Principled Programming

Introduction to Coding in Any Imperative Language

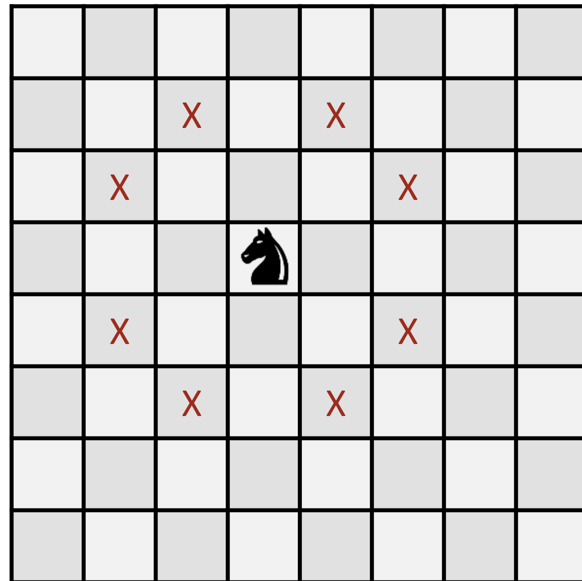
Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Knight's Tour

Copyright©2023 by Tim Teitelbaum; Most recent revision, 8/17/2023

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.

Introduction

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.
- Find a minimal value.
- Sentinels.

New techniques introduced include:

- Generalization and re-instantiation.
- Use of symbolic constants, and tables of constants.
- Incremental testing.

Two new programming approaches that, while not guaranteed to solve 3 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.**

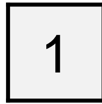
☞ **Confirm your understanding with concrete examples.**

Understanding the Problem

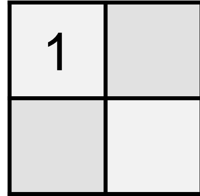
1

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

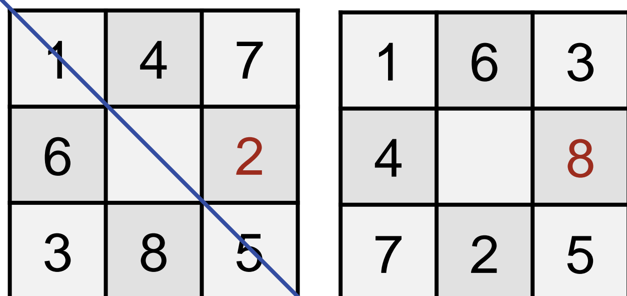
Understanding the Problem

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.



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?

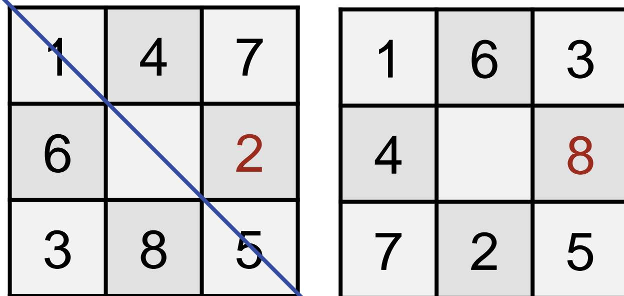
Understanding the Problem

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	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.

Top-level Code Structure

Establish a framework:

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight’s Tour. */
      static void main() { } /* main */
   } /* KnightsTour */
```

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

Top-level Code Structure

Establish a framework:

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight's Tour. */
      static void main() { } /* main */
   } /* KnightsTour */
```

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

☞ **A method header-comment specifies the effect of invoking it, and (if the method has non-void type) the value returned. If the method has parameters, the specification is written in terms of those parameters.**

Top-level Code Structure

Establish a framework:

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight's Tour. */
      static void main() { } /* main */
   } /* KnightsTour */
```

- ☞ **A class header-comment is descriptive, and omits the details of the methods and variables of the class. Reference available auxiliary documentation.**
- ☞ **A method header-comment specifies the effect of invoking it, and (if the method has non-void type) the value returned. If the method has parameters, the specification is written in terms of those parameters.**
- ☞ **Label the end of a long class or method definition with its name in a comment.**

Top-level Code Structure

Establish a solution architecture …

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight’s Tour. */
      static void main() {
         /* Initialize. */
         /* Compute. */
         /* Output. */
         } /* main */
   } /* KnightsTour */
```

