

# 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.

		X		X			
	X				X		
							
	X				X		
		X		X			

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.

**Where to begin:** *Get your feet wet.*

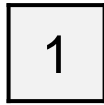
You can start by working the problem by hand, but may find it a bit overwhelming.

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

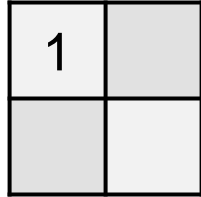
- 
- ☞ **Make sure you understand the problem.**
  - ☞ **Confirm your understanding with concrete examples.**
-

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.



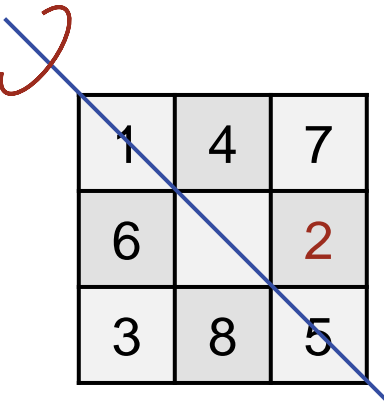
N=2. Impossible, and stuck from the get-go. So, the problem may be not solvable, in general.

1
---

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

1	

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



1	4	7
6		2
3	8	5

1	6	3
4		8
7	2	5

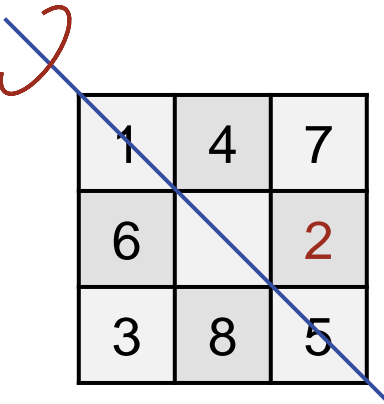
N=3. There is a choice at the begin, but thereafter the (partial) tour is proscribed. However, no tour can reach the central square. Taking symmetry into account, the initial choices were not fundamentally different. Might symmetry play a role?

1
---

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

1	

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



1	4	7
6		2
3	8	5

1	6	3
4		8
7	2	5

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?

1	8	3	
	5	12	9
11	2	7	4
6		10	

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:  
    """Knight's Tour."""  
    _____
```

---

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

---

## Establish a framework

```
class Tour:
    """
    Knight's Tour.
    Find a path for a Knight, from the upper-left square of an 8-by-8 chess board,
    that visits each square exactly once, if possible.

    # Methods.
    main(cls) -> None

    # See also.
    Chapter 14 of Principled Programming.

    """
    _____
```

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

## Establish a framework

```
class Tour:  
    """Knight's Tour ... """  
  
    _____
```

Many IDE editors support folding a region of code into an ellipsis. We will follow that practice here.

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

## Establish a framework

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's 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.
-

## Establish a framework

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's Tour."""
        #.Initialize.
        #.Compute.
        #.Output.
```

A standard pattern.

---

 Master stylized code patterns, and use them.

---

**Representation invariants:** Rely on them in statement comments.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's Tour."""
        #.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.
```

A standard pattern, elaborated for the problem at hand.

---

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

---

**Code structure:** Invoke separate methods to do the work.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's Tour."""
        # Initialize: Establish a tour of length 1.
        Tour.initialize()

        # Compute: Extend the tour, if possible.
        Tour.solve()

        # Output: Print the tour as numbered cells in an N-by-N grid of 0s.
        Tour.display()
```

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



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

**Code structure:** Create stubs for methods

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def initialize(cls) -> None:
        """Initialize: Establish a tour of length 1."""
        pass

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        pass

    @classmethod
    def display(cls) -> None:
        """Output: Print the tour as numbered cells in an N-by-N grid of 0s."""
        pass
```

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



---

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

---



## Test early and often.

Add a temporary output statement to `Tour.main`

```
class Tour:
    """Knight's Tour ..."""
    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's Tour."""
        # Initialize: Establish a tour of length 1.
        Tour.initialize()

        # Compute: Extend the tour, if possible.
        Tour.solve()

        # Output: Print the tour as numbered cells in an N-by-N grid of 0s.
        Tour.display()

        # Temporary output.
        print("done");
```

- 
- 👉 **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()
```

```
class Tour:
    """Knight's Tour ..."""
    @classmethod
    def main(cls) -> None:
        """Output a (possibly partial) Knight's Tour."""
        # Initialize: Establish a tour of length 1.
        Tour.initialize()

        # Compute: Extend the tour, if possible.
        Tour.solve()

        # Output: Print the tour as numbered cells in an N-by-N grid of 0s.
        Tour.display()

        # Temporary output.
        print("done");
```

- 
- 👉 **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

What has not been validated?

- That the 3 methods could be executed.

---

👉 **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:

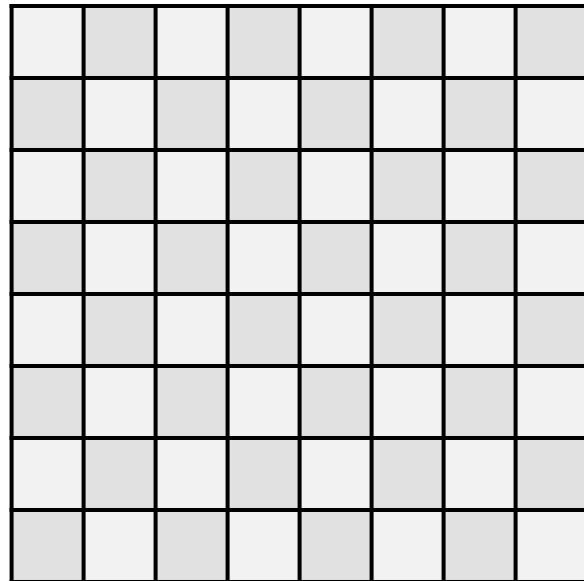
We need representations of the **board** and a (partial) **tour**.



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

---

**Board Representation 1:** The 2-D physical board can be modeled directly in a 2-D array.



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

**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>B</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	

- 
- 👉 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>lo</b>						<b>hi</b>		
<b>B</b>		0	1	2	3	4	5	6	7	<b>N</b>
<b>lo</b>	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	
<b>hi</b>	7	0	0	0	0	0	0	0	0	
<b>N</b>										

**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	

BLANK	<input type="text" value="0"/>
move	<input type="text" value="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 a specification.


