

Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Creative Representations

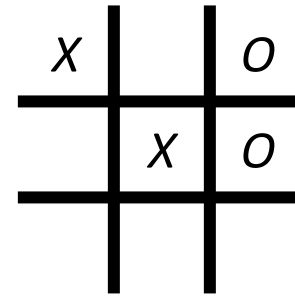
Choice of representations. Straightforward data representations are often effective and sufficient. However, creative representations can result in significant improvements. We present four examples of innovative data representations:

- Tic-Tac-Toe
- Checkers
- The Eight Queens Problem
- The Ricocheting Bee-Bee Problem

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

Background: Tic-Tac-Toe is played on a 3-by-3 board. Two players take turns marking cells with “X” or “O”. The objective is obtain 3 of your marks in a row, column, or diagonal.

Problem Statement: Write a program that plays Tic-Tac-Toe.



X		O
	X	O

Data Representation.

Update the data representation on a move by "X" at $\langle r,c \rangle$.

Test for a win by "X".

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

T	0	1	2
0	X		O
1		X	O
2			

Data Representation. Cells of (' ', 'X', 'O') for (blank, "X", "O").
 T: list[list[str]] = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

Update the data representation on a move by "X" at (r,c).
 T[r][c] = 'X' # Mark T with 'X' for "X".

Test for a win by "X".

if T[0][0] == 'X' and T[0][1] == 'X' and T[0][2] == 'X' or
 T[1][0] == 'X' and T[1][1] == 'X' and T[1][2] == 'X' or
 ...
 T[0][0] == 'X' and T[1][1] == 'X' and T[2][2] == 'X' or
 T[0][2] == 'X' and T[1][1] == 'X' and T[2][0] == 'X' : # "X" wins.

IN PROGRESS

T	0	1	2
0	1	0	-1
1	0	1	-1
2	0	0	0

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

```
T: list[list[int]] = [ [0,0,0],[0,0,0],[0,0,0] ]
```

Update the data representation on a move by "X" at $\langle r,c \rangle$.

```
T[r][c] = 1 # Mark T with 1 for "X".
```

Test for a win by "X".

```
if T[0][0] + T[0][1] + T[0][2] == 3 or
   T[1][0] + T[1][1] + T[1][2] == 3 or
   ...
   T[0][0] + T[1][1] + T[2][2] == 3 or
   T[0][2] + T[1][1] + T[2][0] == 3 : # "X" wins.
```

IN PROGRESS

T	0	1	2
0	1	0	-1
1	0	1	-1
2	0	0	0

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

```
T: list[list[int]] = [[0,0,0],[0,0,0],[0,0,0] ]
```



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

Update the data representation on a move by "X" at $\langle r,c \rangle$.

```
T[r][c] = 1 # Mark T with 1 for "X".
```

Test for a win by "X".

```
if T[0][0] + T[0][1] + T[0][2] == 3 or
    T[1][0] + T[1][1] + T[1][2] == 3 or
    ...
    T[0][0] + T[1][1] + T[2][2] == 3 or
    T[0][2] + T[1][1] + T[2][0] == 3 : # "X" wins.
```

T	0	1	2
0	1	0	-1
1	0	1	-1
2	0	0	0

movesX 2

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

```
T: list[list[int]] = [ [0,0,0],[0,0,0],[0,0,0] ]
```

```
movesX: int = 0      # Number of moves made by "X".
```

Update the data representation on a move by "X" at $\langle r,c \rangle$.

```
T[r][c] = 1          # Mark T with 1 for "X".
```

```
movesX += 1         # Increment count of "X" marks.
```

Test for a win by "X".

```
if movesX < 3: #.Not a win for "X".
```

```
else: #.Win for "X" is still possible, but only in 3, 4, or 5 moves.
```

IN PROGRESS

T	0	1	2
0	1	0	-1
1	0	1	-1
2	0	0	1

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

```
T: list[list[int]] = [ [0,0,0],[0,0,0],[0,0,0] ]
```

```
movesX: int = 0 # Number of moves made by "X".
```

```
M: list[list[int]] = [ [8,1,6],[3,5,7],[4,9,2] ] # Magic Square.
```

```
sumX: int = 0 # Sum of magic values corresponding to "X".
```

movesX **3** sumX **15**

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

```
T[r][c] = 1 # Mark T with 1 for "X".
```

```
movesX += 1 # Increment count of "X" marks.
```

```
sumX = sumX + M[r][c] # Add magic value corresponding to  $\langle r,c \rangle$ .
```

Test for a win by "X".

```
if movesX < 3: #.Not a win for "X".
```

```
elif movesX == 3 and sumX == 15: #."X" wins.
```

```
else: #.Win for "X" is still possible, but only in 4 or 5 moves.
```



T	0	1	2
0	1	1	-1
1	0	1	-1
2	0	0	0

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

```
T: list[list[int]] = [ [0,0,0],[0,0,0],[0,0,0] ]
```

```
movesX: int = 0 # Number of moves made by "X".
```

```
M: list[list[int]] = [ [8,1,6],[3,5,7],[4,9,2] ] # Magic Square.
```

```
sumX: int = 0 # Sum of magic values corresponding to "X".
```

movesX **3** sumX **14**

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

```
T[r][c] = 1 # Mark T with 1 for "X".
```

```
movesX += 1 # Increment count of "X" marks.
```

```
sumX = sumX + M[r][c] # Add magic value corresponding to  $\langle r,c \rangle$ .
```

Test for a win by "X".

```
if movesX < 3: #.Not a win for "X".
```

```
elif movesX == 3 and sumX == 15: #."X" wins.
```

```
else: #.Win for "X" is still possible, but only in 4 or 5 moves.
```

IN PROGRESS

T	0	1	2
0	1	1	-1
1	0	1	-1
2	0	-1	0

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

`T: list[list[int]] = [[0,0,0],[0,0,0],[0,0,0]]`

`movesX: int = 0` # Number of moves made by "X".

`M: list[list[int]] = [[8,1,6],[3,5,7],[4,9,2]]` # Magic Square.

`sumX: int = 0` # Sum of magic values corresponding to "X".

`movesX` 3 `sumX` 14

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

`T[r][c] = 1` # Mark T with 1 for "X".

`movesX += 1` # Increment count of "X" marks.

`sumX = sumX + M[r][c]` # Add magic value corresponding to $\langle r,c \rangle$.

Test for a win by "X".

`if movesX < 3: #.Not a win for "X".`

`elif movesX == 3 and sumX == 15: #."X" wins.`

`else: #.Win for "X" is still possible, but only in 4 or 5 moves.`

IN PROGRESS

T	0	1	2
0	1	1	-1
1	0	1	-1
2	0	-1	1

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

T: list[list[int]] = [[0,0,0],[0,0,0],[0,0,0]]

movesX: int = 0 # Number of moves made by "X".

M: list[list[int]] = [[8,1,6],[3,5,7],[4,9,2]] # Magic Square.

sumX: int = 0 # Sum of magic values corresponding to "X".

TT: int = [0] * 10 # TT[m] is 1 iff "X" in T[r][c] M[r][c]==m.

movesX **4** sumX **16**

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

T[r][c] = 1 # Mark T with 1 for "X".

movesX += 1 # Increment count of "X" marks.

sumX = sumX + M[r][c] # Add magic value corresponding to $\langle r,c \rangle$.

TT[M[r][c]] = 1 # Set TT[m] to 1 iff square with magic
value m is "X".

TT	0	1	2	3	4	5	6	7	8	9
	0	1	1	0	0	1	0	0	1	0

Test for a win by "X".

if movesX < 3: #.Not a win for "X".

elif movesX == 3 and sumX == 15: #."X" wins.

elif movesX == 4 and 9 <= sumX and sumX <= 24 and TT[sumX-15] == 1: #."X" wins.

else: #.Win for "X" is still possible, but only in 5 moves.

WIN!

T	0	1	2
0	1	1	-1
1	0	1	-1
2	1	-1	0

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, "X", "O" \rangle$.

T: list[list[int]] = [[0,0,0],[0,0,0],[0,0,0]]

movesX: int = 0 # Number of moves made by "X".

M: list[list[int]] = [[8,1,6],[3,5,7],[4,9,2]] # Magic Square.

sumX: int = 0 # Sum of magic values corresponding to "X".

TT: int = [0] * 10 # TT[m] is 1 iff "X" in T[r][c] M[r][c]==m.

movesX **4** sumX **18**

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

T[r][c] = 1 # Mark T with 1 for "X".

movesX += 1 # Increment count of "X" marks.

sumX = sumX + M[r][c] # Add magic value corresponding to $\langle r,c \rangle$.

TT[M[r][c]] = 1 # Set TT[m] to 1 iff square with magic value m is "X".

TT	0	1	2	3	4	5	6	7	8	9
	0	1	0	0	1	1	0	0	1	0

Test for a win by "X".

if movesX < 3: #.Not a win for "X".

elif movesX == 3 and sumX == 15: #."X" wins.

elif movesX == 4 and 9 <= sumX and sumX <= 24 and TT[sumX-15] == 1: #."X" wins.

else: #.Win for "X" is still possible, but only in 5 moves.

IN PROGRESS

T	0	1	2
0	1	0	-1
1	0	-1	1
2	-1	1	1

Data Representation. Cells of $\langle 0, +1, -1 \rangle$ for $\langle \text{blank}, \text{"X"}, \text{"O"} \rangle$.

T: list[list[int]] = [[0,0,0],[0,0,0],[0,0,0]]

movesX: int = 0 # Number of moves made by "X".

M: list[list[int]] = [[8,1,6],[3,5,7],[4,9,2]] # Magic Square.

sumX: int = 0 # Sum of magic values corresponding to "X".

TT: int = [0] * 10 # TT[m] is 1 iff "X" in T[r][c] M[r][c]==m.

movesX **4** sumX **26**

M	0	1	2
0	8	1	6
1	3	5	7
2	4	9	2

Update the data representation on a move by "X" at $\langle r,c \rangle$.

T[r][c] = 1 # Mark T with 1 for "X".

movesX += 1 # Increment count of "X" marks.

sumX = sumX + M[r][c] # Add magic value corresponding to $\langle r,c \rangle$.

TT[M[r][c]] = 1 # Set TT[m] to 1 iff square with magic
value m is "X".

TT	0	1	2	3	4	5	6	7	8	9
	0	0	1	0	0	0	0	1	1	1

Test for a win by "X".

if movesX < 3: #.Not a win for "X".

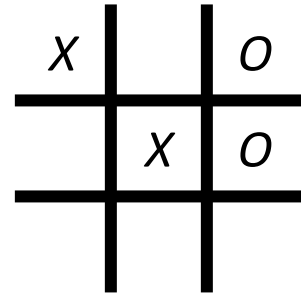
elif movesX == 3 and sumX == 15: #."X" wins.

elif movesX == 4 and 9 <= sumX and sumX <= 24 and TT[sumX-15] == 1: #."X" wins.

else: #.Win for "X" is still possible, but only in 5 moves.

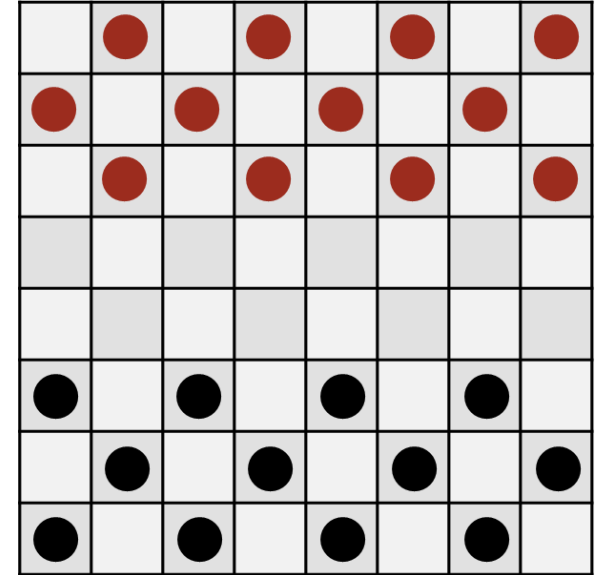
IN PROGRESS

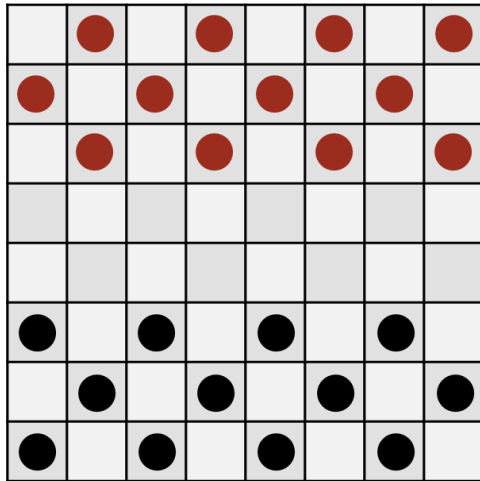
Reflection: Because Tic-Tac-Toe is so simple, the efficiency of the above code is not really needed. It is intended to stretch your imagination, and exhibit principles that can lead to significant code improvement.



