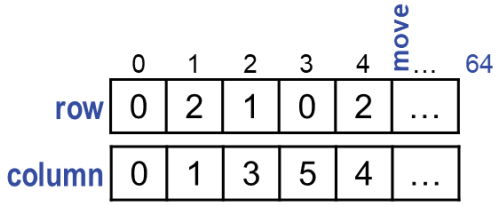
☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Alternative Representation: Why do we need the board B at all?

Why not just represent the tour by the two ordered collections, `row` and `column`?

- **Extend** the tour, if possible.





Without the board B, testing whether an ⟨r′,c′⟩ is "unvisited" would require determining whether it is on the current tour, which would require a **search** of the tour in `row` and `column`.

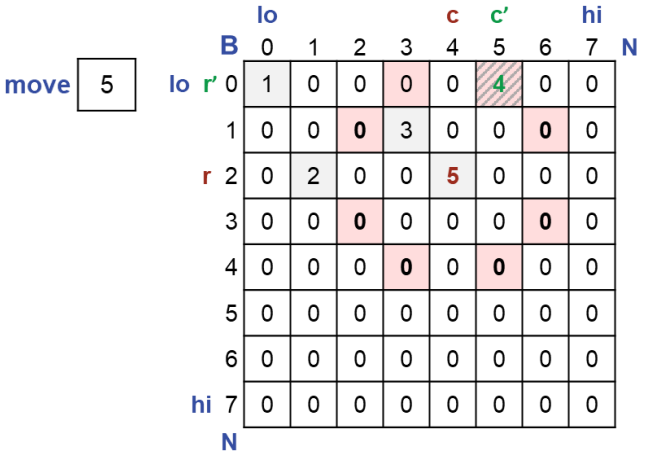
Of course, an auxiliary 2-D **`boolean`** array B indicating "visited" would obviate a search.

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

Representation 1:

Primary: tour recorded in cells of 2-D **`int`** array `B`.

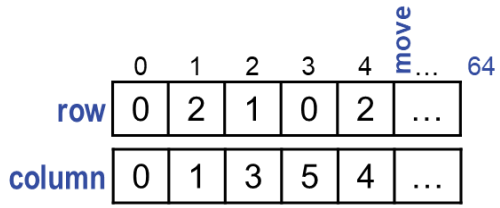
Auxiliary: Variables `row` and `column` to facilitate finding predecessor square, for **Retract**.



Representation 2:

Primary: tour recorded in variables `row` and `column`.

Auxiliary: 2-D **`boolean`** array `B` to facilitate testing whether a square is unvisited, for **Extend**.



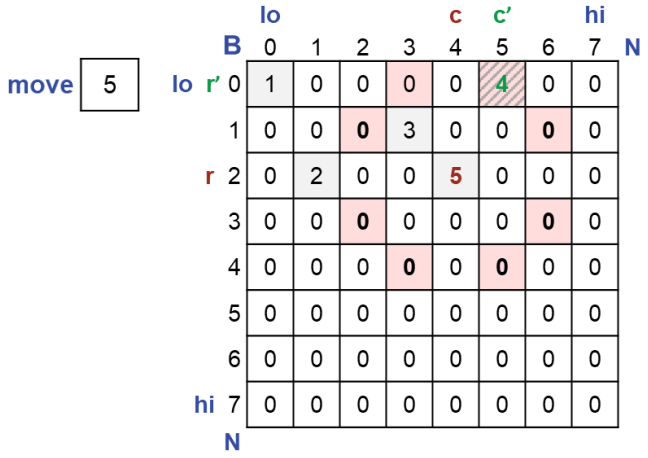
☞ **The touchstone of a data representation is its utility in performing the needed operations.**

Alternative Representation: Which is better? Or is it "six or half dozen the other"?

Representation 1:

Primary: tour recorded in cells of 2-D **`int`** array B.

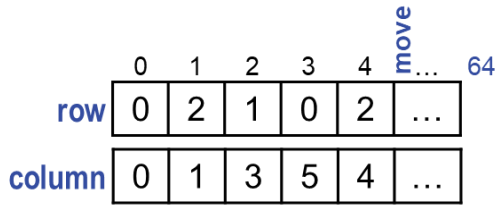
Auxiliary: Variables `row` and `column` to facilitate finding predecessor square, for **Retract**.



Representation 2:

Primary: tour recorded in variables `row` and `column`.

Auxiliary: 2-D **`boolean`** array B to facilitate testing whether a square is unvisited, for **Extend**.



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

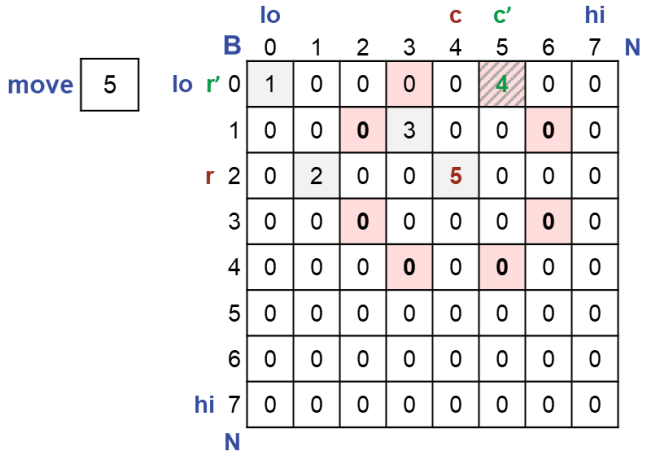
Alternative Representation: Which is better? Or is it "six or half dozen the other"?

(a) We don't know yet that we need **retract**. (b) Won't 2-D output require `int B[][]` anyway?

Representation 1:

Primary: tour recorded in cells of 2-D **`int`** array `B`.

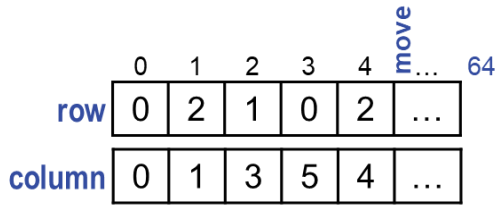
Auxiliary: Variables `row` and `column` to facilitate finding predecessor square, for **Retract**.



Representation 2:

Primary: tour recorded in variables `row` and `column`.

Auxiliary: 2-D **`boolean`** array `B` to facilitate testing whether a square is unvisited, for **Extend**.