		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	<input type="text" value="0"/>
move	<input type="text" value="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.**

```
class Tour:
    """Knight's 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.
    N: int = 8          # N is the size of chess board.
    BLANK: int = 0      # BLANK signifies a square not on the current path.
    lo: int = 2         # lo is the row and column index of the upper-left square.
    hi: int = lo + N - 1 # hi is the row and column index of the lower-right square.
    B: list[list[int]] # B[][] is the chess board.

    ...
```



**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.

```
class Tour:
    """Knight's Tour ..."""

    # A tour of length move is given by squares of B numbered 1 to move.
    # Squares numbered consecutively go from (lo,lo) to (r,c), and
    # correspond to legal moves for a Knight.
    r: int          # r is the Knight's row coordinate.
    c: int          # c is the Knight's column coordinate.
    move: int       # move is the length of the current 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 64<sup>th</sup> 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 64<sup>th</sup> 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 64<sup>th</sup> 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 64<sup>th</sup> 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  $\langle r,c \rangle$  to  $\langle r',c' \rangle$ , 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 64<sup>th</sup> 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  $\langle r,c \rangle$  to  $\langle r',c' \rangle$ , increment move, and store move in  $B[r'][c']$ .
- **Retract** the tour, if the strategy calls for backtracking.
  - To undo previous **extend**, locate previous square  $\langle r',c' \rangle$ , set  $\langle r,c \rangle$  to  $\langle r',c' \rangle$ , 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 previous **extend**, locate previous square  $\langle r', c' \rangle$ , set  $\langle r, c \rangle$  to  $\langle r', c' \rangle$ , and decrement move.

For Representation 1, a **search** would be required to find  $\langle r', c' \rangle$ .

move 5

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

Such a **search** would inspect the eight neighbors of  $\langle r, c \rangle$  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: Address a shortcoming of Representation 1.

- **Retract** the tour, if the strategy calls for backtracking.
  - To undo previous **extend**, locate previous square  $\langle r', c' \rangle$ , set  $\langle r, c \rangle$  to  $\langle r', c' \rangle$ , and decrement move.

For Representation 1, a **search** would be required to find  $\langle r', c' \rangle$ .

move 5

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

row

	0	1	2	3	4	...	64
row	0	2	1	0	2	...	

column

	0	1	2	3	4	...
column	0	1	3	5	4	...

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.

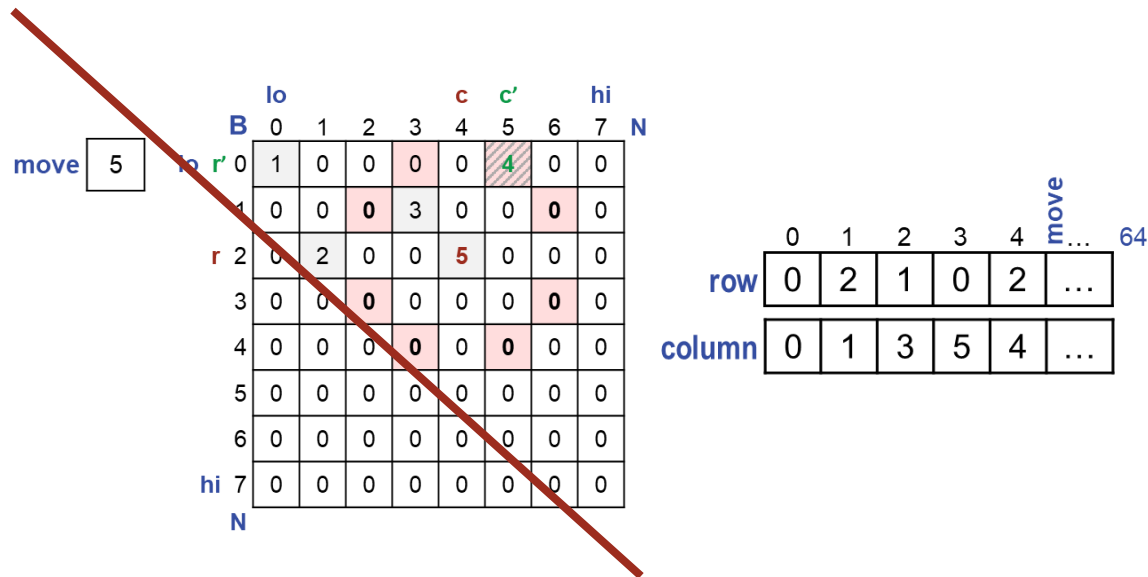
---

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?




---

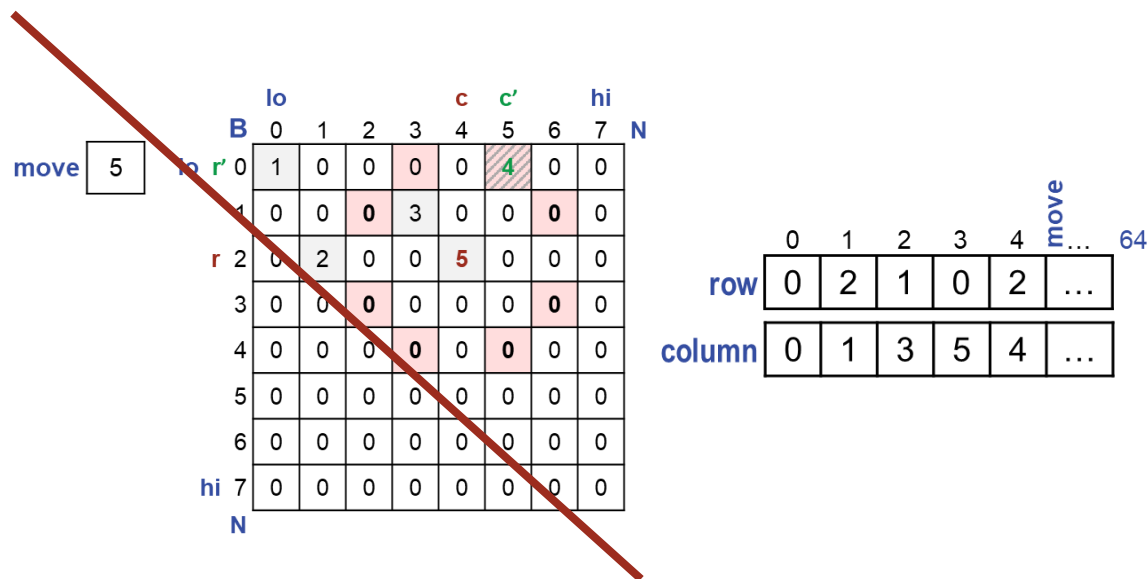
 The touchstone of a data representation is its utility in performing the needed operations.

---

**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  $\langle r', c' \rangle$  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.**

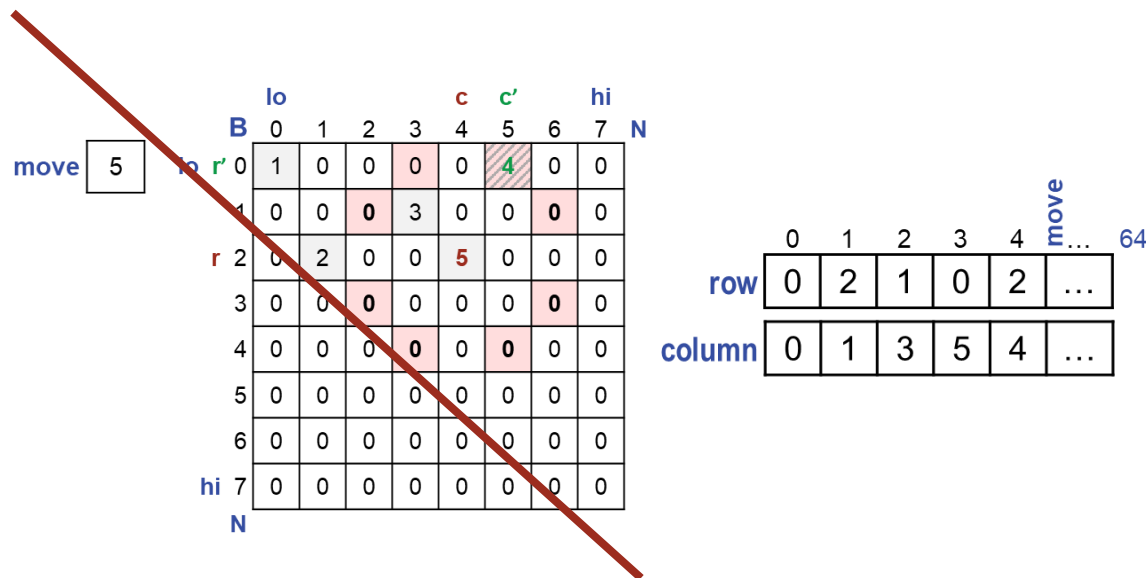
---



**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  $\langle r', c' \rangle$  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 **bool** array B indicating “visited” would obviate a search.



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

## Alternative Representation: Which is better? Or is it “six of one, 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**.

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

### Representation 2:

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

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

		0	1	2	3	4	...	64
row		0	2	1	0	2	...	
column		0	1	3	5	4	...	



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

**Alternative Representation:** Which is better? Or is it “six of one, 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**.

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

### Representation 2:

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

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

	0	1	2	3	4	...	64
row	0	2	1	0	2	...	
column	0	1	3	5	4	...	



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

**Alternative Representation:** Which is better? Or is it “six of one, 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**.

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

### Representation 2:

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

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

		0	1	2	3	4	...	64
row		0	2	1	0	2	...	
column		0	1	3	5	4	...	

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 of one, 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**.

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

### Representation 2:

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

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

	0	1	2	3	4	...	64
row	0	2	1	0	2	...	
column	0	1	3	5	4	...	

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.**

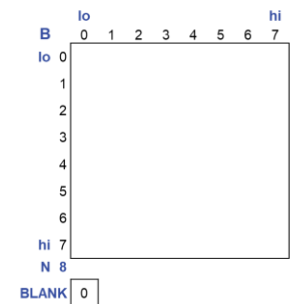
---

## Create initial data representation:

```
class Tour:
    """Knight's 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.
    N: int          # N is the size of chess board.
    BLANK: int      # BLANK signifies a square not on the current path.
    lo: int         # lo is the row and column index of the upper-left square.
    hi: int         # hi is the row and column index of the lower-right square.
    B: list[list[int]] # B[][] is the chess board.

    ...
```



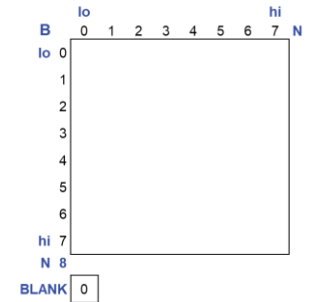
**A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

## Create initial data representation:

```
class Tour:
    """Knight's 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.
    N: int          # N is the size of chess board.
    BLANK: int      # BLANK signifies a square not on the current path.
    lo: int         # lo is the row and column index of the upper-left square.
    hi: int         # hi is the row and column index of the lower-right square.
    B: list[list[int]] # B[][] is the chess board.

    ...
```



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



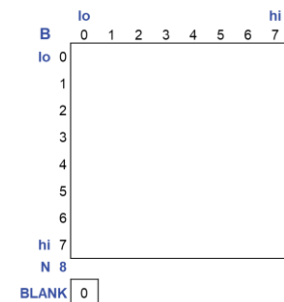
**A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

## Create initial data representation

```
class Tour:
    """Knight's 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.
    N: int = 8           # N is the size of chess board.
    BLANK: int = 0      # BLANK signifies a square not on the current path.
    lo: int = 0         # lo is the row and column index of the upper-left square.
    hi: int = lo + N - 1 # hi is the row and column index of the lower-right square.
    B: list[list[int]] # B[][] is the chess board.

    ...
```



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



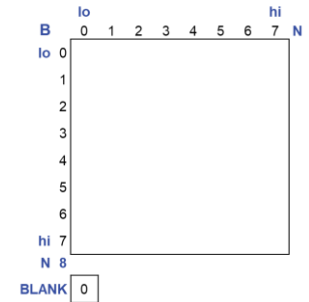
**A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**



## Create initial data representation:

```
class Tour:
    """Knight's 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.
    N: int = 8          # N is the size of chess board.
    BLANK: int = 0      # BLANK signifies a square not on the current path.
    lo: int = 0         # lo is the row and column index of the upper-left square.
    hi: int = lo + N - 1 # hi is the row and column index of the lower-right square.
    B: list[list[int]] = [] # B[][] is the chess board. Initialization to [] is a
                           # violation of the representation invariant because N is 8
                           # and B is empty; this is corrected in method initialize.
```



The initial value of a 2-D array must be created in method initialize because the needed construct (list comprehension) is not permitted here.



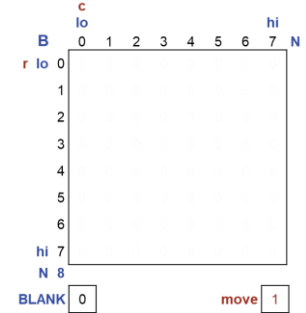
**A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

## Create initial data representation:

```
class Tour:
    """Knight's Tour ..."""

    # A tour of length move is given by squares of B numbered 1 to move. Squares numbered
    # consecutively go from (lo,lo) to (r,c), and correspond to legal moves for a Knight.
    r: int          # r is the Knight's row coordinate.
    c: int          # c is the Knight's column coordinate.
    move: int       # move is the length of the current tour.

    ...
```



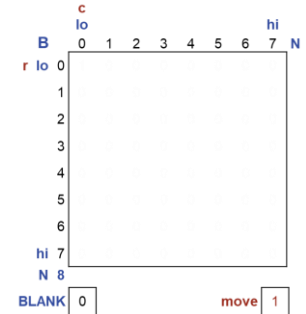
**A representation invariant describes the value(s) of one or more program variables, and their relationships to one another as the program runs. The invariant is typically written as a comment associated with the declaration(s) of the relevant variable(s).**

## Create initial data representation

```
class Tour:
    """Knight's Tour ..."""

    # A tour of length move is given by squares of B numbered 1 to move. Squares numbered
    # consecutively go from (lo,lo) to (r,c), and correspond to legal moves for a Knight.
    r: int = lo          # r is the Knight's row coordinate.
    c: int = lo          # c is the Knight's column coordinate.
    move: int = 1       # move is the length of the current tour.