Background: Checkers is played on a 8-by-8 board, starting with the initial layout shown. The two players take turns making diagonal moves. In addition to the initial “men” shown, players can earn “queens”. Because all moves are diagonal, only half the squares are used. No other rules are relevant for the purpose of this presentation.

Problem Statement: Write a program that plays Checkers.

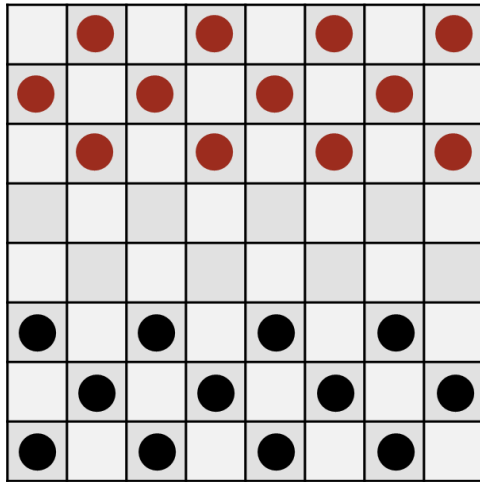




Data Representation.



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



Data Representation.

<Blank, BlackMan, BlackQueen, RedMan, RedQueen>

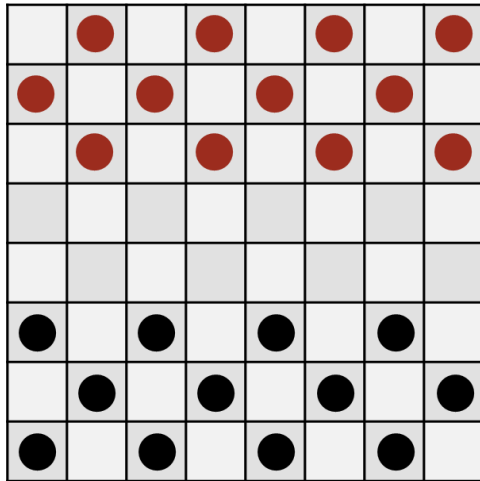
represented by <0,1,2,-1,-2>.

Board: list[list[int]]

0	-1	0	-1	0	-1	0	-1
-1	0	-1	0	-1	0	-1	0
0	-1	0	-1	0	-1	0	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
+1	0	+1	0	+1	0	+1	0
0	+1	0	+1	0	+1	0	+1
+1	0	+1	0	+1	0	+1	0

This representation requires 64*32 bits.

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



Data Representation.

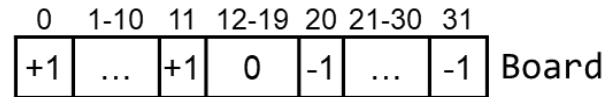
<Blank, BlackMan, BlackQueen, RedMan, RedQueen>

represented by <0,1,2,-1,-2>.

Board: list[list[int]]

	28		29		30		31
24		25		26		27	
	20		21		22		23
16		17		18		19	
	12		13		14		15
8		9		10		11	
	4		5		6		7
0		1		2		3	

Number the used squares 0..31 and adopt a 1-dimensional representation indexed by those numbers. This requires only half the space, i.e., 32*32 bits.



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

	28		29		30		31
24		25		26		27	
	20		21		22		23
16		17		18		19	
	12		13		14		15
8		9		10		11	
	4		5		6		7
0		1		2		3	

Data Representation.

<Blanks, BlackMen, BlackQueens, RedMen, RedQueens>
 # represented by 5 separate Boolean arrays.

Blanks: list[bool] = [0] * 32

BlackMen: list[bool] = [0] * 32

BlackQueens: list[bool] = [0] * 32

RedMen: list[bool] = [0] * 32

RedQueens: list[bool] = [0] * 32

0	1-10	11	12-19	20	21-30	31
F	...	F	T	F	...	F

Blanks

0	1-10	11	12-19	20	21-30	31
T	...	T	F	F	...	F

BlackMen

0	1-10	11	12-19	20	21-30	31
F	...	F	F	T	...	T

RedMen

Because there are only 5 things to represent, the same information can be stored in 5 **bool** arrays. Thus, only 5*32 bits are required, i.e., 5/64 of the original.

N.B. The implementation of a programming language does not necessarily implement **bool** as 1 bit, or **int** as 32 bits, but in principle it could.



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

	28		29		30		31	
+4	24		25		26		27	
+5		20		21		22		23
+4	16		17		18		19	
+5		12		13		14		15
+4	8		9		10		11	
+5		4		5		6		7
+4	0		1		2		3	

Data Representation.

```
# <Blanks, BlackMen, BlackQueens, RedMen, RedQueens>
# represented by 5 separate Boolean arrays.
Blanks: list[bool] = [0] * 32
BlackMen: list[bool] = [0] * 32
BlackQueens: list[bool] = [0] * 32
RedMen: list[bool] = [0] * 32
RedQueens: list[bool] = [0] * 32
```

0	1-10	11	12-19	20	21-30	31
F	...	F	T	F	...	F

Blanks

0	1-10	11	12-19	20	21-30	31
T	...	T	F	F	...	F

BlackMen

0	1-10	11	12-19	20	21-30	31
F	...	F	F	T	...	T

RedMen

Compute targets of all black's forward-right diagonal moves.

Forward-right diagonal moves go to a square numbered 4 (or 5) higher, an irregularity that prevents uniformly shifting the 1-dimensional arrays right 4 (or 5) places to compute reachable squares.

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

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

Data Representation.

```
# (Blanks, BlackMen, BlackQueens, RedMen, RedQueens)
# represented by 5 separate Boolean arrays.
Blanks: list[bool] = [0] * 36
BlackMen: list[bool] = [0] * 36
BlackQueens: list[bool] = [0] * 36
RedMen: list[bool] = [0] * 36
RedQueens: list[bool] = [0] * 36
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
T	T	T	T	F	T	T	T	T	T	T	T	F	F	F	F	F	...	F	

BlackMen

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	T	T	T	T	F	T	T	T	T	T	T	...	F		

BlackMen
right shifted 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

Blanks

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

all targets of
forward-right
moves

Compute targets of all black's forward-right diagonal moves.

We introduce 4 phantom squares, and renumber. All forward-right diagonal moves are now to a square that is **uniformly numbered 5 higher**.

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

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

Data Representation.

(Blanks, BlackMen, BlackQueens, RedMen, RedQueens)

represented by 5 separate Boolean arrays.

Blanks: list[bool] = [0] * 36

BlackMen: list[bool] = [0] * 36

BlackQueens: list[bool] = [0] * 36

RedMen: list[bool] = [0] * 36

RedQueens: list[bool] = [0] * 36

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
T	T	T	T	F	T	T	T	T	T	T	T	T	F	F	F	F	F	...	F

BlackMen

Compute targets of all black's forward-right diagonal moves.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	T	T	T	T	F	T	T	T	T	T	T	T	...	F	

BlackMen
right shifted 5

As a result, we can compute all black's forward-right diagonal moves in a single Right-Shift-5 operation...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

Blanks

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

all targets of
forward-right
moves



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

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

Data Representation.

```
# <Blanks, BlackMen, BlackQueens, RedMen, RedQueens>
# represented by 5 separate Boolean arrays.
Blanks: list[bool] = [0] * 36
BlackMen: list[bool] = [0] * 36
BlackQueens: list[bool] = [0] * 36
RedMen: list[bool] = [0] * 36
RedQueens: list[bool] = [0] * 36
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
T	T	T	T	F	T	T	T	T	T	T	T	F	F	F	F	F	...	F	

BlackMen

Compute targets of all black's forward-right diagonal moves.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	T	T	T	T	F	T	T	T	T	T	T	...	F		

BlackMen
right shifted 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

Blanks

perform a bitwise-logical-and operation with the array of non-blank non-phantom squares...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	35
F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	...	F

all targets of
forward-right
moves

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

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

Data Representation.

(Blanks, BlackMen, BlackQueens, RedMen, RedQueens)
represented by 5 separate Boolean arrays.

Blanks: list[bool] = [0] * 36

BlackMen: list[bool] = [0] * 36

BlackQueens: list[bool] = [0] * 36

RedMen: list[bool] = [0] * 36

RedQueens: list[bool] = [0] * 36

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 35
T T T T F T T T T T T T F F F F F ... F

BlackMen

Compute targets of all black's forward-right
diagonal moves.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 35
F F F F F T T T T F T T T T T T T ... F

BlackMen
right shifted 5

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 35
F F F F F F F F F F F F F T T T T ... F

Blanks

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 35
F F F F F F F F F F F F F T T T T ... F

all targets of
forward-right
moves

and obtain the positions of all blank squares reachable by BlackMen in a forward-right diagonal move.

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

History and Serendipity: Arthur Samuels, a founder of the field of Machine Learning, wrote a Checkers playing program in the 1950s, for which he invented the above representation.



At that time, the fundamental word length of computers was **36** bits. An operation like Shift-Right-5 could be performed in a single machine instruction, and so too could an operation like bitwise-logical-**and**. Thus, the target of all forward-right moves by BlackMen to a vacant square could be performed in just 2 machine instructions!



Shortly thereafter, IBM changed to 32-bit words, which eliminated the possibility of representing phantom squares in words, and the efficiency of uniform shifts of them to compute possible moves in parallel.



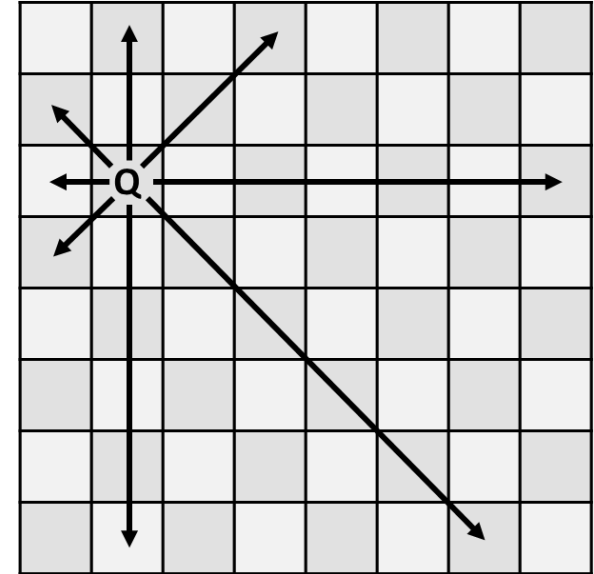
Words are now 64-bits (or more), and so the representation is viable once again.



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

Background: Chess is played on a 8-by-8 board. A Queen can move or capture at any distance vertically, horizontally, or diagonally. It is possible to place eight Queens on the board so that no two are on the same row, column, or diagonal..

Problem Statement: Write a program that finds a layout that solves the problem.



Data Representation.

`list[list[bool]]` # 8 by 8 bool's

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

But why choose a representation (with 2^{64} possible states) that allows two-in-the-same row to be expressed?

Not a solution because there are two Queens in same row.



Choose representations that by design do not have nonsensical configurations.

Data Representation.

`R: list[int] = [0] * 8 # R[c] is r if Queen in (r,c).`

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

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

Why not choose a representation (with 8^8 states) in which two-in-the-same-row cannot even be expressed.

Not a solution because there are two Queens in same row.

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

Data Representation.

`R: list[int] = [0] * 8 # R[c] is r if Queen in (r,c).`

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

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

But why choose a representation (with 8^8 states) that allows two-in-the-same column to be expressed?

Require R to be a permutation of 1,2,3,4,5,6,7,8.