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

Top-level Code Structure

… but don’t go far before thinking about the (internal) data representation.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight’s Tour. */
      static void main() {
         /* Initialize. */
         /* Compute. */
         /* Output. */
         } /* main */
   } /* KnightsTour */
```

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

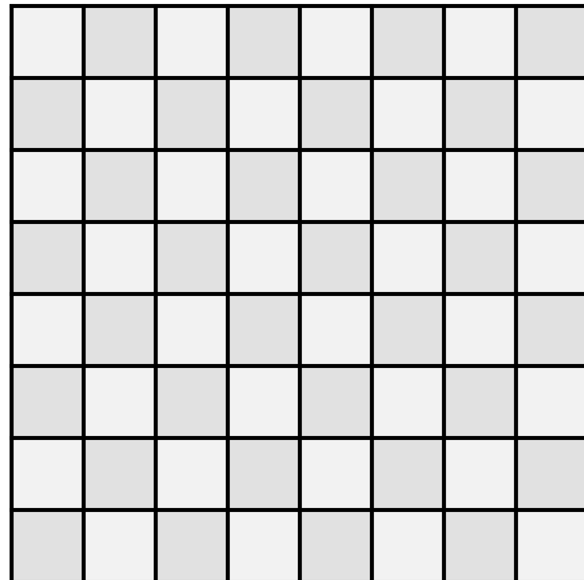
Data Representation:

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

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

Data Representation

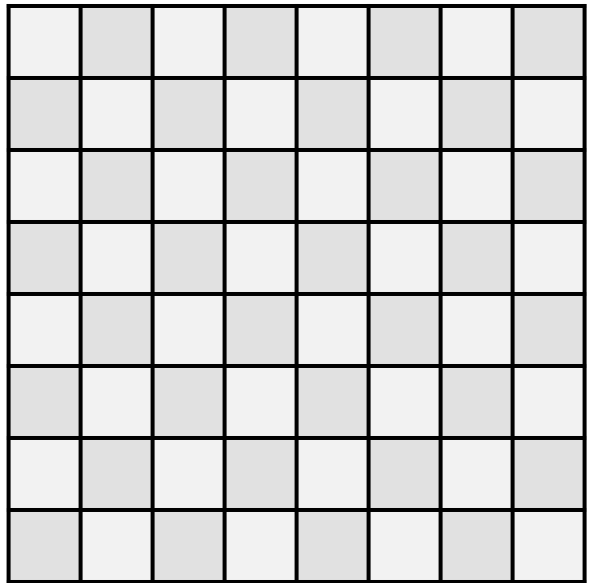
Board Representation 1: The 2-D physical board can correspond directly to a 2-D array.



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

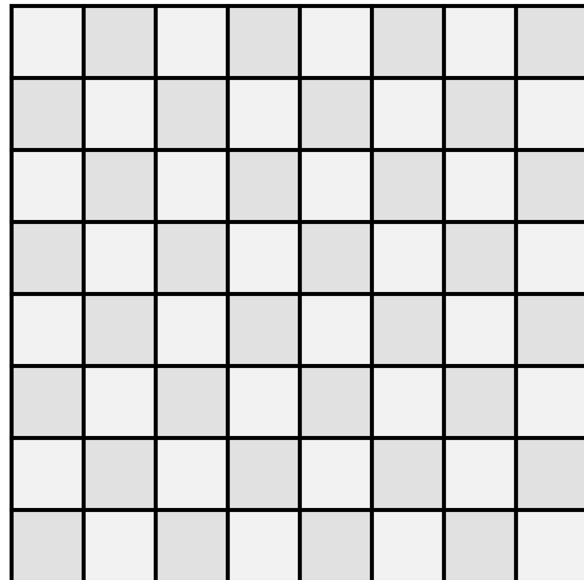
Data Representation

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

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.

B	0 (lo)	1	2	3	4 (c)	5	6	7 (hi)	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.

B	lo 0	1	2	3	c 4	5	6	hi 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									

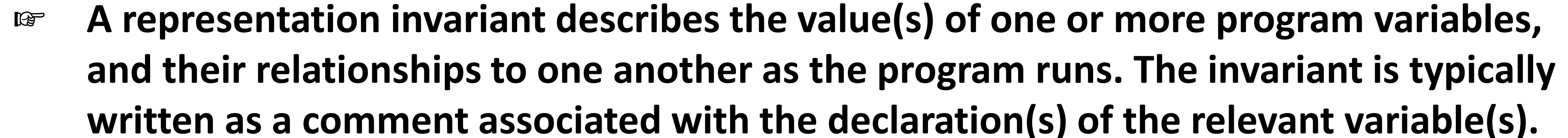
BLANK: 0

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: Write invariants for the data representations.

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* 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[][] = new int[N][N]; // 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 = lo+N-1;     // Last row or column index.
   ...
   } /* KnightsTour */
```