    ...
```



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



**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).**

## Create initial data representation:

```

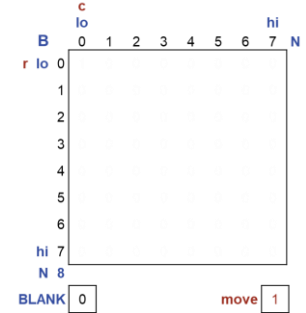
class Tour:
    """Knight's Tour ..."""

    # A tour of length move is given by squares of B numbered 1 to move. Squares numbered
    # consecutively go from (lo,lo) to (r,c), and correspond to legal moves for a Knight.
    r: int = lo           # r is the Knight's row coordinate.
    c: int = lo           # c is the Knight's column coordinate.
    move: int = 1        # move is the length of the current tour.
    # B[r][c] = move     # The Knight is initially in the upper-left square.
                        # Initialization of B[r][c] to move is deferred until method
                        # initialize because B itself is not initialized until then.

    ...

```

Commented out because B is not yet initialized.



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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def display(cls) -> None:
        """Output: Print the tour as numbered cells in an N-by-N grid of 0s."""
        for r in range(Tour.lo, Tour.hi + 1):
            for c in range(Tour.lo, Tour.hi + 1):
                print(Tour.B[r][c], " ", end='')
            print()

    ...
```

**Test early and often.**

```
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 the problem statement is: Write a program that **attempts** to find a complete Knight's Tour, so our program is technically correct.

**We just didn'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.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        pass
```

Let's try a little harder.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
```

---

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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""

        while _____:
            _____
```

---

 If you “smell a loop”, write it down.

---

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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        #.Initialize.
        while not-finished:
            #.Compute.
            #.Go-on-to-next.
```

**Refine:** Specialize the *general-iteration* pattern for an itinerary.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        #.Start at the beginning.
        while not-beyond-the-end:
            #.Process the current place.
            #.Advance to the next place or beyond-the-end.
```

**Refine:** Specialize the itinerary for the Knight's Tour'.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        #.Start at the beginning.
        while not-beyond-the-end:
            #.Process the current place.
            #.Advance to the next place or beyond-the-end.
```

place is the Knight's coordinates (r,c) in the chess board.

---

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

---



**Refine:** Specialize the itinerary for the Knight's Tour.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        #.Start at the beginning.
        while not-in-cul-de-sac:
            #.Process the current place.
            #.Advance to the next place or discover a cul-de-sac.
```

*beyond-the-end* is when we are stuck in a cul-de-sac.

---

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

---

**Refine:** Specialize the itinerary for the Knight's Tour.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        #.Start at the beginning.
        while not-in-cul-de-sac:
            #.Extend the tour an additional square, if possible.
```

Combine "Process" and "Advance" into "Extend".

---

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

---

Recall the context: The data representation invariant.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            #.Extend the tour an additional square, if possible.
```

Drop “Start at the beginning” because it was already done by establishing the data representation invariant.

---

👉 Master stylized code patterns, and use them.

---

**Standard Pattern:** Deploy the *search-use* pattern in extending the tour.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            #.Search.
            #.Use
```

**Refine:** Specialize the *search-use* pattern for the case in hand.

```
class Tour:
    """Knight's Tour ..."""

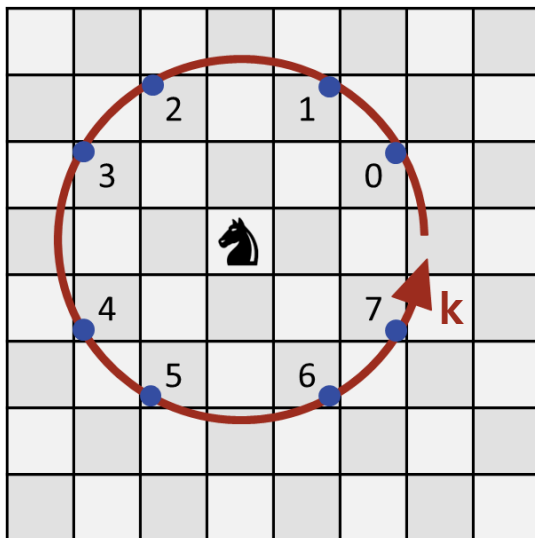
    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            #.Locate an unvisited neighbor, or indicate cul-de-sac.
            #.Use
```

**Refine:** Specialize the *search-use* pattern for the case in hand.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            #.Locate an unvisited neighbor, or indicate cul-de-sac.
            if not-in-cul-de-sac:
                #.Extend the tour to the unvisited neighbor.
```

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



---

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

---

**Refine:** Adopt the terminology of the introduced coordinate system.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            #.Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

---

 **Write comments as an integral part of the coding process, not as afterthoughts.**

---



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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while (k <= maximum) and condition:
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

---

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

---

**Refine:** **Instantiate** the *sequential-search* pattern to find an unvisited neighbor.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while (k < CUL_DE_SAC) and neighbor-k-already-visited:
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

---

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

---

**Uniformity:** Have faith in the expressive power of the programming language.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while (k < CUL_DE_SAC) and (Tour.B[___][___] != Tour.BLANK):
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

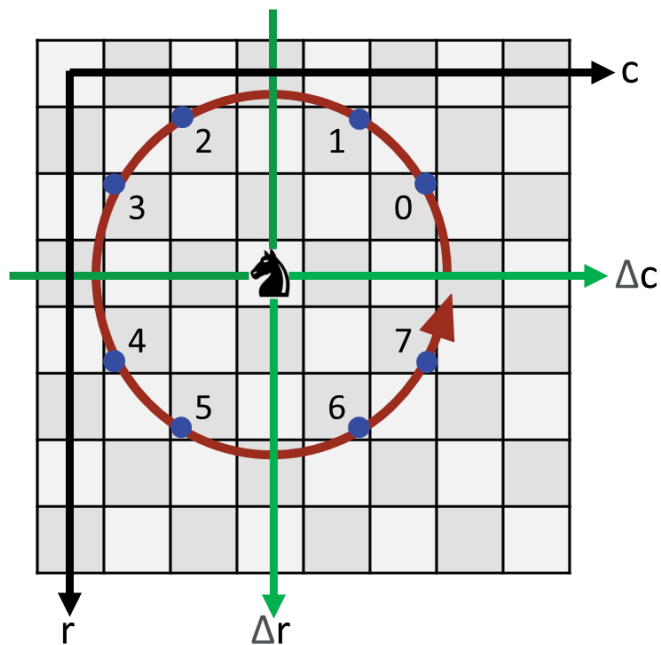
Hope to find uniform subscript expressions in terms of k.