Choose representations that by design do not have nonsensical configurations.

Data Representation.

`R: list[int] = [0] * 8 # R[c] is r if Queen in (r,c).`

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

R permutation of 1,2,3,4,5,6,7,8

But not choose a representation (with 8! states) in which two-in-the-same-column cannot even be expressed.

The failing configuration (above) cannot be expressed as a permutation of 1,2,3,4,5,6,7,8.



Choose representations that by design do not have nonsensical configurations.

R

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

 permutation of 1,2,3,4,5,6,7,8

```
def main() -> None:
    """Solve the Eight Queens problem."""
    # R[c] is r if Queen in (r,c).*/
    R: list[int] = [0, 1, 2, 3, 4, 5, 6, 7]

    # Consider each permutation of R until one is found that represents a
    # solution. (At least one such permutation is known to exist.)
    while has_same_diagonal(R): next_permutation(R)

    # Output solution R.
    ...
```

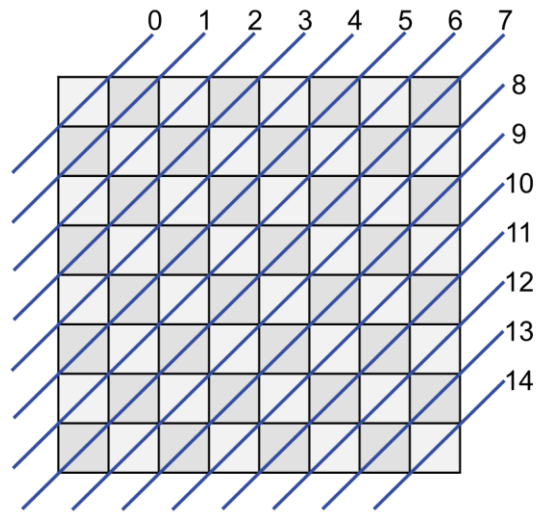
 **The touchstone of a data representation is its utility in performing the needed operations.**

R

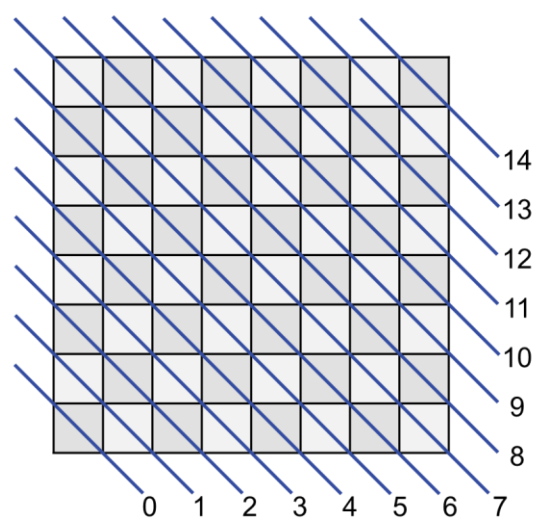
0	1	2	3	4	5	6	7
permutation of 1,2,3,4,5,6,7,8							

...

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



Positive diagonal index: row+column



Negative diagonal index: column-row+7

```
def has_same_diagonal( R: list[int] ) -> bool:
    """Return True iff R has two Queens on same diagonal."""
    # PosDiag[k] (resp., NegDiag[k]) True iff a Queen in R[0..c] occurs on the
    #   positive (resp., negative) diagonal with index k.
    PosDiag: list[bool] = [False] * 15
    NegDiag: list[bool] = [False] * 15

    # Return the column c (or 8, if none) of a Queen for which there are duplicates
    #   on a diagonal.
    c: int = 0
    while c < 8 and not(PosDiag[R[c] + c]) and not(NegDiag[c - R[c] + 7]):
        PosDiag[R[c] + c] = True; NegDiag[c - R[c] + 7] = True
        c += 1
    return c != 8
```

Iterate through the 8 queens until the first that occurs on an already-occupied diagonal. If no such queen is found, then the permutation represents a solution, or stop at 8 if none.

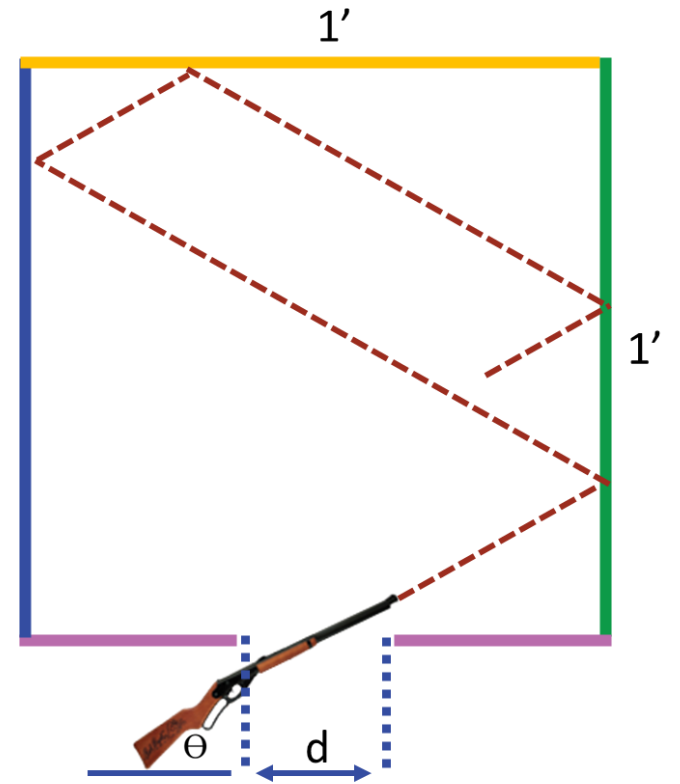
Commentary. Who would have guessed that the Eight Queens Problem reduces to:

```
def next_permutation(R: list[int]) -> None:
    """
    Update R[0..7] to be the next permutation of
    1,2,3,4,5,6,7,8 in a cycle of all 8! such permutations.
    """
    ...
```

which is left as a (not so easy) exercise.

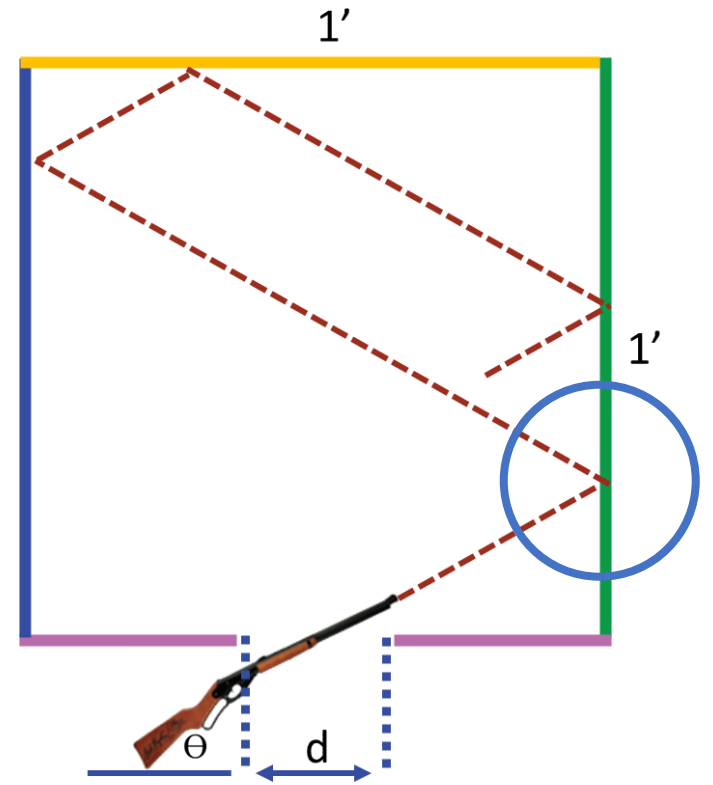
Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ , and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ , and outputs the total distance the bee-bee travels before it exits.

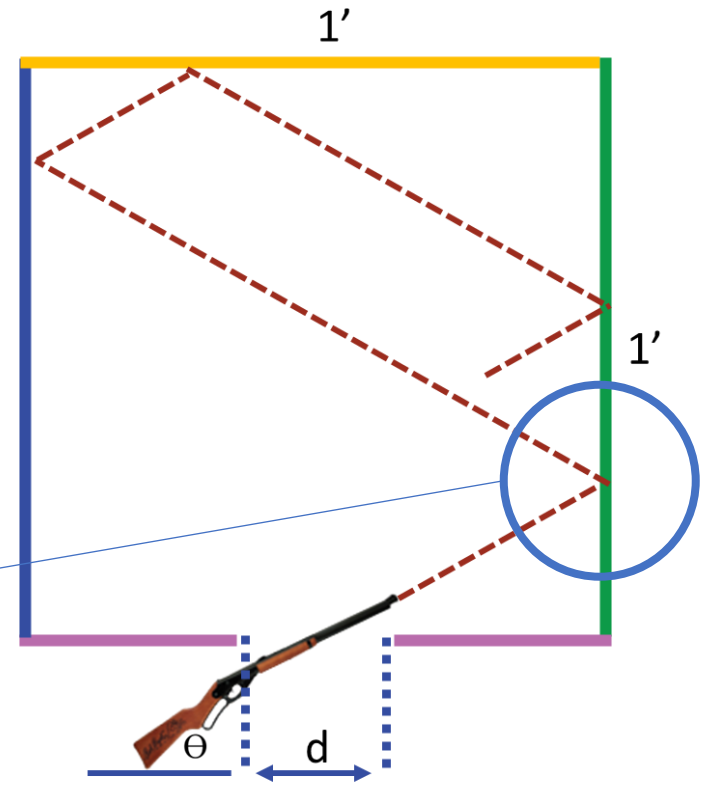
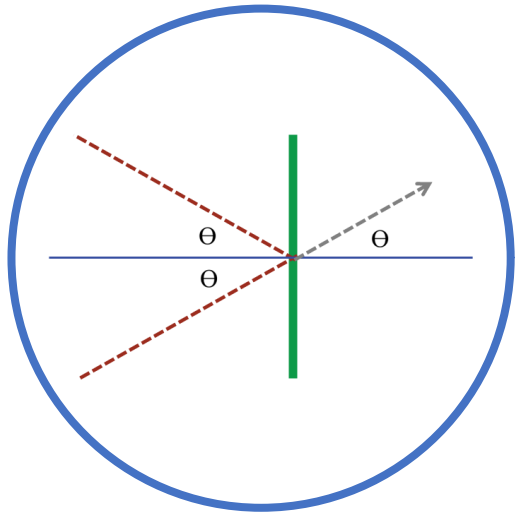


Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ , and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