Board Representation 1: Write invariants for the data representations.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* 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 move;                 // Length of Tour.
      static int r, c;                 // Position of Knight.
   ...
   } /* KnightsTour */
```

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 path, if possible.
- **Retract** the path, if the strategy calls for backtracking.

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 path, if possible.
- **Retract** the path, 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 path, if possible.
- **Retract** the path, if the strategy calls for backtracking.

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

Alternative Representation: Address a shortcoming of Representation 1.

- **Retract** the path, 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′`⟩.

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									

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 path, 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'`⟩.

move: 5

B	0 (lo)	1	2	3	4 (c)	5 (c')	6	7 (hi)	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									

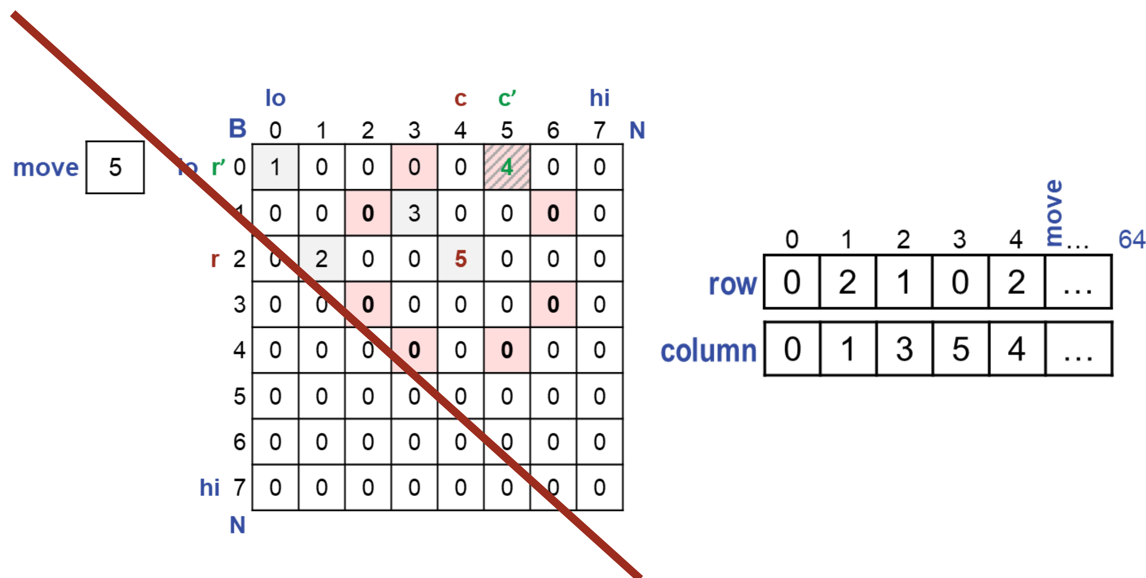
	0	1	2	3	4	… (move)	64
row	0	2	1	0	2	…	
column	0	1	3	5	4	…	

But if the coordinates of path 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 path by the two ordered collections, `row` and `column`?

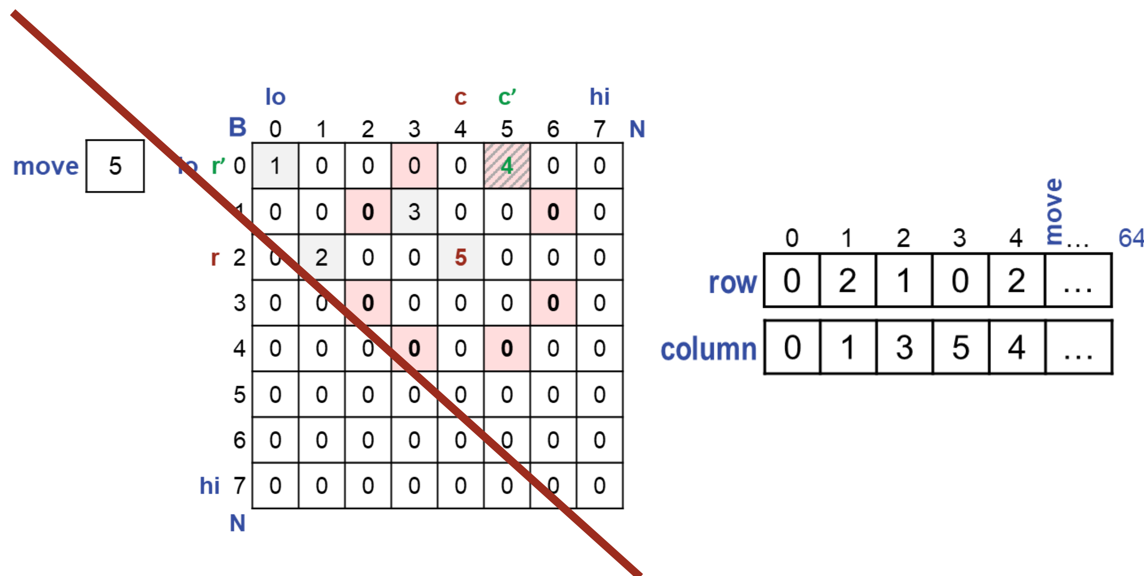


☞ **The touchstone of a data representation is its utility in performing the needed operations.**

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

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

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



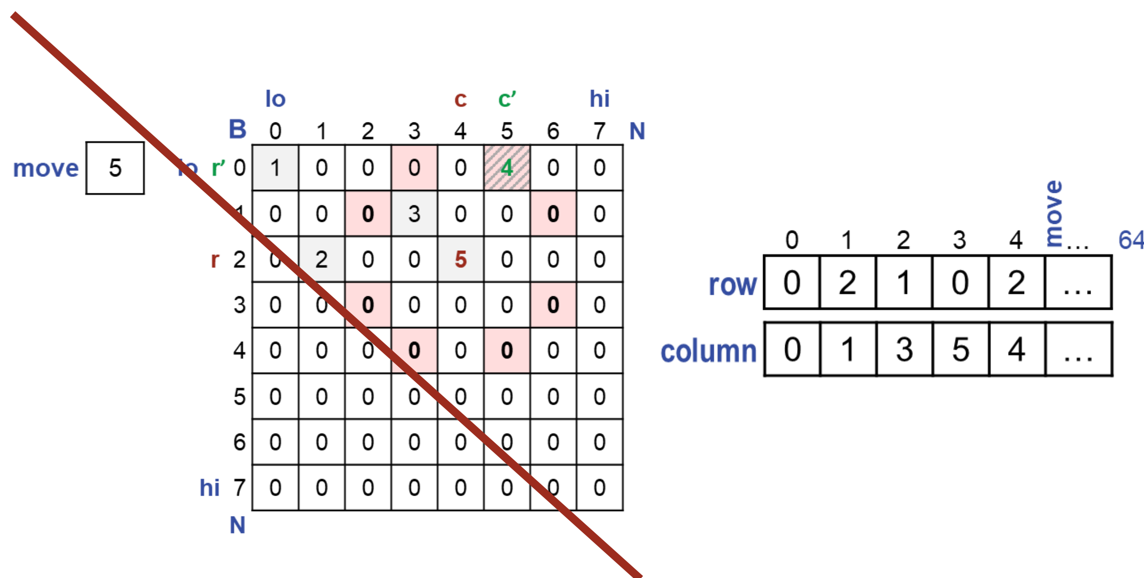
Without the board B, testing whether an ⟨`r’`,`c’`⟩ is “unvisited” would require determining whether it is on the current path, which would require a **search** of the path.

☞ **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 path by the two ordered collections, `row` and `column`?

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



Without the board B, testing whether an ⟨`r'`,`c'`⟩ is “unvisited” would require determining whether it is on the current path, which would require a **search** of the path.

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.**

Data Representation

Representation 1:

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

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

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									

Representation 2:

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

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

	0	1	2	3	4	move …	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 or half dozen the other"?

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

Representation 1:

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

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

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									

Representation 2:

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

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

	0	1	2	3	4	move …	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 or half dozen the other”?

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

Representation 1:

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

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

move: 5

B	0 (lo)	1	2	3	4 (c)	5 (c’)	6	7 (hi)
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

Representation 2:

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

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

	0	1	2	3	4	… (move)	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 path 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 the **int** B anyway?

Representation 1:

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

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

Representation 2:

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

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

move: 5

B	0 (lo)	1	2	3	4 (c)	5 (c')	6	7 (hi)
0 (lo, r')	1	0	0	0	0	4	0	0
1	0	0	0	3	0	0	0	0
2 (r)	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 (hi)	0	0	0	0	0	0	0	0

N

	0	1	2	3	4	move ...	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 path 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.**

Representation invariants: Refer to them in statement comments.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight’s Tour. */
      static void main() {
         /* Initialize: Establish invariant for a tour of length 1. */
         /* Compute: Extend the tour, if possible. */
         /* Output: Print tour as numbered cells in N-by-N grid of 0s. */
} /* main */
   } /* KnightsTour */
```