---

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

---

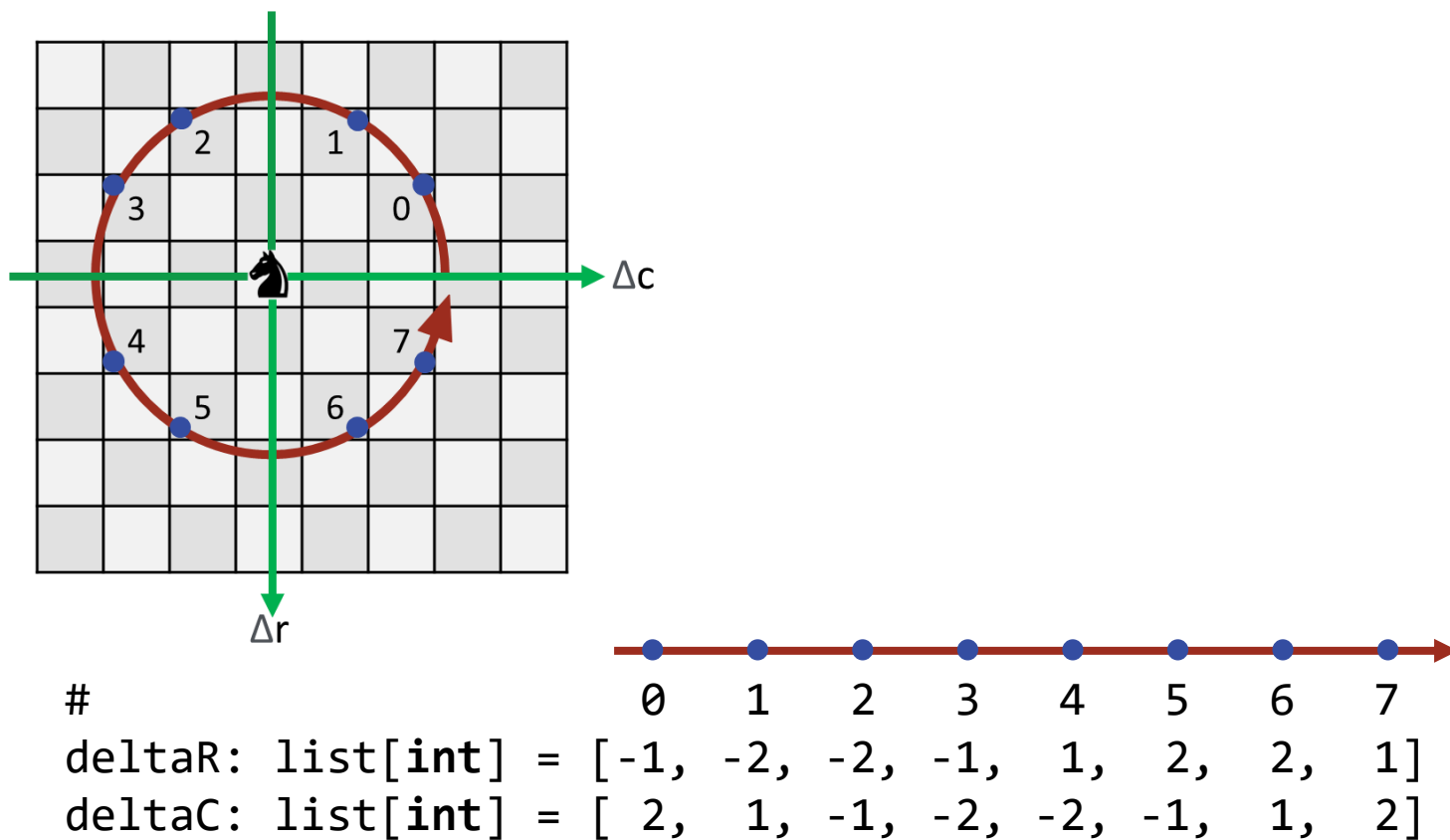
## Introduce another Coordinate System: $\langle \Delta r, \Delta c \rangle$



Introduce a **local** coordinate system  $\langle \Delta r, \Delta c \rangle$  with origin at the location of a Knight at  $\langle r, c \rangle$  in the global coordinate system.

If the Knight has a neighbor (●) at  $\langle \Delta r, \Delta c \rangle$  in the **local** system, then that neighbor is at  $\langle r + \Delta r, c + \Delta c \rangle$  in the global system.

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

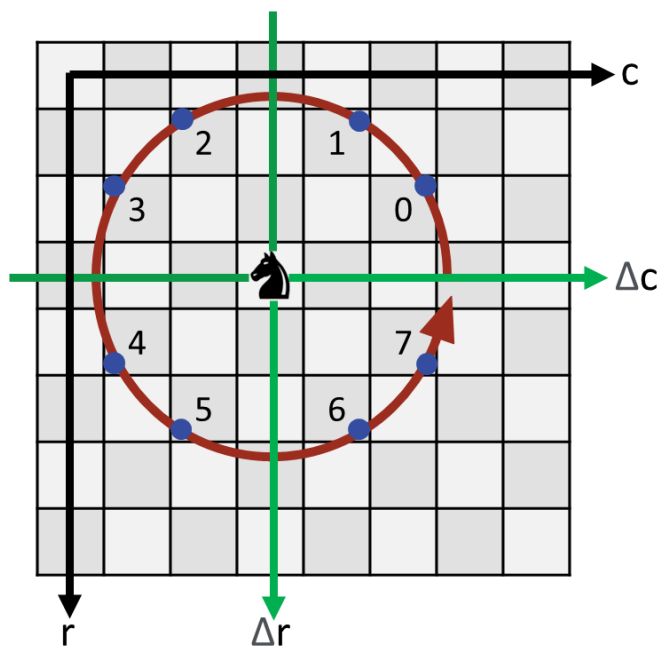



---

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

---

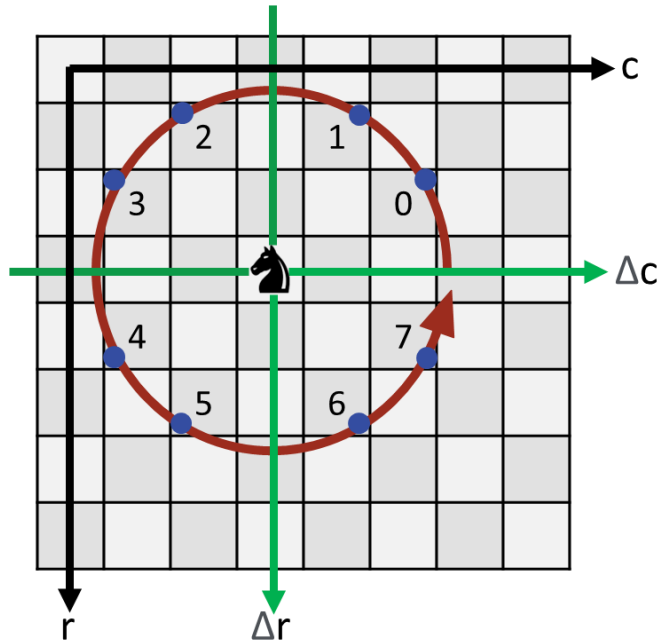
**Introduce a Table of Constants:** It can obviate an explicit Case Analysis.



If the Knight has a neighbor (●) at  $\langle \Delta r, \Delta c \rangle$  in the **local** system, then that neighbor is at  $\langle r + \Delta r, c + \Delta c \rangle$  in the **global** system.