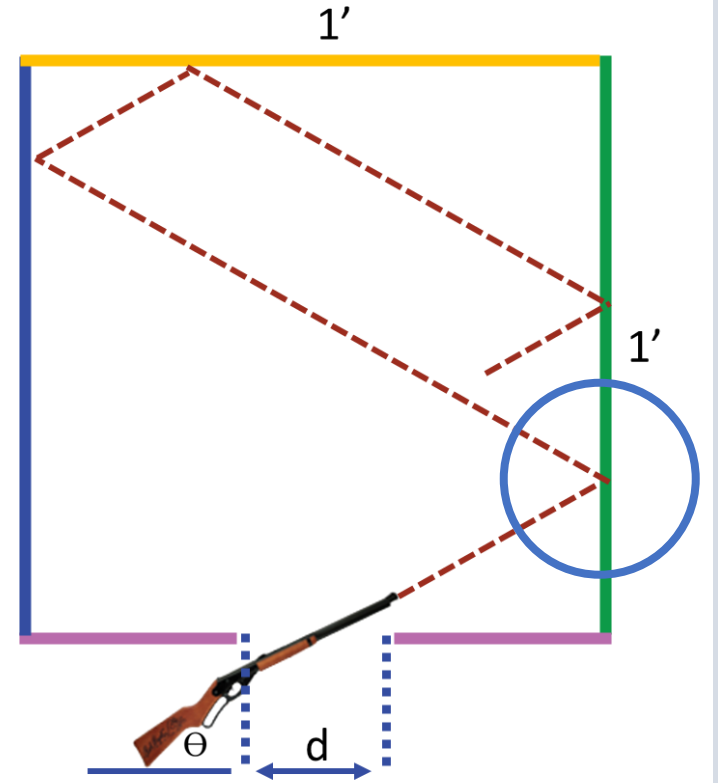
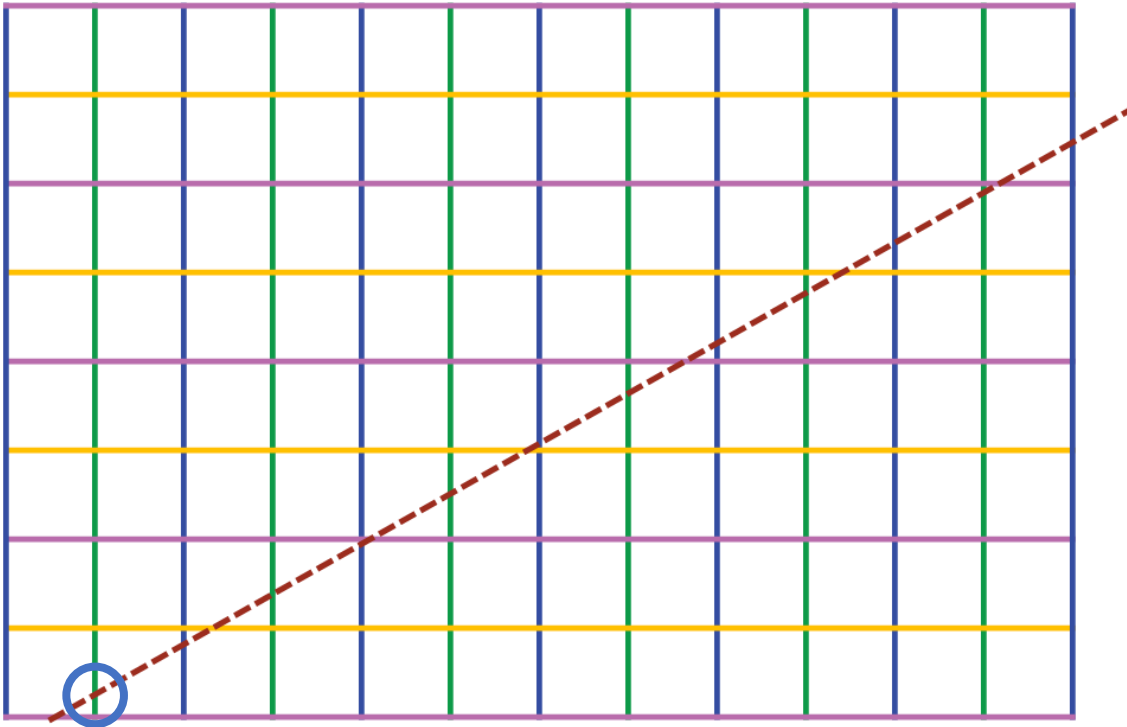
Problem Statement. Write a program that inputs d and Θ , and outputs the total distance the bee-bee travels before it exits.



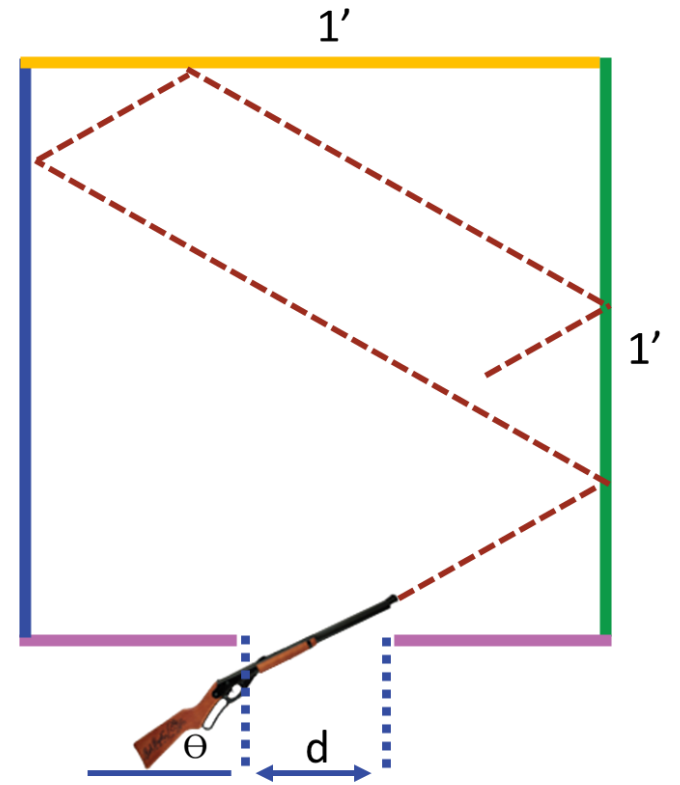
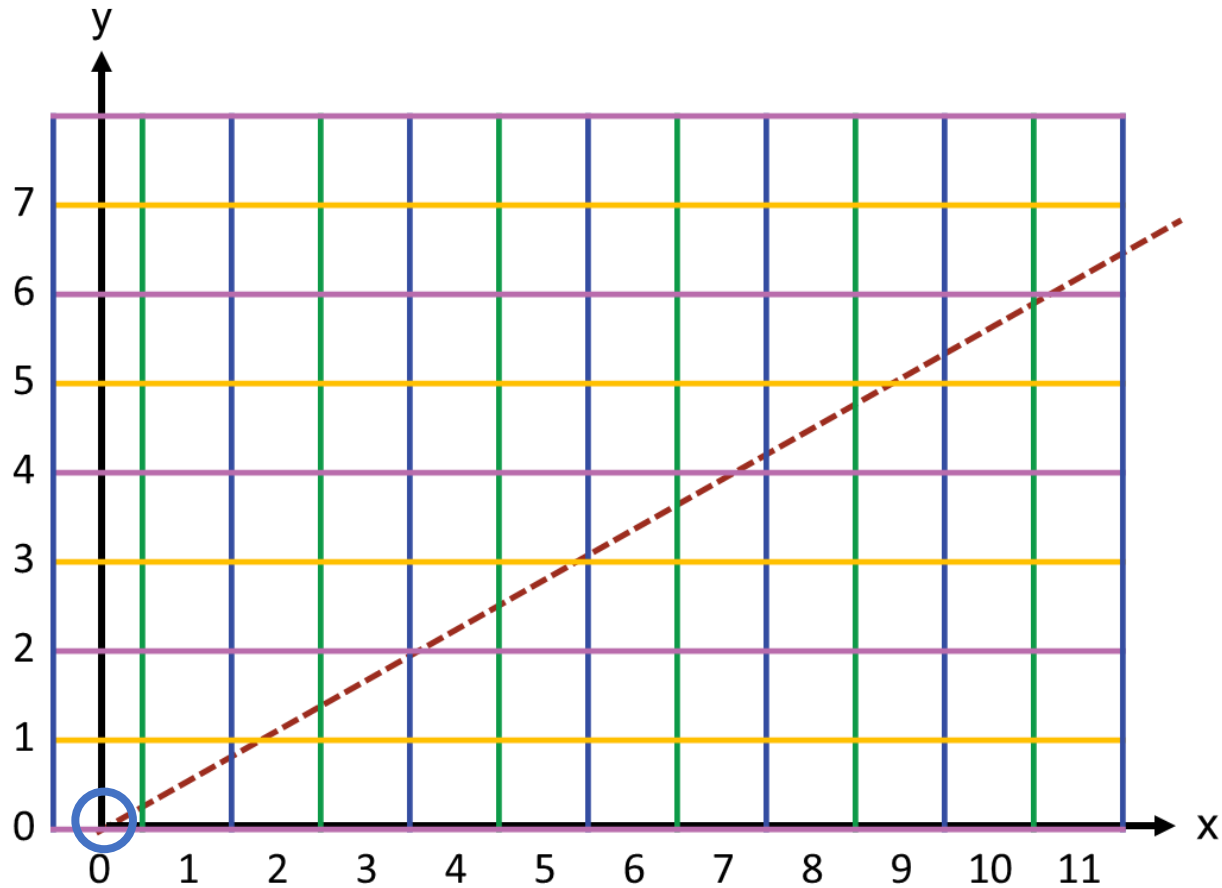
Problem Reduction. Allow the bee-bee to break through the wall of the box, and proceed in a straight line trajectory.



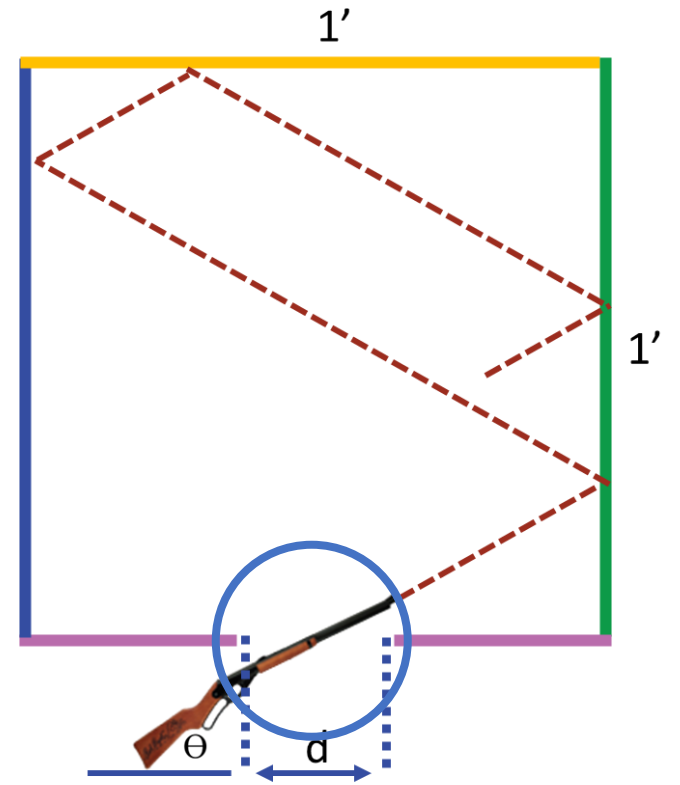
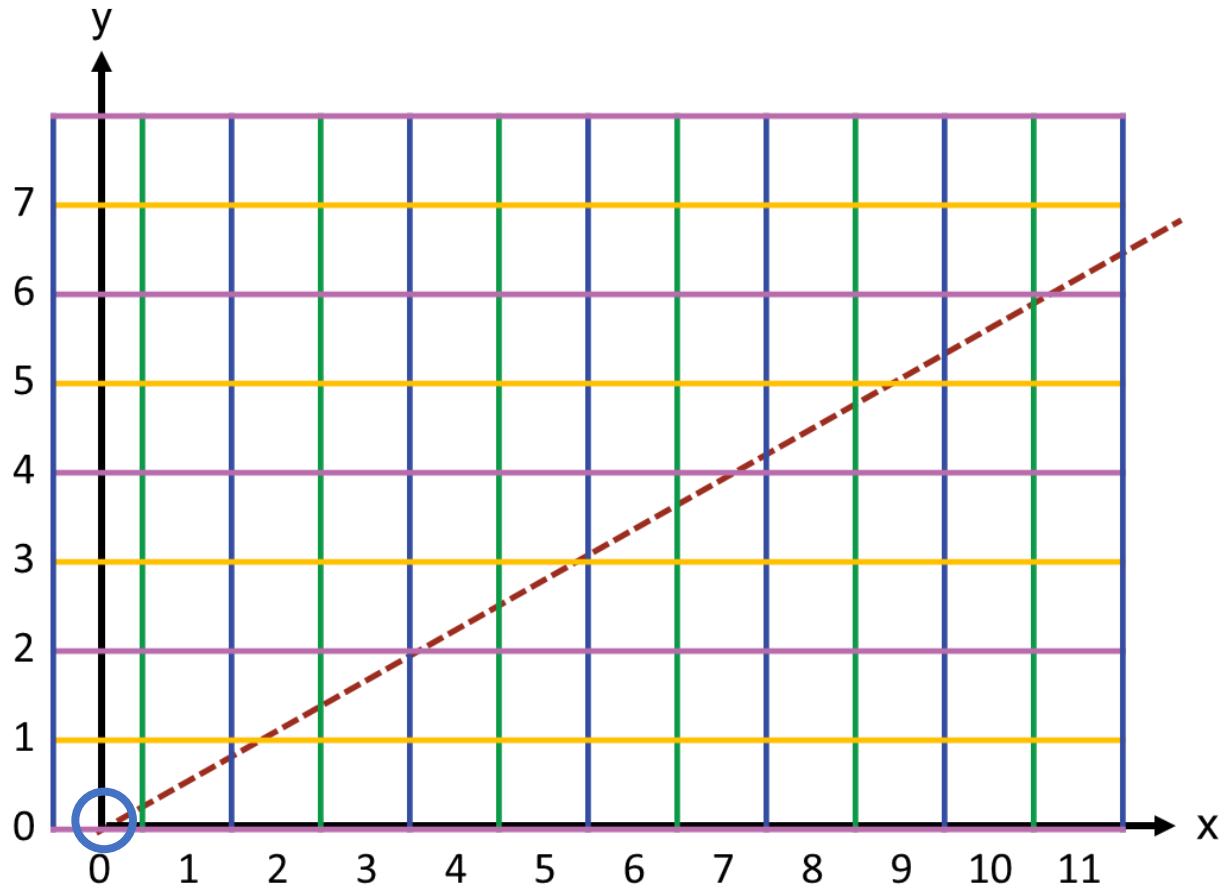
Problem Reduction. Consider the plane to be tiled with reflections of the box.