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

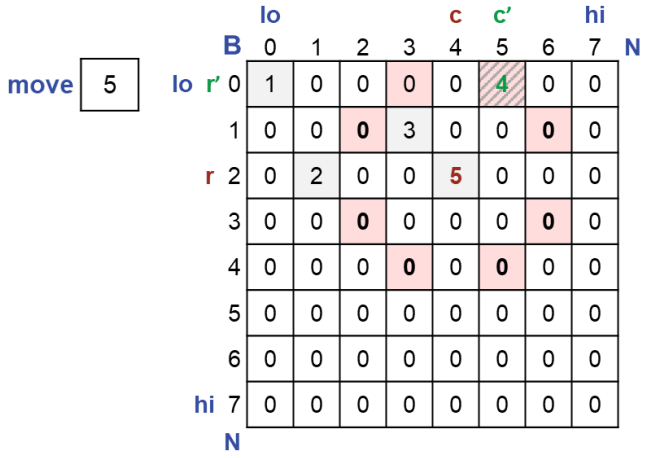
Alternative Representation: Which is better? Or is it "six or half dozen the other"?

(a) We don't know yet that we need **retract**. (b) Won't 2-D output require `int B[][]` anyway?

Representation 1:

Primary: tour recorded in cells of 2-D **`int`** array `B`.

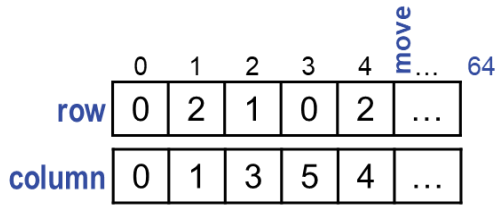
Auxiliary: Variables `row` and `column` to facilitate finding predecessor square, for **Retract**.



Representation 2:

Primary: tour recorded in variables `row` and `column`.

Auxiliary: 2-D **`boolean`** array `B` to facilitate testing whether a square is unvisited, for **Extend**.



Choose Representation 1 (without the auxiliary collections), for now. Revisit later if tour retraction becomes an issue.

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

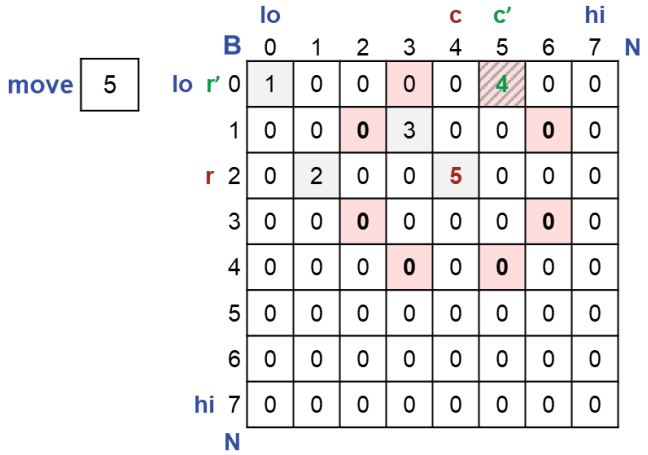
Alternative Representation: Which is better? Or is it "six or half dozen the other"?

(a) We don't know yet that we need **retract**. (b) Won't 2-D output require `int B[][]` anyway?

Representation 1:

Primary: tour recorded in cells of 2-D **`int`** array `B`.

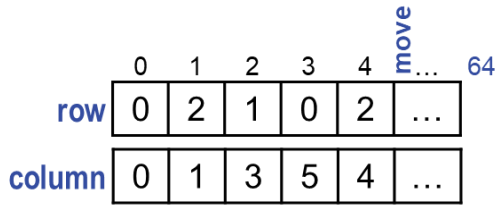
Auxiliary: Variables `row` and `column` to facilitate finding predecessor square, for **Retract**.



Representation 2:

Primary: tour recorded in variables `row` and `column`.

Auxiliary: 2-D **`boolean`** array `B` to facilitate testing whether a square is unvisited, for **Extend**.



Choose Representation 1 (without the auxiliary collections), for now. Revisit later if tour retraction becomes an issue.

☞ **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.**

Define methods:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Establish a tour of length 1. */
   static void Initialize() {
      /* Start a tour with the Knight in the upper-left corner. */
         B[lo][lo] = move;
      } /* Initialize */
   ...
   } /* Tour */
```

Define methods: Row-major order enumeration should be second nature.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Print tour as numbered cells in N-by-N grid of 0s. */
   static void Display() {
      for (int r=lo; r<=hi; r++) {
         for (int c=lo; c<=hi; c++)
            System.out.print(B[r][c] + " ");
         System.out.println();
         }
      } /* Display */
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

Test early and often.

Invoke `Tour.main()`

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Test early and often.

Output:

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

What has been validated?

- Syntactic correctness of overall framework
- Creation of initial data representation
- Correct 2-D output format
- That the 3 methods were actually executed.

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Test early and often.

Output:

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

It's no secret why the tour isn't very long: `Solve` is just a stub.

But if the problem statement is: Write a program that attempts to find a complete Knight's Tour, our program is correct.

It just doesn't try very hard!

- ☞ **Test programs incrementally.**
- ☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**
- ☞ **Validate output thoroughly.**

Let's try a little harder.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      __________
      } /* Solve */
   ...
   } /* Tour */
```