#	0	1	2	3	4	5	6	7
deltaR:	-1	-2	-2	-1	1	2	2	1
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  $\langle \Delta r, \Delta c \rangle$  in the **local** system, then that neighbor is at  $\langle r + \Delta r, c + \Delta c \rangle$  in the global system.

If the Knight has a neighbor ( $k$ ) at  $\langle \text{deltaR}[k], \text{deltaC}[k] \rangle$  in the **local** system, then that neighbor is at  $\langle r + \text{deltaR}[k], c + \text{deltaC}[k] \rangle$  in the global system.

#	0	1	2	3	4	5	6	7
deltaR: list[int]	[-1,	-2,	-2,	-1,	1,	2,	2,	1]
deltaC: list[int]	[ 2,	1,	-1,	-2,	-2,	-1,	1,	2]

**Uniformity:** Have faith in the expressive power of the programming language.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                    (Tour.B[_____][_____] != Tour.BLANK)):
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```



**Uniformity:** Have faith in the expressive power of the programming language.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                    (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK))
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                   (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)):
                k += 1

            if k != CUL_DE_SAC:
                #.Extend the tour to the unvisited neighbor.
```

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

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        while not-in-cul-de-sac:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                   (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK))
                k += 1

            if k != CUL_DE_SAC:
                # Extend the tour to the unvisited neighbor.
                Tour.r += deltaR[k]; Tour.c += deltaC[k];
                Tour.move += 1; Tour.B[Tour.r][Tour.c] = Tour.move
```

**Termination:**

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

```

class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        k = 0 # A neighbor number that is not CUL_DE_SAC.
        while k != CUL_DE_SAC:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                   (Tour.B[Tour.r + deltaR[k]][Tour.c + deltaC[k]] != Tour.BLANK)):
                k += 1

            if k != CUL_DE_SAC:
                # Extend the tour to the unvisited neighbor.
                Tour.r += deltaR[k]; Tour.c += deltaC[k];
                Tour.move += 1; Tour.B[Tour.r][Tour.c] = Tour.move

```

## Local Auxiliary Constants:

```

class Tour:
    """Knight's Tour ..."""

    @classmethod
    def solve(cls) -> None:
        """Compute: Extend the tour, if possible."""
        # Row and column offsets for eight neighbors.
        deltaR: list[int] = [-1, -2, -2, -1, 1, 2, 2, 1]
        deltaC: list[int] = [ 2, 1, -1, -2, -2, -1, 1, 2]
        CUL_DE_SAC: int = 8      # Not a neighbor.

        k = 0    # A neighbor number that is not CUL_DE_SAC.
        while k != CUL_DE_SAC:
            # Extend the tour an additional square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                   (Tour.B[Tour.r + deltaR[k]][Tour.c + deltaC[k]] != Tour.BLANK)):
                k += 1

            if k != CUL_DE_SAC:
                # Extend the tour to the unvisited neighbor.
                Tour.r += deltaR[k]; Tour.c += deltaC[k]
                Tour.move += 1; Tour.B[Tour.r][Tour.c] = Tour.move

```



Declare variables with as small a scope as possible.

## Incremental Testing: But don't be overeager.

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

```
...
# Let k = # # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
k = 0
while ((k < CUL_DE_SAC) and
       (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)): k += 1
...
```

- 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.

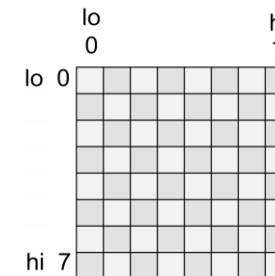
**Incremental Testing:** But don't be overeager.

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

```
...
# Let k = # # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
k = 0
while ((k < CUL_DE_SAC) and
      (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)): k += 1
...
```

- 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.

## Sentinels to the Rescue: Original representation invariant.



```
class Tour:
    """Knight's 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.
    N: int = 8          # N is the size of chess board.
    BLANK: int = 0      # BLANK signifies a square not on the current path.
    lo: int = 0         # lo is the row and column index of the upper-left square.
    hi: int = lo + N - 1 # hi is the row and column index of the lower-right square.
    B: list[list[int]] = [] # B[][] is the chess board. Initialization to [] is a
                            # violation of the representation invariant because N is 8
                            # and B is empty; this is corrected in method initialize.

    ...
```

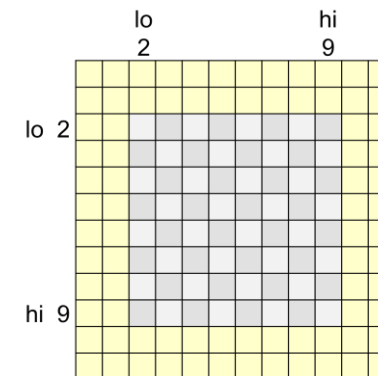


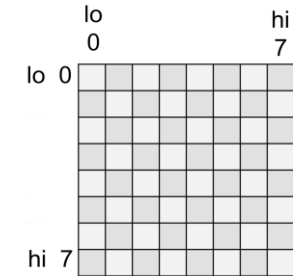
Sentinels to the Rescue: Revised representation invariant.

```
class Tour:
    """Knight's Tour ..."""

    # Chess board B is an N-by-N int array, for N=8, embedded in a 2-cell
    # ring of non-BLANK sentinel squares. Unvisited squares are BLANK,
    # and row and column indices range from lo to hi.
    N: int = 8          # N is the size of chess board.
    BLANK: int = 0      # BLANK signifies a square not on the current path.
    lo: int = 2         # lo is the row and column index of the upper-left square.
    hi: int = lo + N - 1 # hi is the row and column index of the lower-right square.
    B: list[list[int]] = [] # B[][] is the chess board. Initialization to [] is a
    # violation of the representation invariant because N is 8
    # and B is empty; this is corrected in method initialize.

    ...
```





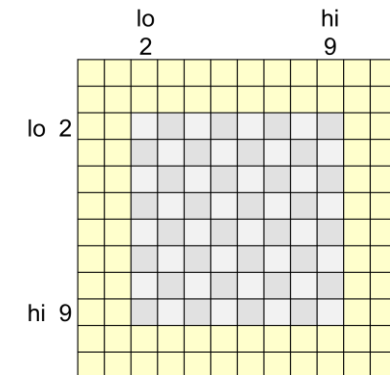
**Sentinels to the Rescue:** Original definition of method `initialize`.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def initialize(cls) -> None:
        """Initialize: Establish a tour of length 1."""

        # Complete the representations of the board and a tour of length 1 that could
        # not be done within the declarations of the class variables:
        # -----
        # Set B to an N-by-N array of all BLANK, with the Knight in the upper-left corner.
        Tour.B = [[Tour.BLANK for _ in range(0, Tour.N)] for _ in range(0, Tour.N)]

        Tour.B[Tour.lo][Tour.lo] = Tour.move
```



## Sentinels to the Rescue: Revised definition of method initialize.

```
class Tour:
    """Knight's Tour ..."""

    @classmethod
    def initialize(cls) -> None:
        """Initialize: Establish a tour of length 1."""