Top-level Procedures

Invoke methods:

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* Output a (possibly partial) Knight's Tour. */
      static void main() {
         /* Initialize: Establish invariant for a tour of length 1. */
            Initialize();
         /* Compute: Extend the tour, if possible. */
            Solve();
        /* Output: Print tour as numbered cells in N-by-N grid of 0s. */
            Display();
      } /* main */
   } /* KnightsTour */
```

Define methods: Consult the representation invariant, as needed.

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Initialize: Establish invariant for a tour of length 1. */
   static void Initialize() {
      r = lo; c = lo; move = 1; B[r][c] = move;
      } /* Initialize */
   ...
} /* KnightsTour */
```

Define methods: A stub can facilitate incremental testing.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() { } /* Solve */
   ...
   } /* KnightsTour */
```

Top-level Procedures

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

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Output: 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 */
   ...
   } /* KnightsTour */
```

Any small bugs will be caught in this controlled setting.

Incremental Testing:

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
```

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

Test early:

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!

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

☞ **Test programs incrementally.**

Method Solve

Define methods: It's time to try a little harder.

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      __________
      } /* Solve */
   ...
   } /* KnightsTour */
```

Method Solve

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

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      _________
      while ( _________ ) _________
      _________
      } /* Solve */
   ...
   } /* KnightsTour */
```

General iteration pattern: As a model.

```
/* Start at “the beginning”. */
while ( /* not “beyond the end” */ ) {
   /* Process the current “place”. */
   /* Advance to “the next place” (or to “beyond the end”). */
   }
```

"Place" is ⟨r,c⟩.

General iteration pattern: Instantiate.

```
/* Start at "the beginning". */
while ( /* not "beyond the end" */ ) {
   /* Process the current "place". */
   /* Advance to "the next place" (or to "beyond the end"). */
   }
```

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

Combine “Process” and “Advance” into “Extend”.

General iteration pattern: Instantiate.

```
/* Start at “the beginning”. */
while ( /* not “beyond the end” */ ) {
   /* Extend the tour 1 square, if possible. */
   }
```

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

"The end" is when we are stuck in a cul-de-sac.

General iteration pattern: Instantiate.

```
/* Start at "the beginning". */
while ( /* not in cul-de-sac */ ) {
   /* Extend the tour 1 square, if possible. */
   }
```

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

Method Solve

"Start at the beginning" done by `Initialize`.

General iteration pattern: Instantiate.