Iterative Refinement: Indeterminate form, because we can't predict when to stop.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
     before
     while ( condition ) each-time
     after
    } /* Solve */
   ...
   } /* Tour */
```

☞ **If you "smell a loop", write it down.**

Standard Pattern: Specialize the loop as an instance of the *general-iteration pattern*.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      /* Initialize. */
      while ( not-finished ) {
         /* Compute. */
         /* Go on to next. */
         }
      } /* Solve */
   ...
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

Refine: Express the *general-iteration pattern* as a journey through an abstract space.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      /* Start-at-the-beginning. */
      while ( not-beyond-the-end ) {
         /* Process-the-current-place. */
         /* Advance-to-the-next-place. */
         }
      } /* Solve */
   ...
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

This pattern is not a good match because, by definition, a cul-de-sac is a place from which there is *no* next place.

Refine: Express the *general-iteration pattern* as a journey through an abstract space.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      /* Start-at-the-beginning. */
      while ( not-beyond-the-end ) {
         /* Process-the-current-place. */
         /* Advance-to-the-next-place. */
         }
      } /* Solve */
   ...
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

Specialize as an alternative version that goes no further than the end.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      /* Start-at-the-beginning. */
      /* Process-the-current-place. */
      while ( not-at-the-end ) {
         /* Determine-a-next-place-to-go-or-at-the-end. */
         if ( not-at-the-end ) {
            /* Advance-to-the-next-place. */
            /* Process-the-current-place. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Processing instructions appear twice in the code, which is a disadvantage.

Specialize as an alternative version that goes no further than the end.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      /* Start-at-the-beginning. */
      /* Process-the-current-place. */
      while ( not-at-the-end ) {
         /* Determine-a-next-place-to-go-or-at-the-end. */
         if ( not-at-the-end ) {
            /* Advance-to-the-next-place. */
            /* Process-the-current-place. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Omit first two lines, which are done by class-variable declarations and `Initialize`.

Specialize for the Knight's Tour.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {

      while ( not-at-the-end ) {
         /* Determine-a-next-place-to-go-or-at-the-end. */
         if ( not-at-the-end ) {
            /* Advance-to-the-next-place. */
            /* Process-the-current-place. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Specialize for the Knight's Tour.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {

      while ( not-at-the-end ) {
         /* Locate an unvisited neighbor, or indicate cul-de-sac. */
         if ( not-at-the-end ) {
            /* Advance-to-the-next-place. */
            /* Process-the-current-place. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Specialize for the Knight's Tour.

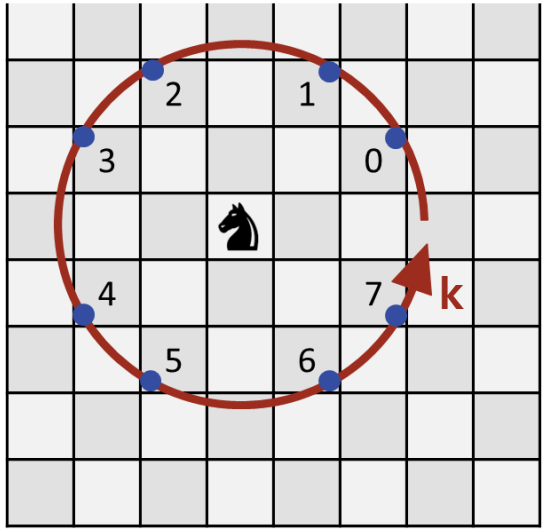
```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {

      while ( not-in-cul-de-sac ) {
         /* Locate an unvisited neighbor, or indicate cul-de-sac. */
         if ( not-in-cul-de-sac ) {
            /* Advance-to-the-next-place. */
            /* Process-the-current-place. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Specialize for the Knight's Tour.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Locate an unvisited neighbor, or indicate cul-de-sac. */
         if ( not-in-cul-de-sac ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Introduce a Coordinate System: Polar-like neighbor numbers, k.



☞ **Invent (or learn) vocabulary for concepts that arise in a problem.**

Refine: Using polar-like neighbor numbers, k.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Refine: Use sequential search pattern to find an unvisited neighbor.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<=maximum) && condition ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

☞ **Master stylized code patterns, and use them.**

Variable k will automatically be set to CUL_DE_SAC on a failed search if we choose CUL_DE_SAC to be 8.

Refine: Use sequential search pattern to find an unvisited neighbor.

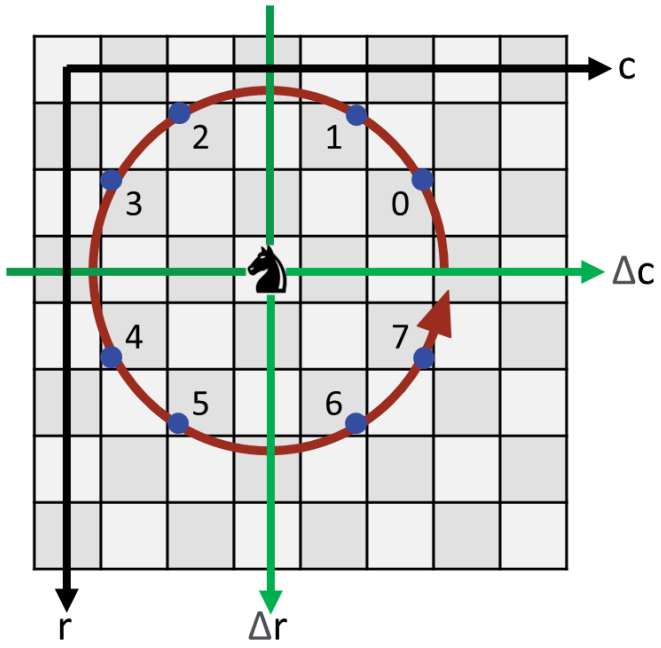
```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && /* neighbor k is visited */ ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Refine: Have faith in the expressive power of the language.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[___________][___________]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

☞ **Beware of unnecessary Case Analysis; hope for code uniformity; avoid code bloat.**

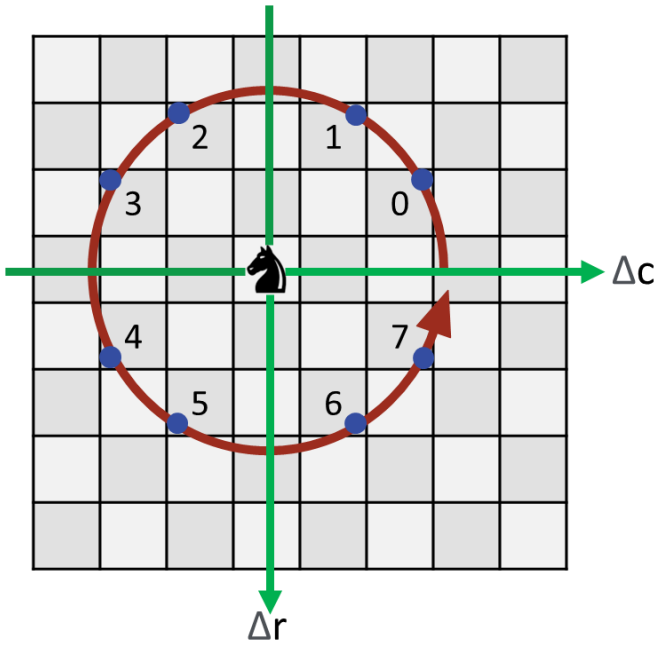
Introduce another Coordinate System: ⟨Δr,Δc⟩



Introduce a local coordinate system ⟨Δr,Δc⟩ with origin at the location of a Knight at ⟨r,c⟩ in the global coordinate system.

If the Knight has a neighbor (●) at ⟨Δr,Δc⟩ in the local system, then that neighbor is at ⟨r+Δr,c+Δc⟩ in the global system.

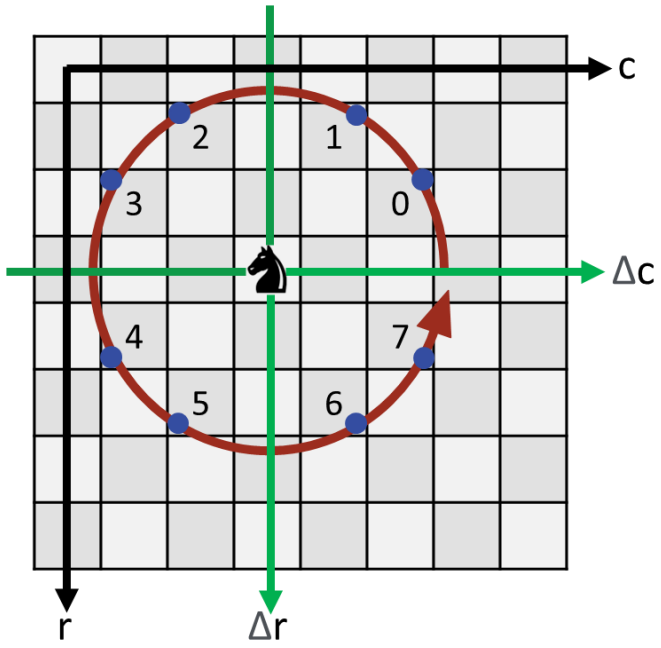
Introduce a Table of Constants: It can obviate an explicit Case Analysis.