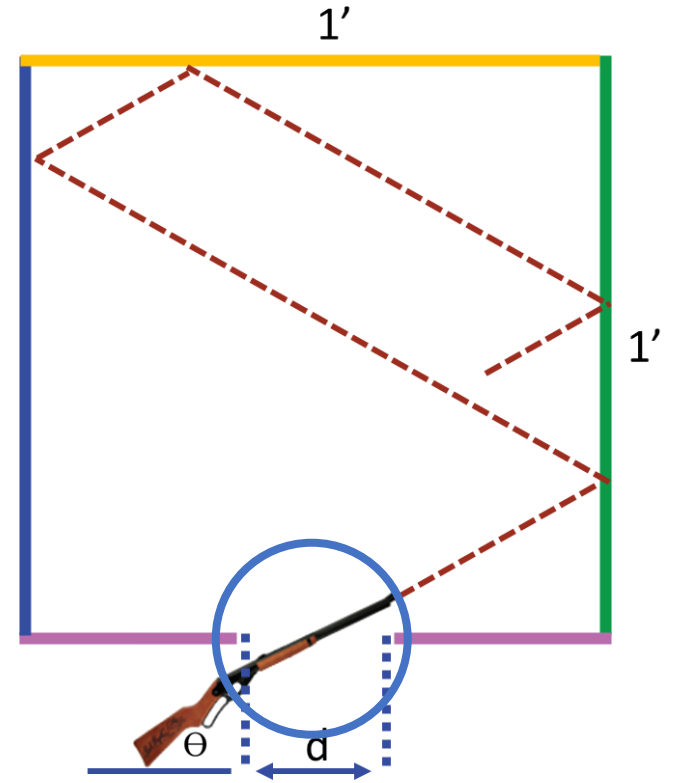
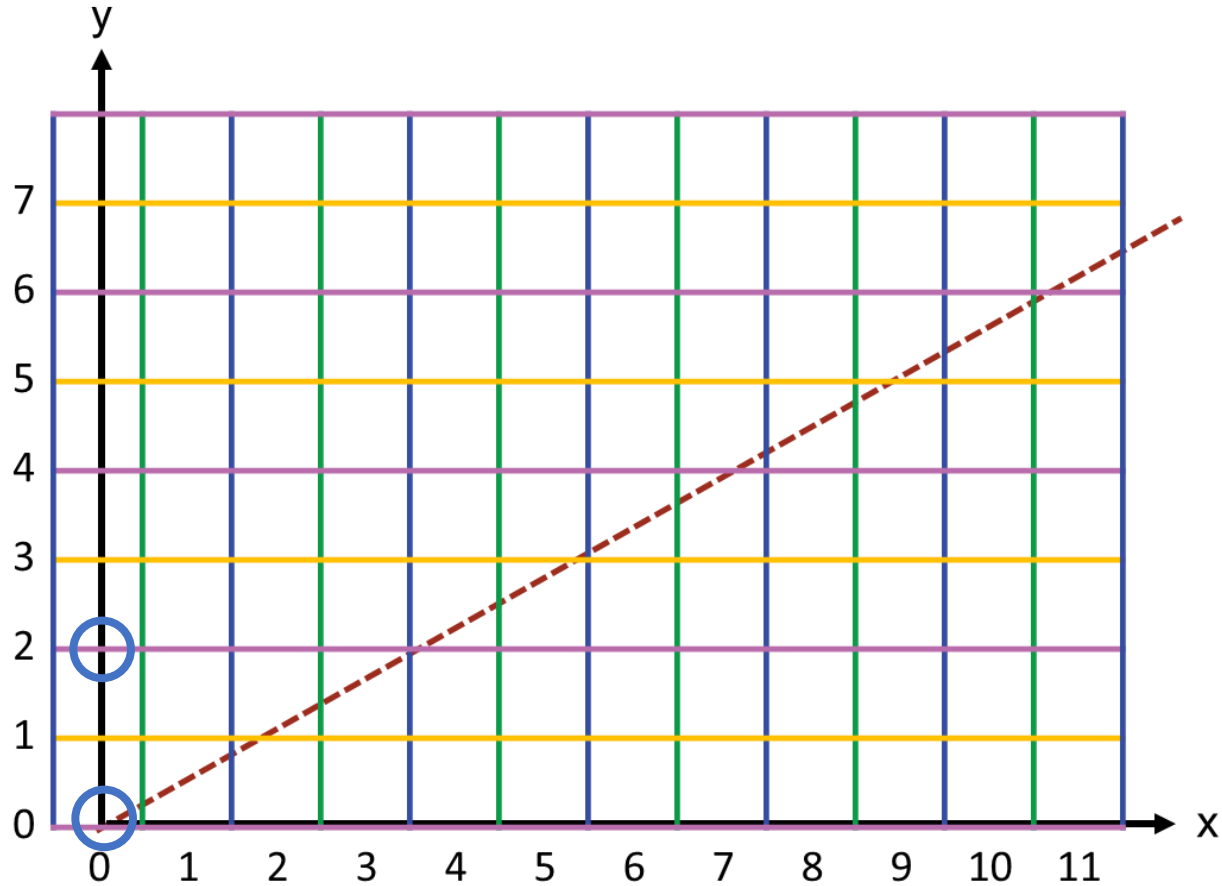
Problem Reduction. Add Cartesian coordinates with origin centered in the middle of the slit of the lower-leftmost box.



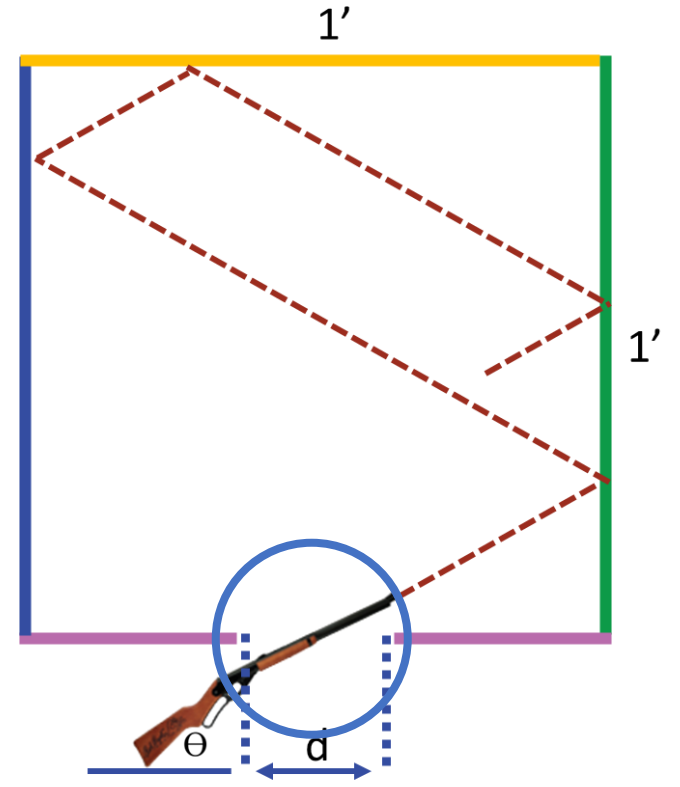
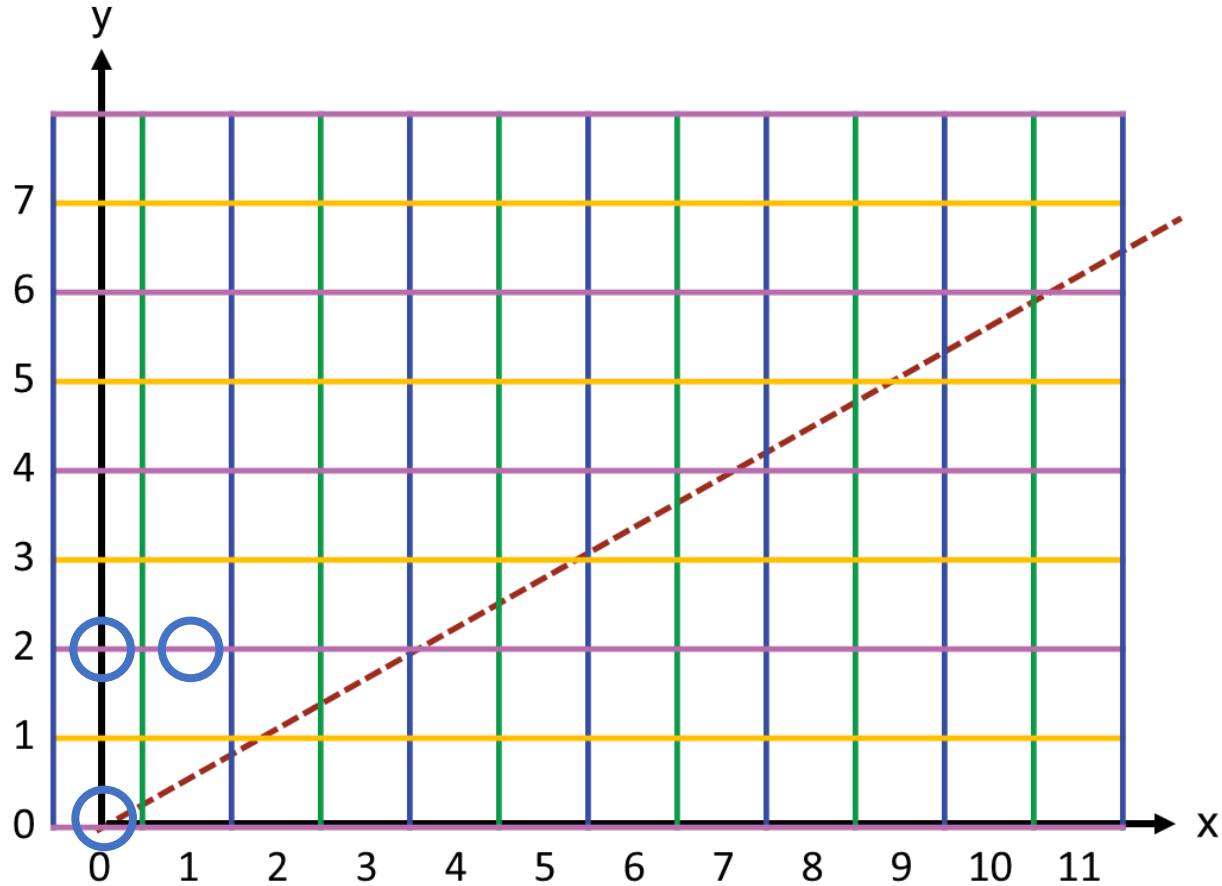
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x .