```
while ( /* not in cul-de-sac */ ) {
   /* Extend the tour 1 square, if possible. */
   }
```

Method Solve

General iteration pattern: Instantiate, in place**.**

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ )
         /* Extend the tour 1 square, if possible. */
      } /* Solve */
   ...
   } /* KnightsTour */
```

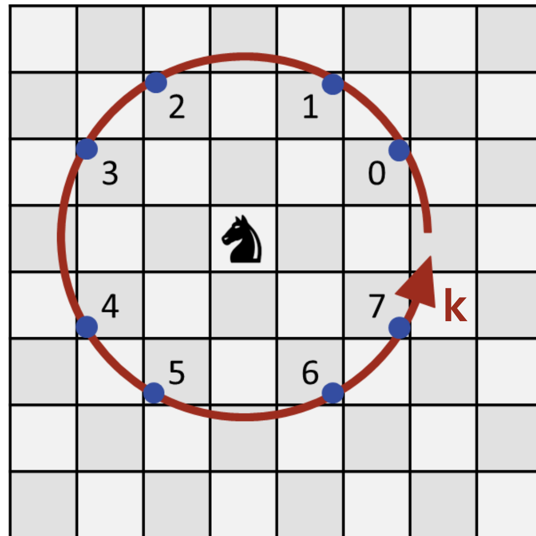
Search-Use Pattern: Extend is an example of the pattern.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Locate unvisited neighbor, or indicate cul-de-sac. */
            if ( /* not in cul-de-sac */ )
               /* Extend the tour to unvisited neighbor. */
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

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

Method Solve

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



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

Introduce a Coordinate System: Adopt it.

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of unvisited neighbor, or CUL_DE_SAC. */
            if ( k!=CUL_DE_SAC )
               /* Extend the tour to unvisited neighbor. */
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Method Solve

Method Solve

Sequential Search: To find an unvisited neighbor.

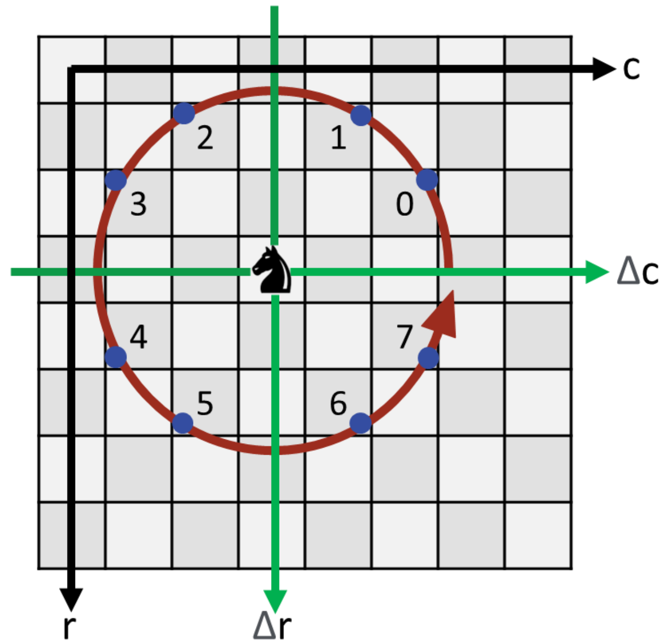
```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of unvisited neighbor, or CUL_DE_SAC. */
               int k = 0;
               while ( k<CUL_DE_SAC && /* neighbor k visited */ ) k++;
            if ( k!=CUL_DE_SAC )
               /* Extend the tour to unvisited neighbor. */
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Method Solve

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

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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 unvisited neighbor. */
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

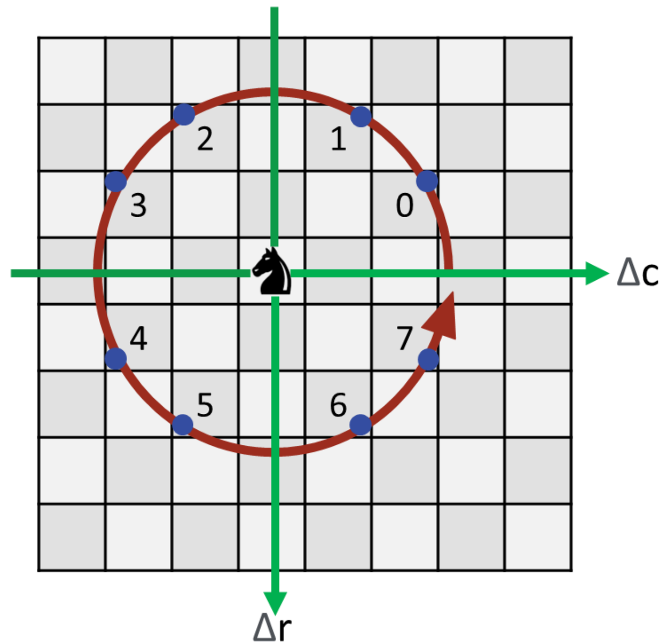
Introduce another Coordinate System: ⟨Δr,Δc⟩



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.

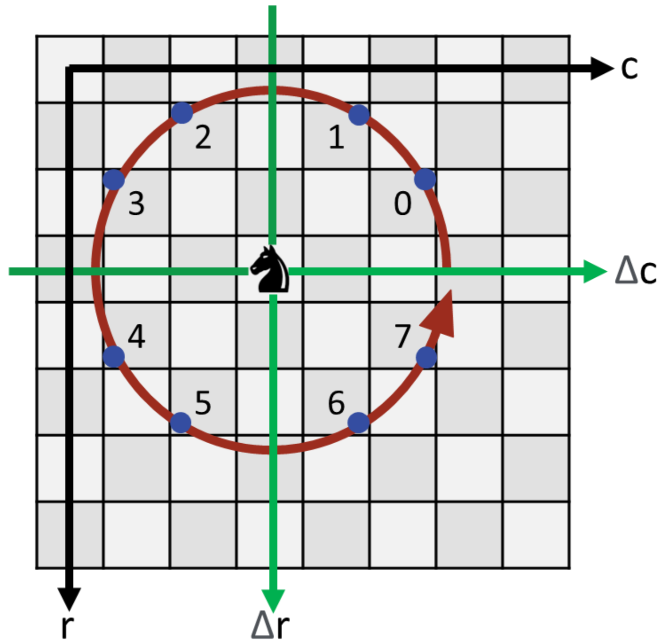