```
//                                0   1   2   3   4   5   6   7
static final int deltaR[] = {-1, -2, -2, -1,  1,  2,  2,  1};
static final int deltaC[] = { 2,  1, -1, -2, -2, -1,  1,  2};
```

☞ **Introduce auxiliary data to allow code to be uniform.**

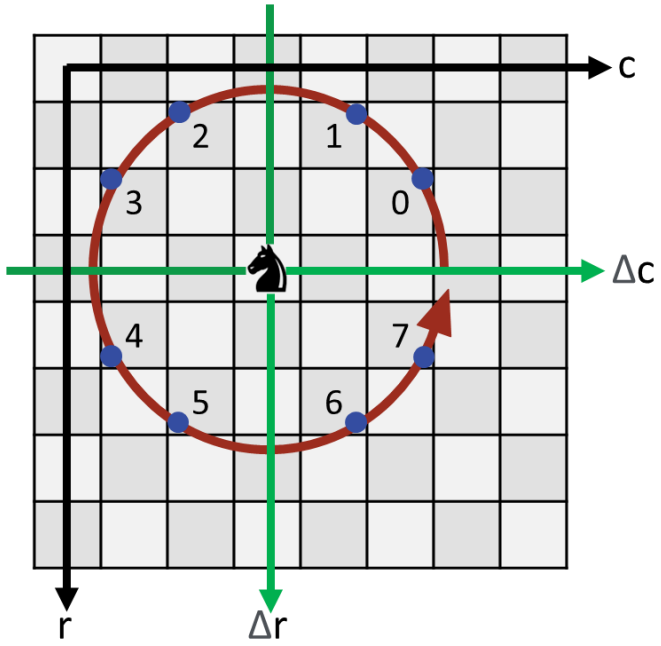
Introduce a Table of Constants: It can obviate an explicit Case Analysis.



If the Knight has a neighbor (●) at ⟨Δr,Δc⟩ in the local system, then that neighbor is at ⟨r+Δr,c+Δc⟩ in the global system.


```
//                                0  1  2  3  4  5  6  7
static final int deltaR[] = {-1, -2, -2, -1,  1,  2,  2,  1};
static final int deltaC[] = { 2,  1, -1, -2, -2, -1,  1,  2};
```

Introduce a Table of Constants: It can obviate an explicit Case Analysis.



If the Knight has a neighbor (●) at ⟨Δr,Δc⟩ in the local system, then that neighbor is at ⟨r+Δr,c+Δc⟩ in the global system.

If the Knight has a neighbor (k) at ⟨deltaR[k],deltaC[k]⟩ in the local system, then that neighbor is at ⟨r+deltaR[k],c+deltaC[k]⟩ in the global system.

```
//                                0   1   2   3   4   5   6   7
static final int deltaR[] = {-1, -2, -2, -1,  1,  2,  2,  1};
static final int deltaC[] = { 2,  1, -1, -2, -2, -1,  1,  2};
```

Refine: Have faith in the expressive power of the language.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Refine: Update tour, referring to its data-representation invariant to know what must change.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

```
/* A Tour of length move is given by elements of B numbered 1 to move.
   Squares numbered consecutively go from ⟨0,0⟩ to ⟨r,c⟩, and
   correspond to legal moves for a Knight. */
   static int r, c;                  // Position of Knight.
   static int move;                  // Length of Tour.
```

Refine: Update tour, referring to its data-representation invariant to know what must change.

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
               r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

```
/* A Tour of length move is given by elements of B numbered 1 to move.
   Squares numbered consecutively go from ⟨0,0⟩ to ⟨r,c⟩, and
   correspond to legal moves for a Knight. */
   static int r, c;                  // Position of Knight.
   static int move;                  // Length of Tour.
```

Termination:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( not-in-cul-de-sac ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
               r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Termination can use failure to find an unvisited neighbor on the previous iteration,

Termination:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( k!=CUL_DE_SAC ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            int k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
               r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Termination can use failure to find an unvisited neighbor on the previous iteration, but we must make sure the loop iterates the first time.

Termination:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      int k = 0;  // Neighbor number not CUL_DE_SAC.
      while ( k!=CUL_DE_SAC  ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
               r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Notice that we have moved the declaration of k outside the loop.

Termination:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      int k = 0;  // Neighbor number not CUL_DE_SAC.
      while ( k!=CUL_DE_SAC  ) {
         /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
            k = 0;
            while ( (k<CUL_DE_SAC) && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
         if ( k != CUL_DE_SAC ) {
            /* Extend the tour to the unvisited neighbor. */
               r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
         }
      } /* Solve */
   ...
   } /* Tour */
```

Auxiliary Constants:

```
/* Knight's Tour. */
class Tour {
   ...
   /* Auxiliary constants. */
      final int[] deltaR = {-1, -2, -2, -1,  1,  2,  2,  1};
      final int[] deltaC = { 2,  1, -1, -2, -2, -1,  1,  2};
      int CUL_DE_SAC = 8;
   ...
   } /* Tour */
```

Incremental Testing: But don't be overeager.