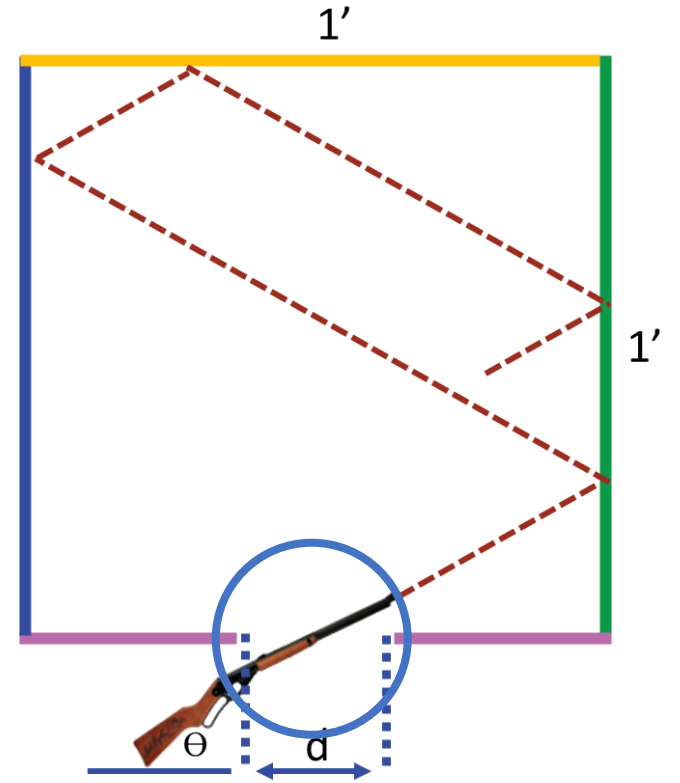
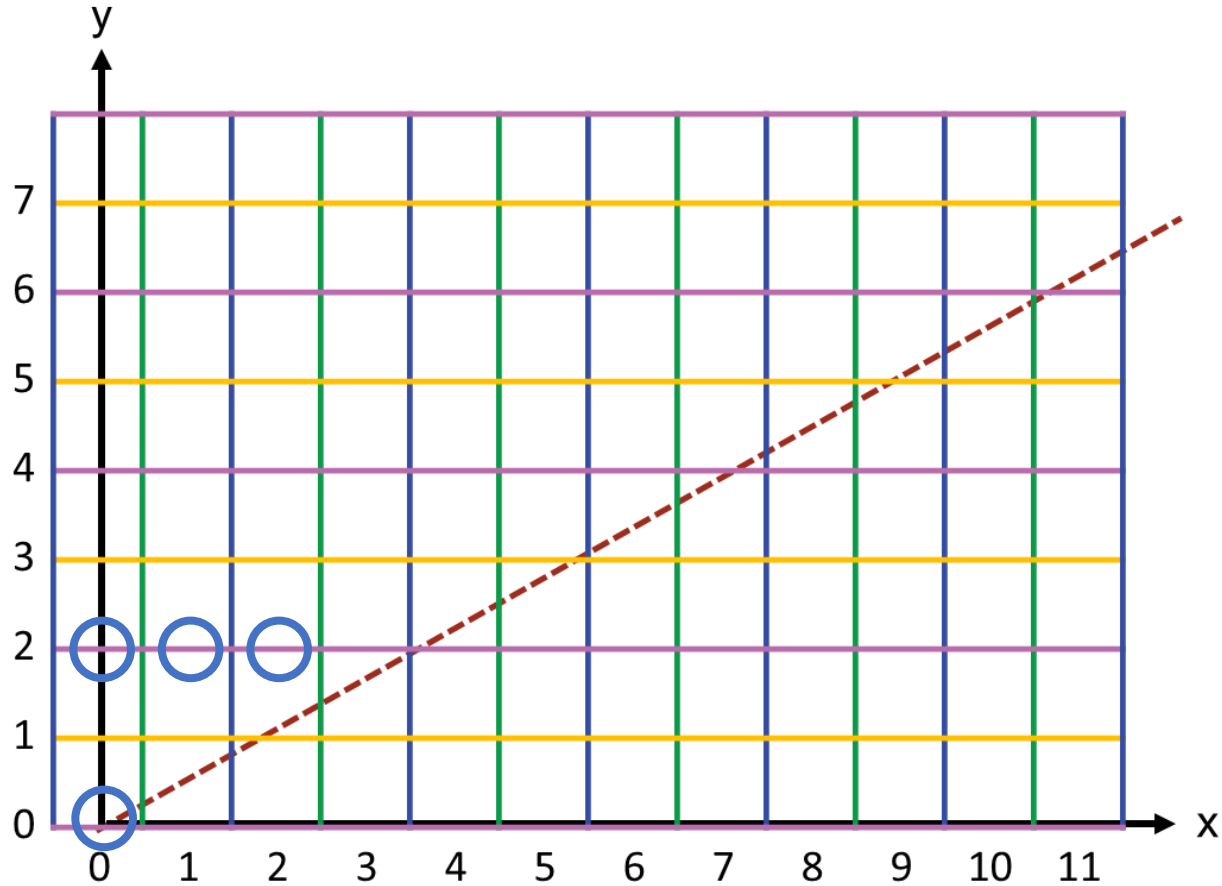
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x .



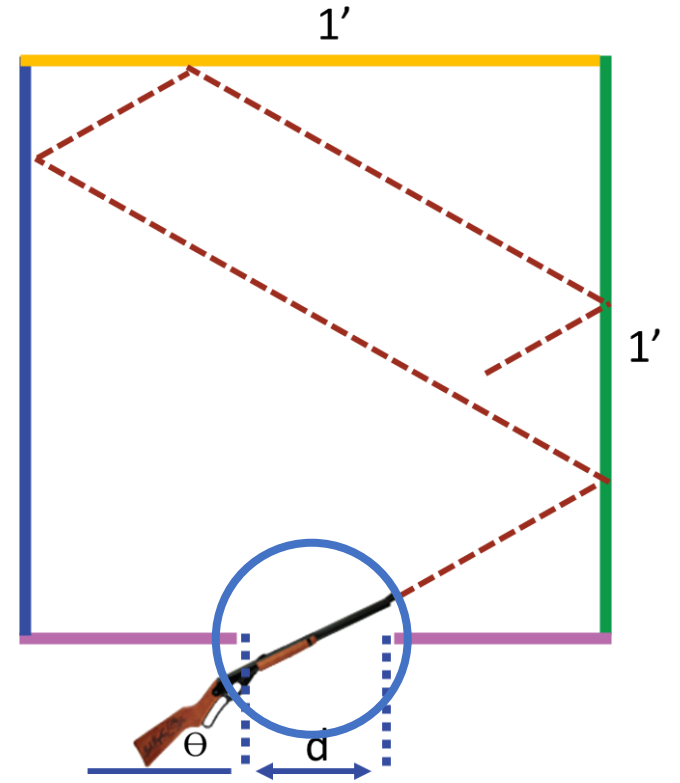
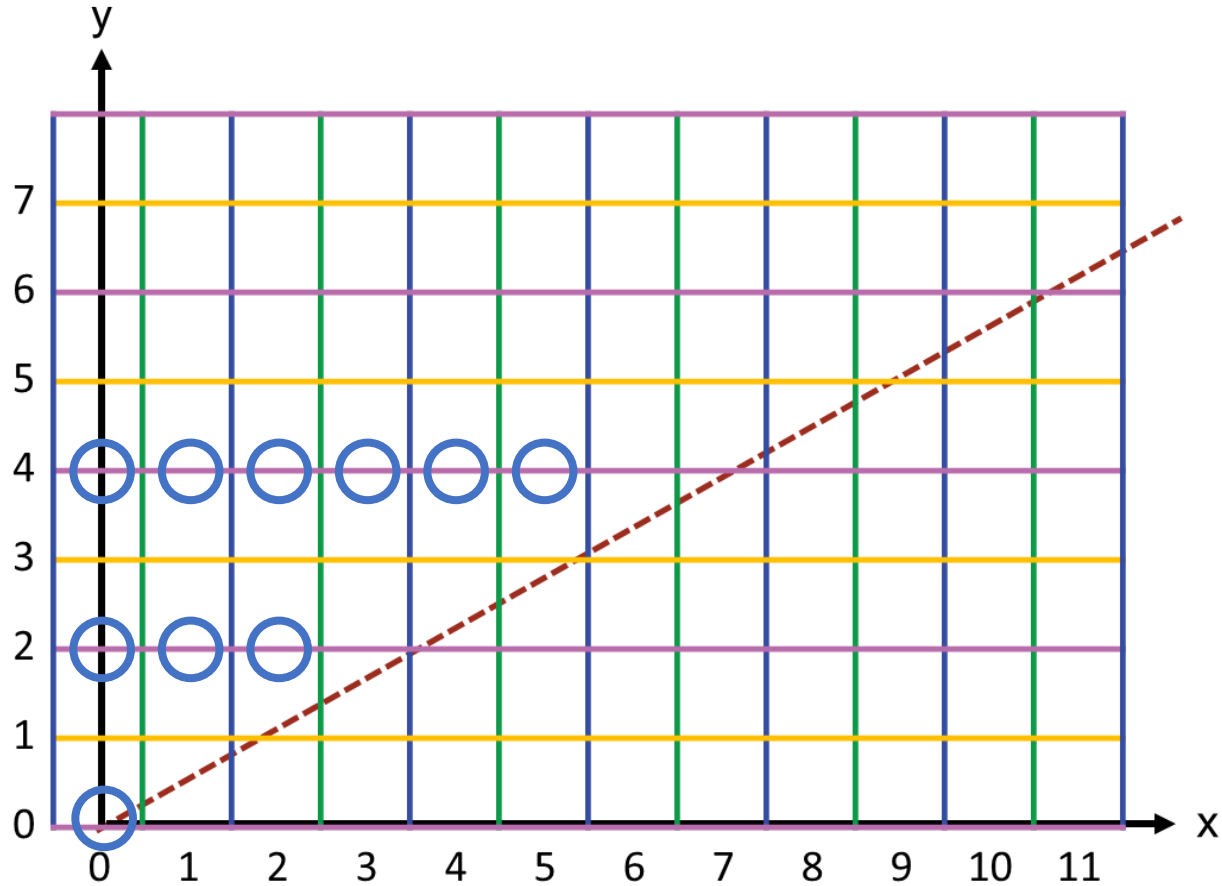
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x .