```
//                                 0    1    2    3    4    5    6    7
static final int deltaR[] = {-1, -2, -2, -1,  1,  2,  2,  1};
static final int deltaC[] = { 2,  1, -1, -2, -2, -1,  1,  2};
```

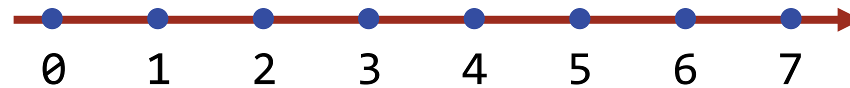
☞ **Introduce auxiliary data to allow code to be uniform.**

Method Solve

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

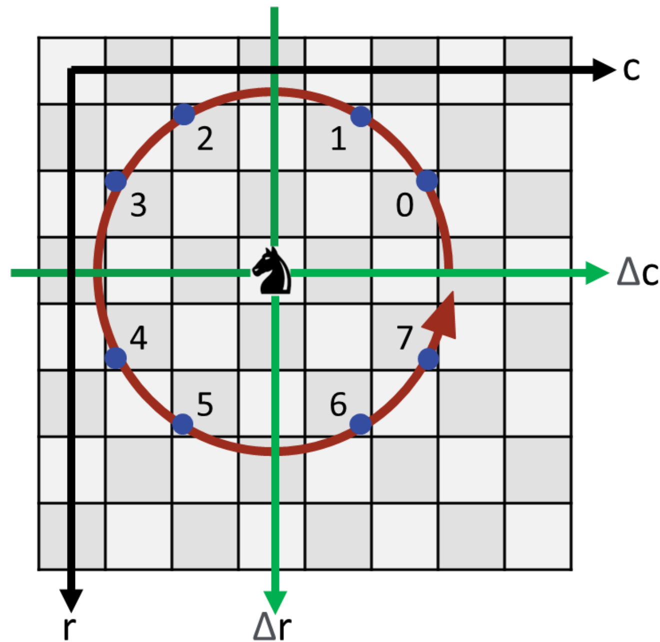


If the Knight has a neighbor (●) at ⟨Δr,Δc⟩ in the local system, then that neighbor is at ⟨r+Δr,c+Δc⟩ in the global system.



```
//                                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};
```

Method Solve

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

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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 unvisited neighbor. */
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Extend the Tour: Update the invariant.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      while ( /* not in cul-de-sac */ ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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 unvisited neighbor. */
                  r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Method Solve

Termination:

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

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      int k = 0;  // Neighbor number not CUL_DE_SAC.
      while ( k!=CUL_DE_SAC ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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. */
                  r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Method Solve

Termination:

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

Move declaration of k outside the loop.

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      int k = 0;  // Neighbor number not CUL_DE_SAC.
      while ( k!=CUL_DE_SAC ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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. */
                  r = r+deltaR[k]; c = c+deltaC[k]; move++; B[r][c] = move;
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

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 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.**

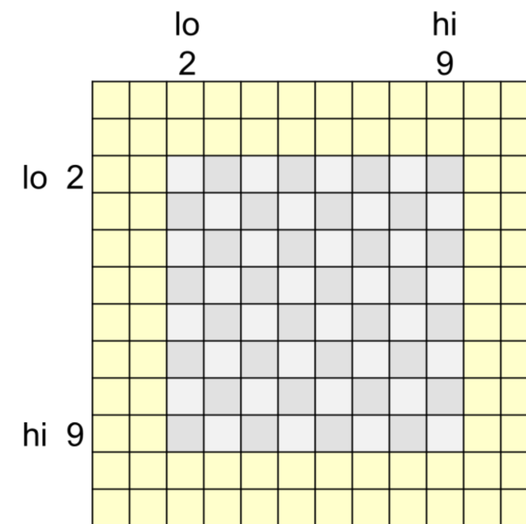
Boundary Conditions

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

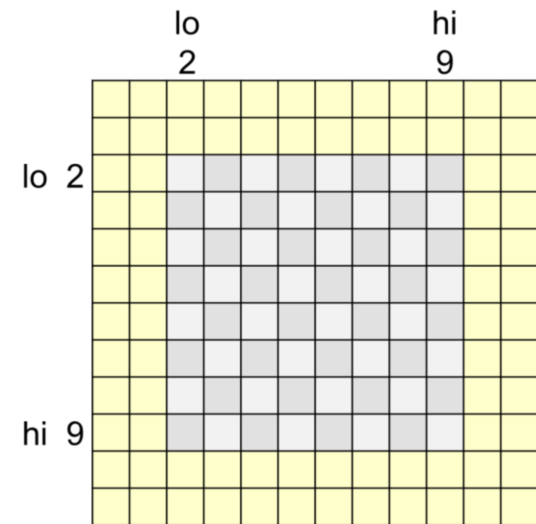
- You can waste a lot of time debugging things you could have anticipated if you had thought a little more deeply: Some squares have fewer than eight neighbors because they are at the board boundary.
- We seek a way to deal with the boundaries without doing major surgery on the code.



Sentinels to the Rescue:

```
/* Knight's Tour: See problem statement in Chapter 14. */
class KnightsTour {
   /* 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 = 8; // Size of B.
      static int B[][] = new int[N+4][N+4]; // Chess board, initially 0s.
      static final int BLANK = 0;            // Unvisited square in board.
      static final int lo = 2;               // First row or column index.
      static final int hi = lo+N-1;          // Last row or column index.
   ...
   } /* KnightsTour */
```

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



Sentinels to the Rescue:

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Initialize: Establish invariant for a tour of length 1. */
   static void Initialize() {
      /* Set B to N-by-N board of BLANKs in 2-cell ring of 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;
         for (int r=lo; r<=hi; r++)
            for (int c=lo; c<=hi; c++) B[r][c] = BLANK;
      r = lo; c = lo; move = 1; B[r][c] = move;
      } /* Initialize */
   ...
} /* KnightsTour */
```

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
```

Not too bad 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
```

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

Fix a minor formatting issue by modifying a line in `Output`:

```
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.

Neighbor Selection: Greedy algorithm, pick any neighbor.

```
/* Knight’s Tour: See problem statement in Chapter 14. */
class KnightsTour {
   ...
   /* Compute: Extend the tour, if possible. */
   static void Solve() {
      int k = 0;  // Neighbor number not CUL_DE_SAC.
      while ( k!=CUL_DE_SAC ) {
         /* Extend the tour 1 square, if possible. */
            /* Let k = # of 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. */
                  r = r+deltaR[k]; c = c+deltaC[k];  move++; B[r][c] = move;
            }
      } /* Solve */
   ...
   } /* KnightsTour */
```

Neighbor Selection: Greedy algorithm, pick any neighbor.

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

Neighbor Selection: Heuristic algorithm, pick “best” neighbor.

```
/* Let k = # of unvisited neighbor, or CUL_DE_SAC. */
   /* Let bestK be favored unvisited neighbor, or CUL_DE_SAC, if all
      neighbors are already visited. */
   k = bestK;
```

Neighbor Selection: Heuristic algorithm, pick "best" neighbor, for some Score function.

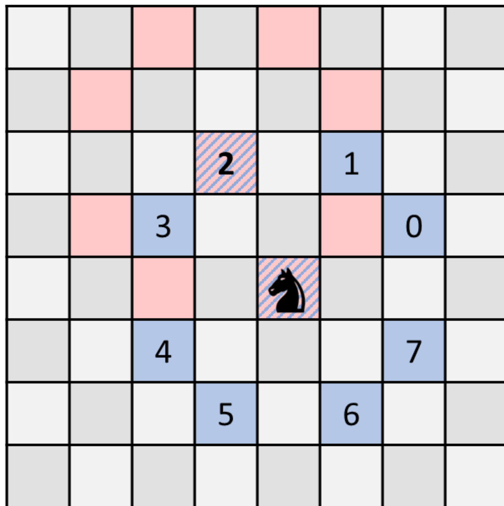
```
/* Let k = # of unvisited neighbor, or CUL_DE_SAC. */
   /* Let bestK be favored unvisited neighbor, or CUL_DE_SAC, if all
      neighbors are already visited. */
      int bestK = CUL_DE_SAC;              // Neighbor # of favored neighbor.
      int bestScore = Integer.MAX_VALUE; // Score of neighbor bestK.
      for (k=0; k<8; 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;
```

"Best" is minimal score.

Choosing a 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).

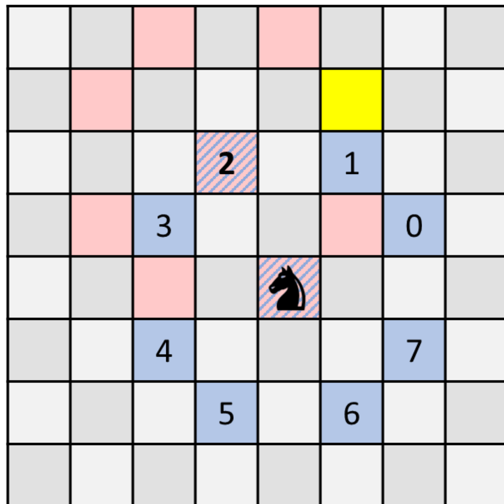
m=0. The Knight's current square is the only way to get to square 2, and if it doesn't go there now, it won't ever get another chance. Yes, it will then be in a cul-de-sac, so, if we hope for a tour of length 64, this better be the 64th move. If not, the Knight is effectively cutting its losses, and 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".

Choosing a 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).



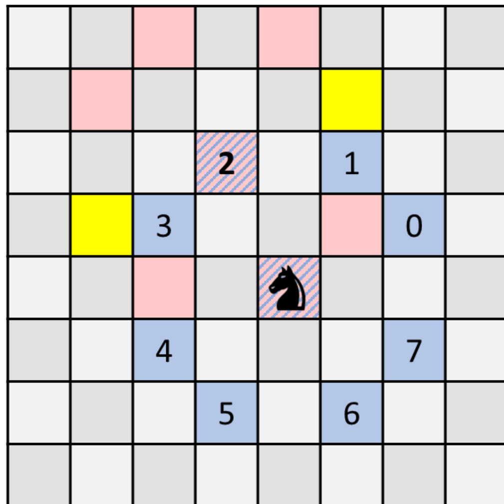
Rationale. Let Knight's neighbor (e.g., 2) have *m* unvisited neighbors (a subset of the pink squares).

m=1. There is only one way out (shown in yellow). If the Knight goes to square 2 now, the next move (to yellow) removes 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 end the tour in a cul-de-sac. So, best to pass through 2 now, for otherwise it will loom as a hazard.

Choosing a 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.

Coding Score:

```
/* Return # of unvisited neighbors of ⟨r,c⟩. (Warnsdorff's Rule) */
static int Score(int r, int c) {
   int count = 0;  // # unvisited neighbors among 0..k.
   for (int k=0; k<8; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) count++;
   return count;
   }
```

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

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

Coding Score:

```
/* Return # of unvisited neighbors of ⟨r,c⟩. (Warnsdorff's Rule) */
static int Score(int r, int c) {
   int count = 0;  // # unvisited neighbors among 0..k.
   for (int k=0; k<8; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==BLANK ) count++;
   return count;
   }
```

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

⟨r,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.**

Testing, revisited yet again

Incremental Testing: 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
```

Testing, revisited yet again

Incremental Testing: 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 = # of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor #’s of 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 = # of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor #’s of unvisited neighbors
      of ⟨r,c⟩. */
      int unvisited[] = new int[8];
      int count = 0;  // # unvisited neighbors
      for (k=0; k<8; 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] */;
```

Monte Carlo Tours

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

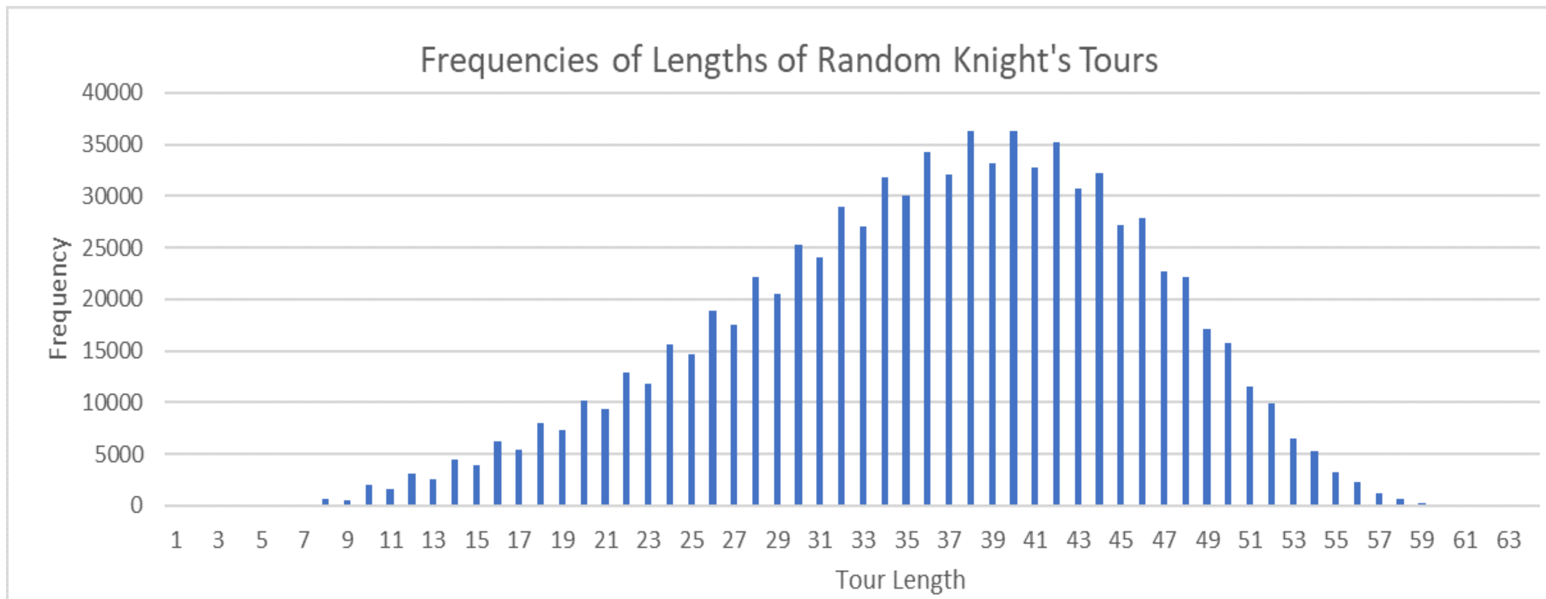
```
/* Let k = # of unvisited neighbor, or CUL_DE_SAC. */
   /* Let unvisited[0:count-1] be neighbor #’s of unvisited neighbors
      of ⟨r,c⟩. */
      int unvisited[] = new int[8];
      int count = 0;  // # unvisited neighbors
      for (k=0; k<8; k++)
      if ( B[r+deltaR[k]][c+deltaC[k]]==Blank ) {
         unvisited[count]=k; count++;
         }
   if ( count==0 ) k = CUL_DE_SAC;
   else k = unvisited[rand.nextInt(count)];
```

Omitted Details:

Access to the random number generator `rand`.

A driver that repeatedly invokes `Solve` until a solution is found.

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



Reflections:

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