- Hit the execute button now, and you will get a "subscript out of bounds" error.

```
...
/* Let k = # of an unvisited neighbor, or CUL_DE_SAC. */
   k = 0;
   while ( k<CUL_DE_SAC && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
...
```

- You can waste a lot of time debugging things you could have anticipated if you had thought a little more deeply: Some squares have fewer than eight neighbors because they are at the board boundary.

☞ **Boundary conditions. Dead last, but don't forget them.**

Incremental Testing: But don't be overeager.

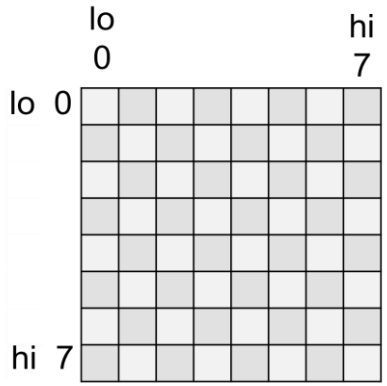
- Hit the execute button now, and you will get a "subscript out of bounds" error.

```
...
/* Let k = # of an unvisited neighbor, or CUL_DE_SAC. */
   k = 0;
   while ( k<CUL_DE_SAC && B[r+deltaR[k]][c+deltaC[k]]!=BLANK ) k++;
...
```

- You can waste a lot of time debugging things you could have anticipated if you had thought a little more deeply: Some squares have fewer than eight neighbors because they are at the board boundary.
- We seek a way to deal with the boundaries without doing major surgery on the code.

☞ **Boundary conditions. Dead last, but don't forget them.**

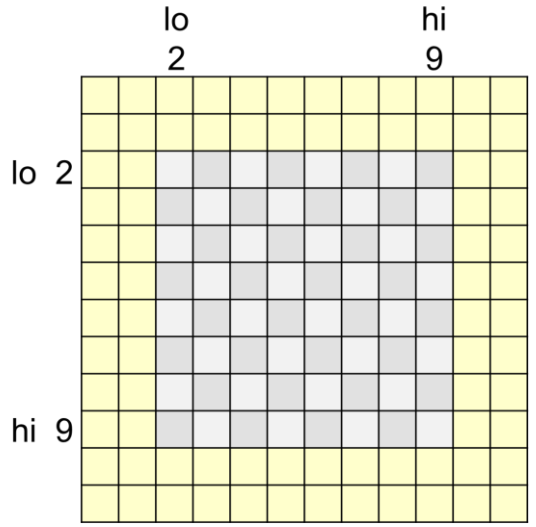
Sentinels to the Rescue: Original representation invariant




```
/* Knight's Tour. */
class Tour {
   /* Chess board B is an N-by-N int array, for N==8. Unvisited squares
      are BLANK, and row and column indices range from lo to hi. */
      static final int N = 8;           // Size of B.
      static int B[][];                 // Chess board, initially 0s.
      static final int BLANK = 0;       // Unvisited square in board.
      static final int lo = 0;          // First row or column index.
      static final int hi = hi+N-1;     // Last row or column index.
   ...
   } /* Tour */
```

☞ **Boundary conditions. Dead last, but don't forget them.**

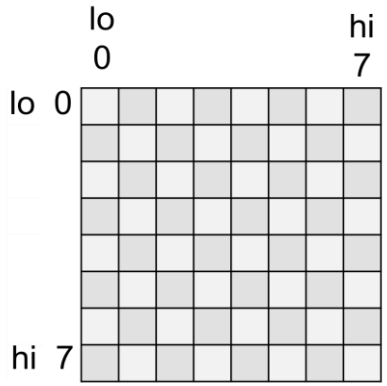
Sentinels to the Rescue: Updated representation invariant




```
/* Knight's Tour. */
class Tour {
   /* Chess board B is an N-by-N int sub-array, for N==8, embedded in a
      2-cell ring of sentinel squares. Unvisited squares are BLANK, and
      row and column indices range from lo to hi. */
      static final int N;              // Size of B.
      static int B[][];                // Chess board, initially 0s.
      static final int BLANK = 8;      // Unvisited square in board.
      static final int lo = 2;         // First row or column index.
      static final int hi = hi+N-1;    // Last row or column index.
   ...
   } /* Tour */
```

☞ **Boundary conditions. Dead last, but don't forget them.**

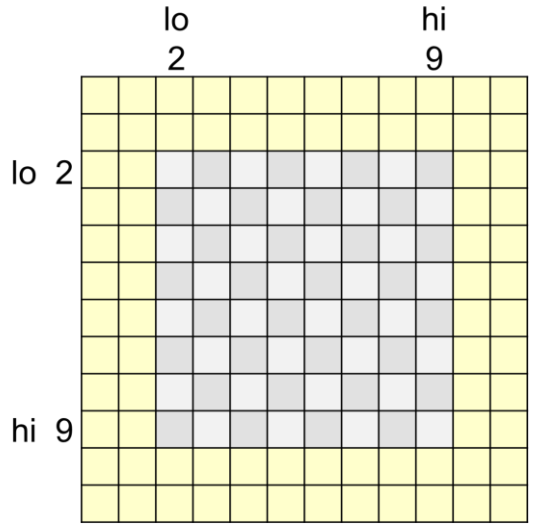
Sentinels to the Rescue: Original Initialize




```
/* Knight's Tour. */
class Tour {
   ...
   /* Initialize: Establish invariant for a tour of length 1. */
   static void Initialize() {
      /* Start a tour with the Knight in the upper-left corner. */
         B[lo][lo] = move;
      } /* Initialize */
   ...
} /* Tour */
```

☞ **Boundary conditions. Dead last, but don't forget them.**

Sentinels to the Rescue: Revised `Initialize`




```
/* Knight's Tour. */
class Tour {
   ...
   /* Initialize: Establish invariant for a tour of length 1. */
   static void Initialize() {
      /* Set B to an (N+4)-by-(N+4) array of all non-BLANK. */
         for (int r=lo-2; r<=hi+2; r++)
            for (int c=lo-2; c<=hi+2; c++)
               B[r][c] = BLANK+1;
      /* Reset inner N-by-N array to all BLANK. */
         for (int r=lo; r<=hi; r++)
            for (int c=lo; c<=hi; c++)
               B[r][c] = BLANK;
      /* Start a tour with the Knight in the upper-left corner. */
         B[lo][lo] = move;
      } /* Initialize */
   ...
} /* Tour */
```

Incremental Testing: Good to go!