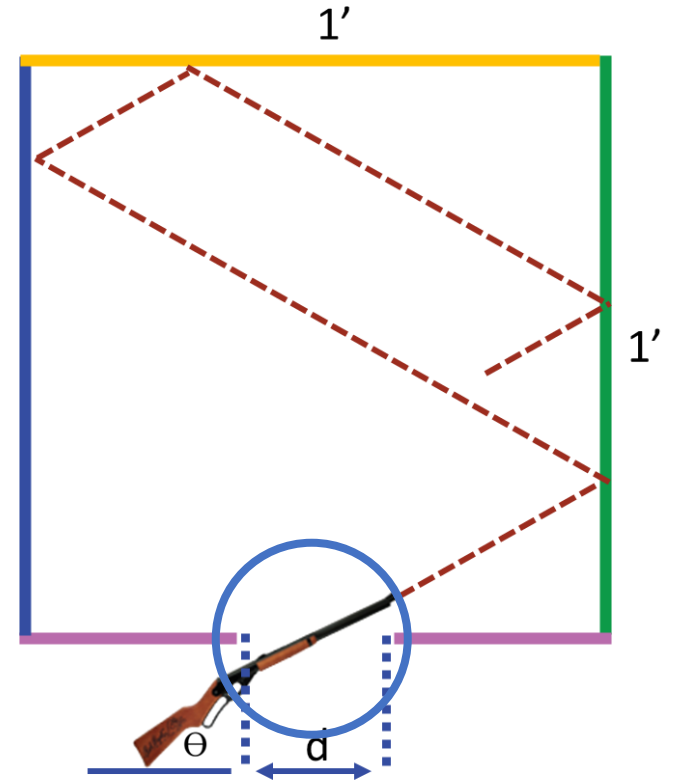
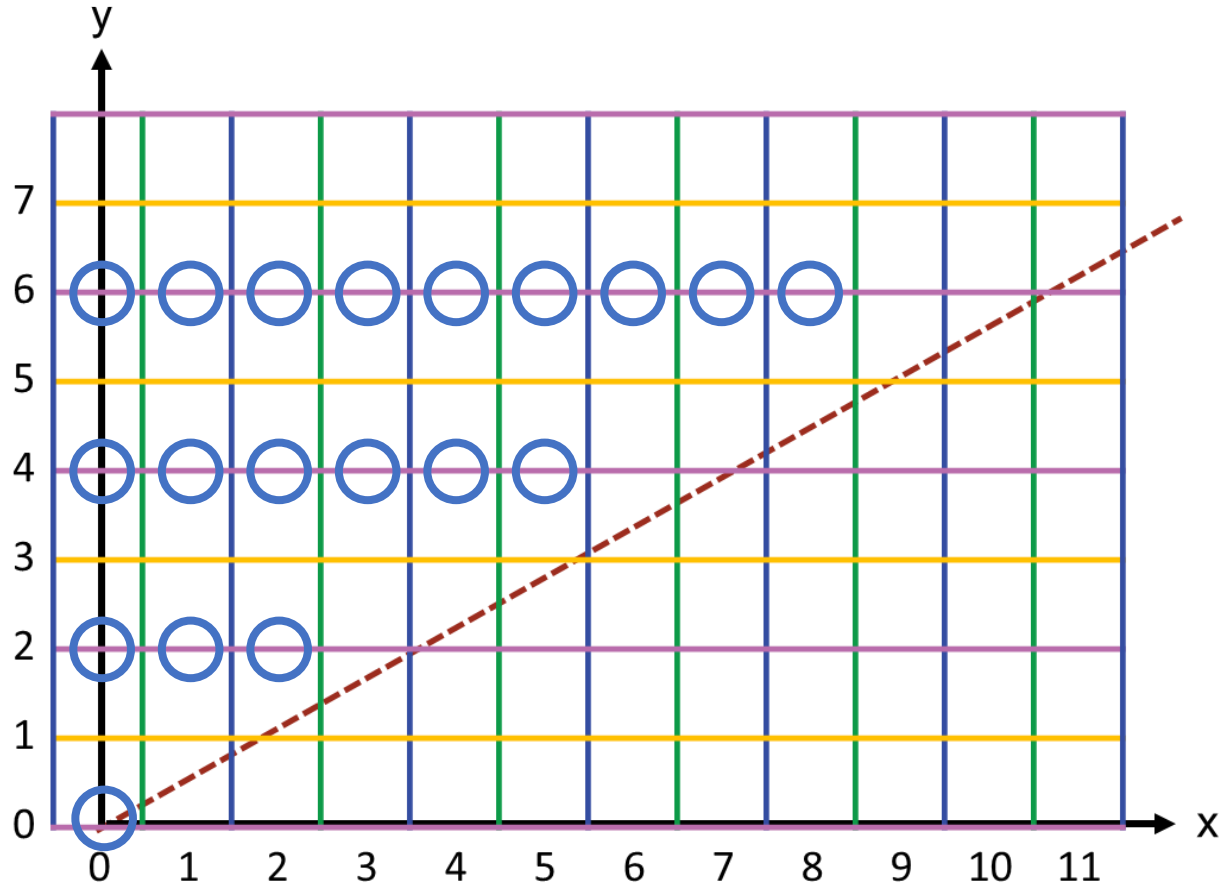
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x .



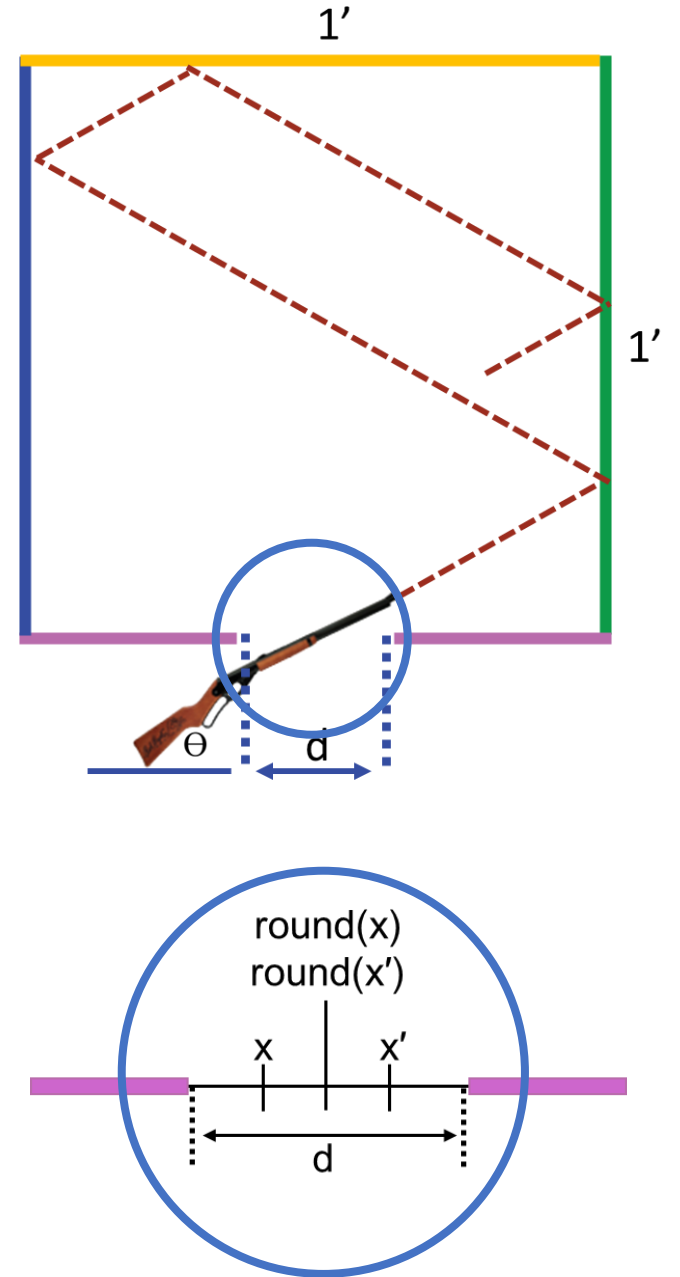
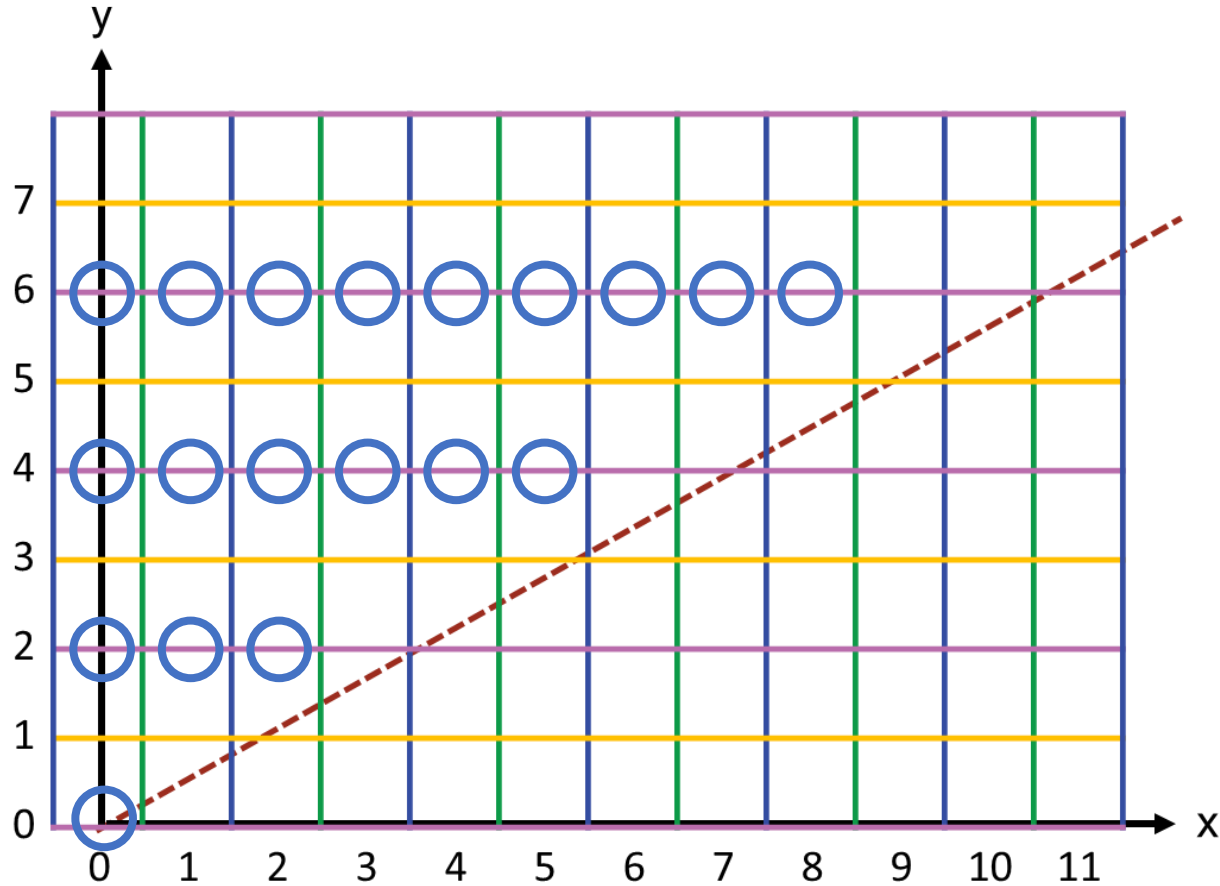
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x .