        # Complete the representations of the board and a tour of length 1 that could not be
        # done within the declarations of the class variables:
        # -----
        # Set B to an N-by-N array of all BLANK, with the Knight in the upper-left corner,
        # embedded in a 2-cell ring of non-BLANK sentinel squares.
        # ~~~~~
        # Set B to an (N+4)-by-(N+4) array of all non-BLANK.
        Tour.B = [[Tour.BLANK + 1 for _ in range(0, Tour.N+4)] for _ in range(0, Tour.N+4)]

        # Reset inner N-by-N array to all BLANK.
        for r in range(Tour.lo, Tour.hi + 1):
            for c in range(Tour.lo, Tour.hi + 1):
                Tour.B[r][c] = Tour.BLANK

        Tour.B[Tour.lo][Tour.lo] = Tour.move
```

**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:** Fix up and retest.

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:

```
print(Tour.B[r][c], " ", end='')
```

in method Output, as follows:

```
print((str(Tour.B[r][c]) + " ")[0:3], end='')
```

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

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

```
class Tour:
    """Knight's Tour ..."""

    # Compute: Extend the tour, if possible.
    @classmethod
    def solve(cls) -> None:
        ...
        k = 0 # A neighbor number that is not CUL_DE_SAC.
        while k != CUL_DE_SAC:
            # Extend the tour 1 square, if possible.
            # -----
            # Let k = index of an unvisited neighbor, or CUL_DE_SAC.
            k = 0
            while ((k < CUL_DE_SAC) and
                    (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)):
                k += 1

            if k != CUL_DE_SAC:
                # Extend the tour to the unvisited neighbor.
                Tour.r += deltaR[k]; Tour.c += deltaC[k];
                Tour.move += 1; Tour.B[Tour.r][Tour.c] = Tour.move
```



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

```
# Let k = index of an unvisited neighbor, or CUL_DE_SAC.  
k = 0  
while ((k < CUL_DE_SAC) and  
       (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)):  
    k += 1
```

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

```
# Let k = index of an unvisited neighbor, or CUL_DE_SAC.
k = 0
while ((k < CUL_DE_SAC) and
      (Tour.B[r + deltaR[k]][c + deltaC[k]] != Tour.BLANK)):
    k += 1
```

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

```
# 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;
```

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

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

```
# 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.
bestK = CUL_DE_SAC          # Index of favored neighbor.
best_score = CUL_DE_SAC + 1 # Score of favored neighbor.
for k in range(0, CUL_DE_SAC):
    if Tour.B[Tour.r + deltaR[k]][Tour.c + deltaC[k]] == Tour.BLANK:
        s = score(Tour.r + deltaR[k], Tour.c + deltaC[k])
        if s < best_score:
            bestK = k
            best_score = s

k = bestK
```

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

---

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

---

**score:**

```
def score(r: int, c: int) -> int:  
    """Return 0."""  
    return 0
```

N.B. `score` is auxiliary to method `solve`, and can be defined therein. This will have a benefit (later) when we want to access the auxiliary constants of `solve`. For a language that does not support such nested definitions, both `score` and the auxiliary constants would have to be defined at the top level of the class.

---

 **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?

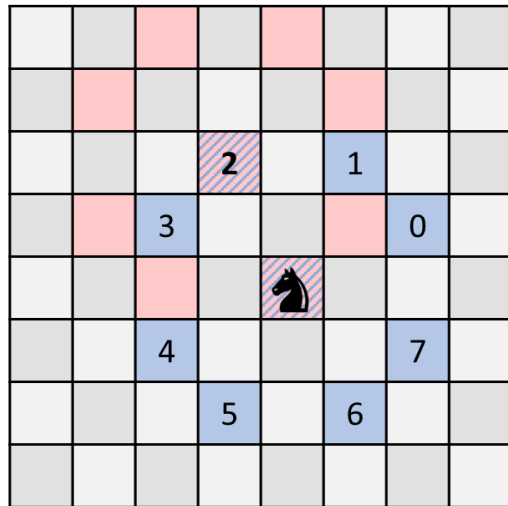
- Syntactic correctness of overall framework
- Creation of initial data representation
- Correct 2-D output format
- Correct search for an unvisited neighbor
- Correct extension of tour to that neighbor
- Correct treatment of boundaries
- Exercising of search for a favored neighbor, albeit still just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

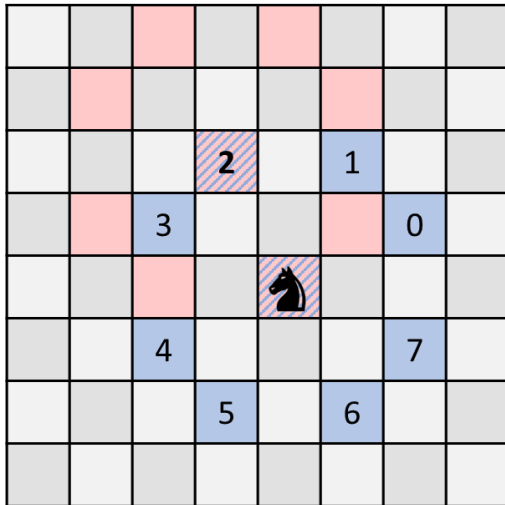
That is, the score of a given neighbor of the Knight (e.g., 2) should be a count of that neighbor's unvisited neighbors (pink).



## 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).

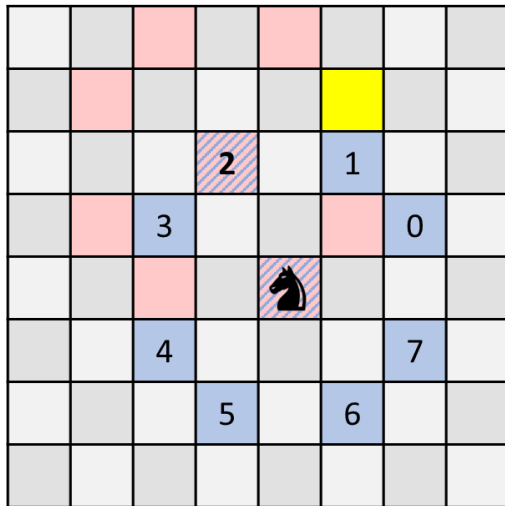
**$m=0$ .** The Knight's current square is the only way to get to square 2, and if it doesn't go there now, it won't ever get another chance.

Yes, it will then be in a cul-de-sac, so, if we hope for a tour of length 64, this better be the 64<sup>th</sup> 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).

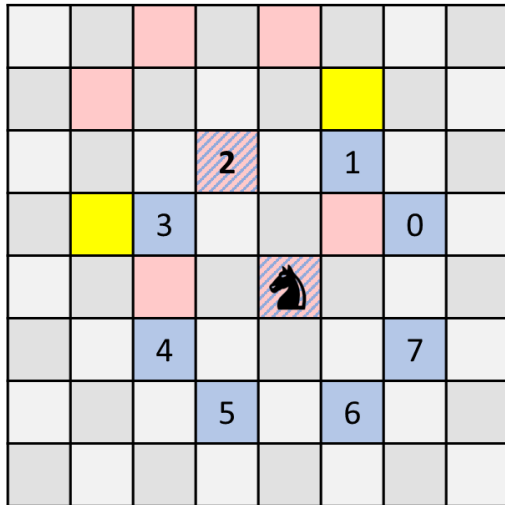
**$m=1$ .** There is only one way out (e.g, the yellow square). If the Knight were to go to square 2 now, then the next move (to yellow) would remove 2 from further concern. But if it doesn't go there now, then when it eventually gets to the yellow square, it will be forced to go to 2, which will then 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.