Incremental Testing: Good to go!

Output:

```
1 10 23 42 7 4 13 18
24 41 8 3 12 17 6 15
9 2 11 22 5 14 19 32
0 25 40 35 20 31 16 0
0 36 21 0 39 0 33 30
26 0 38 0 34 29 0 0
37 0 0 28 0 0 0 0
0 27 0 0 0 0 0 0
```

What has been validated?

- Syntactic correctness of overall framework
- Creation of initial data representation
- Correct 2-D output format
- Correct search for an unvisited neighbor
- Correct extension of tour to that neighbor
- Correct treatment of boundaries

Unanticipated problem detected

- Ragged output due to variable-length integers

Incremental Testing: Good to go!

Output:

```
1 10 23 42 7 4 13 18
24 41 8 3 12 17 6 15
9 2 11 22 5 14 19 32
0 25 40 35 20 31 16 0
0 36 21 0 39 0 33 30
26 0 38 0 34 29 0 0
37 0 0 28 0 0 0 0
0 27 0 0 0 0 0 0
```

What has been validated?

- Syntactic correctness of overall framework
- Creation of initial data representation
- Correct 2-D output format
- Correct search for an unvisited neighbor
- Correct extension of tour to that neighbor
- Correct treatment of boundaries

Unanticipated problem detected

- Ragged output due to variable-length integers

Not too shabby considering that we just went to an arbitrary unvisited square, an approach called a *greedy algorithm*.

Incremental Testing: Good to go!

Output:

```
1  10 23 42 7  4  13 18
24 41 8  3  12 17 6  15
9  2  11 22 5  14 19 32
0  25 40 35 20 31 16 0
0  36 21 0  39 0  33 30
26 0  38 0  34 29 0  0
37 0  0  28 0  0  0  0
0  27 0  0  0  0  0  0
```

Fix the minor formatting issue by modifying the line:

```
System.out.print(B[r][c] + " ");
```

in method `Output`, as follows:

```
System.out.print( (B[r][c]+"  ").substring(0,3) );
```

Concatenate a blank at the end of the `String` representation of the integer, and then truncate it to 3 characters.

Greedy Selection: A greedy algorithm just picks the first available neighbor.

```
/* Extend the tour, if possible. */
static void Solve() {
   int k = 0; // A neighbor number that is not CUL_DE_SAC.
   while ( k!=CUL_DE_SAC ) {
      /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
         k = 0;
         while ( (k<CUL_DE_SAC) && (B[r+deltaR[k]][c+deltaC[k]]!=BLANK) )
            k++;
      if ( k!=CUL_DE_SAC ) {
         /* Extend the tour to unvisited neighbor k. */
            r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c]=move;
         }
      }
   } /* Solve */
```

Heuristic Selection: A better algorithm picks a favored neighbor, which we optimistically refer to as the "best choice".

```
/* Extend the tour, if possible. */
static void Solve() {
   int k = 0; // A neighbor number that is not CUL_DE_SAC.
   while ( k!=CUL_DE_SAC ) {
      /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
      /* Let bestK be a favored unvisited neighbor, or CUL_DE_SAC if all
         neighbors are already visited. */
      k = bestK;
      if ( k!=CUL_DE_SAC ) {
         /* Extend the tour to unvisited neighbor k. */
            r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c]=move;
         }
      }
   } /* Solve */
```

A *heuristic* is an aid to problem solving that may help.

Heuristic Selection: A better algorithm picks a favored neighbor, which we optimistically refer to as the "best choice".

```
/* Extend the tour, if possible. */
static void Solve() {
   int k = 0; // A neighbor number that is not CUL_DE_SAC.
   while ( k!=CUL_DE_SAC ) {
      /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
      /* Let bestK be a favored unvisited neighbor, or CUL_DE_SAC if all
         neighbors are already visited. */
      k = bestK;
      if ( k!=CUL_DE_SAC ) {
         /* Extend the tour to unvisited neighbor k. */
            r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c]=move;
         }
      }
   } /* Solve */
```

Adapt the pattern from Chapter 7: Find an argument k that minimizes a function's value.

A *heuristic* is an aid to problem solving that may help.

Heuristic Selection: Pick the neighbor that minimizes a score, for some `score` function.

```
/* Extend the tour, if possible. */
static void Solve() {
   int k = 0; // A neighbor number that is not CUL_DE_SAC.
   while ( k!=CUL_DE_SAC ) {
      /* Let k = index of an unvisited neighbor, or CUL_DE_SAC. */
         /* Let bestK be a favored unvisited neighbor, or CUL_DE_SAC if all
            neighbors are already visited. */
            int bestK = CUL_DE_SAC;
            int bestScore = CUL_DE_SAC;
            for (k = 0; k<CUL_DE_SAC; k++) {
               if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) {
                  int s = score(r+deltaR[k], c+deltaC[k]);
                  if (s<bestScore) {bestScore = s; bestK = k; }
                  }
               }
         k = bestK;
      if ( k!=CUL_DE_SAC ) {
         /* Extend the tour to unvisited neighbor k. */
            r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c]=move;
         }
      }
   } /* Solve */
```

Score:

```
/* Return 0. */
int Score(int r, int c) { return 0; }
```

☞ **Write method stubs that allow partial programs to execute.**

Incremental Testing: Good to go!

Output:

```
1  10 23 42 7  4  13 18
24 41 8  3  12 17 6  15
9  2  11 22 5  14 19 32
0  25 40 35 20 31 16 0
0  36 21 0  39 0  33 30
26 0  38 0  34 29 0  0
37 0  0  28 0  0  0  0
0  27 0  0  0  0  0  0
```

What has been validated?

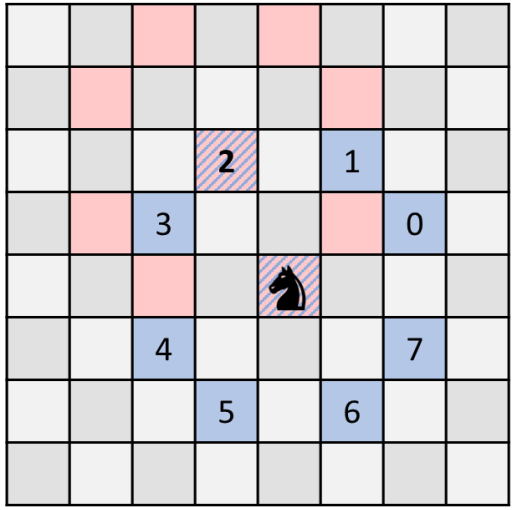
- Syntactic correctness of overall framework
- Creation of initial data representation
- Correct 2-D output format
- Correct search for an unvisited neighbor
- Correct extension of tour to that neighbor
- Correct treatment of boundaries
- Exercising of search for a favored neighbor, albeit still just selects first unvisited neighbor

Same output as before, because any unvisited neighbor has a Score of 0.