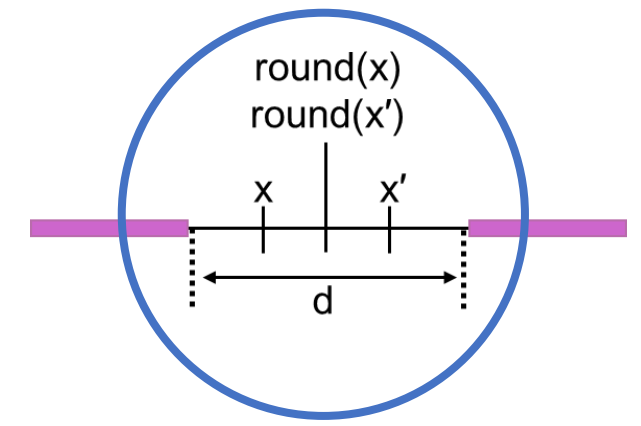
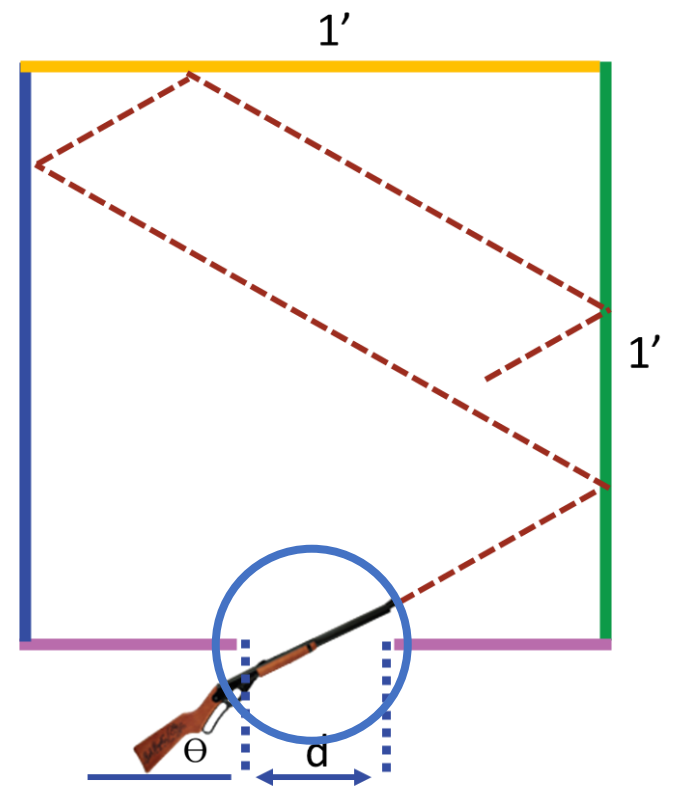
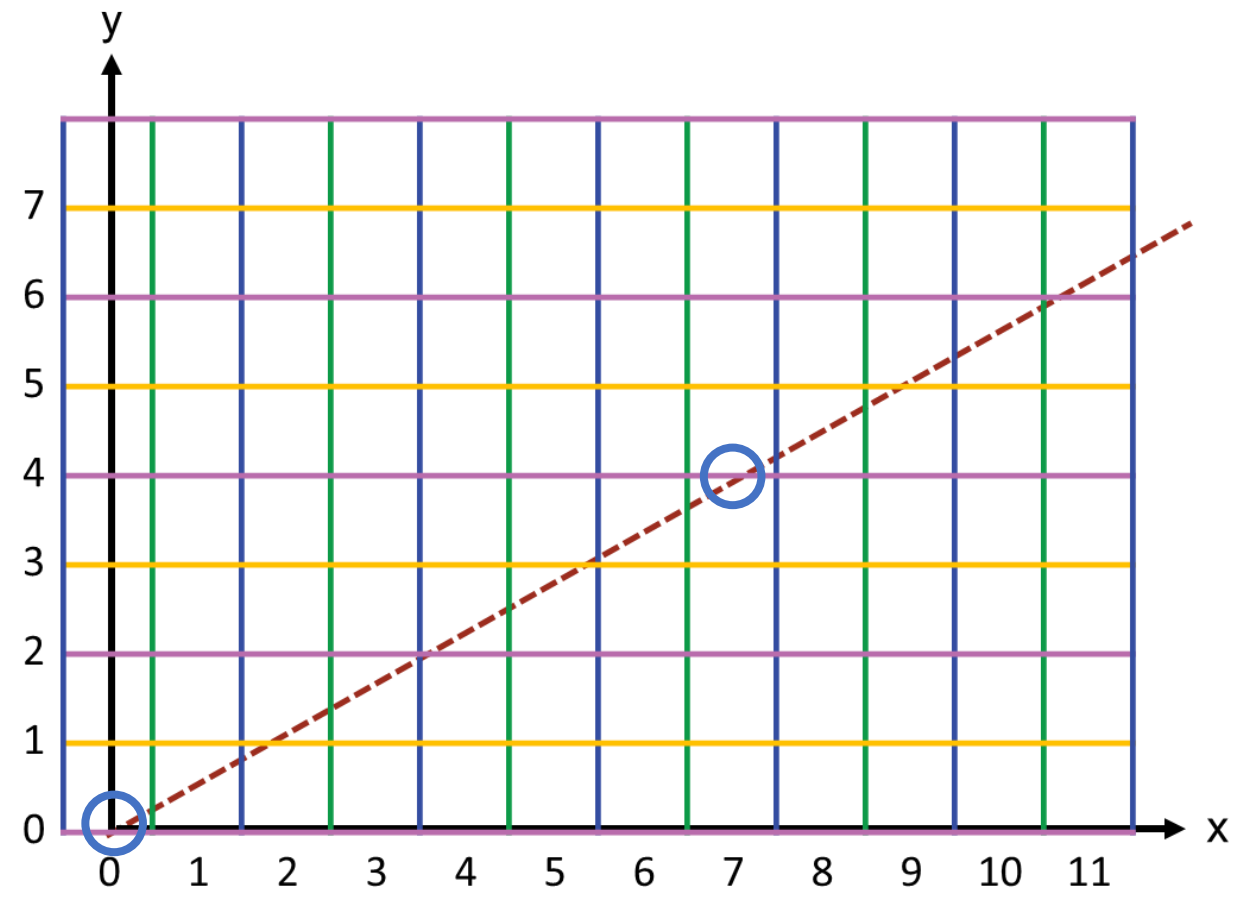
Problem Reduction. The slit replicas are on even values of y , centered on integer values of x , etc.



Problem Reduction. We seek the smallest (even) y such that the trajectory falls within $d/2$ of the center of a slit replica.



Problem Reduction. We seek the smallest (even) y such that the trajectory falls within $d/2$ of the center of a slit replica.

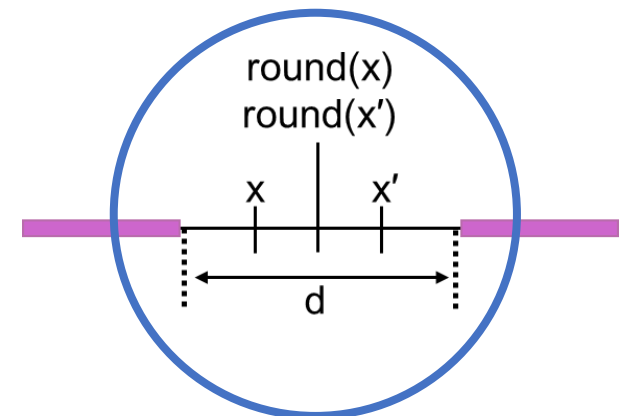
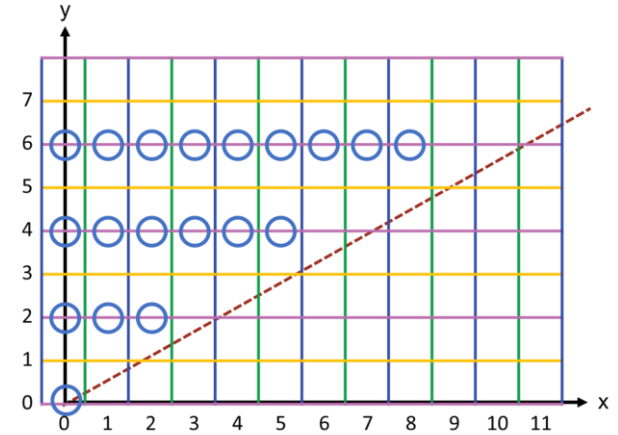


Problem Reduction. We seek the smallest (even) y such that the trajectory falls within $d/2$ of the center of a slit replica.

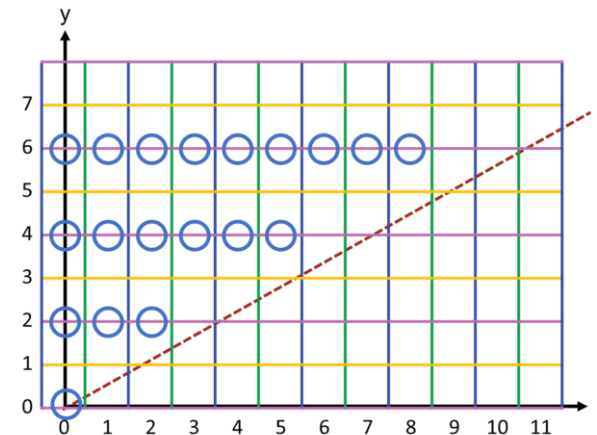
```

y: int = 2
while line-does-not-pass-through-a-slit-at-y:
    y += 2
# Output the length of the line between  $\langle 0,0 \rangle$  and  $\langle x,y \rangle$ , where  $x$  is
# computed from  $y$  and  $\theta$ .

```



Problem Reduction. We seek the smallest (even) y such that the trajectory falls within $d/2$ of the center of a slit replica.



```

y: int = 2
while math.abs(x(y, theta) - math.round(x(y, theta))) >= d / 2:
    y += 2
# Output the length of the line between <math>\langle 0,0 \rangle</math> and <math>\langle x,y \rangle</math>, where  $x$  is
# computed from  $y$  and  $\theta$ .

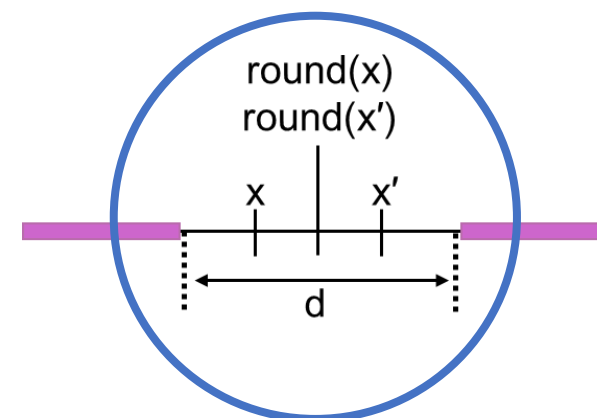
```

where

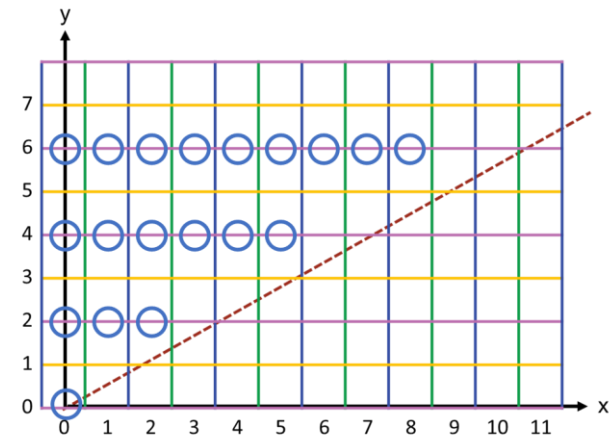
```

def x(y: float, theta: float) -> float:
    <compute x from y and theta>

```



Problem Reduction. Output the length of the trajectory for that y and the corresponding x .



```

y: int = 2
while math.abs(x(y, theta) - math.round(x(y, theta))) >= d / 2:
    y += 2
print(hypotenuse(x(y, theta), y))

```

where

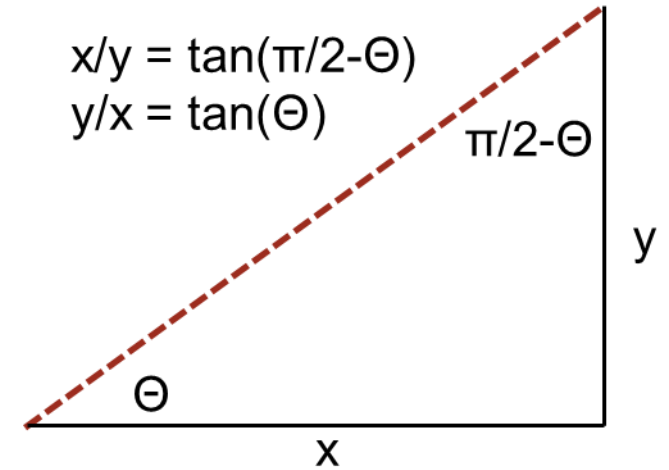
```

def hypotenuse(x: float, y: float ) -> float:
    """Return length of hypotenuse of triangle with sides x and y."""
    return math.sqrt(x * x + y + y)

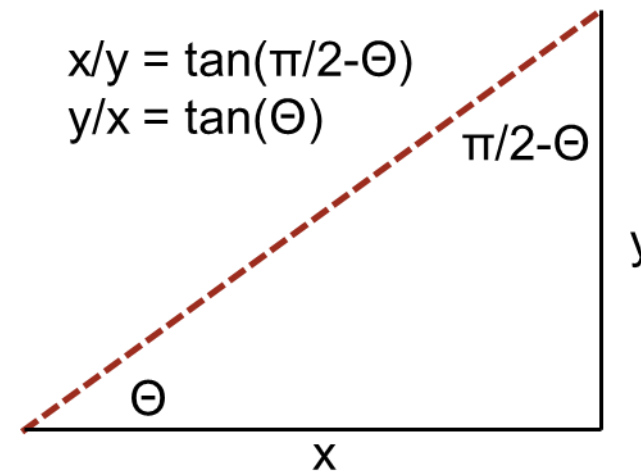
```

Final detail: Compute x from y and θ using a bit of trigonometry and care for numerical stability.

```
def x(y: float, theta: float) -> float:  
    <compute x from y and theta>
```



Final detail: Compute x from y and θ using a bit of trigonometry and care for numerical stability.



```
def x(y: float, theta: float) -> float:
```

```
    """
```

```
    Return the x at which a line with slope theta radians has a given
    y intercept.
```

```
    """
```

```
    if theta < math.pi/4 or theta > 3*math.pi/4:
```

```
        return y / math.tan(theta)
```

```
    else: return y * math.tan(math.pi/2 - theta)
```

Reflection. Analysis and problem reduction resulted in simple code.

Contrast it with the complexity of the code that would have been needed to simulate each leg the bee-bee's trajectory.

```
def main() -> None:
    d: float = float(input())
    theta: float = (math.pi/180) * float(input()) # Input theta in degrees.
    y: int = 2
    while math.fabs(x(y, theta) - round(x(y, theta))) >= d / 2:
        y += 2
    print(hypotenuse(x(y, theta), y))
```

N.B. Boundary conditions not considered, e.g., d or θ of zero.