```
def score(r: int, c: int) -> int:
    """Return the number of unvisited neighbors of (r,c). (Warnsdorff's Rule)."""
    count = 0          # The number of unvisited neighbors of (r,c) found so far.
    for j in range(CUL_DE_SAC):
        if Tour.B[r + deltaR[j]][c + deltaC[j]] == Tour.BLANK:
            count += 1
    return count
```

**score:** Replace stub by implementation of Warnsdorff's Rule.

```
def score(r: int, c: int) -> int:
    """Return the number of unvisited neighbors of (r,c). (Warnsdorff's Rule)."""
    count = 0          # The number of unvisited neighbors of (r,c) found so far.
    for j in range(CUL_DE_SAC):
        if Tour.B[r + deltaR[j]][c + deltaC[j]] == Tour.BLANK:
            count += 1
    return count
```

N.B. `score` is auxiliary to method `solve`, and can be defined therein, which has the benefit (**now**) that we want access to the auxiliary constants `deltaR` and `deltaC` of `solve`.

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

**score:** Replace stub by implementation of Warnsdorff's Rule.

```
def score(r: int, c: int) -> int:
    """Return the number of unvisited neighbors of (r,c). (Warnsdorff's Rule)."""
    count = 0          # The number of unvisited neighbors of (r,c) found so far.
    for j in range(CUL_DE_SAC):
        if Tour.B[r + deltaR[j]][c + deltaC[j]] == Tour.BLANK:
            count += 1
    return count
```

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

---

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

---

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

**score:** Replace stub by implementation of Warnsdorff's Rule.

```
def score(r: int, c: int) -> int:
    """Return the number of unvisited neighbors of (r,c). (Warnsdorff's Rule)."""
    count = 0 # The number of unvisited neighbors of (r,c) found so far.
    for j in range(CUL_DE_SAC):
        if Tour.B[r + deltaR[j]][c + deltaC[j]] == Tour.BLANK:
            count += 1
    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 picks a random neighbor.

```
# Let k = index of an unvisited neighbor, or CUL_DE_SAC.
```

```
# -----
```

```
#.Let unvisited be the set of unvisited neighbors of (r,c).
```

```
#.If |unvisited| is zero, k = CUL_DE_SAC; else pick k at random from unvisited.
```



**Neighbor Selection:** Monte Carlo algorithm picks a random neighbor.

```
# Let k = index of an unvisited neighbor, or CUL_DE_SAC.  
# -----  
# Let unvisited be the set of unvisited neighbors of (r,c).  
unvisited = []  
for k in range(CUL_DE_SAC):  
    if Tour.B[Tour.r + deltaR[k]][Tour.c + deltaC[k]] == Tour.BLANK:  
        unvisited.append(k);  
  
#.If |unvisited| is zero, k = CUL_DE_SAC; else pick k at random from unvisited.
```

**Neighbor Selection:** Monte Carlo algorithm picks a random neighbor.

```
# Let k = index of an unvisited neighbor, or CUL_DE_SAC.
# -----
# Let unvisited be the set of unvisited neighbors of (r,c).
unvisited = []
for k in range(CUL_DE_SAC):
    if Tour.B[Tour.r + deltaR[k]][Tour.c + deltaC[k]] == Tour.BLANK:
        unvisited.append(k);

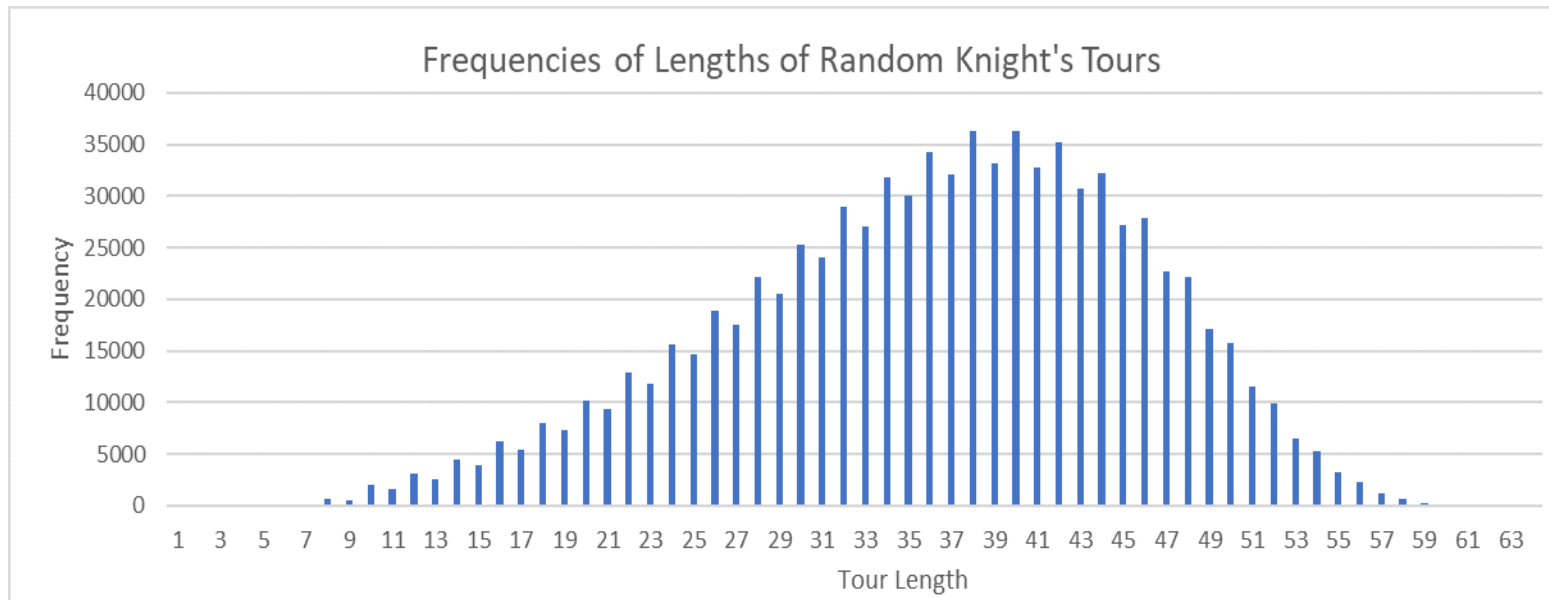
# If |unvisited| is zero, k = CUL_DE_SAC; else pick k at random from unvisited.
if len(unvisited) == 0: k = CUL_DE_SAC
else: k = random.choice(unvisited)
```

## 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.