A beneficial heuristic: Warnsdorff's Rule.

Go to an unvisited neighbor (blue) that has the fewest unvisited neighbors.

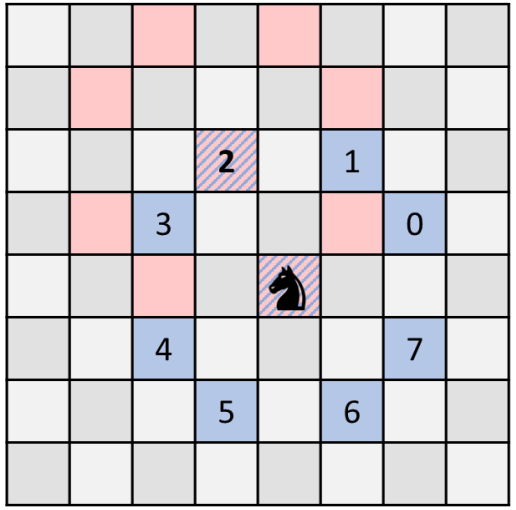
That is, the score of a given neighbor of the Knight (e.g., 2) should be a count of that neighbor's unvisited neighbors (pink).



A beneficial heuristic: Warnsdorff's Rule.

Go to an unvisited neighbor (blue) that has the fewest unvisited neighbors.

That is, the score of a given neighbor of the Knight (e.g., 2) should be a count of that neighbor's unvisited neighbors (pink).



Rationale. Let Knight's neighbor (e.g., 2) have *m* unvisited neighbors (a subset of the pink squares).

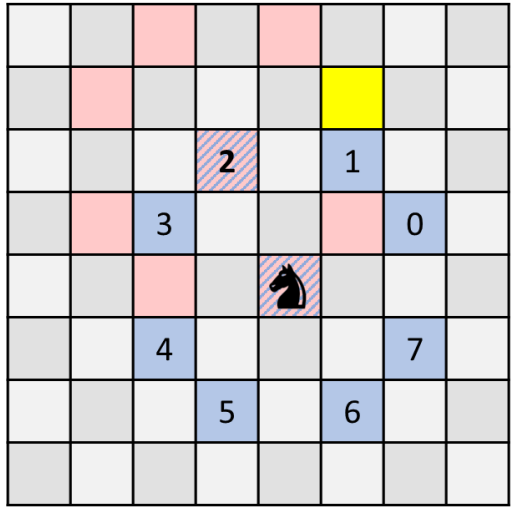
m=0. The Knight's current square is the only way to get to square 2, and if it doesn't go there now, it won't ever get another chance.

Yes, it will then be in a cul-de-sac, so, if we hope for a tour of length 64, this better be the 64th move. If not, the Knight is effectively cutting its losses, and ending a doomed tour. If the goal were to maximize tour length, it would be better not to go there now, unless this is move 64. Warnsdorff 's Rule is "going for broke".

A beneficial heuristic: Warnsdorff's Rule.

Go to an unvisited neighbor (blue) that has the fewest unvisited neighbors.

That is, the score of a given neighbor of the Knight (e.g., 2) should be a count of that neighbor's unvisited neighbors (pink).



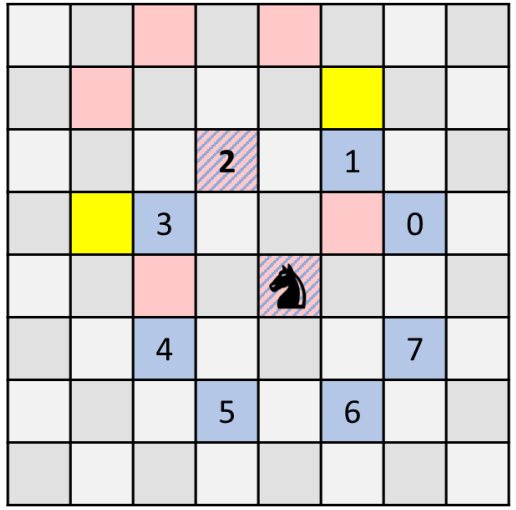
Rationale. Let Knight's neighbor (e.g., 2) have *m* unvisited neighbors (a subset of the pink squares).

m=1. There is only one way out (e.g, the yellow square). If the Knight were to go to square 2 now, then the next move (to yellow) would remove 2 from further concern. But if it doesn't go there now, then when it eventually gets to the yellow square, it will be forced to go to 2, which will then end the tour in a cul-de-sac. So, it is best to pass through 2 now, for otherwise it will loom as a hazard.

A beneficial heuristic: Warnsdorff's Rule.

Go to an unvisited neighbor (blue) that has the fewest unvisited neighbors.

That is, the score of a given neighbor of the Knight (e.g., 2) should be a count of that neighbor's unvisited neighbors (pink).



Rationale. Let Knight's neighbor (e.g., 2) have *m* unvisited neighbors (a subset of the pink squares).

m=2. Too hard to think about. Perhaps the advantages of m=0 and m=2 are good enough to complete a tour.

Score: Replace stub by implementation of Warnsdorff's Rule.

```
/* Return # of unvisited neighbors of ⟨r,c⟩. */
static int Score(int r, int c) {
   int count = 0;  // Number of unvisited neighbors of ⟨r,c⟩ found so far.
   for (int k=0; k<CUL_DE_SAC; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) count++;
   return count;
   }
```

Call site: `int s = Score(r+deltaR[k],c+deltaC[k]);`

Score: Replace stub by implementation of Warnsdorff's Rule.

```
/* Return # of unvisited neighbors of ⟨r,c⟩. */
static int Score(int r, int c) {
   int count = 0;  // Number of unvisited neighbors of ⟨r,c⟩ found so far.
   for (int k=0; k<CUL_DE_SAC; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) count++;
   return count;
   }
```

r and c are class variables that are part of the tour's representation invariant, and are the Knight's current coordinates.

Call site: `int s = Score(r+deltaR[k],c+deltaC[k]);`

Parameters: `Score(int r, int c)`

Score: Replace stub by implementation of Warnsdorff's Rule.

```
/* Return # of unvisited neighbors of ⟨r,c⟩. */
static int Score(int r, int c) {
   int count = 0;  // Number of unvisited neighbors of ⟨r,c⟩ found so far.
   for (int k=0; k<CUL_DE_SAC; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) count++;
   return count;
   }
```

r and c are class variables that are part of the tour's representation invariant, and are the Knight's current coordinates.

r and c are parameters of Score. On each call, they are the coordinates of the Knight's k-th neighbor.

☞ **Avoid gratuitously different names for parameters and variables whose use is essentially the same. Practice conceptual economy.**

Incremental Testing: A complete tour!

Output:

```
1  22 3  18 25 30 13 16
4  19 24 29 14 17 34 31
23 2  21 26 35 32 15 12
20 5  56 49 28 41 36 33
57 50 27 42 61 54 11 40
6  43 60 55 48 39 64 37
51 58 45 8  53 62 47 10
44 7  52 59 46 9  38 63
```

Incremental Testing: A complete tour!

Output:

```
1   22 3   18 25 30 13 16
4   19 24 29 14 17 34 31
23 2   21 26 35 32 15 12
20 5   56 49 28 41 36 33
57 50 27 42 61 54 11 40
6   43 60 55 48 39 64 37
51 58 45 8   53 62 47 10
44 7   52 59 46 9   38 63
```

Neighbor Selection: Monte Carlo algorithm, pick a random neighbor.

```
/* Let k = index of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor numbers of the count
      unvisited neighbors of ⟨r,c⟩. */
   if ( count==0 ) k = CUL_DE_SAC;
   else k = a-random-neighbor-selected-from-unvisited[0:count-1];
```

Neighbor Selection: Monte Carlo algorithm, pick a random neighbor.

```
/* Let k = index of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor numbers of the count
      unvisited neighbors of ⟨r,c⟩. */
      int unvisited[] = new int[CUL_DE_SAC];
      int count = 0;  // # unvisited neighbors
      for (k=0; k<CUL_DE_SAC; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) {
         unvisited[count]=k; count++;
         }
   if ( count==0 ) k = CUL_DE_SAC;
   else k = a-random-neighbor-selected-from-unvisited[0:count-1];
```

Neighbor Selection: Monte Carlo algorithm, pick a random neighbor.

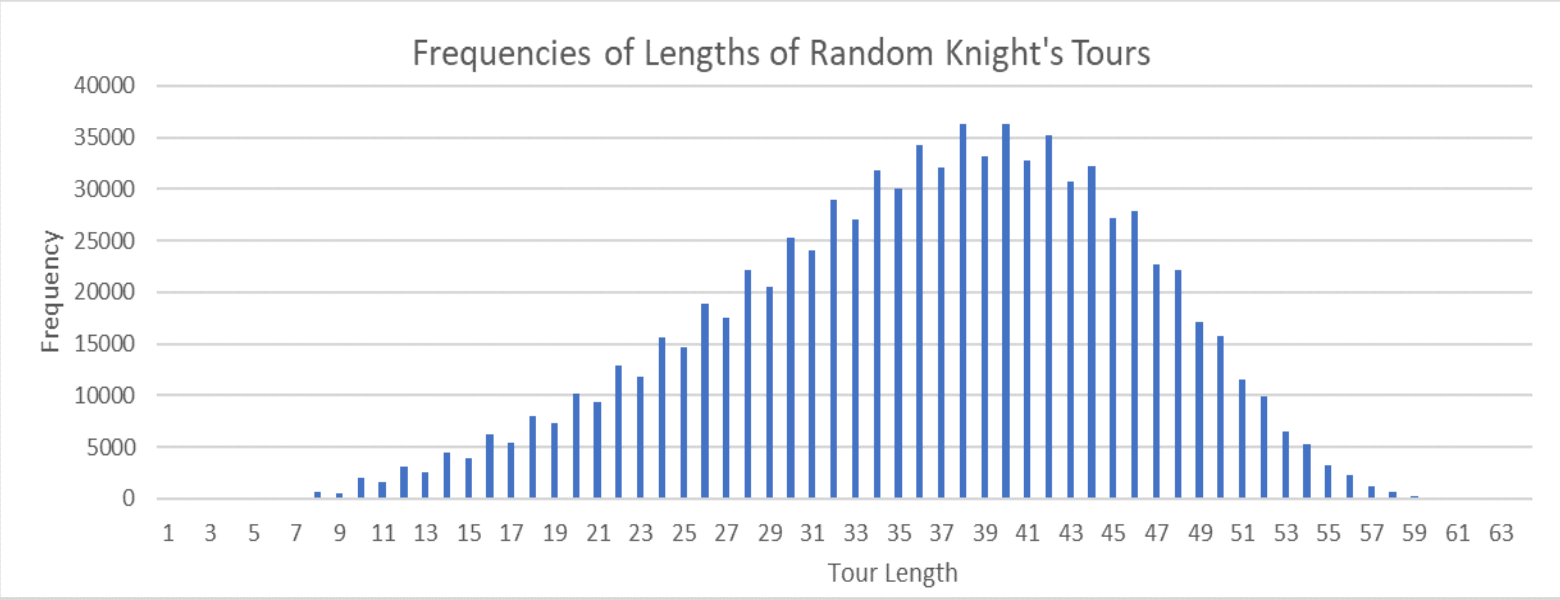
```
/* Let k = index of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor numbers of the count
      unvisited neighbors of ⟨r,c⟩. */
      int unvisited[] = new int[CUL_DE_SAC];
      int count = 0;  // # unvisited neighbors
      for (k=0; k<CUL_DE_SAC; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) {
         unvisited[count]=k; count++;
         }
   if ( count==0 ) k = CUL_DE_SAC;
   else k = unvisited[rand.nextInt(count)];
```

Omitted Details:

Importing of the random library.

A driver that repeatedly invokes the Monte Carlo `solve` until a solution is found.

Instrumentation of the driver to histogram the tour lengths of each trial.



Who could have guessed that a Knight could be so stupid as to get himself into a cul-de-sac in just 8 moves!

Summary:

Many standard precepts, patterns, and established coding techniques have been illustrated.

The importance of data representations and invariants was stressed.

The notions of greedy, heuristic, and Monte Carlo algorithms were introduced.